Integrated Tagging and Pruning via Shift-Reduce CCG Parsing

STEPHEN MERITY

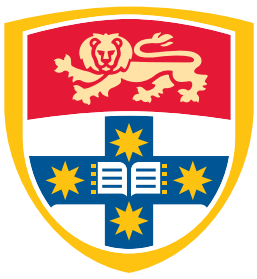
SID: 308147804

Supervisor: Dr. James R. Curran

Bachelor of Information Technology (Honours)

School of Information Technologies
The University of Sydney
Australia

7 November 2011



THE UNIVERSITY OF SYDNEY

Student Plagiarism: Compliance Statement

I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: Stephen Merity

Signature: **Date**:

Abstract

Statistical parsers are a crucial tool for understanding and analysing natural language. Many computing applications would benefit from an understanding of the text it is processing. Unfortunately, this hasn't been possible due to two primary issues in parsing. First, parsers are too slow for practical application on large data sets. The situation becomes worse when applied to real-time applications, such as speech recognition or predictive text editing. The core algorithm in many parsers, the CKY algorithm, requires the entire sentence is provided before parsing can begin. For applications where only a fragment of the input is available, this prevents the use of parsing.

This work demonstrates several novel ideas focused on incremental parsing. The core idea is that state-of-the-art accuracy should be achievable using incremental parsing with no loss in parsing speed. This was particularly challenging as full incremental parsing traditionally has an exponential time complexity. To allow for practical incremental parsing, a data structure called a graph-structured stack had to be implemented for CCG parsing. This allows processing an exponential number of parser derivations in polynomial time. Additionally, incremental parsing can allow for a tight integration between parsing components traditionally considered separate by enabling these components to retrieve a partial understanding of the sentence as it is being parsed. By providing a partial understanding of the sentence to these other components, we can improve both accuracy and speed across many components in the parser. This means incremental parsing is not only useful for real-time applications, but can be used to improve accuracy across the entire parsing pipeline.

By applying the novel features acquired from incremental parsing, we have improved sentence level accuracy in POS tagging by 3.40% and per token supertagging accuracy by 2.00%. These novel features were also used in a form of pruning that improved incremental parsing speed by 39%. This work will lead directly to improvements in a range of Natural Language Processing tasks by enabling a state-of-the-art high-speed incremental CCG parser.

To my friends, family, and mentors.

CONTENTS

List of Figures

List of Tables

CHAPTER 1

Introduction

Parsing is the process of analysing a sentence and extracting its underlying grammatical structure. In artificial languages, there is usually a guarantee that only one valid interpretation of the structure exists. In contrast, natural language is inherently ambiguous and multiple valid interpretations of a sentence may exist. Natural language parsing attempts to resolve these ambiguities, analysing the multiple possible interpretations of a sentence and returning the most likely interpretation.

Parsing assists in a wide range of tasks in the field of Natural Language Processing, including machine translation, automatic summarisation, predictive text editing, semantic role labeling and speech recognition. Unfortunately, parsers are traditionally too slow for large-scale use in these applications. For real-time applications, such as predictive text editing and speech recognition, current parsers cannot be used at all as they are not incremental. Incremental parsers process a sentence a single word at a time and can also handle incomplete sentences. Both of these are necessary for speech recognition, for example, where speech processing needs to occur whilst the person is speaking or where the sentence is interrupted.

The aim of this project was to allow for an high-speed incremental CCG parser that can be used to improve the accuracy in other components in the parsing pipeline. This is achieved by providing a partial understanding of the sentence being parsed to other components. Traditionally, most systems separate components and allow for little or no interaction and tightly integrating the parser with other components in this manner has not previously been explored.

Traditionally, parsing involves a pipeline of components that each are responsible for understanding a small portion of the sentence. The results of each stage of the pipeline are then passed on to later components. This prevents incremental parsing, necessary in a number of real-time applications, and most importantly prevents earlier components in the pipeline from developing a deep understanding of the sentence being parsed. As early errors in the parsing pipeline can lead to reduced accuracy for all

later components using its output, this is a fundamental issue. A clear need exists for a more integrated approach to parsing that allows for all components to understand the deep structure of the sentence being parsed.

1.1 Contributions

1.1.1 Incremental Parsing for the C&C Parser

We extend the high-speed state-of-the-art C&C parser to allow for incremental parsing. As the basis of our approach, we use the shift-reduce algorithm. As this algorithm has a worst case exponential time complexity, we enable practical shift-reduce parsing by implementing the first graph-structured stack for CCG parsing in the literature.

During evaluation, we found our graph-structured stack allowed our incremental parsing implementation to produce output of near equivalent accuracy with only a 34% speed penalty compared to the traditional CKY parsing algorithm used in the C&C parser. Even with this speed penalty, the resulting parser is faster than the majority of constituent parsers in the literature and parses at near state-of-the-art accuracy. We conclude that with further engineering optimisations, the incremental CCG parser could be directly competitive against the traditional C&C parser on both speed and accuracy.

1.1.2 Improved POS tagging and Supertagging using Incremental Parsing

Using the new capabilities of incremental parser, we tightly integrate parsing and tagging for improved accuracy and explore the accuracy impact of errors early in the parsing pipeline. Our core finding is that, by providing the partial understanding of the sentence that the incremental parser has generated as novel features, accuracy in components that traditionally do not interact with the parser can be substantially increased. With these novel features, we improve sentence level POS tagging accuracy by 3.40% and per token supertagging accuracy by 2.00%. This suggests incremental parsing not only allows for real-time applications but could improve accuracy across non real-time NLP applications as well.

We also explore the impact that improved accuracy has further on in the parsing pipeline by evaluating components with and without the earlier accuracy improvements. We show that mistakes early in the parsing pipeline can result in a substantial loss in accuracy as the error propagates to later components. We also show that small improvements to the parsing pipeline, such as an accuracy improvement of

0.21% in per token POS tagging, can result in larger improvements later in the pipeline, such as an accuracy improvement of 0.69% in per token supertagging.

1.1.3 Frontier Pruning for High-speed Incremental Parsing

We implement a form of pruning that improves parsing speed by using the features produced by the incremental parser. As the C&C parser is already so efficient, we show that speed improvements can be difficult to achieve with pruning due to the overhead that pruning introduces. We show that naïve but constrained feature sets can produce higher parsing speeds that more expressive feature sets as the time used in processing the more expressive features outweighs the speed benefits of pruning. This insight will prove useful for other high-speed linguistic parsers.

Frontier pruning allows for a 39% speed improvement for the incremental CCG parser with little impact on accuracy. This negates the 34% speed loss caused by replacing the CKY algorithm with the shift-reduce algorithm in the incremental parser and enables the incremental shift-reduce parser to be directly competitive with the CKY parser on which it is based.

1.2 Outline

Chapter 2 provides an outline of different parsing models, the Combinatory Categorial Grammar formalism and the CCGbank corpus. Chapter 3 describes statistical parsing in detail, with focus on the CKY and shift-reduce parsing algorithms. The graph-structured stack for shift-reduce parsing is also introduced. Chapter 4 explains the evaluation methodology for our experiments and the theoretical motivations behind them. Chapter 5 provides a background of the machine learning methods relevant to the work in this thesis.

Our implementation of both the original shift-reduce algorithm and the graph-structured stack for CCG parsing is described in Chapter 6. Chapter 7 describes the novel features generated by the incremental shift-reduce parser that are used to improve accuracy in POS tagging and supertagging, providing motivation behind the use of incremental parsing in non real-time applications. Chapter 8 introduces frontier pruning which uses the new features from incremental parsing to improve parsing speed. Chapter 9 summarises our work and describes how it impacts on the field of parsing.

Preliminary work on the shift-reduce algorithm, the graph-structured stack and frontier pruning is to be published under the title *Frontier Pruning for Shift-Reduce* CCG *Parsing* at the Australasian Language Technology Workshop in December 2011.

CHAPTER 2

Background

2.1 Models of Parsing

Understanding the structure of a sentence is vital as both the words and the structure of a sentence dictate its semantics, or meaning. Although the sentences (1) and (2) are composed of the same words, they describe distinctly different actions. Sentence (3) is structurally different to both but in fact has the same meaning as (2).

(1) The policeman accused the head inspector of corruption.

(2) The head inspector accused the policeman of corruption.

