Full Forest Treebanking

Woodley Packard

A thesis submitted
in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2015

Reading Committee:

Emily M. Bender, Chair

Stephan Oepen

Program Authorized to Offer Degree:
Computational Linguistics

©Copyright 2015

Woodley Packard

University of Washington

Abstract

Full Forest Treebanking

Woodley Packard

Chair of the Supervisory Committee:
Professor Emily M. Bender
Department of Linguistics

In this thesis, I present a new method of producing treebanks using constraint-based grammars. Rather than requiring an explicitly enumerated set of candidate analyses per utterance, my method works from an implicit representation, allowing the annotator to efficiently select the correct analysis from trillions of possibilities, without requiring the user or the computer to store or iterate over all of them.

I explain the advantages and disadvantages of this method, and show the details and motivation for the algorithms that make it possible. Relative to comparable prior art (i.e. top-N treebanking), my solution enables higher coverage treebanks without a significant reduction in annotation speed, and reduces storage and computational resource consumption.

TABLE OF CONTENTS

Page

List of Figures ii

Acknowledgements iii

Chapter 1: Introduction 1
- 1.1 Constraint-based Grammar 1
- 1.2 Treebanks 2
- 1.3 Motivation 2
- 1.4 Roadmap 3

Chapter 2: Literature Review 5
- 2.1 Hand-corrected Treebanking 5
- 2.2 Discriminant-based Treebanking 6
- 2.3 Ambiguity Packing 9
- 2.4 Summary 11

Chapter 3: Implementing Full-Forest Treebanking 12
- 3.1 Representing a Forest 12
- 3.2 Counting Readings 16
  - 3.2.1 Computing an Upper Bound Using Dynamic Programming 16
  - 3.2.2 Counting a Subforest 17
  - 3.2.3 Collapsing Unary Chains 19
  - 3.2.4 Counting a Subforest Revisited 24
- 3.3 Enumerating Discriminants 26
- 3.4 Human-Computer Interaction 29
  - 3.4.1 Navigating the Discriminants 29
  - 3.4.2 Viewing Partial Results 29
  - 3.4.3 Latent Unification Failures 30
- 3.5 Summary 31

Chapter 4: Preliminary Experiments and Results 32
4.1 500 Chances 32
4.2 How Much Effort? 36
4.3 Summary 38

Chapter 5: Conclusion 40

References 42

LIST OF FIGURES

Figure Number Page

3.1 Example non-packed parse chart 14
3.2 Example packed parse chart 15
3.3 Example unary-chain packed parse chart 22
4.1 Top-500 recall overestimate graph 35

LIST OF ALGORITHMS

3.1 Computing unary chains 20
3.2 Eliminating unary edges from the chart 21
3.3 Enumerating discriminants 28

ACKNOWLEDGEMENTS

I am indebted to Emily Bender and Stephan Oepen for their helpful comments and advice throughout this project. I additionally am grateful to Daniel Flickinger for his insights into the treebanking process, the time he devoted to treebanking trials, and his feedback on user interface design. Finally, I wish to thank my wife Elizabeth for making it possible for me to spend the time I needed to complete it.

Chapter 1

INTRODUCTION

This thesis concerns methodology for the task of *treebanking* with a *constraint-based grammar*. I seek to shed light on the advantages and disadvantages of asking a human annotator to consider all available analyses, as compared to considering just an automatically selected subset. This chapter will introduce some of the key terms and concepts, explain the motivation for large-scale treebanking, and orient the reader with respect to the following chapters.

1.1 Constraint-based Grammar

A *grammar* of a natural language, for the purposes of this thesis, is a formal system embodying an effective procedure to determine whether or not a given string of sounds or letters constitutes a meaningful sentence in that language; moreover, a grammar typically associates to sentences of the language a proof of grammaticality known as a *derivation* or *analysis*. These analyses can expose linguistically interesting properties of the sentence, such as constituent structure or predicate–argument relationships, and hence are valuable artifacts both in the study and exploitation of natural language.

A *constraint-based grammar* is a grammar in which some portion of each rule is encoded as a body of intersecting constraints, collectively determining the rule's applicability in any given context. Examples include Generalized Phrase Structure Grammar (Gazdar, 1985), Head-driven Phrase Structure Grammar (HPSG; Pollard & Sag, 1994), and Lexical Functional Grammar (Bresnan, 2001). The present work focuses on HPSG, but most of the points are applicable to other grammatical formalisms, whether constraint-based or not.

1.2 Treebanks

One common use of trees in linguistics is to describe the constituent structure of an utterance, together with the syntactic categories of the constituents (e.g. encoding the part of speech and valency conditions of a phrase's head with symbols such as V for a verb and VP for a constituent headed by a verb that has already realized all of its complements). In formal grammar, the related *derivation tree* describes constituent structure together with the identity of the *rules* that license those constituents; a derivation tree constitutes a proof of grammaticality.[1] Ambiguity inherent to natural language typically causes grammars to license more than one distinct analysis for some sentences; indeed, broad-coverage grammars frequently license tens of thousands of analyses for long sentences. This complicates the use of a grammar as a tool, since frequently it is only the analysis corresponding to the meaning that was intended by the speaker that is of interest for further study.[2] The solution, both for detailed study by linguists and for enabling automatic exploitation by computer systems, is to invest some manual effort in recording the intended analysis for each sentence in some corpus of interest. In the case of automatic systems, machine learning can be used to automatically select the correct analysis (or at least a reasonably good analysis) for unseen text on the basis of patterns discovered in a manually annotated treebank.

With the advent of ubiquitous personal computers and growing interest in computer-assisted analysis of language, the development of large corpora of natural language accompanied by syntactic analyses in the form of trees became a practical possibility. Such an annotated corpus is known as a *treebank*.

1.3 Motivation

The primary purpose of a treebank is to facilitate the estimation of various statistics about languages and the grammars that describe them. In general, larger treebanks provide more accurate statistics. Even a small treebank can help a grammarian gauge the grammar's

---

[1] This work will not discuss dependency trees, although many of the same principles apply.

[2] I take the notational liberty of presuming that the analysis corresponding to the speaker's intended meaning is the *intended analysis*.

coverage on some particular genre or domain of text, which in turn can help focus grammar development effort towards areas where coverage is the most impoverished. Larger treebanks can be mined for frequency information on syntactic phenomena, for instance shedding light on the question of whether predicative or attributive usage is more common for some specific adjective or class of adjectives. The larger the treebank is, the more detailed the questions become (e.g. investigating rarer phenomena) that can be answered. In addition to such “hands-on” research applications, treebanks are invaluable for enabling computers to automatically resolve ambiguity, where the size of the treebank again is a major factor in determining the effectiveness.

The creation of large-scale treebanks is a labor-intensive task, and much research has been devoted to developing methods to facilitate the process (e.g. Marcus et al., 1993, Carter, 1997). In the context of HPSG, the most common procedure is to automatically select the N most probable analyses (with $N = 500$ often) according to a statistical model, and subsequently ask a human expert to rule out swaths of unintended analyses until either the intended analysis is found or all analyses have been ruled out (Oepen, Flickinger, et al., 2004). The motivation of the present work is to continue this research trajectory by eliminating the arbitrary (from a linguistic perspective) limit of 500 candidates; I hypothesized that discarding all but the 500 most probable analyses was leading to reduced coverage for treebanks created in that manner. I further hypothesized that ambiguity management techniques from the parsing literature could be adapted to resolve the engineering problems raised by treebanking with a very large number of candidate analyses.

1.4 Roadmap

Chapter 2 gives an overview of treebanking techniques. In Chapters 3 and 4, I address four research questions: First, how computationally tractable is it to present all of the analyses licensed by the grammar even for very complex sentences? Second, how practical is it from a user interface perspective? Third, what is lost by automatically discarding all but 500 analyses? And fourth, how much manual decision making is required when presenting all analyses to the human expert, relative to the amount required for top-N treebanking? The methods I use to approach these questions are engineering-oriented and experimental in

nature. I answer the first two of these research questions in Chapter 3, by designing and implementing algorithms and a user interface capable of supporting treebanking with all analyses available. The resulting system, called the Full Forest Treebanker[3], is already in use by several grammarians. I approach the second two research questions by conducting numerical experiments on existing treebanks constructed under the top-500 paradigm to extrapolate estimates for the desired quantities. A more accurate approach would be to build a large scale treebank using the new platform, paralleling an existing top-500 treebank, and compare the two, but unfortunately as of yet no such large scale full forest treebank exists.

The primary conclusions of these investigations are that full forest treebanking is indeed practical and efficient, that top-500 treebanking likely discards desirable analyses for at least 2% of sentences, and that full forest treebanking probably does not require in excess of 15% more human effort than top-500 treebanking. The main practical contributions of this work are the algorithms that support full forest treebanking and the implemented platform enabling annotators to actually use the technique. In chapter 5, I summarize the contributions of the preceding chapters and offer thoughts on future research directions.

[3]Abbreviated FFTB; the software is open source under the MIT license, and is available from svn://sweaglesw.org/svn/treebank/trunk/.

Chapter 2

LITERATURE REVIEW

This chapter paints a summary of the landscape of previous research on treebanking methodology, with an emphasis on discriminant-based treebanking, in order to give the reader an idea of the current state of the art, and to situate the present work with respect to alternative approaches.

2.1 Hand-corrected Treebanking

Recall that a treebank is a corpus of text annotated with syntactic analyses in the form of trees. While early corpora (e.g. the Brown Corpus of Francis & Kucera, 1982) contained text annotated with part of speech information, the earliest sizeable published corpus containing syntactic annotations was the Penn Treebank (Marcus et al., 1993). The PTB consists of 4.5 million words of American English, of which about 3 million words have been annotated with syntax trees.

Both layers of annotation in the PTB (part of speech and syntax) were accomplished by first using an automated tool to assign a best-guess analysis, and then manually correcting the result to produce gold-standard annotations. The tool used to perform the initial syntactic annotation was Fidditch (Hindle, 1983), a partial parser which outputs a sequence of tree fragments. Fidditch attaches the fragments together to the degree that it is confident, and leaves more difficult attachments to the manual annotators. Marcus et al. (1993, p. 320) describes the process of designing the syntactic annotation scheme as “highly pragmatic and strongly influenced by the need to create a large body of annotated material given limited human resources.” In other words, linguistic precision (e.g. the distinction between arguments and adjuncts) was sometimes sacrificed in favor of annotation speed. Experienced annotators were able to maintain an output of 750–1000 annotated words per hour (Marcus et al., 1993).

The flexibility of analysis afforded by the process of treebanking by hand-correcting trees is simultaneously an advantage, in that the scope of annotatable text is limited only by the user's imagination, and a liability, since the resulting structures are not held formally accountable to any linguistic standard of consistency. In cases where the annotation guidelines are murky, annotators are apt to make inconsistent decisions (for instance, see Dickinson, 2008 for some insight into the consistency of the PTB). Furthermore, regardless of how carefully worded the annotation guidelines are, human beings make mistakes.

