Chapter 1
LITERATURE REVIEW

1.1 Syntax Trees

Perhaps the most 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. This work will not discuss dependency trees, although many of the same principles apply.

1.2 Hand-corrected Treebanking

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

While early corpora (e.g. 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. Experienced annotators were able to maintain an output of 750 annotated words per hour. The authors note that several design decisions were made that reduced the richness and complexity of the annotations in order to improve annotator efficiency — a key consideration when annotating a large amount of text.

1.3 Discriminant-based Treebanking

The hand-edited treebanking process has a number of drawbacks, perhaps the most important of which is that the analyses produced are not held accountable to any formal standard. In cases where the annotation guidelines are murky, annotators are apt to make inconsistent decisions. 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. 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.

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. Since average sentence length in that corpus is about 11 words (Hemphill et al., 1990), we can extrapolate a speed of about 1870 annotated words per hour — more than 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 et al., 2004) using Head-driven Phrase Structure Grammar (Pollard & Sag, 1994) and the TREPIL project (Rosén 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), a broad-coverage precision computational grammar of American English formally grounded in HPSG and Minimal Recursion Semantics (henceforth MRS; Copestake et al., 2005). The authors 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] Whereas Carter's original work employed several types of discriminants, including constituent types, word senses, attachment ambiguities, and grammar rules, the Redwoods work initially only deployed one type of discriminant: labels off of a derivation tree, with their associated constituent spans. Oepen and Lønning (2006) later extended the system to offer discriminants derived from the MRS rather than from the derivation trees. No information is reported about annotation speed in either work.

The Redwoods infrastructure was subsequently deployed to produce a number of additional treebanks for various languages. Examples include the Tibidabo treebank of Spanish (Marimon, 2010), the Hinoki treebank of Japanese (Bond et al., 2004), the CINTIL corpus of Portuguese (Branco et al., 2012), and the ParDeepBank multilingual treebank of the Wall Street journal (Flickinger et al., 2012).

The question of annotation speed with discriminant-based HPSG treebanking was first 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 a rate of about 500 words per hour. However, 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.

[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 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 prominant 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 by constituent length. Compared to this baseline, the authors report an impressive 50% increase in annotation throughput, increasing 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.

The LFG Parsebanker (Rosén et al., 2009) implements a treebanking environment similar to Redwoods, in the LFG universe. 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, featuring dual annotation on 40% of the items. Unfortunately, interannotator agreement and efficiency are not reported.

1.4 Ambiguity Packing

The algorithms and data structures that support large scale parsing and treebanking with precision, broad-coverage computational grammars have evolved vigorously in disparate directions over the years, and to illuminate even the main hallways of that cavernous space would quickly exhaust the available space (and the reader). However, there is one facet of that literature which is very relevant to the present work: the efficient handling of the stupendous ambiguity inherent in natural language. Grammars like the ERG can assign literally trillions of interpretations to longer sentences. Of course, nearly all of these are patent nonsense given an ordinary context, and it can be presumed that only *one* of them was intended by the speaker, but in order to to anything with the intended meaning, the system needs an efficient way of overlooking all of the other tempting-looking analyses. In order to get any traction when interpreting such sentences (and they are not uncommon), it is necessary to avoid at all costs any operation which must iterate over all of the interpretations. At first glance, the idea of a system that can sift through exponentially many analyses in non-exponential time seems preposterous, but that is the very idea that underlies the notion of *ambiguity packing*.

Oepen and Carroll (2000) show how to apply ambiguity packing to the problem of parsing with an HPSG grammar. 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) shows how to apply a similar algorithm for the generation direction, and also how to extract the N top-ranked trees (according to an appropriate maximum entropy 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 unpacking *all* of the analyses.

Miyao and Tsujii (2002) and Miyao and Tsujii (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 & 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).