(3) The policeman was the one accused of corruption by the head inspector.

As these examples show, the syntactic structure is necessary to understand the meaning of a sentence, making naïve textual representations, such as the bag-of-words models used in information retrieval, inadequate.

Parsing is the process of determining the syntactic structure of a sentence in a given language. The set of rules and principles that determine the structure of a sentence in a given language is called the language's *grammar* or *syntax*. Each grammar allows *derivations* of valid sentences, where a derivation lists the sequence of rules and actions from the grammar that are required to form a correct analysis of the sentence. If no valid derivation of a sentence can be found then the sentence is considered *ungrammatical* or structurally ill-formed. Notice that a sentence does not need to be semantically valid to be grammatical, as seen by this oft-quoted sentence from Chomsky (1957):

(4) Colorless green ideas sleep furiously.

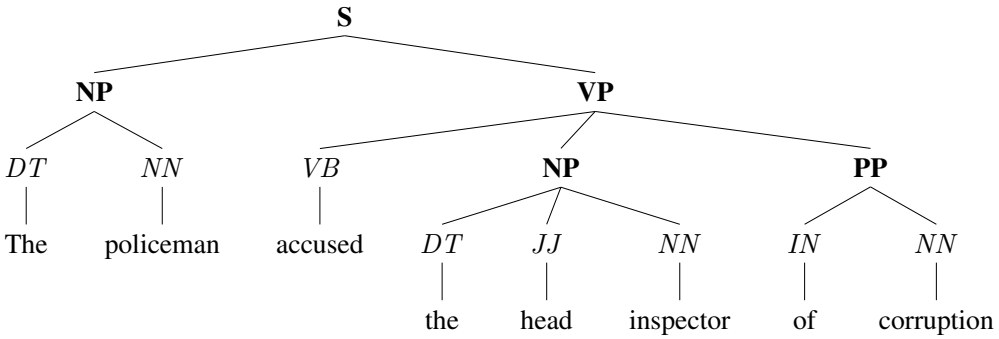


FIGURE 2.1. A constituent structure tree for sentence (1) with phrasal category labels in bold.

The two main types of grammar are based on *constituent structure* and *dependency relations*.

2.1.1 Constituent Structures

A constituent structure grammar breaks down natural language constructions into constituent parts. *Constituents*, also referred to as phrases, are groups of adjacent words that function as a single unit within a sentence. The output of many constituent grammars can be represented as a tree, such as the one seen in Figure 2.1.

These constituent parts are labeled as either phrasal categories or lexical categories. *Phrasal categories* are structures, such as a Noun Phrase (NP) or a Verb Phrase (VP), which are composed of other constituents. These categories can be seen in bold and are the internal nodes of the tree. *Lexical categories* correspond to components, such as nouns, verbs, or determiners, that cannot be further decomposed. These categories are the leaf nodes of the tree.

In each constituent, we identify one element which is known as the *head* of a phrase. This element roughly determines the syntactic properties of the phrase and is modified by all other phrasal elements. For example, the heads of noun phrases and verb phrases are respectively the noun and the verb. In the constituent "the head inspector", we mark the inspector as the head as it is modified by the other elements.

Constituency tests dictate a set of behaviours that constituents must have, including the ability to re-order and substitute constituents. With *substitution* or *replacement*, if you can replace a phrase or clause with a pronoun (*it*, *he*, *her*, *him*, ...) and the change yields a grammatical sentence where the sentence structure has not been altered, then the sequence of words tested are a constituent. Sentence (6) demonstrates

a successful constituent test whilst sentence (5) is ungrammatical. This demonstrates that the whole sequence "the head inspector" is a constituent functioning as a unit.

(5) The policeman accused the head him of corruption.

(6) The policeman accused him of corruption.

These behaviours are useful for a number of tasks in sub-fields of natural language processing (NLP) such as machine translation and automatic summarisation. If we were to translate this sentence to another language, such as Latin, we would have to re-order the constituents. English generally has the form *Subject-Verb-Object* whilst Latin has the form *Subject-Object-Verb*. With constituents, it is trivial to re-order the constituents from one form to the other:

(7) The policeman the head inspector accused of corruption.

2.1.2 Dependency Relations

Dependency parsing is the process of identifying dependencies or relationships between pairs of words. Such relationships include *subject*, *object*, and *modifier* relationships and have been variously known as *dependency relations*, *dependencies*, *grammatical relations* and *predicate-argument structures*.

In each dependency relation, we identify one word as the *head* of the relation and the other as the *dependent* of the relation. The terminology for *head* is similar to that found in constituent structures due to the way that all elements within a constituent are related to the head by dependency relations.

If the grammar permits, the dependency structure may also contain a label for each dependency, indicating the grammatical function. In Figure 2.2, it is clear that the verb "accused" is linked to "policeman" (verb's subject) and "inspector" (verb's direct object).

Note that these dependency structures can be represented as a graph, with words as nodes and the edges as the dependencies between words. Applying graph-theoretic constraints on dependency structures can affect expressivity and parsing efficiency. The tree in Figure 2.2 is projective, for example, meaning that if we put the words in their linear order, preceded by the root, the edges can be drawn above the words without crossings. Projectivity in dependency structures is equivalent to the concept of not allowing crossing branches in a constituency tree. In some cases, a non-projective dependency structure would be preferable and allow for crossing dependencies. Consider the sentence "Branson demonstrated the

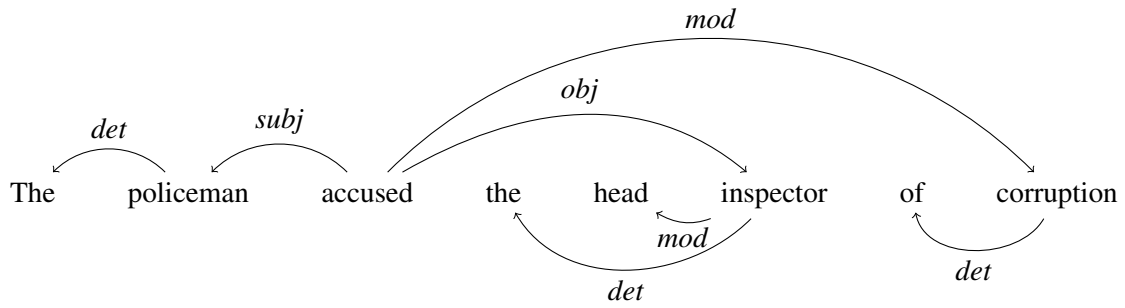


FIGURE 2.2. A directed graph representing the dependencies in sentence (1), where $subj$ = subject, obj = object, det = determiner, and mod = modifier relationships.

new plane *yesterday* which used four jet engines" for example. The word "yesterday" should modify "demonstrated" but separates "the new plane" from the constituent modifying it. These constructions are common in languages with flexible word order, such as German, Dutch and Czech.

In many cases, it is possible to derive dependency relations from an existing phrase structure tree. Dependency grammars and constituency grammars are strongly equivalent provided the constituency grammar is restricted (Covington, 2001). As many dependency grammars are intended to be theory-neutral, this introduces the possibility of using dependency relations as a form of cross-formalism evaluation (Briscoe and Carroll, 2006).

2.1.3 Comparisons between Constituent Structures and Dependency Relations

This thesis focuses on applying a specific algorithm for parsing, the shift-reduce algorithm, to an existing constituent parser. As most shift-reduce parsers perform dependency parsing, it is worthwhile comparing the two representations.

A substantial advantage of dependency relations over constituent structures is that there is no need to build a constituent tree over the sentence. Having a more constrained representation enables both conceptually simpler implementations and computationally more efficient methods of parsing. When the constituent structure is not needed for all tasks, dependency relations can be attractive due to the high-speed dependency parsers available. This is especially true for large-scale systems, where slow parsing speeds may make the task entirely impractical.

For some NLP tasks, constituent structures are a necessity. This is because dependency relations can lack important information contained in constituent structures. For example, when translating into different

languages, statistical machine translation tools use the constituent structure of a sentence to help with word re-ordering (Charniak et al., 2003; Collins et al., 2005). Constituent structure has also been shown to be an invaluable feature in semantic role labeling, leading to considerably better performance than surface-oriented features (Gildea, 2001; Gildea and Palmer, 2002; Chen and Rambow, 2003).