One method of alleviating this concern somewhat is the use of a formal grammar to validate the analyses. Alternately, a parser can be used to enumerate all of the grammatical analyses for each utterance to be annotated. The annotator's job is then to select which of the competing analyses (if any) is correct. This process offers the guarantee that every analysis in the resulting treebank is well-formed with respect to the accompanying formal grammar.

2.2 Discriminant-based Treebanking

Since the number of candidate analyses for an utterance can be very large, it is unrealistic to expect the annotator to quickly and reliably select the correct analysis from a list by simple inspection. Carter (1997) presents a more practical method of treebanking based on this paradigm. By presenting the annotator with a list of *differences* between the analyses rather than a list of the analyses themselves, his system enables the annotators to much more quickly comprehend and resolve the choices they are being presented with. Annotator efficiency is reported as 170 sentences per hour in the ATIS domain (Air Travel Information System; Hemphill et al., 1990). Since average sentence length in that corpus is about 11 words, we can extrapolate a speed of about 1870 annotated words per hour — roughly twice the speed reported for the manual annotation of the PTB, with the added benefit of enforced grammatical consistency.

Carter's method of discriminant-based treebanking gained popularity and was adopted by a number of other grammar-based treebanking projects, notably the Redwoods project (Oepen, Flickinger, et al., 2004) using Head-driven Phrase Structure Grammar (Pollard & Sag, 1994) and the TREPIL project (Rosén, De Smedt, et al., 2005) using Lexical Functional

Grammar (Bresnan, 2001).

The Redwoods project adapted discriminant-based treebanking to the LinGO English Resource Grammar (henceforth ERG; Flickinger, 2000, 2011), a broad-coverage precision computational grammar of American English formally grounded in HPSG and a framework for meaning representations called Minimal Recursion Semantics (henceforth MRS; Copestake et al., 2005). Oepen et al. argue that the discriminant-based method allows the creation of a treebank that is (i) both rich and consistent in its linguistic analyses, and (ii) dynamic, in the sense that the analyses can be semiautomatically kept up-to-date with ongoing development of the ERG with relatively little human effort.[1] Oepen et al. report an annotation rate of about 2000 sentences per week for a single annotator, on text with an average sentence length of 7.5 words. Assuming the annotator worked 40 hours per week, this translates to 375 words per hour. This is substantially slower than the speed reported by Carter; however, it is not clear whether the complexity of the analyses and coverage of the grammar are similar enough for direct comparison to be meaningful.

One of the most compelling properties of discriminant-based treebanking is that, provided the decisions used to find the desired tree are recorded, they can be replayed with a slightly different forest (arising, say, from reparsing the same text with an updated grammar) with a good chance of isolating the new desired tree, if one is still available. This technique has allowed the Redwoods treebanks to be kept up-to-date with the ERG as it has undergone continued development.

The Redwoods infrastructure was subsequently deployed to produce a number of additional treebanks for various languages. The treebank bundled with the ERG as of 2013 consisted of more than 75,000 sentences. Examples in other languages include the Hinoki treebank of Japanese (Bond et al., 2004, about 25,000 sentences), the Tibidabo treebank of Spanish (Marimon, 2010, about 15,000 sentences), the CINTIL corpus of Portuguese (Branco et al., 2012, 1,204 sentences), and the ParDeepBank multilingual treebank of the Wall Street journal (Flickinger, Kordoni, et al., 2012).

[1] Oepen et al. also emphasize another benefit, viz. (iii) that the information can be exported into many different formats, each containing a different view on the rich annotations, but this is less relevant to the topic at hand.

The question of annotation speed with discriminant-based HPSG treebanking was first methodologically addressed by Tanaka et al. (2005), in the process of building the Hinoki treebank of Japanese dictionary definitions. This work used a statistical part-of-speech tagger to preselect certain discriminants in cases where the tagger was determined to be reliable, yielding approximately a 16% speedup over the baseline where none of the discriminants were preselected. Absolute speed in the faster configuration is reported as 70 seconds per item, with an average item length of 10, yielding an annotation rate of about 500 words per hour. Again, it is difficult to compare this figure to those reported by Carter and by Marcus et al., since the languages and domains are very different. Treebanking speed can also be expected to depend on the level of ambiguity and complexity of the underlying grammar and its analyses, further confounding comparisons.

The question of speed was later revisited in the process of annotating Wall Street Journal text with ERG analyses (Zhang & Kordoni, 2010). A model was trained to predict which discriminants an annotator was most likely to want to make a decision about, and used to present discriminants to which the model assigned a high score in a more prominent place than those to which a low score was assigned. In previous Redwoods work, the list of discriminants was presented to the user in sorted order, with constituent length as the primary sorting criterion (and left-to-right in-sentence position as a secondary criterion). Compared to this baseline, the authors report an impressive 50% improvement in annotation throughput, from 62 sentences per hour to 96 sentences per hour when annotating the PARC 700 (a random sample of 700 sentences from section 23 of the Wall Street Journal portion of the PTB; King et al., 2003). Since the average sentence length for the PARC 700 is 19.8 words, these results correspond to a throughput of 1900 words per hour, which is very similar to the speed reported by Carter.

Lexical Functional Grammar (Bresnan, 2001) is another unification-based theory of syntax. LFG posits several separate layers of analysis (called c-structure, f-structure, etc.), whereas HPSG signs are monostratal. The LFG Parsebanker (Rosén et al., 2009), built as part of the LFG Pargram research program (a concerted effort to develop computational grammars for a number of languages; Butt et al., 2002), implements a treebanking environment similar to Redwoods. Discriminants are based on c-structure and f-structure instead of

derivation trees and MRS, but the principles involved are similar; c-structure discriminants allow users to make decisions about constituency, while f-structure discriminants facilitate disambiguating more abstract structures like predicate–argument relationships. The LFG Parsebanker enables users to view and apply both types of discriminants at once, whereas the Redwoods environment requires that the user select which type of discriminants to use before beginning a treebanking session. Baird and Walker (2010) describe using the Parsebanker system to annotate over 100,000 sentences from Wikipedia as training data for the Powerset search engine, a commercial venture that deployed LFG to provide natural language web search. Unfortunately, annotation speed is not reported.

These efforts have made it clear that discriminant-based treebanking is both practical and useful for building large-scale annotated corpora. Although the work is labor-intensive, much research focused on how to optimize the use of human annotators' time has allowed tens or hundreds of thousands of sentences to be annotated using the technique, under a variety of grammatical frameworks and implementations—a much more positive result than could have been expected if annotators had to select from among competing analyses without the aid of discriminants. I adopt this well-proven framework for my treebanking experiments in this thesis.

2.3 Ambiguity Packing

Grammars like the ERG can assign literally trillions of analyses to longer sentences. Nearly all of these are nonsensical given an ordinary context, and it can generally be presumed that only *one* of them was intended by the speaker, but in order to do anything with the intended meaning (such as recording it into a treebank), processing systems need an efficient way of overlooking all of the other candidate analyses, which (to the computer) look just as tempting as the intended one. The literature on the algorithms and data structures that support large-scale parsing is vast. I content myself to explore just one facet of that literature which is very relevant to the present work: the efficient handling of the enormous ambiguity inherent in natural language. In order to get any traction when interpreting highly ambiguous sentences (and they are not uncommon), it is necessary to avoid at all costs any operation which must iterate over all of the (trillions of) analyses. At first glance,

the notion of a system that can sift through exponentially[2] many analyses in non-exponential time seems preposterous, but that is the very idea that underlies the technique of *ambiguity packing*.

Oepen and Carroll (2000) show how to apply ambiguity packing to the problem of parsing with an HPSG, adopting and refining closely related techniques from context-free chart parsing. The resulting structure is called a *packed parse forest*, and can represent all of the exponentially many analyses of a sentence in comparatively little space. Carroll and Oepen (2005) show how to apply a similar algorithm for the generation direction, and also how to extract the N top-ranked trees (according to an appropriate probabilistic parse ranking model) from such a packed parse forest in time that is empirically negligible compared to the forest construction time, which in turn is frequently empirically negligible compared to the time required to exhaustively unpack *all* of the analyses.

Operations on packed parse forests often require nonobvious algorithms. In addition to the selective unpacking algorithms of Oepen and Carroll, Miyao and Tsujii (2002, 2008) show how to train such a maximum entropy parse ranking model from such a packed parse forest without exhaustive unpacking, using a derived structure called a *feature forest*. Clark and Curran (2004) deploy the feature forest mechanism to train parse selection models for Categorial Combinatory Grammar (Steedman, 1987; Steedman & Baldridge, 2011), giving a novel algorithm showing how to automatically annotate the nodes in a forest to indicate whether derivations using those nodes are consistent with a set of gold standard dependencies or not (they train a constituency parsing model using gold dependency data rather than gold constituency data). Geman and Johnson (2002) developed a method for training the parameters of stochastic unification grammars based on the packed representations of Maxwell and Kaplan (1991), which are distinct in form but similar in purpose to packed parse forests.

The literature is surprisingly silent on the use of packed representations for treebanking. In the Redwoods architecture, analyses must be unpacked before treebanking can be performed; to avoid handling exponentially many analyses for complex sentences, a parse

[2]The average number of analyses of sentences in open-domain text is typically an exponential function of sentence length.

selection model must be used to selectively unpack only the most promising N (typically 500) analyses. The LFG Parsebanker is capable of processing packed representations directly (Rosén, Meurer, et al., 2005 and personal communication with Paul Meurer in 2013), but the algorithms used to process the packed structures do not appear to be published. However, Baird and Walker (2010) report discarding all sentences that receive more than 200 analyses when using the LFG Parsebanker, suggesting that the tool may not be capable of efficiently processing exponentially many results.

2.4 Summary

Treebanking is an important task that requires nontrivial support software in order to be practical and efficient. Discriminant-based treebanking allows human annotators to navigate through a potentially large space of candidate analyses without having to inspect each tree individually. Ambiguity packing, on the other hand, allows *computers* to navigate a large space of candidate analyses without having to explicitly enumerate each tree, again saving time. Together, these two techniques make the large-scale annotation of linguistically complex corpora possible within practical resource limits.

Recall that the existing Redwoods system implements discriminant-based treebanking over an explicitly enumerated subset of the parse forest, typically the best 500 trees as ranked by a statistical parse selection component. In the next chapter, I will describe the algorithms required to perform discriminant-based treebanking over the *full forest*, without explicitly enumerating any trees.

Chapter 3

IMPLEMENTING FULL-FOREST TREEBANKING

This chapter will address the challenges involved in supporting a treebanking environment capable of operating over the complete forest of derivation trees without enumerating them. The challenges include designing data structures to represent the complete packed forest, algorithms to efficiently determine what discriminants remain to be resolved, and a user interface to facilitate navigation of the discriminants and resulting trees.

3.1 Representing a Forest

Before any algorithms can be described, it is necessary to describe the shape of the data to be manipulated. This section explains how the complete forest can be represented by a relatively compact *parse chart*.