Independently, Geman and Johnson (2002) developed a method for training the parameters of stochastic unification grammars based on the packed representations of Maxwell III 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 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 to some extent, including rendering graphical representations of the packed structures, but there is no indication as to whether discriminants are computed from and applied to packed structures or to an exhaustively unpacked set of trees, and no description of algorithms used to process the packed structures. 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.

Chapter 2
IMPLEMENTING FULL-FOREST TREEBANKING

2.1 Representing a Forest

A (non-packed) parse chart consists of a set of *edges*, each of which represents an analysis of 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 adjacent daughter edges. When a grammar assigns N readings to a sentence, there are at least N edges in the non-packed parse chart. For highly ambiguous sentences, both time and space constraints on the parser dictate that a *packed* parse chart be produced instead. Edges in a packed parse chart each (implicitly) represent a *set* of analyses of some particular substring of the sentence to be parsed. As in the non-packed parse chart, the local analysis building blocks come in two flavors: lexemes and rules. For a sentence with a large number N of readings, the number of edges in the packed parse chart is typically dramatically smaller than N.

In order to represent the complete analyses 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 or packed in which other edges), and one additional piece of information: the set R of edges which satisfy the grammar's root condition. Note that while a parser may produce edges which are not reachable from root edges (by daughter or packing links), these are superfluous from the point of view of processing of the (grammatically licensed) parse forest, and need not be stored for further processing.

Note that a formal constraint on HPSG analyses requires that the structures are non-cyclic, which implies that no edge can ever be its own (grand-)daughter. Furthermore, the parser's packing system is designed in such a way that an edge can never pack inside of itself. These properties ensure that the parse chart, when viewed as a graph[1], is both directed and

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

Figure 2.1: Non-packed (left) and packed (right) charts.

acyclic.

Figure 2.1 shows examples of a non-packed parse chart and a packed parse chart for the sentence "Kim bought donuts from Lafeen's.".

2.2 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 analysis, for example. 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[2]. Furthermore, some grammars stipulate additional non-unification-based requirements on wellformedness that cannot be straightforwardly represented as constraints on the packed forest[3]. 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 (such as the ERG's idiom filter). 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.

2.2.1 Dynamic Programming

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

[2]Specifically, when packing under subsumption or when the packing restrictor is inadmissible. This is quite common.

[3]For example, the English Resource Grammar incorporates an *idiom filter*, which is a procedural operation on the MRS structure extracted from an analysis.

filters) 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 the forest, a tractible 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:

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

Intuitively, the computation proceeds recursively, with memoization. If an edge p is a packed representative for several "building-block" edges $\{p_1, p_2, \ldots, p_k\}$, then:

$$T(p) = \sum_{i=1}^{k} T(p_k)$$

If an edge u represents the application of a rule to several daughter edges $\{u_1, u_2, \ldots, u_k\}$, then:

$$T(u) = \prod_{i=1}^{k} T(u_k)$$

Finally, if an edge l represents a single lexeme, then:

$$T(l) = 1$$

To compute T, it suffices to compute $T(e)$ for each edge in the forest. The fact that the daughter and packing links between edges form a directed acyclic graph ensures that recursion terminates (in fact, an implementation may choose to first perform a topological sort on the edges and not use recursion at all).

2.2.2 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 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 outlined 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, it is still necessary to consider 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 represented by e conflict with C or not purely on the basis of the information encoded in e. The constraints used in Redwoods can be broken down into two forms[4]:

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

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

Constraints of type 2 can be evaluated with local information, since if edge e with span S is part of a complete derivation, it must comply with X. However, constraints of type 1 cannot be evaluated, 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.

Since constraints of type 1 are common (much more common, in fact, than constraints of type 2), a revised approach is needed.

[4]Positive discriminants take form 1; negative discriminants take form 2.

2.2.3 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 forest into a new, equivalent, but restructured parse forest 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.