Constituent structure grammars are commonly more deeply motivated by linguistic theory than dependency grammars. As dependency structures have fewer linguistic constraints, theory-neutral dependency structures can be created. As constituent structure grammars can be converted into dependencies in many cases, this enables the possibility of using theory-neutral dependencies as a form of cross-formalism evaluation (Briscoe and Carroll, 2006). Unfortunately, even theory-neutral dependency relations still require the use of non-trivial mapping schemes. These mappings can have a negative impact on evaluation by forcing a substantially reduced upper-bound on achievable accuracy in the task (Clark and Curran, 2007b; Cahill et al., 2008).

Annotation of dependency relations is also considered easier and more natural than annotating the equivalent corpus with constituent structure (Nivre and Scholz, 2004), but there have not been detailed studies on the effectiveness or inter-annotator agreement.

For these reasons, constituent parsers are likely to become progressively less attractive for large-scale practical systems unless significant improvements to either their performance or accuracy occur.

2.2 Combinatory Categorial Grammar

Combinatory Categorial Grammar (Steedman, 2000), referred to as CCG, is a lexicalised grammar formalism that incorporates both constituent structure and dependency relations into its analyses. CCG was derived from a combination of the original Categorial Grammar (Wood, 1993) and combinatory logic (Curry and Feys, 1958). The idea of using combinatory logic to understand natural language was first introduced by Lambek (1958), who eventually formalised these ideas as the Lambek calculus, an extension of categorial grammars.

CCG is termed a lexicalised grammar as each word is associated with a lexical category that defines how the word behaves in the sentence. Categories are either atomic (representing stand-alone constituents) or complex (functions that require other categories in order to produce a grammatical construction).

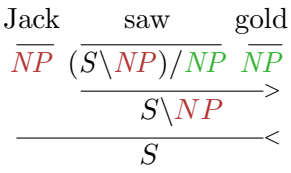


FIGURE 2.3. An example CCG derivation using the complex CCG category $(S\backslash NP)/NP$.

In CCGbank (further described in Section 2.3.2), there are only four atomic categories: S (sentence), N (noun), NP (noun phrase) and PP (prepositional phrase). Complex categories are represented by X/Y or $X\backslash Y$ where X and Y are either atomic or complex categories. Y can be considered a necessary argument in order for the complex category to generate X as a result. The forward and backward slashes indicate that the argument must be on the right or left respectively.

An example of a complex category is the transitive verb $(S\backslash NP)/NP$ in Figure 2.3. The transitive verb takes an NP on the right (money) to produce another complex category, $S\backslash NP$, which then takes an NP on the left (Jack). The final result is the sentence S.

2.2.1 CCG Combinatory Rules

Since categories encode so much of the grammatical information in CCG, only a few generic rules are necessary for combining constituents. These rules are termed *combinatory rules*. The basic combinatory rules are adopted from the original context-free Categorial Grammar (Bar-Hillel, 1953) but are extended with additional rules to allow for more complex linguistic phenomena. These additional combinatory rules increase the grammar's expressive power from context-free to mildly context-sensitive (Shanker and Weir, 1994), meaning more complex linguistic structures can be represented.

2.2.1.1 Forward Application and Backward Application

These two functional application rules, termed forward application and backward application, state that a complex category can be combined with the outermost of its nested arguments to produce the complex category's result. They form the basis of Categorial Grammar.

$$\text{Forward application:} \quad X/Y \quad Y \quad \Rightarrow \quad X$$
$$\text{Backward application:} \quad Y \quad X\backslash Y \quad \Rightarrow \quad X$$

2.2.1.2 Co-ordination

Co-ordination is handled by the addition of a ternary operator (Φ) to CCG.

$$\text{Co-ordination:} \quad X \; conj \; X \quad \Rightarrow_{\Phi} \quad X$$

When two categories of the same type are found to the left and right, they form a single new category of the same type. This allows for other categories to treat multiple constituents as a single category. In the example below, the verb "saw" remains unmodified when it maps to additional direct objects as co-ordination replaces the two CCG categories with a single CCG category representing both of them.

$$\begin{array}{cccccc} \text{Jack} & \text{saw} & & \text{gold} & \text{and} & \text{silver} \\ \overline{NP} & \overline{(S\backslash NP)/NP} & & \overline{NP} & \overline{conj} & \overline{NP} \\ & & & \multicolumn{3}{c}{\overline{\qquad\qquad NP \qquad\qquad}^{<\Phi>}} \\ & & \multicolumn{4}{c}{\overline{\qquad\qquad\qquad\qquad S\backslash NP \qquad\qquad\qquad\qquad}^{>}} \\ & \multicolumn{5}{c}{\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad S \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}^{<}} \end{array}$$

To convert CCG to a binary branching grammar, co-ordination exists in a modified form in most CCG parsers and resources. Conjunction is implemented by using a CCG category of the form $(X\backslash X)/X$ where X is a placeholder and is substituted as needed. In the example below, the derivation above is represented using the binary branching co-ordination rule. In this case, X becomes NP, producing the category $(NP\backslash NP)/NP$.

$$\begin{array}{ccccccc} \text{Jack} & \text{saw} & & \text{gold} & \text{and} & & \text{silver} \\ \overline{NP} & \overline{(S\backslash NP)/NP} & & \overline{NP} & \overline{(NP\backslash NP)/NP} & & \overline{NP} \\ & & & & \multicolumn{2}{c}{\overline{\qquad\qquad NP\backslash NP \qquad\qquad}^{>}} & \\ & & & \multicolumn{3}{c}{\overline{\qquad\qquad\qquad\qquad NP \qquad\qquad\qquad\qquad}^{>}} & \\ & & \multicolumn{5}{c}{\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad S\backslash NP \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}^{>}} \\ & \multicolumn{6}{c}{\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad S \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}^{<}} \end{array}$$

2.2.1.3 Type-Raising

Type-raising is used in CCG to convert atomic categories into complex categories.

$$\begin{array}{rrcl} \text{Forward Type-raising:} & X & \Rightarrow_T & T/(T\backslash X) \\ \text{Backward Type-raising:} & X & \Rightarrow_T & T\backslash(T/X) \end{array}$$

Without type-raising, there are many instances in which the co-ordination rule does not behave correctly. Below the NP category has been raised to a new category $S/(S\backslash NP)$. This category consumes a verb

of the type $S\backslash NP$ and produces a sentence. As there is only a single direct object for both "saw" and "stole", this sentence cannot be represented yet. The rules from the next section are required.

$$\dfrac{\dfrac{\text{Jack}}{NP}}{\dfrac{}{S/(S\backslash NP)}\,{>}\mathbf{T}} \quad \dfrac{\text{saw}}{(S\backslash NP)/NP} \quad \dfrac{\text{and}}{conj} \quad \dfrac{\dfrac{\text{Jill}}{NP}}{\dfrac{}{S/(S\backslash NP)}\,{>}\mathbf{T}} \quad \dfrac{\text{stole}}{(S\backslash NP)/NP} \quad \dfrac{\text{gold}}{NP}$$

2.2.1.4 Composition

In composition, two categories can combine to form a new category if the result of one of them is a required argument of the other.

$$\begin{aligned}
\text{Forward composition:} \quad & X/Y \quad Y/Z \quad \Rightarrow_B \quad X/Z \\
\text{Backward composition:} \quad & Y\backslash Z \quad X\backslash Y \quad \Rightarrow_B \quad X\backslash Z \\
\text{Forward crossed composition:} \quad & X/Y \quad Y\backslash Z \quad \Rightarrow_{B\times} \quad X\backslash Z \\
\text{Backward crossed composition:} \quad & Y/Z \quad X\backslash Y \quad \Rightarrow_{B\times} \quad X/Z
\end{aligned}$$

This allows us to complete the derivation begun in the previous section. By using forward composition, we can combine the phrases "Jack saw" and "Jill saw". Both have CCG categories of $S/(S\backslash NP)$ and $(S\backslash NP)/NP$ after type-raising the NP. $S/(S\backslash NP)$ requires an $S\backslash NP$ on the right, whilst on the right $(S\backslash NP)/NP$ will produce an $S\backslash NP$ when an NP is found on the left. By combining these two functions, we are left with the function S/NP that will produce an S when an NP is found on the right.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{Jack}}{NP}}{\dfrac{}{S/(S\backslash NP)}\,{>}\mathbf{T}} \quad \dfrac{\text{saw}}{(S\backslash NP)/NP}}{\dfrac{}{S/NP}\,{>}\mathbf{B}} \quad \dfrac{\text{and}}{conj} \quad \dfrac{\dfrac{\dfrac{\text{Jill}}{NP}}{\dfrac{}{S/(S\backslash NP)}\,{>}\mathbf{T}} \quad \dfrac{\text{stole}}{(S\backslash NP)/NP}}{\dfrac{}{S/NP}\,{>}\mathbf{B}} \quad \dfrac{\text{gold}}{NP}}{\dfrac{}{S/NP}\,{<}\Phi{>}}}{\dfrac{}{S}\,{>}}$$

Thus we can represent the sentence using co-ordination, type-raising and composition.