A (non-packed) parse chart consists of a set of *edges*, each of which represents a derivation tree dominating some particular substring of the sentence to be parsed. An edge will either represent a single lexeme or the application of a grammar rule to a sequence of one or more adjacent daughter edges. Since no edge can ever be its own (grand-)daughter, the parse chart, when viewed as a graph,[1] is both directed and acyclic. When a grammar assigns N readings to a sentence, there are typically far more than N edges in the non-packed parse chart. For highly ambiguous sentences, time and space constraints on the parser therefore dictate that a *packed* parse chart be produced instead (Oepen & Carroll, 2000).

The difference between a non-packed parse chart and a packed parse chart is the addition of the *packing relation*, an equivalence relation that partitions the edges into equivalence classes. An edge representing the application of a rule to a daughter refers to an equivalence class $\hat{Y}$ of daughters, rather than to a specific daughter Y. Any edge X referencing an edge

[1] Note that the *edges* of the parse chart form the *nodes* of the graph, and the daughter links form the edges of the graph.

Y as the daughter of its rule is understood to also be able to be built from other edges $Y' \in \hat{Y}$.[2] Whereas an edge in an ordinary parse chart represents a single subtree, an edge X in a packed parse chart implicitly represents a *set* $\texttt{unpack}(X)$ of derivation trees dominating some particular substring of the sentence to be parsed. As in the non-packed parse chart, edges come in two flavors: lexemes and rules. If X represents a lexeme L, then $\texttt{unpack}(X) = \{L\}$. Otherwise, X represents a rule R and:

$$\texttt{unpack}(X) = \left\{ R(D) : D \in \bigotimes_{\hat{d} \in \text{daughters of } X} \bigcup_{d \in \hat{d}} \texttt{unpack}(d) \right\} \quad (3.1)$$

For a sentence with a very large number N of readings, the number of edges explicitly represented in the packed parse chart is typically dramatically smaller than N.

In order to represent the complete set of derivation trees licensed by the grammar, it suffices to have access to the list of edges in the parse chart (together with data about which edges are daughters of other edges, and the packing relation), and one additional piece of information: the set R of edges which satisfy the grammar's root condition.[3] Note that while a parser may produce edges which are not reachable from root edges (by daughter or packing equivalence links), these are superfluous from the point of view of processing the set of grammatically licensed analyses, and need not be stored for further processing.[4]

Figures 3.1 and 3.2 show examples of a non-packed parse chart and a packed parse chart for the sentence *I think three zebras flew zeppelins over Zimbabwe*. While the reduction in chart size from packing is not particularly impressive in this example, the effect becomes dramatic for longer, more ambiguous sentences. Unfortunately, available space does not permit the exhibition of larger parse charts.

---

[2]In actual parsers, the packing relation is implemented by designating a distinguished representative Y of each equivalence class $\hat{Y}$, and enumerating the other members of the equivalence class on a so-called packing list linked from Y. Other edges built from members of $\hat{Y}$ always refer to the distinguished representative Y.

[3]The root condition corresponds to what is known as a start symbol in context free grammars. The root condition is a feature structure F which determines which derivation trees are considered to describe complete utterances and which are not. Specifically, for a derivation tree T to describe a complete utterance, F must be unifiable with the feature structure found at the top level of T. Root edges are those which span the entire sentence and are unifiable with F (Pollard & Sag, 1994).

[4]Indeed, such unreachable edges are not stored by default in the implemented system.

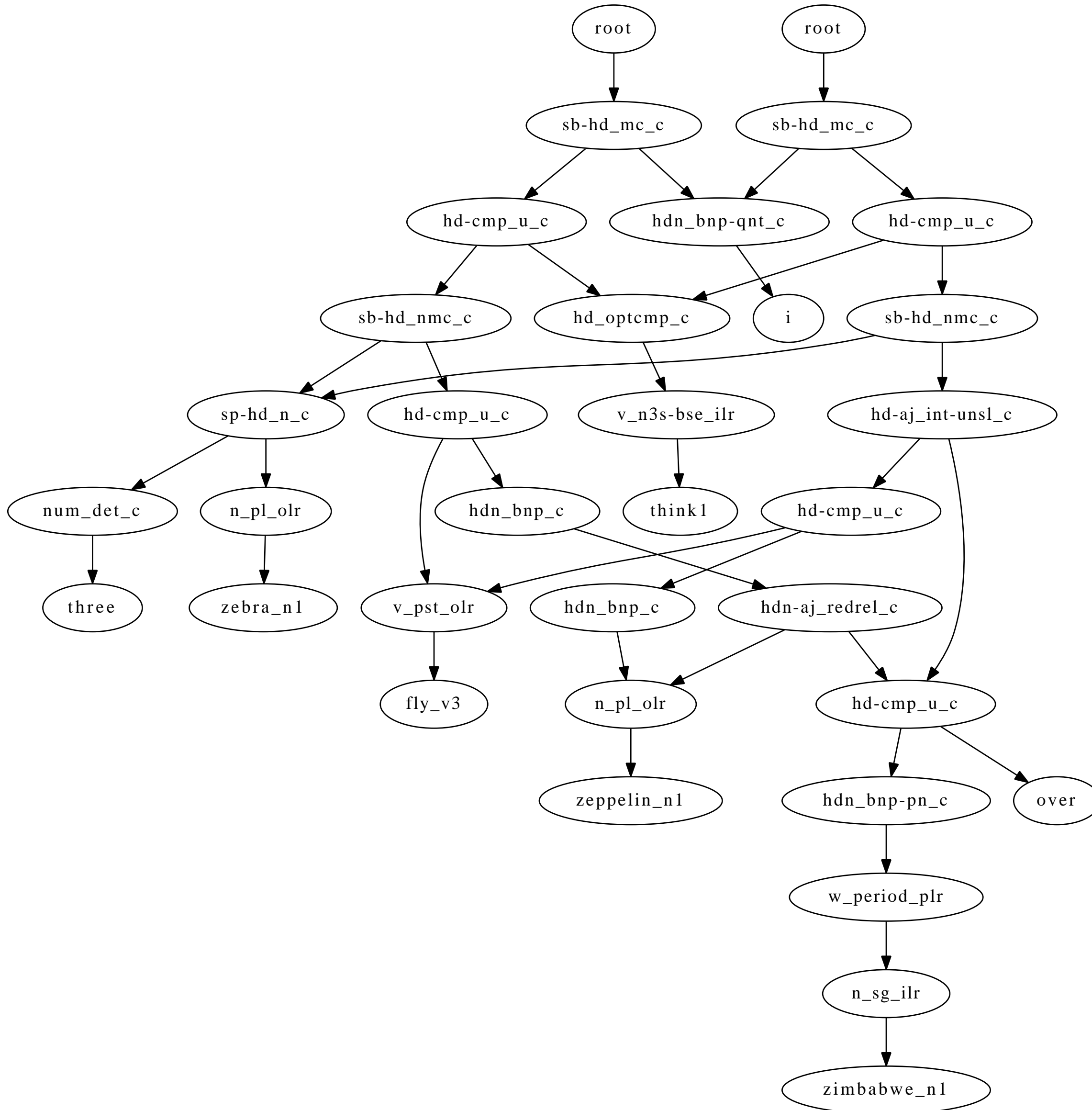

Figure 3.1: Non-packed parse chart for *I think three zebras flew zeppelins over Zimbabwe.* The attachment ambiguity from the embedded clause percolates up, resulting in multiple "root" edges. For brevity, edges corresponding to the highest attachment of the PP are omitted. Note that in this figure, chart *edges* are rendered as graph *nodes*, with arcs corresponding to daughters in rules.

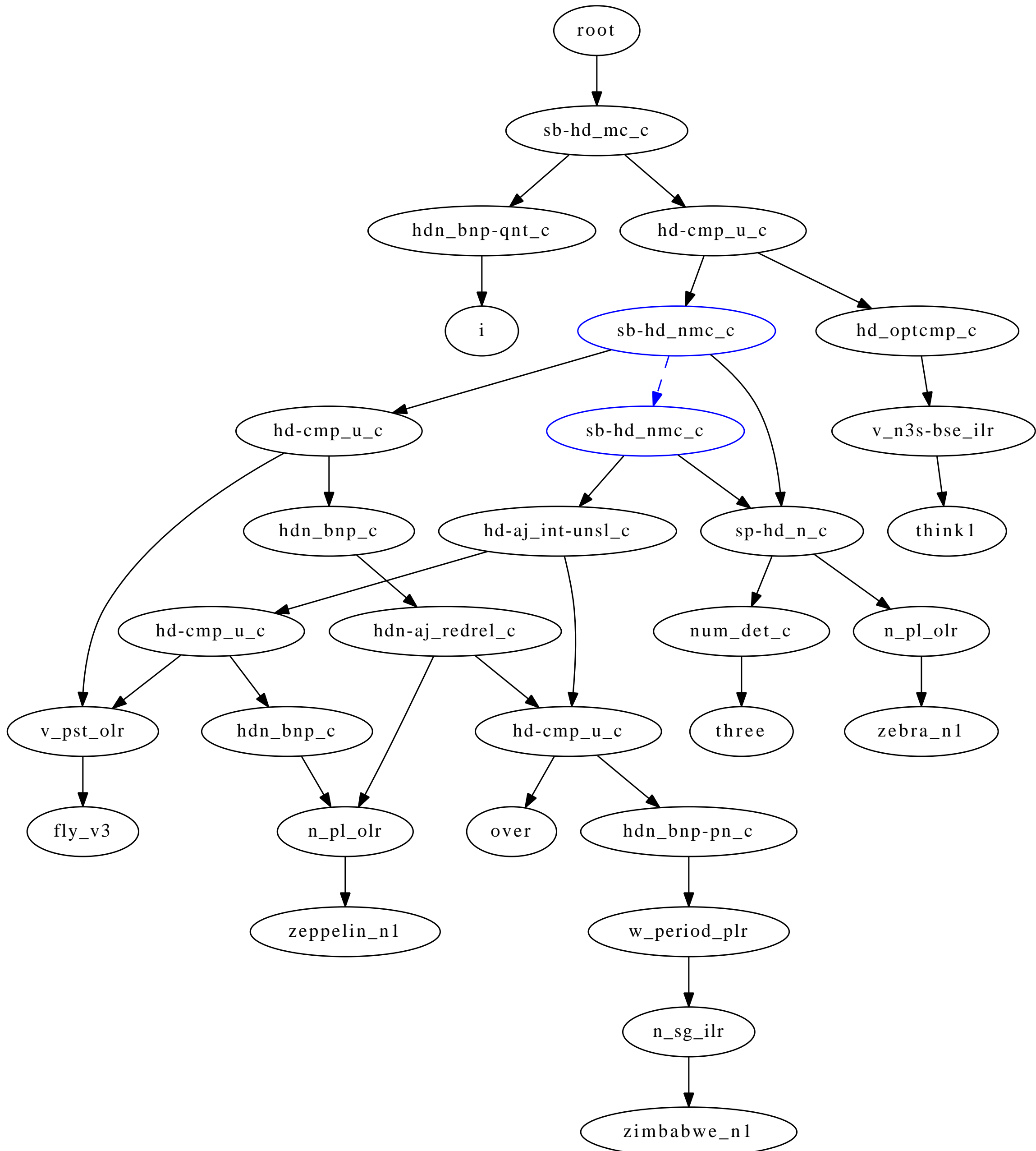

Figure 3.2: Packed parse chart for *I think three zebras flew zeppelins over Zimbabwe.* Attachment ambiguity from the embedded clause is effectively contained. The two edges shown in blue are in the same equivalence class. For brevity, edges corresponding to the highest attachment of the PP are omitted.