The result of this operation is that the derivation trees represented by the transformed forest never contain a configuration where 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 forest may be larger than the number of edges in the original forest, since the local ambiguity of the unary rules is unpacked. In practice, this has not proven to be a significant problem; in fact, some forests even shrink slightly (e.g. when several unambiguous unary edges are eliminated).

2.2.4 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 and packed edges:

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

$$T_C(p) = \sum_{e \text{ packed in } p} T_C(e)$$

For an edge e representing a chain $u_1, \ldots, u_n, r$ terminating in a rule r, with daughters $e_1, \ldots, e_k$, and span S:

$$T_C(e) = \begin{cases} \prod_{j=1}^{k} T_C(e_j) & C \text{ is consistent with } u_1, \ldots, u_n, r \text{ at } S \\ 0 & \text{otherwise} \end{cases}$$

For an edge e representing a chain $u_1, \ldots, u_n, l$ terminating in a lexeme l with span S:

$$T_C(e) = \begin{cases} 1 & C \text{ is consistent with } u_1, \ldots, u_n, l \text{ at } S \\ 0 & \text{otherwise} \end{cases}$$

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 the chain $u_1, \ldots, u_n, x$ at span S. Note that since there can be at most one edge with a given span S, the difference between constraints of types 1 and 2 is that the former asserts the existence of an edge with that span, while the latter does not. 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 the chain $u_1, \ldots, u_n, x$ if and only if every element of C is consistent with the chain. The following rules apply:

1. A constraint with span S_i is consistent with an edge with span S if they are disjoint or nested, i.e. $(S \cap S_i = \emptyset) \vee (S \subsetneq S_i) \vee (S_i \subsetneq S)$. Otherwise,

2. Crossing spans (i.e. $S_i \neq S$) are inconsistent for type 1 constraints but consistent for type 2 constraints.

3. Otherwise $S = S_i$. Type 1 constraints are consistent if and only if $X_i(x)$ or $X_i(u_j)$ for some j. Type 2 constraints are consistent if and only if $Y_i(x)$ and $Y_i(u_j)$ for all j.

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 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.

2.2.5 Enumerating Discriminants

During an automatic 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 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 of trees. These constraints are called *discriminants*.

This of course presents a new engineering problem: how can the list of discriminants be efficiently computed? We have gone to considerable length to ensure that our constraints describe only 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. Figure 2.2 shows how to compute a set of discriminants, given a count $S_C(e)$ of how many complete trees each edge takes part in.

The latter quantity can be computed using dynamic programming from the $T_C(e)$ figure described above. $T_C(e)$ describes how many unique trees are described by the edge e (i.e. trees rooted at that edge). By contrast, $S_C(e)$ describes how many fully spanning trees (rooted at an edge that meets the root condition) containing e somewhere within them are described by the forest. The equations for computing $S_C(e)$ are as follows. If e is an edge representing packed edges $\{e_1, \ldots, e_k\}$, then:

$$S_C(e) = \sum_{p \in \text{ edges of which e is a daughter}} S_C(p)$$

$$S_C(e_j) = S_C(e)\frac{T_C(e_j)}{T_C(e)}$$

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.

The algorithm in Figure 2.2 computes constraints on the full unary chain present at a particular span. These are more informative (in the technical sense) than the single-rule or single-lexeme constraints, 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.

Figure 2.2: Algorithm for Enumerating Discriminants

```
$T$ = total number of trees
constraints= $\emptyset$
treecounts=[]
discriminants= $\emptyset$
**for** each edge $e$ with unary chain $u_1, \ldots, u_k, x$ and span $S$ **do**
  $c$ =constraint that span $S$ has unary chain $u_1, \ldots, u_k, x$
  **if** $c \notin$ treecounts **then**
    constraints = constraints $\cup\{c\}$
    treecounts[$c$] = 0
  **end if**
  treecounts[$c$] = treecounts[$c$] + # of complete trees $e$ is in
**end for**
**for** each constraint $c$ such that treecounts[$c$] $< T$ **do**
  discriminants = discriminants $\cup\{c\}$
**end for**
```

2.3 Human-Computer Interaction

2.3.1 Navigating the Discriminants

2.3.2 Viewing Partial Results

2.3.3 Delayed Unification Failures

Chapter 3
WHAT CAN WE LEARN?