Certain restrictions have been placed on composition rules in CCG to prevent over-generation. Over-generation is when a grammar allows for representations of invalid sentences. For example, the sentence "Jill stole because gold, she is poor" (i.e. "Jill stole gold because she is poor") is allowed by CCG when no restrictions are enforced. By disallowing variations of forward crossing composition and backward-type raising, this over-generation can be prevented.

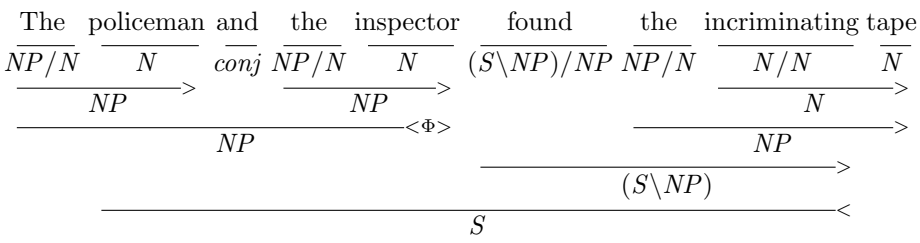


FIGURE 2.4. A CCG derivation

2.2.2 Spurious Ambiguity in CCG

Although Categorial Grammar is context-free and efficiently parseable, the additional rules introduced by CCG lead to what is known as *spurious ambiguity*. This is when even simple sentences can combine constituents in many different ways but still produce identical dependencies. These redundant derivations still need to be considered by the parser, resulting in substantially slower parsing. Wittenburg (1986, 1987) showed that the same categories can be defined in numerous ways due to the flexibility afforded by both combinatory rules and type-raising. In a specific example, the sentence "I can believe that she will eat cakes" is shown to produce 469 equivalent derivations. An example of spurious ambiguity from the sentence "Jill stole the gold" is shown in Figure 2.5. A parser may need to enumerate an enormous space of possible derivations for even a trivially short sentence.

To avoid such redundant analyses, Eisner (1996) introduced a set of simple constraints, now called *Eisner constraints*, that eliminate spurious ambiguity altogether in CCG without the type-raising rule. This is achieved by enforcing two simple rules:

(1) The output of right composition may not compose or apply to anything to its right
(2) The output of left composition may not compose or apply to anything to its left

A parse tree or sub-tree that satisfies these constraints is referred to as the *normal-form derivation*.

The first rule eliminates anything but right branching parses (forward chains such as $A/B\ B/C\ C$) and the second rule eliminates anything but left branching parses (backward chains such as $A\ B\backslash A\ C\backslash B$). When a single forward or backward chain is insufficient, the two merge. Eisner proves that this removes all spurious ambiguity in parsers without type-raising and does not remove necessary trees. Even when type-raising is still used, Eisner constraints have been shown to produce significant speed increases in CCG parsers by reducing the search space (Clark and Curran, 2004a).

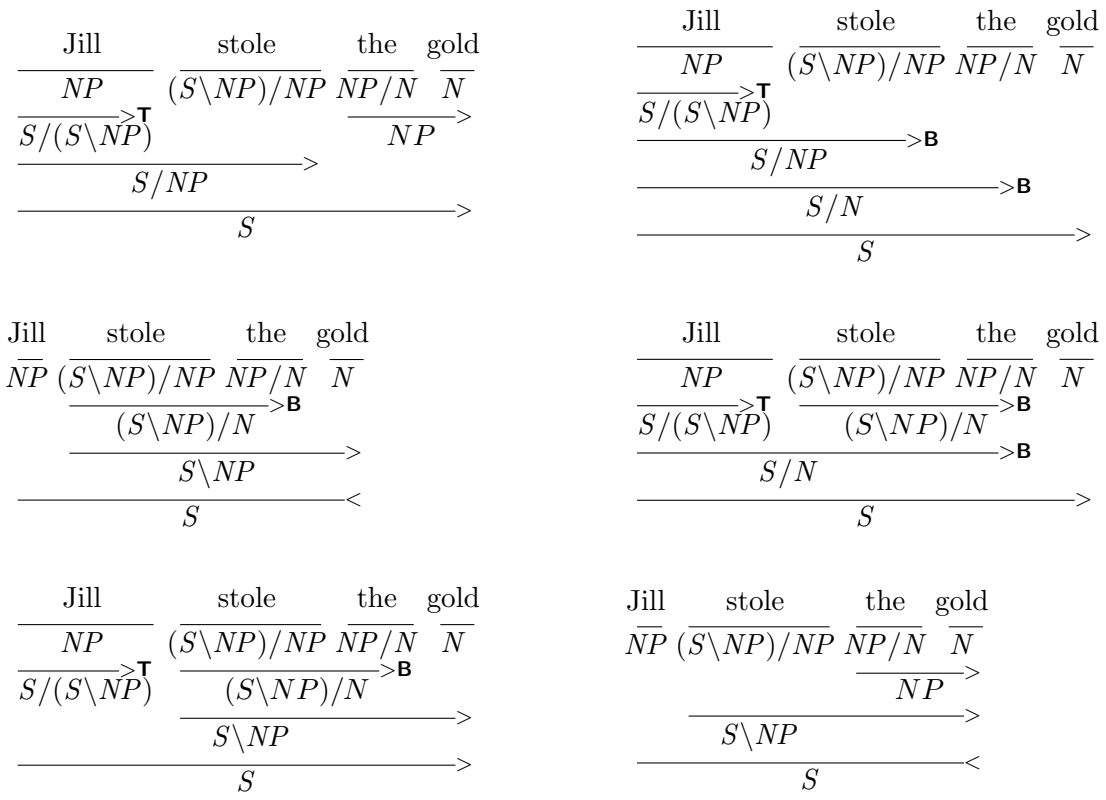


FIGURE 2.5. An example illustrating spurious ambiguity in CCG for the short sentence "Jill stole the gold". All six derivations have the same category assignment, but different derivation trees.

2.3 Corpora

A corpus is a large and usually structured collection of texts commonly created for the purposes of linguistic study. They are often annotated to contain additional linguistic information, such as sentence break locations, the parts-of-speech of words or any other useful structures present in the document. In NLP, most tasks are dominated by data-driven empirical approaches. Corpora are central to both evaluating the tools and providing training data for the statistical models.

Banko and Brill (2001) show that the performance of different machine learning algorithms on a given NLP task perform similarly when given a large enough corpus for training. This implies the more data available for statistical machine learning, the more accurate the results. As statistical parsers use machine learning algorithms for many components, it is likely that parsing performance will also improve with more training data.

```
(S (NP-SBJ There)
   (VP 's
       (NP-PRD no doubt
               (SBAR that
                     (S (NP-SBJ he)
                        (VP 'll
                            (VP go)))))))
```

FIGURE 2.6. An example of the Penn Treebank bracketing structure for "There's no doubt that he'll go". Note the use of the S-expression format from LISP.

Unfortunately, it has been shown that when working on a specific target domain, a large amount of data from a related but out-of-domain corpus can result in worse performance than a small amount of training data from the target domain itself (Biber, 1993; Gildea, 2001). This means that the performance of a parser is strongly related to the genre that the parser is trained and tested against (domain dependence).

As such, it would be optimal to have an extensive, annotated corpus for any domain we may be interested in parsing. Unfortunately, the annotation speeds recorded during the Penn Treebank project (see Section 2.3.1) demonstrated that this process is both slow and expensive. The recent increase in computing power and the explosion of available raw text in electronic form have done little to simplify the annotation process. Thus, the size of the available annotated corpora in NLP are relatively small compared to real world data.