3.2 Counting Readings

One of the most basic tasks involved in implementing a full-forest treebanker is the computation of how many trees (complete analyses) are represented by a packed parse forest. This can be employed to give the user an estimate of the difficulty of treebanking an item before starting, or to provide feedback to the grammarian on the level of ambiguity assigned by his or her grammar, for example. As we shall see, the ability to produce such a count is important to other system-internal processes as well.

In certain modes of operation, a packed parse forest may represent derivation trees that are not globally consistent with respect to the unification constraints in the grammar (Oepen & Carroll, 2000).[5] Furthermore, some grammars stipulate additional non-unification-based requirements on wellformedness that cannot be straightforwardly represented as constraints on the packed forest.[6] To count readings with the granularity imposed by these considerations, it appears that the only possible solution is to unpack every analysis from the forest, replay all of the unifications, and check any additional constraints. This is referred to as exhaustive unpacking. Unfortunately, the running time of such a solution is necessarily at least linear in the number of readings represented by the forest, making it prohibitive for highly ambiguous sentences. Although it does not appear to be possible to efficiently compute an *exact* count of the number of readings represented by a parse chart, it is possible to quickly compute estimates. Below, I present an efficient technique for approximating the number of readings.

3.2.1 Computing an Upper Bound Using Dynamic Programming

Let T be the number of "abstract" derivation trees represented by a packed parse forest (i.e. without consideration to latent unification failures or additional non-unification–based

---

[5] Specifically, when packing under subsumption or when a packing restrictor is employed to discard certain feature paths (in an effort to cause more edges to become equivalent to each other), an edge's feature structure may not represent all of the relevant constraints of all of the unpacked subtrees it represents. When the full set of constraints are evaluated on an unpacked tree, failures may occur.

[6] For example, the English Resource Grammar incorporates an *idiom filter* (Copestake, 1994), which is a wellformedness constraint stated over the complete MRS structure extracted from an analysis in a way that does not easily factor over the parts of the derivation.

filters). T is an *upper bound* on the number N of fully consistent analyses licensed by the grammar. As it turns out, latent unification failures are relatively rare (depending on the aggressiveness of the parser's packing settings), and additional non-unification–based filters are both rarely deployed and generally very permissive. As a result, T is typically quite close to N, and can be used as a proxy.

This is good news, because dynamic programming can be used to compute T from a packed parse forest in time proportional to the number of edges in it, a tractable computation. The number of trees represented by a packed forest is trivially equal to the sum of the number of trees represented by each of its root edges $r \in R$:[7]

$$T = \sum_{r \in R} T(r) \tag{3.2}$$

The computation proceeds recursively, with memoization. If an edge e represents a single lexeme, then:

$$T(e) = 1 \tag{3.3}$$

Otherwise, e represents the application of a rule, with k daughters $\{\hat{d}_1, \ldots, \hat{d}_k\}$ which are equivalence classes (under the packing relation) of other edges. We then have:

$$T(e) = \prod_{\hat{d} \text{ is a daughter of } e} \sum_{d \in \hat{d}} T(d) \tag{3.4}$$

To compute T, it suffices to compute $T(e)$ for each edge in the packed forest. The acyclicity of the daughter links between edges ensures that recursion terminates.[8]

3.2.2 Counting a Subforest

We have seen how to efficiently compute the number T of trees represented by a parse forest. A related problem is counting the number T_C of trees that are consistent with some particular set of constraints C. This problem arises during discriminant-based treebanking, when a set of decisions has been made, and the tool needs to present the user with status information about the remaining ambiguity. It also is relevant when performing an automated

[7] I assume R is a full listing of individual root edges, rather than a listing of equivalence classes.

[8] In fact, my implementation first performs a topological sort on the edges and does not use recursion at all.

update of a treebank (i.e. applying stored decisions to a new parse forest, perhaps derived from an updated grammar with additional or different ambiguities). In such a setting, it is useful to know whether the stored decisions select a unique tree ($T_C = 1$), select a set of trees ($T_C > 1$), or rule out all trees ($T_C = 0$).

The exhaustive unpacking approach to counting referred to above can be trivially modified to count how many trees are consistent with arbitrary constraints C, making it again the most flexible and precise algorithm. However, even when the constraint set C eliminates all or nearly all of the trees in the forest, this approach still requires considering each and every tree, which as before can frequently be prohibitively expensive.

The dynamic programming approach is tempting, but the interaction between the type of useable constraints and the locality of the available information needs careful consideration. $T_C(e)$ should be the number of analyses represented by e that are consistent with the constraints C. For this to be well-defined, it must be possible to positively determine whether analyses extended to include e conflict with C or not purely on the basis of the information local to the dynamic programming states (i.e. edges). The constraints used in Redwoods can be broken down into two forms:

1. For *some* edge with span S, local property X holds.

2. For *every* edge with span S, local property X holds.

Redwoods constraints are not usually written in a way that makes the correspondence to these types clear, so a pair of examples will be useful. Two typical constraints that might be employed when analyzing the sentence *Inspect bags at the airport!* would be:

1. The rule `hd_imp_c` dominates span 0-5.

2. The rule `hdn_bnp_c` does not dominate span 1-5.

These can be reformulated as the following equivalent statements:

1. *Some* edge with span 0-5 is an application of the rule `hd_imp_c`.

2. *Every* edge with the span 1-5 is not an application of the rule `hdn_bnp_c`.

Constraints of type 2 can be evaluated with information that is local to an edge, since if edge e with span S is part of a complete derivation, it must comply with property X. However, constraints of type 1 cannot be evaluated on an edge-by-edge basis, since even if an edge e with span S fails to comply with X, it could still be part of a complete derivation if X is satisfied by some other edge with the same span, e.g. a unary rule taking e as its daughter. Since constraints of type 1 are common (much more common, in fact, than constraints of type 2), a revised approach is needed.

3.2.3 Collapsing Unary Chains

The basic problem with evaluating constraints of type 1 is that a unary rule may satisfy them somewhere outside of the local context represented in an edge. This difficulty can be overcome by rewriting the packed parse chart into a new, equivalent, but restructured parse chart that contains no unary edges. To make this possible, instead of using single rule and lexeme names as edge labels, *chains* of names are used. A chain always ends in either a non-unary rule name or a lexeme name, but contains a prefix of any number of unary rule names.

In the example from the preceding section, `hd_imp_c` is a unary rule dominating the entire sentence, taking as its input the result of the binary rule `hd-aj_int-unsl_c`. Together these form a chain of length 2. Algorithm 3.1 shows how to compute the list of such *unary chains* that can be unpacked starting from a given edge, and Algorithm 3.2 shows how to convert an ordinary packed parse chart into one where all unary rules have been compacted into unary chains.

The result of this operation is that the derivation trees represented by the transformed chart never contain a configuration where, within one tree, there are multiple nodes with the same span. This makes it possible to evaluate constraints of both type 1 and type 2 using edge-local information. A side-effect is that the number of edges in the transformed chart may be larger than the number of edges in the original chart, since the local ambiguity of the unary rules is unpacked. In practice, this has not proven to be a significant problem; in fact, some charts even shrink slightly (e.g. when several unambiguous unary edges are

```
1:  function UnaryChains(ê)
2:      chainset = ∅
3:      for all e ∈ ê do
4:          if e is unary then
5:              for all chain ∈ UnaryChains(daughter(e)) do
6:                  chain = [e] + chain
7:                  chainset = chainset ∪{chain}
8:              end for
9:          else
10:             chainset = chainset ∪{[e]}
11:         end if
12:     end for
13:     return chainset
14: end function
```

Algorithm 3.1: Computing unary chains

eliminated, as in the example below).[9]

To see how the transformation helps, consider the packed parse chart for *I think three zebras flew zeppelins over Zimbabwe.*, shown in Figure 3.2. Suppose one wishes to use the following constraint[10] to resolve the ambiguity:

`hdn_bnp_c` dominates the span *zeppelins.*

In order to give the user feedback on the effect of the constraint, a count is needed of how many analyses remain, i.e. are consistent with the constraint. The first step is to inspect the edge `n_pl_olr(zeppelin_n1)`, and determine that there is a set S consisting of exactly one subtree that can be unpacked from it. That subtree so far has not satisfied the constraint. For a larger edge, however, there could be a very large number of such

[9]Charts for short sentences (10 or less words) tend to shrink on the order of 10%, while charts for long sentences (30+ words) tend to expand on the order of 10%.

[10]This constraint amounts to saying *zeppelins* is a complete NP, so that the PP must attach elsewhere.

```
1: function CollapseUnaries(old_chart)
2:     new_chart = {roots=∅, edges=∅, eqclasses=∅}
3:     for all r ∈ roots(old_chart) do
4:         roots(new_chart) = roots(new_chart) ∪ UnaryCollapsedEdge(new_chart, r̂)
5:     end for
6:     return new_chart
7: end function
8: function memoized UnaryCollapsedEdge(new_chart, old_edge)
9:     chainset = UnaryChains(old_edge)
10:    eqclass = ∅
11:    for all chain ∈ chainset do
12:        edge = UnaryCollapsedEdgeFromChain(new_chart, chain)
13:        eqclass = eqclass ∪{edge}
14:    end foreqclasses(new_chart) = eqclasses(new_chart) ∪{eqclass}
15:    return eqclass
16: end function
17: function memoized UnaryCollapsedEdgeFromChain(new_chart, chain)
18:    e = {label: chain, daughters: []}
19:    for all d̂ ∈ daughters(last(chain)) do
20:        new_daughter = UnaryCollapsedEdge(new_chart, d̂)
21:        daughters(e).append({new_daughter})
22:    end for
23:    return e
24: end function
```

Algorithm 3.2: Eliminating unary edges from the chart

subtrees, some of which have satisfied the constraint and some of which have not, making it impractical in general to carry forward detailed information about which elements of S have satisfied the constraint and which have not. In this simple example it might be possible to keep track of a simple *count* of how many subtrees have satisfied the constraint and how many still need to, but it appears to be nontrivial to extend that idea to the case of multiple interacting constraints. When it is time to count how many subtrees can be built with `hdn-aj_redrel_c(zeppelins over Zimbabwe)`, the window of opportunity for the constraint to be satisfied has passed, but it is unknown how many of the subtrees have satisfied it.