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 three such inquiries: First, when treebanking, how often is the desired tree present in the forest but not ranked in the top-500? Second, when again treebanking, how much additional effort is required to select a unique tree from the full forest compared to from the top-500? Finally, what effect does full forest treebanking have on the statistical models that are trained from Redwoods-style treebanks?

3.1 500 Chances

Although the number 500 is quite arbitrary, it is the de-facto standard for how many trees to consider in Redwoods-style treebanking[1]. It is considered 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. To consolidate this apparent victory, it remains to quantify the risk involved in discarding all but the top-500.

A good numerical measurement of the risk is 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

[1]Some work with the Japanese grammar has used 5000 trees.

[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.

figure will vary considerably with the quality of the 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 model, the top-500 recall would come out to be 100%[3] The only straightforward solution to this problem is to produce gold annotations using the full forest — a labor-intensive process. For lack of a better tool, I proceed to use this biased estimate of the top-500 recall, but it must be understood as an *over*-estimate rather than an accurate figure.

Table 3.1: Top-500 Recall. In-domain results are in bold. The `redwoods.mem` model was trained on both WESCIENCE and LOGON data, as well as other material.

		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%

Table 3.1 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.

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

The WSJ domain consists of newspaper text annotated by the DeepBank project (?, ?), the WESCIENCE domain comprises Wikipedia articles (?, ?), and the LOGON domain is a collection of text from Norwegian tourism brochures (?, ?). 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.

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 processed in a top-500 treebanking system. 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 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 annotatble sentences. 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]

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

[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.

3.2 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. 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 $\frac{D}{M}$ 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 contain 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 $\log \frac{T}{T_C}$. This allows the computation of an average information gain per stored decision:

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

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

$$H^{\text{total}} = \sum_{\text{all items}} \log T^{\text{item}}$$

The expected number of decisions required to completely disambiguate all of the forests is therefore:

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

Table 3.2 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.

Table 3.2: Decisions required for full-forest treebanking

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

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 (decisions made prior to rejecting an item are not recorded in the existing treebanks, so more accurate predictions are hard to make).

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. 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 item with an updated grammar) with a good chance of still finding the desired tree, if it is still available. In some cases, additional 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.

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. In practice, this is implemented by re-treebanking the item from scratch, and correspondingly a significant portion of the update process is spent revisiting each rejected item. If the proportion of items that are rejected can be reduced significantly, the update process can be sped up significantly, leading to greater grammar development productivity. Typical rejection rates for the treebanks above are on the order of 10%; 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 20% reduction in time spent revisiting rejected items during treebank updates.

3.3 Models Trained on Full Forest Treebanks

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.

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.* Stroudsburg, PA, USA: Association for Computational Linguistics.

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

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

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.

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

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). Stroudsburg, PA, USA: 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, 2010, vol. 45, num. 1, p. 113-119*.

Maxwell III, J. T., & Kaplan, R. M. (1991). *A method for disjunctive constraint satisfaction.* Springer.

Miyao, Y., & Tsujii, J. (2002). Maximum entropy estimation for feature forests. In *Proceedings of hlt* (Vol. 2).

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

Oepen, S., & Lønning, J. T. (2006). Discriminant-based MRS banking. In *Proceedings of the 5th International Conference on Language Resources and Evaluation* (pp. 1250–1255). Genoa, Italy.

Pollard, C., & Sag, I. A. (1994). *Head-Driven Phrase Structure Grammar.* Chicago, 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).

Steedman, M., & Baldridge, J. (2011). Combinatory categorial grammar. *Non-Transformational Syntax*.

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.*

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