2.3.1 Penn Treebank

The Penn Treebank (Marcus et al., 1993) project was a substantial multi-year endeavour to develop a large, annotated corpus for use in corpus linguistics. The first release contained around 2.9 million words of text annotated with constituent structure trees (see Figure 2.6). The composition of the Treebank was highly varied but included approximately 1.2 million words re-tagged from an existing corpus, the Brown corpus, and 1.1 million words from 2,499 stories from the Wall Street Journal (WSJ). A further 2 million words of text from the WSJ were annotated with part-of-speech (POS) tags but not constituent structure.

Since its introduction, a subset of the corpus, the Wall Street Journal section, has become the standard for both the training and evaluation of parsers in NLP. Commonly, the WSJ section has been converted into a new grammar formalism or extended with novel linguistic information and released as a derived corpus. Examples of this include PropBank (Kingsbury and Palmer, 2002), which adds predicate-argument

```
(<T S[dcl] 0 2>
  (<T S[dcl] 1 2>
    (<T NP 1 2>
      (<L NP[nb]/N DT DT The NP[nb]_137/N_137>)
      (<L N NNS NNS rifles N>)
    )
    (<T S[dcl]\NP 0 2>
      (<T (S[dcl]\NP)/(S[pss]\NP) 0 2>
        (<L (S[dcl]\NP)/(S[pss]\NP) VBD VBD were ...>)
        (<L (S\NP)\(S\NP) RB RB n't ...>)
      )
      (<L S[pss]\NP VBN VBN loaded S[pss]\NP_130>)
    )
  )
  (<L . . . . .>)
)
```

FIGURE 2.7. An example of the CCGbank bracketing structure for "The rifles weren't loaded."

relations, and the Penn Discourse Treebank (Miltsakaki et al., 2004), which adds discourse relations. Although there have been proposals for using the Penn Treebank as a standard evaluation corpus for cross-formalism comparison (Matsuzaki and Tsujii, 2008), major issues have been raised regarding the complexity of the conversion process needed by parsers using different formalisms (Clark and Curran, 2009).

2.3.2 CCGbank

CCGbank (Hockenmaier and Steedman, 2007) is a corpus of CCG normal-form derivations and is the primary corpus used to train and test wide-coverage CCG parsers. It was derived semi-automatically from the WSJ section of the Penn Treebank by a CCG extraction program and then manually re-annotated to ensure consistency (Hockenmaier, 2003). An example of bracketing structure can be seen in Figure 2.7.

CCGbank maintains 99.44% of the sentences from the Penn Treebank, but for the sentences successfully converted the process was not lossless. CCGbank introduces non-standard rules to handle co-ordination, punctuation and other linguistic complexities found in the Penn Treebank that are difficult or impossible to convert to CCG (Hockenmaier, 2003). Co-ordination is implemented via two binary-branching operations instead of a ternary operation, as shown in Section 2.2.1.2. The corpus has recently been been extended to include quotation marks (Tse and Curran, 2007), which were removed during the Penn Treebank conversion, and provide internal structure to noun phrases (Vadas and Curran, 2007, 2008).

These conversion issues present additional obstacles in the implementation of a state-of-the-art CCG parser, particularly as this missing information could be used for more accurate results during the CCG parsing process.

2.3.3 The PARC 700 Dependency Bank (DepBank)

The PARC 700 Dependency Bank, commonly known as DepBank, consists of 700 sentences randomly selected from the Penn Treebank WSJ section 23. King et al. (2003) created the corpus by parsing the sentences, capturing the dependency information, and then manually correcting the produced dependencies.

Briscoe and Carroll (2006) re-annotated DepBank in order to produce a derived corpus with an annotation scheme closer in style to their RASP parser (Briscoe et al., 2006). Their dependency annotation was designed to be as theory-neutral as possible and also contained less grammatical detail to allow for easy cross-formalism comparisons. Even with this in mind, mapping from one representation format to another appears to be difficult. When CCG dependencies generated by the C&C parser were evaluated against this modified DepBank, a number of non-trivial issues led to an upper-bound on accuracy of 84.8% for the task (Clark and Curran, 2007c). This evaluation shows that work still needs to be done on using dependencies as a method of cross-formalism parser evaluation.

2.4 Summary

In this chapter, Combinatory Categorial Grammar (CCG) was introduced. CCG incorporates both constituency structure and dependency relations into its analyses and can model a wide range of linguistic phenomena. The rest of this thesis will be based upon this grammar.

Natural languages, as opposed to programming languages, are inherently ambiguous. Even the most trivial of sentences can have millions of possible derivations. This ambiguity leads to an immense search space that can make parsing impractical for large-scale use.

Whilst dependency parsing is faster and more efficient in many cases than constituent parsing, there are many tasks in which constituent structures are necessary. This includes machine translation, automatic summarisation and semantic role labeling.

CHAPTER 3

Statistical Parsing

Early parsing primarily involved the use of manually curated grammars that were carefully constructed by linguists. These manually curated grammars suffered from low coverage. Coverage is the percentage of sentences that a parser can represent and provide an analysis for. For most practical applications, a wide-coverage parser is required. Wide-coverage parsers aim to parse arbitrary input text without restricting domain or complexity. Manually curated grammars require substantial cost and complexity to allow for wide-coverage parsing (Cahill et al., 2008). Many of these complexities are due to the differences in syntax and semantics across both genres of text and languages themselves.

The task of statistical parsing is to extract these rules from a large collection of manually annotated text called a corpus. The rules and statistics produced by analysing a corpus is called a *parsing model*. As a new parsing model can be created from any sufficiently annotated corpus, we process a relevant genre-specific corpus to discover these differences in syntax and semantics automatically. Many of the methods used to automatically acquire this linguistic information have been adapted from the fields of statistics and machine learning.

In this chapter, two parsing algorithms are explored, the CKY algorithm and the shift-reduce algorithm, and we describe previous approaches to CCG parsing and parser pruning. The CKY algorithm is one of the most widely used constituent parsing algorithms and provides strong guarantees on the worst case running time. The shift-reduce algorithm is used heavily by dependency parsers and allows for incremental parsing, where the next word can be left unknown. This chapter also introduces the C&C parsing pipeline and the impact that parser pruning has on parsing speed.

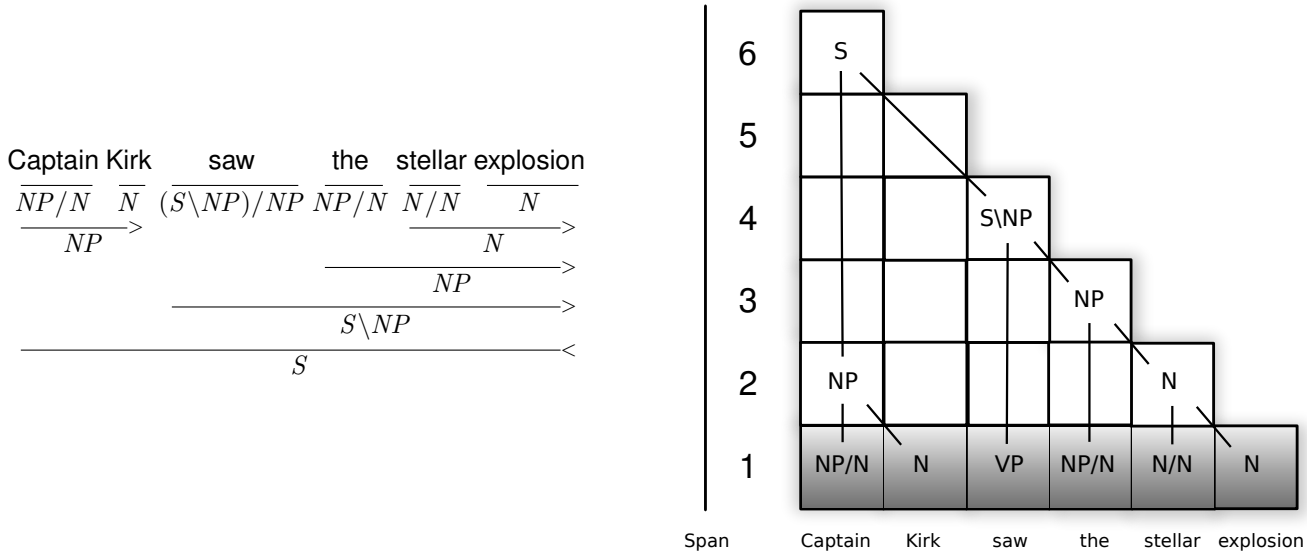


FIGURE 3.1. The resulting chart when parsing a sentence using the CKY algorithm. Note that VP represents $(S\backslash NP)/NP$ but has been shortened for presentation reasons.