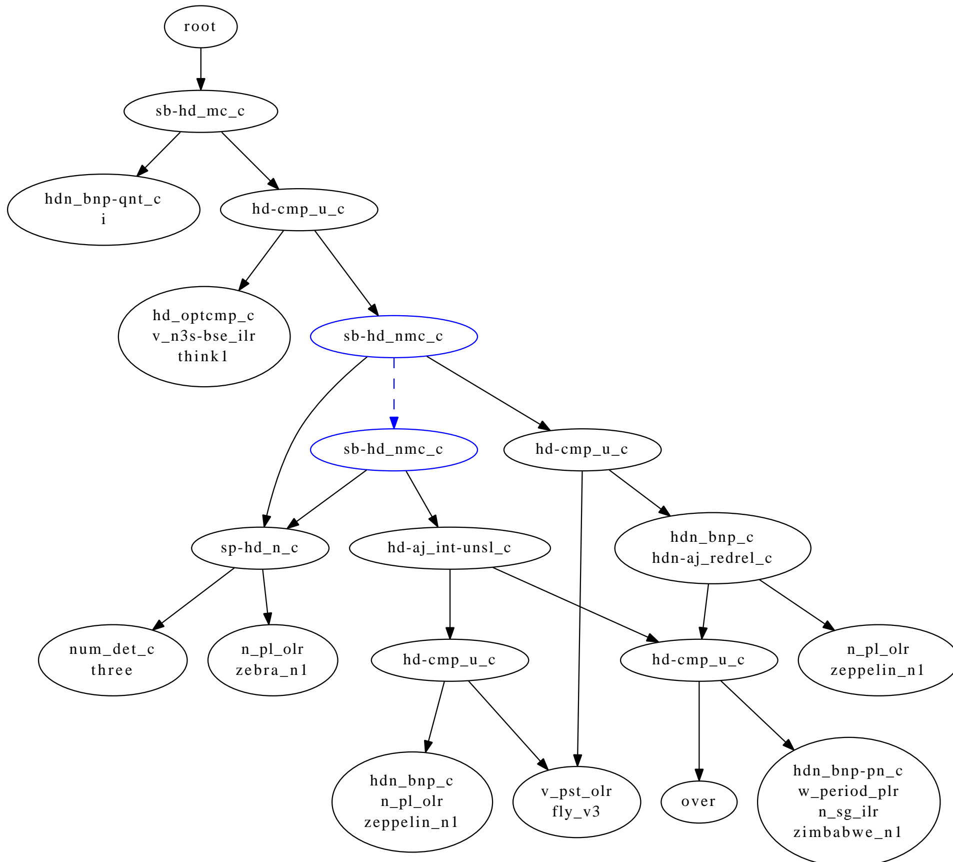

Figure 3.3: Unary-chain packed parse chart for *I think three zebras flew zeppelins over Zimbabwe.*

The story is different with the unary chain chart. After the unary chain transformation, the chart is shown in Figure 3.3. Every edge falls into one of two classes:

1. This edge's local information overtly contradicts the constraint.

2. This edge's local information does not overtly contradict the constraint.

I claim that a derivation tree consisting solely of edges of the latter type is globally consistent with the constraint. The proof proceeds by contradiction. Suppose a derivation tree is not consistent with the constraint. I will demonstrate that some edge in the tree must overtly contradict the constraint. Recall that the general case of the constraints in question is that *some* edge in the tree with span S satisfies a given property P. A tree can be inconsistent with this constraint in two ways: there can be no edge with span S, or all the edges with span S can fail to satisfy property P. Since unary edges have been eliminated, there will always be either zero or one edges with span S, never more than one. If there is one edge with span S, and that edge fails to satisfy property P, then it overtly contradicts the constraint. The other case is that no edge with span S exists. Let w be the first word contained inside of span S.[11] Let X be the largest constituent containing w which is completely contained inside S, and let Y be X's mother. Then Y is not contained inside of S but X is. Then Y overtly contradicts the constraint, since Y-local information forces there to be no constituent with span S (such a constituent would have to be a descendent of Y and an ancestor of X, but Y is the mother of X). Thus in both cases, our derivation contains an edge that overtly contradicts the constraint.

Notice that this implies that the consistent subforest can be counted by simply applying the existing dynamic programming forest counting algorithm to the *subchart* formed by discarding all edges that overtly contradict the constraint. Edges built on emptied equivalence classes must also be discarded for the chart to remain well-formed. In practice, the algorithm is slightly modified to simply consider the subtree count of any overtly contradicting

[11] I assume words are constituents. Multiword lexemes require slightly special treatment, but it is straightforward to consider an edge containing a multiword lexeme as overtly contradicting a constraint that starts or ends inside of it.

edge to be zero. The zeros filter up through the algorithm giving the same effect as actually removing those edges from the chart. The formulas used are given in the next section.

3.2.4 Counting a Subforest Revisited

Once unary chains have been collapsed, it is possible to give the algorithm for counting the number T_C of readings consistent with constraints C. The equations used are similar to those given above, which represent the case when $C = \emptyset$. As before, it is necessary to sum over root edges:

$$T_C = \sum_{r \in R} T_C(r) \tag{3.5}$$

If an edge e overtly contradicts C, then $T_C(e) = 0$. This has the effect that any derivations that contain e are not counted by T_C. If, on the other hand, e is consistent with C, then there are the usual two cases. When e represents a single lexeme, $T_C(e) = 1$. Otherwise, e represents the application of a rule R, and $T_C(e)$ is computed inductively:

$$T_C(e) = \prod_{\hat{d} \text{ is a daughter of } e} \; \sum_{d \in \hat{d}} T_C(d) \tag{3.6}$$

As before, T_C can be evaluated recursively with memoization or by performing a topological sort in advance and simply iterating through the edges. The remaining point to be clarified is the evaluation of whether C is consistent with an edge or not. An edge in a (unary-chain transformed) packed chart has a span[12] S together with a label of the form $u_1, \ldots, u_n$ where u_n is either an instance of a non-unary rule or a lexeme. Assume C is a set $\{(X_1, S_1), \ldots, (X_m, S_m)\} \cup \{(Y_1, V_1), \ldots, (Y_d, V_d)\}$ of constraints, where (X_i, S_i) is a constraint of type 1 with property X_i at span S_i, and (Y_i, V_i) is a constraint of type 2, with property Y_i and span V_i. C is consistent with a unary chain edge $u_1, \ldots, u_n$ if and only if every element of C is consistent with the chain. Assuming the edge has span S and a constraint has span S_i, the following rules apply:

[12] By "span" I mean a contiguous subset of the input tokens.

1. If $S_i \subsetneq S$, and one of the daughters of the edge has a span $S' \subsetneq S_i$, then a type 1 constraint is inconsistent with the edge.[13] Otherwise,

2. If S and S_i are disjoint or nested, i.e. $(S \cap S_i = \emptyset) \vee (S \subsetneq S_i) \vee (S_i \subsetneq S)$, then the constraint is consistent with the edge, regardless of which type of constraint it is. Otherwise,

3. A constraint crossing spans with an edge (i.e. $S_i \neq S$, but not disjoint and not nested) is inconsistent for type 1 but consistent for type 2.

4. Otherwise $S = S_i$. Type 1 constraints are consistent with the edge if and only if $X_i(u_j)$ for some j. Type 2 constraints are consistent with the edge if and only if $Y_i(u_j)$ for all j.

The evaluation of a constraint against an edge is slightly complicated, but not impenetrably so, and crucially is computable in time proportional to the product of the chain length with the number of constraints, both of which are typically small.

Note that some other constraint types can be evaluated as well, including constraints that consider the entirety of a unary chain rather than just individual pieces of it. The ability to stipulate a complete unary chain rather than just the membership of it allows this treebanking regime to clear up ambiguities that the previous-generation Redwoods tools could not resolve, in particular when the grammar licenses multiple unary rules over the same span with more than one ordering. An example where such flexibility affects meaning is the relative order of the extracted complement rule and the optional complement rule for ditransitive verbs: the question *What did you pay?* could be inquiring about the price or about the vendor.[14]

---

[13] This can only occur if u_n is a rule of arity 3 or more.

[14] Another possible case is the relative ordering of prefix and suffix in derivational morphology.

3.3 Enumerating Discriminants

During an automatic treebank update operation, the set of constraints C considered in the last section is provided by the pre-existing gold treebank. However, during manual annotation, the constraints are selected one at a time by the user. It would be rather cumbersome for the user to have to invent and write out each such constraint by hand, with no clues but the edge list. A much more convenient interface (and the one that is always used in discriminant-based treebanking) is to present the user with a *list* of candidate constraints which are guaranteed to reduce the size of the forest without ruling out all trees. These constraints are called *discriminants* (Carter, 1997; Oepen, Flickinger, et al., 2004). An overview of this method of annotation was given in Section 2.2.

This of course presents a new engineering problem: how can the list of discriminants be efficiently computed? I have gone to considerable lengths to ensure that constraints only describe properties that are local to an edge in the parse forest. One of the benefits of that effort is that a list of candidate constraints can be read off directly from the individual edges. The constraints that are of interest, i.e. the discriminants, are those which hold of some but not all of the trees represented by the packed forest. Algorithm 3.3 shows how to compute a set of discriminants, given a count $S_C(e)$ of how many complete (i.e. rooted in a member of R) trees each edge takes part in. Essentially, properties are read off of each edge, and identical properties from non-identical edges are aggregated, keeping track of the total number of trees to which each property applies. Properties that apply to all trees are discarded; those that remain are the discriminants.

The quantity $S_C(e)$ can be computed using dynamic programming from the $T_C(e)$ figure described above. $T_C(e)$ describes how many unique subtrees are described by the edge e (i.e. trees rooted at that edge), given a constraint C. By contrast, $S_C(e)$ describes how many fully spanning trees (rooted at an edge that meets the root condition) consistent with C and containing e somewhere within them are described by the forest. Equation 3.7 computes $S_C(e)$ for an edge e belonging to a packing equivalence class $\hat{e}$, assuming e does not itself

meet the root condition.

$$S_C(e) = \frac{T_C(e)}{\sum_{e' \in \hat{e}} T_C(e')} \sum_{p \in \text{ edges of which } \hat{e} \text{ is a daughter}} S_C(p) \tag{3.7}$$

Intuitively, e appears in as many trees as its parents p do, discounted for the fact that some of the trees that a parent p appears in actually contain a different member of $\hat{e}$ in the position where e would go. In the special case where e meets the root condition, trees where e has no parent at all must also be counted, as in Equation 3.8:

$$S_C(r) = T_C(r) + \frac{T_C(r)}{\sum_{e' \in \hat{r}} T_C(r')} \sum_{p \in \text{ edges of which } \hat{r} \text{ is a daughter}} S_C(p) \tag{3.8}$$

Note that the direction of dependency is reversed relative to the computation of T_C, i.e. if recursion is not used, the edges must be visited in the opposite order to ensure that computation results are ready when they are needed.