3.1 The CKY Parsing Algorithm

The Cocke-Kasami-Younger (CKY) algorithm (Kasami, 1965; Younger, 1967) is a dynamic programming algorithm commonly used to parse with context-free constituent structure grammars. Many of the state-of-the-art results in both constituent structure and dependency parsing have been achieved by parsers using the CKY algorithm at their core.

Chart parsing algorithms are so termed as they use a triangular data structure called a chart to memoise possible analyses, seen in Figure 3.1. The chart begins with all the words in the bottom layer and then combines them until it reaches the root, or top node. The chart begins in the bottom layer, with all the constituents spanning one word. Each step up, the constituents span one more word, incrementally increasing the span until the whole sentence is covered. This is achieved by combining two constituents that together form the current span length. The two constituents must be selected such that one cell is directly beneath the current cell and the other is a cell on the diagonal. As the constituents are built in order of span size, all sub-constituents that are required by the current span have been constructed in previous steps. This is a form of dynamic programming and prevents the reconstruction of smaller constituents.

Consider the CCG category $S\backslash NP$ above "saw" with span length four in Figure 3.1. This is created by combining two categories, the $(S\backslash NP)/NP$ (represented by VP for space reasons) which spans one word ("saw"), and the NP which spans three words ("the stellar explosion"). By combining a CCG category with span length one with another of span length three, we create a new CCG category that spans four words, "saw the stellar explosion".

CKY is attractive as the worst case running time is $O(n^3 \cdot |G|)$, where n is the sentence length and G is the size of the context-free grammar given in Chomsky normal form. This run-time complexity is achieved using dynamic programming on the chart, allowing all possible derivations of a sentence to be calculated without backtracking. Unfortunately, the average case running time for CKY is equivalent to the worst case. Although CKY is the most efficient parsing algorithm in terms of worst case complexity, other algorithms exist for parsing with significantly better average complexity. One of these is the shift-reduce algorithm.

3.2 The Shift-Reduce Algorithm

Shift-reduce parsing is a parsing algorithm composed of two actions and a single base data-structure, the stack. Shift-reduce parsing performs a left-to-right scan of an input sentence and at each step performs one of two parser actions: it either *shifts* the current word onto the stack, or *reduces* the top two items from the stack into a single item (Aho and Ullman, 1972). This type of parsing is attractive for use with deterministic grammars as it is one of the most efficient parsing methods (Knuth, 1965). Deterministic grammars only allow at most one possible action at each point during parsing. As such, the shift-reduce parsing algorithm has been widely used in compiler theory for deterministic grammars. Shift-reduce parsers are also commonly used in NLP due to both their efficiency and also due to their incremental nature. For applications, such as real-time speech recognition, the parser must be able to begin parsing whilst the utterance is still being spoken. As the CKY algorithm requires the words to be known in advance, it is not suitable for such tasks.

The ability to incrementally parse a sentence is vital to the work in Chapters 6, 7 and 8. Incremental parsing allows for parsing in restricted applications, such as speech recognition and predictive text editing, whilst allowing for novel features that can be used in other components (see Chapters 7 and 8).

The deterministic shift-reduce parsing algorithm is described in pseudo-code in Algorithm 1. The deterministic algorithm begins with an empty stack. Until all words from the input have been used, only two

Algorithm 1 The deterministic recogniser version of the shift-reduce algorithm.

```
Q ← input sentence of length N in a queue structure
S ← empty stack
while |Q|and|S| ≠ 1 do
  if top two nodes on S can be reduced then
    A ← pop(S)
    B ← pop(S)
    C ← reduce(A, B) {Reduce the top two nodes according to the grammar}
    push(S, C)
  else
    if Q is empty then
      break
    end if
    A ← pop(Q)
    push(S, A) {Add the new token to the sentence}
  end if
end while
if |S| == 1 then
  return input sentence is in the language
end if
```

actions are possible: *reduce* or *shift*. If a reduce is possible, the two nodes to be reduced are popped off of the stack and replaced with the resulting node. Otherwise, a new node is added from the input. This process continues until all the words from the input have been used and the stack is only composed of a single node.

Figure 3.2 demonstrates the process involved in performing CCG parsing over the sentence from Figure 3.1 using the shift-reduce algorithm. The two examples allow for a direct comparison between CKY and shift-reduce parsing.

Whilst the list of words is provided in the table for instructive purposes, it is important to note that parsing could continue by shifting additional CCG categories on to the stack. The extended sentence found in Figure 3.3 could be successfully parsed from the final step of Figure 3.2. This would not be efficiently possible with the CKY algorithm as the chart data structure must be set up before parsing begins.

The deterministic shift-reduce parsing algorithm can only be used with unambiguous grammars, which prevents anything but the most trivial of natural language grammars. Even the simplest of sentences in natural language contain ambiguity, such as the attachment ambiguity demonstrated in Figure 3.4. If the parser is deterministic, only one possible structure will ever be considered.

Pos	Action	Stack				
1	Shift	NP/N				
2	Shift	NP/N	N			
2	Reduce	NP				
3	Shift	NP	$(S\backslash NP)/NP$			
4	Shift	NP	$(S\backslash NP)/NP$	NP/N		
5	Shift	NP	$(S\backslash NP)/NP$	NP/N	N/N	
6	Shift	NP	$(S\backslash NP)/NP$	NP/N	N/N	N
6	Reduce	NP	$(S\backslash NP)/NP$	NP/N	N	
6	Reduce	NP	$(S\backslash NP)/NP$	NP		
6	Reduce	NP	$S\backslash NP$			
6	Reduce	S				

Pos	Word	CCG Category
1	Captain	NP/N
2	Kirk	N
3	saw	$(S\backslash NP)/NP$
4	the	NP/N
5	stellar	N/N
6	explosion	N

FIGURE 3.2. Walk-through of CCG shift-reduce parsing for the sentence in Figure 3.1. An underline indicates when two categories have been reduced. Pos indicates the total number of words shifted on to the stack so far.

A clear extension is to allow the shift-reduce algorithm to handle non-deterministic grammars. Best-first shift-reduce parsing follows the path deemed most likely to succeed based on a probability model obtained during training. If a dead-end is discovered during parsing, backtracking occurs until the next most likely path is found. This is repeated until the sentence is successfully parsed or a maximum number of paths attempted. Best-first shift-reduce parsing with backtracking can be seen as psycho-linguistically motivated parsing, as it has been suggested that humans analyse sentences incrementally and backtrack for unexpectedly complex or unlikely sentences (Pickering, 1999; Tanenhaus and Brown-Schmidt, 2008).

By allowing backtracking, a non-deterministic shift-reduce parser can be implemented that handles ambiguous context-free grammars. Unfortunately, backtracking results in extreme efficiency issues. Shift-reduce parsing with backtracking can take exponential time in the worst case (Tomita, 1988). Whenever the algorithm backtracks, previous computations are discarded, even if the next path would reuse most

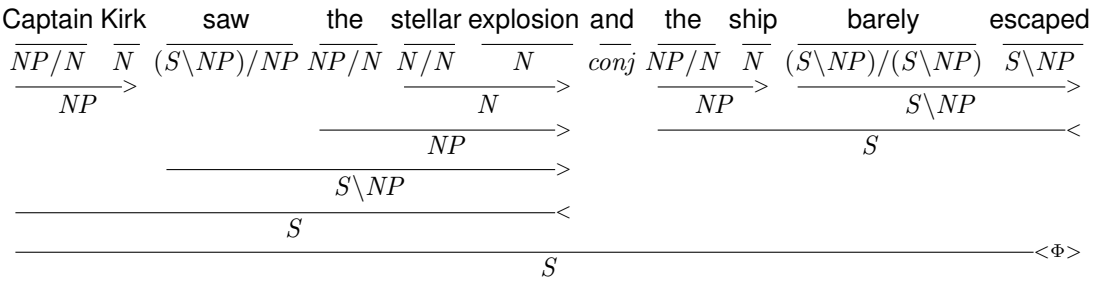


FIGURE 3.3. An extension to the sentence found in Figure 3.1 that could be parsed incrementally using shift-reduce parsing from the state of Figure 3.2.

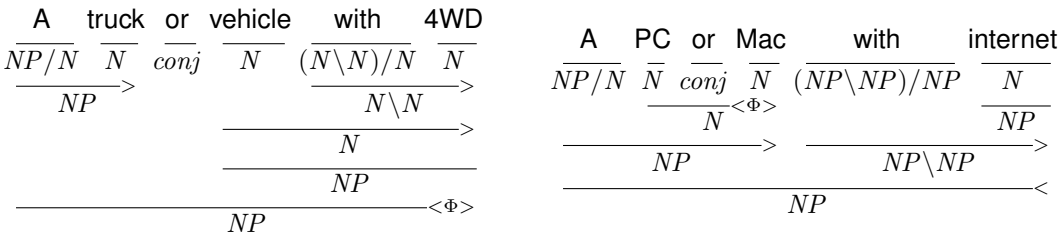


FIGURE 3.4. These two sentences, without additional context, display attachment ambiguity. According to the parse structure, the first sentence states that only the vehicle has 4WD whilst in the second both the PC and Mac have an internet connection.

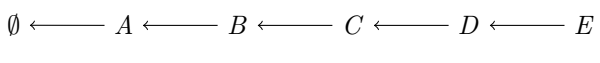
of those computations. CKY and other chart parsers memoise the previous results, preventing this inefficiency. One method that allows for memoisation in shift-reduce parsing is the use of a graph-structured stack, described in Section 3.2.1. Graph-structured stacks have not been actively explored in the shift-reduce parsing community.

As the search space can be exponentially large, some form of pruning must be implemented in order to allow parsing to occur in a practical manner. The majority of shift-reduce parsers in the literature instead use best-first parsing instead of graph-structured stacks to handle the search space. Best-first parsing greedily explores the search space, processing derivations that look most promising first. The two primary approaches have been either a stopping criterion or the use of beam search.

The stopping criterion stops parsing once a maximum number of explored paths have been reached and returns the best parse found so far. This is a naïve method and can result in low coverage but prevents the parser becoming unresponsive when met by challenging sentences.

The more commonly used method is beam search. Beam search holds the n-best partial derivations, scored by a statistical model obtained during training. Searching occurs on the most promising derivation found so far. After each shift-reduce parse action for this derivation, both the n-best partial derivations and the best-scoring completed derivation are retained. This is repeated until no new parse actions can be applied to the partial derivations found in the n-best partial derivation list. At this point, the best-scoring completed derivation found during parsing is returned. If no completed derivation has been found, then the highest scoring of the n-best partial derivations is used. This method allows for a strict guarantee on the time taken to parse a sentence as the value of n can be tailored through experimentation.

The primary drawback to beam search is that if, at any point during the parsing process, the path to the correct derivation is discarded then shift-reduce parsing will not be able to correctly parse the sentence.



Reduction Rules				
F	$\leftarrow$	D	E	
G	$\leftarrow$	D	E	
H	$\leftarrow$	C	D	E

FIGURE 3.5. Consider using shift-reduce parsing to process the stack of tokens by using the grammar on the right. Note a stack composed of only $\emptyset$ is an empty stack.

This can be common when the scoring function does not have sufficient context to appropriately score early derivations.

3.2.1 The Graph-Structured Stack in Shift-Reduce Parsing

In an attempt to improve the efficiency of non-deterministic shift-reduce parsers, Tomita (1987) described an extension to the shift-reduce parsing algorithm for augmented context-free grammars called a *graph-structured stack*. Using a graph-structured stack (Tomita, 1988), the parser was able to maintain multiple derivations without performing inefficient backtracking. This allows for full non-deterministic shift-reduce parsing in polynomial time, removing the major issue caused by backtracking. This work culminated in the release of the Generalized LR Parser/Compiler, designed specifically to be used in practical natural language systems (Tomita, 1990). This technique has seen little use over the years in other shift-reduce parsers. Huang and Sagae (2010) re-introduced the concept and have shown it to be highly effective for high-speed dependency parsing.

To illustrate how a graph-structured stack works, an example will be extended from the original paper where the structure was introduced (Tomita, 1987). In Figure 3.5, an initial stack of tokens has been created and there is a grammar to be applied. This grammar is ambiguous and can result in three different derivations depending on which reduction rule is selected. In traditional shift-reduce parsing only one reduction rule could be applied and without backtracking we would lose the other two derivations.

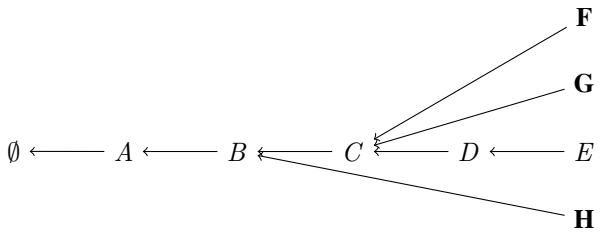


FIGURE 3.6. When a reduce action is applied, *splitting* allows for a graph-structured stack to represent multiple derivations simultaneously in the same structure.

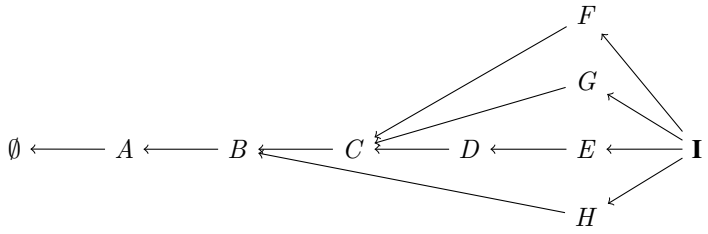
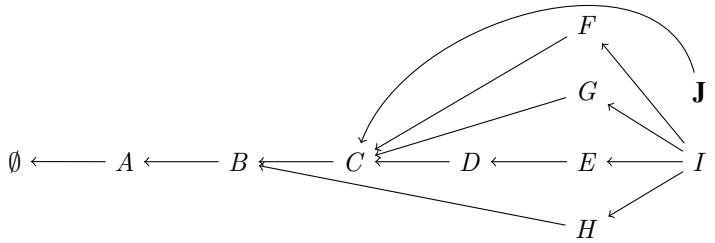


FIGURE 3.7. When a shift operation is performed, the graph-structured stack *combines* the new node to all the heads.

To allow for multiple derivations, the graph-structured stack allows for an action called *splitting*, demonstrated in Figure 3.6. The graph-structured stack allows for multiple heads of the stack. Splitting is performed whenever a reduce operation is performed and creates a new head on the graph-structured stack. The new node created by the reduce operation points to the correct previous node on the stack. As this preserves the previous derivation upon which the reduction takes place, the graph-structured stack makes reduce a non-destructive operation.

When a shift operation is to be performed, pushing a new node to the top of the stack, the graph-structured stack allows for an action called *combine*, demonstrated in Figure 3.7. By *combining* the heads of the graph-structure stack to the new node, we only need to perform a single push to update all possible derivations encoded in the graph-structured stack. In the example, four derivations are updated by the single shift operation.

Finally, when two reduce operations produce the same structure, the graph-structured stack performs *local ambiguity packing* to prevent duplicated work, as see in Figure 3.8. As each reduce operation is completed, the graph-structured stack checks whether the resulting node is equivalent to an existing head. If it is, then we keep track of the ways the given node can be generated and merge them into a single



Reduction Rules			
J	$\leftarrow$	F	I
J	$\leftarrow$	G	I

FIGURE 3.8. Node J has been reduced from both F and G, but only one new node is created. This *local ambiguity packing* allows for polynomial time shift-reduce parsing.