To see this computation in action, suppose one wants to compute $S_C(e)$ for the edge `sp-hd_n_c(three zebras)` in Figure 3.3. The edge does not meet the root condition (it doesn't span the whole sentence), so we use Equation 3.7. I will assume no decisions have been made yet by the user, i.e. $C = \emptyset$. By inspection we see $T_C(e) = 1$, i.e. there is only one way to unpack a subtree rooted at e. No other edges are equivalent to e, so $\hat{e} = \{e\}$. There are two parent edges p_1, p_2 to sum over, both using the rule `sb-hd_nmc_c` (these are the two blue edges in Figure 3.3). Since this is the only instance of packing in the chart and there is only one root, $S_C(p_1) = S_C(p_2) = 1$, i.e. each parent appears in exactly one solution tree. Equation 3.7 yields:

$$S_C(e) \quad = \quad \frac{T_C(e)}{\sum_{e' \in \{e\}} T_C(e')} \sum_{p \in \{p_1, p_2\}} S_C(p) \quad = \quad \frac{1}{1}(1+1) \quad = \quad 2$$

That is, there are two complete trees that e appears in. If one were instead computing $S_C(p_1)$,[15] there would be only one (grand)parent g, with $S_C(g) = 2$, and the equivalence class would be $\hat{p_1} = \{p_1, p_2\}$, yielding:

$$S_C(p_1) \quad = \quad \frac{T_C(p_1)}{\sum_{p' \in \{p_1, p_2\}} T_C(p')} \sum_{g' \in \{g\}} S_C(g) \quad = \quad \frac{1}{(1+1)}(2) \quad = \quad 1$$

[15]Indeed, although I claimed $S_C(p_1) = 1$ by inspection above, the system must actually compute $S_C(p_1)$ and $S_C(p_2)$ before computing $S_C(e)$.

So, the equations correctly compute that there is only one solution tree containing p_1.

Algorithm 3.3 computes constraints in terms of the full unary chain present at a particular span. These are more informative (in a technical sense) than single-rule or single-lexeme constraints,[16] although they may in some circumstances be less convenient. Computation of other types of local discriminants (including the traditional Redwoods-style ones) would follow the same basic procedure.

1: T_C = total number of trees
2: constraints= $\emptyset$
3: treecounts=[]
4: discriminants= $\emptyset$
5: **for** each edge e with unary chain $u_1, \ldots, u_k$ and span S **do**
6:     c =constraint that span S has unary chain $u_1, \ldots, u_k$
7:     **if** $c \notin$ treecounts **then**
8:         constraints = constraints $\cup\{c\}$
9:         treecounts[c] = 0
10:     **end if**
11:     treecounts[c] = treecounts[c] + $S_C(e)$
12: **end for**
13: **for** each constraint c such that treecounts[c] $< T_C$ **do**
14:     discriminants = discriminants $\cup\{c\}$
15: **end for**

Algorithm 3.3: Enumerating discriminants

Thus far in this chapter I have described the technical mechanisms and algorithms making full-forest treebanking possible. In the next section, I will examine ways to enable a human annotator to interact with the algorithms.

---

[16] See the end of section 3.2.4 for an example of how these constraints are more informative than single-rule constraints.

3.4 Human-Computer Interaction

The mechanisms introduced above allow the efficient enumeration of a set of discriminants which a human annotator can use to pare down the parse forest until only one tree remains. Unfortunately, a list that is efficiently enumerable for a computer is by no means guaranteed to be efficiently perusable by a human! Especially for long sentences, there can be tens of thousands of discriminants, or even more. While very small compared to the number of readings (frequently in the trillions, for long sentences), this number is still too large to be exhaustively inspected. Some mechanism is needed to enable the annotator to quickly find easy-to-decide discriminants without having to consider obscure ones.[17]

3.4.1 Navigating the Discriminants

One simple approach is to allow the user to specify which span of the sentence he or she wishes to concentrate on. By using the mouse to select part of the sentence, the user can instantly dismiss the vast majority of the discriminants, and quickly see the relevant choices. For instance, to disambiguate the forest in Figure 3.3, the user might hilight the span *flew zeppelins.* The only discriminant applicable to that span is whether or not the rule `hd-cmp_u_c` applies or not, so the user need waste no further time hunting. In practice this has proven quite effective, although it is an open question whether there are other more effective methods of quickly finding easily decidable discriminants (such as the ranking approach of Zhang and Kordoni (2010), discussed in Chapter 2).

3.4.2 Viewing Partial Results

One drawback to this interface is that a user's intuition about which part of the sentence is causing ambiguity may not always be correct, leading to hunting and frustration. It is frequently the case that the analysis of some substring is unambiguous even when the decisions made thus far are insufficient to identify a unique analysis of the complete sentence.

---

[17]A decision about one discriminant will typically imply decisions about many others, so it is often possible to completely disambiguate a sentence without explicitly considering the more difficult or confusing discriminants.

In such cases, the system can automatically identify such substrings[18] and show the user that no ambiguity resides in those portions of the sentence (in the case of my implementation, by underlining them and disallowing subportions of those unambiguous substrings to be highlighted). An additional benefit of showing these disambiguated substrings is that the system can, at the user's request, display the unambiguous analysis of the substring.

3.4.3 Latent Unification Failures

As discussed in Section 3.2, it is possible for some of the trees that are represented by a packed parse chart to contain hidden errors, which were not detectable at parse time. These errors, known as *latent unification failures*, are generally not detectable by simple static analysis of the parse forest, and can only be detected by examining a specific tree (or subtree). This represents what is perhaps the most significant disadvantage of (packed) full forest treebanking relative to fully enumerated top-N treebanking: it is possible to use system-provided discriminants to select a tree that is not fully consistent with the grammar. While the error can be automatically detected as soon as the tree is produced, a significant amount of the human annotator's time has been wasted by the time a full tree has been selected, and it is difficult at this point to know what incorrect choices were made or how to fix them.

The problem can be alleviated to a large extent by automatically verifying the consistency of the analysis of each unambiguous substring. As the human annotator makes decisions, the number of these unambiguous substrings generally grows gradually, and experience has shown that latent unification failures are usually manifested in unambiguous substrings fairly quickly after a spurious decision is made. The system allows each individual decision to be undone, so usually the amount of wasted effort is minimal. However, avoiding these inefficiencies altogether would be a profitable area for future research.

---

[18] A substring is unambiguous if it is spanned by an edge e where $T_C(e) = 1$ and $S_C(e) = T_C$.

3.5 Summary

I have explained the technical challenges faced by a treebanking system that can efficiently support the disambiguation of sentences with more analyses than can practically be enumerated. The forest must be stored in a *packed* format, using an equivalence relation to show choice points. Mechanisms were introduced to count the number of analyses and to enumerate the discriminants, and both operations needed to be capable of functioning on the subset of the forest delineated by a set of constraints. The lack of locality of information created by unary rules prompted the introduction of the *unary chain* forest, where constraints can always be evaluated locally to a single edge. Finally, I discussed some of the user interface challenges such a system meets, and proposed partial solutions to them.

In the next chapter, I will present some numerical simulations aimed at investigating the utility and practicality of full forest treebanking.

Chapter 4

PRELIMINARY EXPERIMENTS AND RESULTS

Consideration of the difference between utilizing a top-500 list on the one hand and the full forest on the other leads to a number of interesting questions. In this chapter, I launch two such inquiries: First, when treebanking, how often is the desired tree present in the forest but not ranked among the top 500? Second, how much additional effort is required to select a unique tree from the full forest compared to from the top 500?

4.1 500 Chances

Although the number 500 is quite arbitrary, it has been the de-facto standard for how many trees to consider in Redwoods-style treebanking for at least the past five or so years (Flickinger, Kordoni, et al., 2012; Marimon et al., 2014).[1] This number is an attempt to balance two competing desiderata: it needs to be small enough that unpacking and storing the trees from the forest does not take an especially burdensome amount of time or space, and large enough that the desired tree will virtually never be discarded accidentally. This equilibrium does not appear to have been subjected to much scrutiny in the literature. Empirically, full-forest treebanking using the algorithms described in the preceding chapter both is somewhat less expensive computationally and requires dramatically less disk storage[2] than top-500 treebanking, and of course the desired tree (if present) will never be discarded accidentally with full-forest treebanking, making full-forest treebanking superior to top-N treebanking in both of these regards. The reason for the reductions in resource consumption is that when top-500 treebanking is used, subtrees that are identical between many (or most) readings are stored (and sometimes reprocessed) redundantly, whereas such a subtree may

[1] Some work with the Japanese grammar has used 5000 trees (Bond et al., 2004).

[2] The DeepBank Wall Street Journal treebank, for instance, requires about 200GB of disk space for top-500 treebanking, but only 15GB for full-forest treebanking. The benefit in computational time is much less extreme.

not be explicitly represented even once (in full form) in the packed forest. To consolidate this apparent technical victory, it remains to quantify the risk involved in discarding all but the top 500.

A good numerical measurement of the risk is what I will call the top-500 recall, i.e. the proportion of instances in which the desired tree is within the top-500 window. It is self-evident that this figure will vary considerably with the quality of the probabilistic model used to rank the forest. A model trained on a large in-domain corpus may have a relatively high top-500 recall, while a model trained on a small out-of-domain corpus will likely have a low top-500 recall.

The obvious way to measure top-500 recall is to parse some text for which gold annotated trees are available, storing the top-500 analyses according to some particular model, and measure the proportion of items for which the gold trees are among the top 500. Abstractly this is a sound idea. Unfortunately, all of the gold-annotated resources that are large enough for accurate measurements of this figure were built under a top-500 treebanking regime. This means that items for which the desired tree was not among the top 500 at treebanking time simply do not receive an annotation in the treebank. If one were to reparse the annotated text with the same parse selection model that was used to originally annotate it, the top-500 recall would come out to be 100%.[3] When the model used to reparse the annotated text is different (even very subtly), the recall usually comes out at less than 100%, allowing some information to be recovered, but the measurement is still fundamentally biased. The only real solution to this problem is to produce new gold annotations using the full forest — a labor-intensive process. For lack of a better tool, however, I proceed to use the biased estimate (i.e. evaluating against treebanks built from top-500 lists) of the top-500 recall, but it must be understood as an *over*-estimate rather than an accurate figure.

Table 4.1 shows this over-estimate of the top-500 recall rate for three text samples from (somewhat) differing domains, evaluated using three different parse ranking models. The WSJ domain consists of newspaper text annotated by the DeepBank project (Flickinger, Zhang, & Kordoni, 2012), the WeScience domain comprises Wikipedia articles (Ytrestøl et

[3]This is an instance of a self-selecting sample, a problem that plagues statisticians.

		Test Domain		
		WSJ	WeScience	LOGON
	`wsj.mem`	**97.71%**	88.02%	93.27%
Model	`wescience.mem`	73.71%	**97.04%**	90.36%
	`redwoods.mem`	78.65%	**92.84%**	**98.04%**
Geometric Mean Ambiguity		16603	2415	743
Random Choice Baseline		3.01%	20.70%	67.29%
Average Sentence Length		20.4	17.6	14.8

Table 4.1: Top-500 recall overestimate. In-domain results are in bold. The `redwoods.mem` model was trained on both WeScience and LOGON data, as well as other material.

al., 2009), and the LOGON domain is a collection of text from Norwegian tourism brochures (Oepen, Dyvik, et al., 2004). The `wsj.mem` and `wescience.mem` parse ranking models are trained exclusively on annotated data in their respective domains. The `redwoods.mem` model is trained on data from a variety of domains, of which the annotated WeScience and LOGON treebanks form a part. The annotations used for evaluating the top-500 recall were not part of the training data for any of the models. Figure 4.1 shows results from the same experiment aggregated differently: top-500 recall as a function of sentence length. The red line shows results for the case where the test text is in the same domain as the training material (boldface in Table 4.1), while the green line shows the case where the test text is out of domain with respect to the training material (non-boldface in Table 4.1).

These figures suggest a number of observations. First, when parsing using a parse ranking model trained exclusively on in-domain data, the top-500 recall rate can be relatively high, somewhere between 97% and 98%. Even so, 2% or more of inputs with good analyses somewhere in the forest will appear to have no good analysis when annotated in a top-500 treebanking system. Furthermore, the proportion of inputs whose correct analysis survives drops considerably with sentence length, sinking below 90% for inputs longer than 30 words. Keeping in mind that this is an *over*-estimate of the top-500 recall, it is clear that even under the best circumstances an appreciable number of sentences that could have been annotated

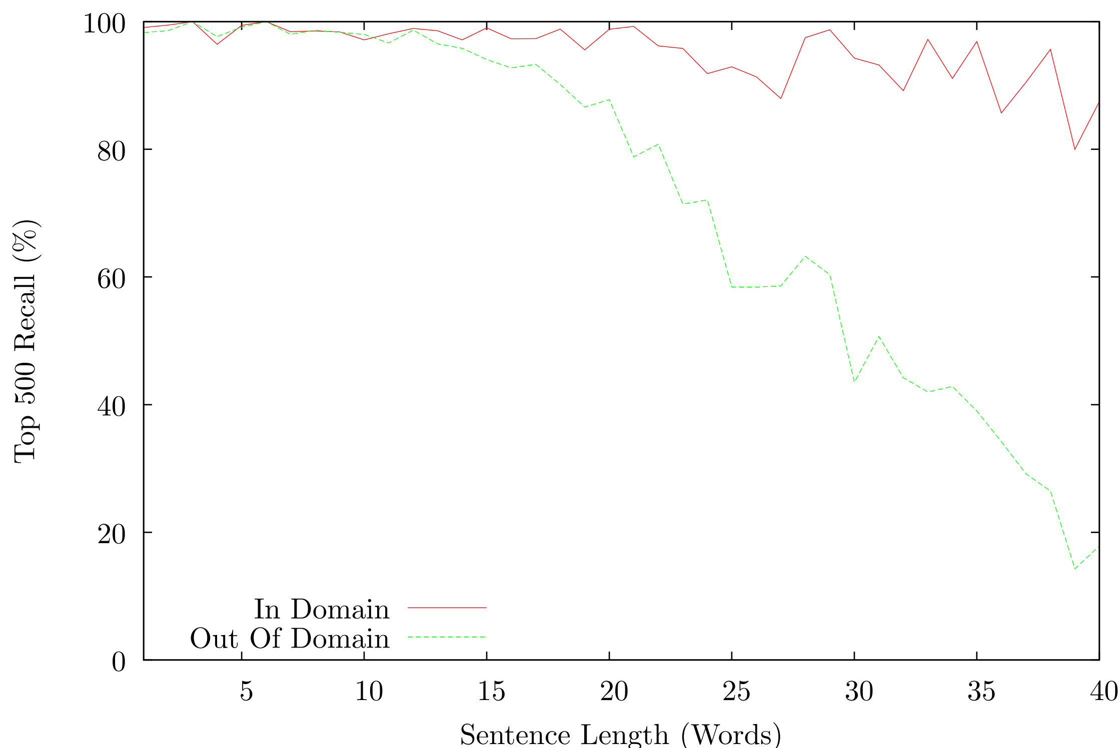

Figure 4.1: Top-500 recall overestimate as a function of sentence length, for sentences that are in-domain and out-of-domain with respect to the parse ranking model.

are lost.

Second, when the parse ranking model is diluted with some out-of-domain data, as is the case for the `redwoods.mem` model, top-500 recall can decrease further. The `redwoods.mem` model is trained on all of the same data that the `wescience.mem` model is trained on, with other data as well,[4] but top-500 recall drops from 97.04% to 92.84%. That corresponds to more than twice as many annotatable sentences lost in top-500 treebanking. Moving to a fully out-of-domain model, losses are even more dramatic. With a highly ambiguous target domain such as WSJ, top-500 recall can plummet from nearly 98% to less than 75%—more than a factor of 10 increase in lost annotatable sentences. Recall also falls off more precipitously with sentence length for the out-of-domain case, dropping below 50%

[4]The WeScience data constitutes roughly half of the `redwoods.mem` training data.

for inputs longer than 30 words. It is clear that top-500 treebanking with an out-of-domain parse selection model is not a viable strategy when a high-coverage treebank is desired.[5]

Combined with the computational and storage efficiency gains cited above, the increase in coverage afforded by full forest treebanking completes the picture of a compelling advantage from a technical point of view. I next examine how efficient full forest treebanking is from the perspective of human effort.

4.2 How Much Effort?

The manual effort that goes into treebanking can be measured either by the number of person-hours invested or by the number of discriminants selected. The existing large-scale Redwoods treebanks nominally include both figures, but the latter are considerably more reliable than the former.[6] Therefore, I base this estimate on the number of discriminant choices required to disambiguate a sentence.

A top-500–style treebank contains N items with gold analyses together with D manual decisions (there are $\frac{D}{N}$ per item) which are sufficient to uniquely select the desired gold analyses from the top-500 lists used to create the treebank. Since an item's full forest frequently contains vastly more than 500 analyses, those same decisions are not always enough to select a unique analysis from the full forest. If a forest contains T trees, and T_C of them are compatible with the stored decisions, then those decisions represent an information gain (with respect to the full forest) of $\log_2 \frac{T}{T_C}$. This allows the computation of an average information gain per stored decision:

$$\bar{\iota} = \frac{1}{D} \sum_{\text{all items}} \log_2 \frac{T^{\text{item}}}{T_C^{\text{item}}} \tag{4.1}$$

[5]To combat this problem, efforts such as the DeepBank project have had to make subsequent passes over the data once a reasonable amount of training data for an in-domain parse selection model is available.

[6]The amount of time invested is recorded in the `t-start` and `t-end` fields of the `tree` relation in the test suite database. Unfortunately, when treebanks are automatically updated for new grammar versions, this information is overwritten with the amount of time the update took. It might be possible to retrieve the original timing data through version control history in some cases. The number of discriminants selected is easily countable (and generally stable across updates) as the number of manual (i.e. not inferred) records in the `decision` relation.

The total entropy of all of the forests in the treebank is:

$$H^{\text{total}} = \sum_{\text{all items}} \log_2 T^{\text{item}} \tag{4.2}$$

The expected number of decisions required to completely disambiguate all of the forests (i.e. starting from scratch, and disambiguating between all T trees rather than just 500) is therefore:

$$\hat{D}' = \frac{H^{\text{total}}}{\bar{\iota}} \tag{4.3}$$

Treebank	N	D	$\frac{D}{N}$	$\bar{\iota}$	H^{total}	$\hat{D}'$	$\frac{\hat{D}'}{N}$	$\frac{\hat{D}'-D}{D}$
WSJ	37741	267648	7.09	1.71	529097	309413	8.20	15.6%
WeScience	8741	50285	5.75	1.69	98229	58124	6.65	15.6%
LOGON	8135	41184	5.06	1.66	77592	46742	5.75	13.5%

Table 4.2: Decisions required for full-forest treebanking

Table 4.2 shows these statistics for the WeScience, LOGON, and DeepBank (WSJ) treebanks. In each case, full-forest treebanking requires roughly 15% more effort than top-500 treebanking, to treebank the same set of items. The additional effort is expended to explicitly reject the candidates that are outside of the top 500. One way to combat this would be to notify the user when all but 1 of the top-500 candidates have been rejected, and offer to allow that single tree to be accepted without making the decisions required to reject the remainder of the forest.

This calculation does not account for the time spent on sentences that are ultimately rejected or for the time spent on sentences ultimately accepted in the full-forest system that would have been rejected in the top-500 system. However, I would expect the figure of 15% more decisions to hold for these classes as well, assuming the annotator makes decisions until only 1 tree remains.

As one final speculation on the effort required for full-forest treebanking compared to top-500 treebanking, it bears considering the effort involved in maintaining treebanks as the underlying grammar undergoes further development. Recall from Chapter 2 that discriminant-

based treebanks can be semiautomatically *updated* when the underlying grammar is revised. In some cases, manual intervention is required during such an update operation: the stored decisions may be compatible with multiple analyses of the new grammar, or they may be not be compatible with any. While these incremental updates can be accomplished relatively quickly, when large treebanks are being maintained, the total annotator effort is still considerable, and can be one of the largest jobs in the release cycle of a precision grammar (D. Flickinger, personal communication, September 17, 2014).

Of particular interest are items that had no desirable analysis in the stored treebank. Whenever the revised grammar introduces new candidate analyses for such items (quite frequent in practice for long sentences, which are the type most likely to be in this class), the annotator is obligated to consider whether each such new analysis might in fact be the correct analysis.[7] In practice, this is often[8] implemented by re-treebanking the item from scratch, and correspondingly a significant portion of the update process is spent revisiting rejected items. If the proportion of items that are rejected can be reduced significantly, the update process can be sped up significantly, leading to greater treebank and grammar development productivity. The average rejection rates for the treebanks above is 7.5% (of all items); the above analysis suggests that full-forest treebanking may allow at least an additional 2% (absolute) of items to be assigned gold analyses, corresponding to a $\frac{2}{7.5} \approx 27\%$ reduction in time spent revisiting rejected items during treebank updates.

4.3 Summary

In this chapter, I have produced estimates to help answer two empirical questions. First, I examined the question of how often the desired analysis is lost by discarding low probability trees in top-500 treebanking. Although exact numbers are hard to produce without access to

[7] When operating under the top-500 regime, the annotator may in fact be unaware that new analyses are available if they do not rank highly enough to be included in the top-500 list. It is unclear to what extent this phenomenon actually occurs; to the extent that it does occur, it may reduce the amount of time spent in treebanking updates, but would of course simultaneously spoil the opportunity to add that item to the coverage of the treebank.

[8] Numbers are hard to come by; another common practice is to maintain a list of reasons for rejection, and to decide (possibly without looking at the new trees) whether the reason is still valid (D. Flickinger, personal communication, September 17, 2014).

large scale full forest treebanks, conservative estimates suggest that at least 2% of all inputs considered in top-500 treebanking with an in-domain parse selection model are rejected when they actually have an acceptable analysis somewhere in the full forest (but outside of the top-500 list), and with an out-of-domain parse selection model the losses are much higher.

Second, I estimated the additional work required to disambiguate the full parse forest relative to the work required to disambiguate from a top-500 list. I predicted that about 15% more decisions need to be made, which may or may not translate to 15% more time spent by a human annotator. Finally, I speculated that the increased coverage afforded by full forest treebanking may actually buy back some of the time spent making those 15% more decisions, in the form of reduced effort in future updates.

These questions will be able to be answered more concretely if a large scale full forest treebank is built sometime in the future. In the next chapter, I summarize the contributions of this thesis, and discuss possible future research directions.

Chapter 5

CONCLUSION

In the preceding chapters, I first reviewed the paradigm of discriminant-based treebanking. This technique allows human annotators to efficiently sift through ambiguity to find the desired analysis among a large number of undesirable (e.g. contextually inappropriate) ones. Largely due to resource limitations, prior work on HPSG treebanking focused on searching for the desired analysis among a subset of the analyses licensed by the grammar (e.g. the most likely 500 candidates according to a statistical model).

In Chapter 3, I showed how the discriminant-based treebanking paradigm can be extended to search for the desired analysis among the complete forest of analyses licensed by the grammar, even when there are so many analyses that enumerating them is prohibitively expensive; I term this *full forest treebanking*. This settles in the affirmative the first two research questions of Section 1.4, i.e. whether such full forest treebanking can in fact be performed within reasonable limits on resources and user-interface complexity; to my knowledge this question has not been answered in the past (at least for HPSG). In fact, the computational resources required for full forest treebanking are typically less (especially in terms of storage) than those required for treebanking with an enumerated top-500 list.

Another advantage, arguably more important than efficiency with computational resources, is that with full forest treebanking the desired analysis is never accidentally discarded in the process of automatically selecting a top-500 list. This leads to increased coverage and less wasted time in future treebank updates when the full forest treebanking methodology is applied, as compared to the top-500 method. The experiments in Chapter 4 show that even with a good in-domain statistical model, at least a couple of percent of sentences appear to have no correct analysis in a top-500 regime when in fact there is an acceptable analysis available outside of the top 500, answering my third research question and confirming my hypothesis that top-500 treebanking is injurious to coverage.

Additional benefits specific to the user interface proposed in this thesis, but not specific to full forest treebanking as compared to top-N treebanking, include the ability to disambiguate cases where a chain of unary rules can appear in multiple orderings, and the ability to quickly navigate to discriminants relevant to a particular span.

One disadvantage of full forest treebanking is that more human decisions are required to disambiguate a sentence compared to the top-500 regime. Answering the fourth research question that I posed in Section 1.4, I estimated the increase in workload at about 15%. It is unclear whether or not this translates into an actual slowdown when accompanied by the proposed user interface changes, and it is also unclear how much time is redeemed by the decrease in update workload concomitant to the increased coverage.

Another disadvantage of full forest treebanking is that the discriminants that can be employed must be local to individual edges in the packed parse chart. This is not a problem when employing syntactic discriminants, since the forest can be transformed to eliminate unary rules, but it makes the use of semantic discriminants (a mode sometimes used with top-N treebanking) impossible. Finding a way to incorporate semantic discriminants into the full forest treebanking architecture would be an excellent area for future research.

A final disadvantage of full forest treebanking is the persistent threat of latent unification failures. Early checking of unambiguous substrings goes a long way towards neutralizing the lost time that these dead ends consume, and annotation guidelines can also help users avoid them, but ultimately they do still happen occasionally. A mechanism to detect such cases earlier would be another profitable area for future research.[1]

I have shown full forest treebanking to be a viable method for producing detailed syntactic annotations with large-scale grammars. If for some particular project semantic discriminants are required, at present top-N treebanking is the only choice. Note however that semantic annotations are still fully accessible in a full forest treebank; they simply are not part of the decision making process when selecting which analyses to record. In any case, if the highest degree of coverage is required, full forest treebanking is the best option, and annotation speed will be comparable to what could be expected with top-500 treebanking.

---

[1] Indeed, such a mechanism would be of great utility to efficient parsing as well.

REFERENCES

Baird, A., & Walker, C. R. (2010). The creation of a large-scale LFG-based gold parsebank. In *Lrec.*

Bond, F., Fujita, S., Hashimoto, C., Kasahara, K., Nariyama, S., Nichols, E., ... Amano, S. (2004). The Hinoki treebank: Working toward text understanding. In *COLING 2004 5th International Workshop on Linguistically Interpreted Corpora* (pp. 7–10). Geneva, Switzerland.

Branco, A., Carvalheiro, C., Pereira, S., Silveira, S., Silva, J., Castro, S., & Graça, J. (2012). A PropBank for Portuguese: the CINTIL-PropBank. In *LREC* (pp. 1516–1521).

Bresnan, J. (2001). *Lexical-Functional Syntax* (Vol. 16). Blackwell Oxford.

Butt, M., Dyvik, H., King, T. H., Masuichi, H., & Rohrer, C. (2002). The parallel grammar project. In *Proceedings of the 2002 Workshop on Grammar Engineering and Evaluation - Volume 15* (pp. 1–7). Stroudsburg, PA, USA: Association for Computational Linguistics.

Carroll, J., & Oepen, S. (2005). High-efficiency realization for a wide-coverage unification grammar. In R. Dale & K. F. Wong (Eds.), *Proceedings of the 2nd International Joint Conference on Natural Language Processing* (Vol. 3651, p. 165 – 176). Jeju, Korea: Springer.

Carter, D. (1997). The TreeBanker. A tool for supervised training of parsed corpora. In *Proceedings of the Workshop on Computational Environments for Grammar Development and Linguistic Engineering* (pp. 9–15). Madrid, Spain.

Clark, S., & Curran, J. R. (2004). Parsing the WSJ using CCG and log-linear models. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics.* Barcelona, Spain: Association for Computational Linguistics.

Copestake, A. (1994). *Representing idioms.* Presentation at the HPSG Conference, Copenhagen.

Copestake, A., Flickinger, D., Pollard, C., & Sag, I. (2005). Minimal recursion semantics: An introduction. *Research on Language & Computation*, *3*(2), 281–332.

Dickinson, M. (2008). Ad hoc treebank structures. In *Acl* (pp. 362–370).

Flickinger, D. (2000). On building a more efficient grammar by exploiting types. *Natural Language Engineering*, *6*(01), 15-28.

Flickinger, D. (2011). Accuracy v. robustness in grammar engineering. In E. M. Bender & J. E. Arnold (Eds.), *Language from a cognitive perspective: Grammar, usage and processing* (pp. 31–50). Stanford, CA, USA: CSLI Publications.

Flickinger, D., Kordoni, V., Zhang, Y., Branco, A., Simov, K., Osenova, P., ... Castro, S. (2012). ParDeepBank: Multiple parallel deep treebanking. In *The 11th International Workshop on Treebanks and Linguistic Theories* (pp. 97–108). Lisbon, Portugal.

Flickinger, D., Zhang, Y., & Kordoni, V. (2012). Deepbank: A dynamically annotated treebank of the Wall Street Journal. In *Proceedings of the Eleventh International Workshop on Treebanks and Linguistic Theories* (pp. 85–96).

Francis, W. N., & Kucera, H. (1982). *Frequency Analysis of English Usage.* New York: Houghton Mifflin Co.

Gazdar, G. (1985). *Generalized phrase structure grammar.* Cambridge, MA, USA: Harvard University Press.

Geman, S., & Johnson, M. (2002). Dynamic programming for parsing and estimation of stochastic unification-based grammars. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics* (pp. 279–286). Philadelphia, Pennsylvania: Association for Computational Linguistics.

Hemphill, C. T., Godfrey, J. J., & Doddington, G. R. (1990). The ATIS Spoken Language Systems Pilot Corpus. In *Proceedings of the DARPA Speech and Natural Language Workshop* (pp. 96–101). Morgan Kaufmann.

Hindle, D. (1983). *User manual for Fidditch, a deterministic parser* (Technical Memorandum 7590–142). Naval Research Laboratory.

King, T. H., Crouch, R., Riezler, S., Dalrymple, M., & Kaplan, R. M. (2003). The PARC 700 Dependency Bank. In *Proceedings of the 4th International Workshop on Linguistically Interpreted Corpora (linc-03)* (pp. 1–8).

Marcus, M., Santorini, B., & Marcinkiewicz, M. A. (1993). Building a Large Annotated Corpora of English: The Penn Treebank. , *19*, 313–330.

Marimon, M. (2010). The Tibidabo Treebank. *Procesamiento del Lenguaje Natural*, *45*(1), 113–119.

Marimon, M., Bel, N., & Padró, L. (2014). Automatic selection of HPSG-parsed sentences for treebank construction. *Computational Linguistics*, *40*(3), 523–531.

Maxwell, J. T., III, & Kaplan, R. M. (1991). A method for disjunctive constraint satisfaction. In *Current Issues in Parsing Technology* (Vol. 126, p. 173-190). Springer.

Miyao, Y., & Tsujii, J. (2002). Maximum entropy estimation for feature forests. In *Proceedings of HLT* (Vol. 2, pp. 292–297).

Miyao, Y., & Tsujii, J. (2008). Feature Forest Models for Probabilistic HPSG Parsing. *Computational Linguistics*, *34*(1), 35–80.

Oepen, S., & Carroll, J. (2000). Ambiguity packing in constraint-based parsing. Practical results. In *Proceedings of the 1st Conference of the North American Chapter of the ACL* (pp. 162 – 169). Seattle, WA.

Oepen, S., Dyvik, H., Lønning, J. T., Velldal, E., Beermann, D., Carroll, J., . . . Meurer, P. (2004). Som å kapp-ete med trollet? Towards MRS-Based Norwegian-English Machine Translation. In *In proceedings of the 10th International Conference on Theoretical and Methodological Issues in Machine Translation* (pp. 11–20).

Oepen, S., Flickinger, D., Toutanova, K., & Manning, C. D. (2004). LinGO Redwoods. A rich and dynamic treebank for HPSG. *Research on Language and Computation*, *2*(4), 575–596.

Pollard, C., & Sag, I. A. (1994). *Head-Driven Phrase Structure Grammar*. Chicago, IL, USA: The University of Chicago Press.

Rosén, V., De Smedt, K., Dyvik, H., & Meurer, P. (2005). TREPIL: Developing methods and tools for multilevel treebank construction. In *in Proceedings of the 4th Workshop on Treebanks and Linguistic Theories.*

Rosén, V., Meurer, P., & De Smedt, K. (2009). LFG Parsebanker: A toolkit for building and searching a treebank as a parsed corpus. In *Proceedings of the 7th International Workshop on Treebanks and Linguistic Theories* (pp. 127–133).

Rosén, V., Meurer, P., De Smedt, K., Butt, M., & King, T. H. (2005). Constructing a parsed corpus with a large LFG grammar. *Proceedings of LFG05*, 371–387.

Steedman, M. (1987). Combinatory grammars and parasitic gaps. *Natural Language & Linguistic Theory*, *5*(3), 403–439.

Steedman, M., & Baldridge, J. (2011). Combinatory categorial grammar. In R. Borsley & K. Brjars (Eds.), *Non-transformational syntax: Formal and explicit models of grammar* (pp. 181–224). Oxford: Wiley-Blackwell.

Tanaka, T., Bond, F., Oepen, S., & Fujita, S. (2005). High Precision Treebanking-Blazing Useful Trees Using POS Information. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics* (pp. 330–337).

Ytrestøl, G., Oepen, S., & Flickinger, D. (2009). Extracting and annotating Wikipedia sub-domains. In *Proceedings of the Seventh International Workshop on Treebanks and Linguistic Theories* (pp. 185–197).

Zhang, Y., & Kordoni, V. (2010). Discriminant Ranking for Efficient Treebanking. In *Proceedings of COLING-2010* (pp. 1453–1461). Beijing, China.