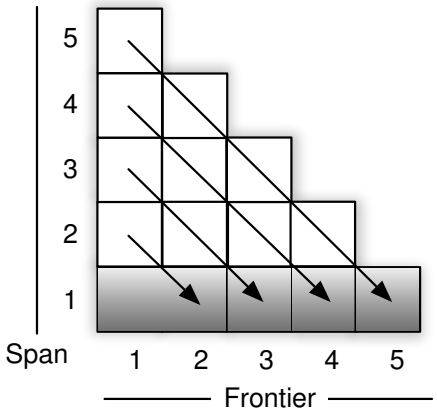


FIGURE 3.9. An illustration of the relationship between the chart in CKY and the GSS in SR. All the diagonal entries in the chart are pushed to the bottom level, with the cell dictating the frontier in the GSS.

node. By producing only one new node, worst case shift-reduce parsing is reduced from exponential time to polynomial time.

These three key notions of splitting, combining and local ambiguity packing are the basis of the graph-structured stack.

3.2.1.1 Frontiers in the Graph-Structured Stack

When parsing an n token sentence, there are n possible stages in the graph-structured stack. We refer to these stages as *frontiers*, with the k^{th} frontier containing all partial derivations that contain a total span of k. This allows us to understand the graph-structured graph through comparing it to a CKY chart. Each frontier in the graph-structured stack can be considered as representing all the cells in the CKY chart on the diagonal from the top left to the bottom right, as seen in Figure 3.9. As the graph-structured stack performs all possible reductions before reaching the next step.

Although the graph-structured stack can be explained briefly, it has been implemented in only a restricted number of shift-reduce parsers (Tomita, 1990; Huang and Sagae, 2010). The main complexity is correctly performing the splitting, combining and local ambiguity packing in new parsing methods and grammars. In Tomita (1987) there is a brief discussion on the feasibility of implementing a

graph-structured stack for Categorial Grammar, which is a precursor to CCG, but many practical implementation issues are omitted. Currently a graph-structured stack for CCG has not been discussed or implemented in the literature.

3.3 Parsing CCG

To illustrate how a CCG parser operates, we will be describing the approach used by the C&C parser. This serves as a basis for the work in this thesis and for other CCG parsers in the literature. The C&C parser will be described in detail in Section 3.4.1. The process of parsing a sentence with the C&C parser can be seen in Figure 3.10 and involves three primary stages: tagging, parsing, scoring and decoding. The tagging stage supplies features that will be used by the parser to improve accuracy or speed, such as part of speech tagging and supertagging. The parsing stage forms attempts to combine the CCG categories together and returns all possible derivations of the sentence. Finally, the scoring and decoding stages process each possible derivation and determines which is most likely to be correct.

3.3.1 Part of Speech Tagging

Tokenised text is initially supplied to a part of speech (POS) tagger. The part of speech tagger used in the C&C framework applies tags from the Penn Treebank tag set. These tags assign a lexical class to each word, helping define the syntactic behaviour of the word in question. Common linguistic categories include *nouns*, *verbs* and *determiners*. The POS tagger uses the surrounding context of the word and features of the word itself to help determine the correct part of speech. The POS tags of each word are then passed on to later stages of the pipeline as additional features to improve accuracy.

3.3.2 Supertagging

There are 1,286 different CCG categories seen in CCGbank Sections 02-21. Enumerating all of them is not feasible. The C&C parser uses supertagging to eliminate tags that are unlikely to be used in the

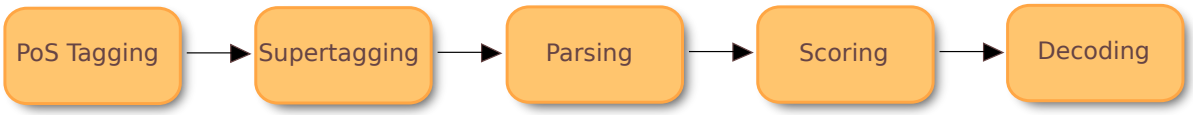


FIGURE 3.10. The traditional parsing pipeline used for CCG parsers.

parsing process, allowing for huge speed increases. The supertagging process has been described as "almost parsing" (Bangalore and Joshi, 1999) since the lexical categories assigned by the supertagger already encode significant amounts of syntactic information. If supertagging were to supply only the correct CCG category for each token in the sentence, CCG parsing is a relatively simple process to combine them to form a correct analysis.

In order to assign the correct tags, the supertagger uses a wide range of features from both the token, the context surrounding the token and the surrounding POS tags. These POS tags are acquired from the previous parsing stage.

Whilst the state-of-the-art per-token supertagging accuracy is around 92% on Section 00 (Clark and Curran, 2007c), this only results in a sentence-level accuracy of 36.8%. This is highly problematic as the average sentence length in the Penn Treebank is 20.54 tokens per sentence. With a supertagging accuracy of 92%, there would be an average of 1.64 incorrect tokens per sentence if the errors were evenly distributed. Due to the high degree of lexicalisation of CCG, a single supertagging error can prevent the parser finding a spanning analysis. A spanning analysis is a derivation that takes all words into account, spanning the entire sentence.

To address this problem, supertaggers can assign multiple lexical categories per token in the sentence in a process known as "multi tagging". In this work, multi tagging will only apply to applying multiple CCG categories to a given word. By returning all lexical categories that are within some factor β of the highest probability lexical category, we can increase the probability that the correct tag will be supplied to a token. By using all categories that are at least 1% as probable as the most probable category, the per-token supertagging accuracy can be raised to 98.5% and the sentence-level accuracy to 78.4%.

Multi-tagging requires a delicate balancing act for optimal performance. If too many categories are assigned, then the parser will have to do increased amounts of work due to the increased ambiguity, becoming slower and slower the more lexical tags that are assigned. If too few categories are assigned, then the parser will likely not be able to form an analysis of the sentence. This raises the possibility of a trade-off between speed and accuracy by modifying the number of CCG tags assigned by the supertagger. This will be discussed in detail in Section 4.4.2.

For a while in the 1970s it seemed Mr. Moon was on a spending spree, with such purchases as the former New Yorker Hotel and its adjacent Manhattan Center; a fishing/processing conglomerate with branches in Alaska, Massachusetts, Virginia and Louisiana; a former Christian Brothers monastery and the Seagram family mansion (both picturesquely situated on the Hudson River); shares in banks from Washington to Uruguay; a motion picture production company, and newspapers, such as the Washington Times, the New York City Tribune (originally the News World), and the successful Spanish-language Noticias del Mundo.

FIGURE 3.11. A sentence, taken from the Penn Treebank, which contains only 108 tokens yet has $6.39 \times 10^{23} = 2^{79}$ different possible analyses.

3.3.3 Parsing

Parsing attempts to combine the CCG categories together using CCG combinatory rules until a spanning analysis is found. If the parser enumerates all the possible derivations and cannot find a spanning analysis, the parser fails. The most common algorithms for performing CCG parsing, the CKY and shift-reduce algorithms, have been previously described.

3.3.4 Scoring and Decoding

Due to ambiguity in natural languages, it is possible that a sentence could have multiple spanning analyses. To select the most likely analysis, a scoring model is applied. The scoring model is most commonly probabilistic in nature and uses parameters chosen during training.

For certain sentences, it is not even computationally feasible to enumerate all possible derivations the parser has produced. The sentence in Figure 3.11 demonstrates the challenge that wide-coverage parsers face when handling complex sentences.

Decoding is the process of finding the highest-scoring derivation from the derivations the parser has produced. Different decoders can be used in an attempt to optimise towards different goals. For the C&C parser, two primary decoders exist: the dependency decoder and the normal-form decoder. The dependency decoder involves summing the probabilities of all possible derivations that yield a particular dependency structure. This includes the nonstandard derivations eliminated by normal-form constraints. The normal-form model scores normal-form derivations more highly than the alternate derivations.

3.4 Existing Parsers

As CCG, as used in CCGbank, is a binary branching grammar, it is use possible to use the CKY algorithm and the shift-reduce algorithm for CCG parsing. The choice of algorithm has a high degree of impact on the parser and what role supertagging can play. This is explored below by analysing the C&C parser (Clark and Curran, 2007c) and the Zhang and Clark shift-reduce CCG parser.

3.4.1 The C&C Parser

The C&C parser (Clark and Curran, 2007c) is a highly efficient state-of-the-art CCG parser trained on CCGbank. Through a number of evaluations, the C&C parser has been found to be competitive with state-of-the-art parsers across grammatical relations (Clark and Curran, 2007c), Penn Treebank phrase-structure trees (Clark and Curran, 2009) and unbounded dependencies (Rimell et al., 2009).

The parser uses the CKY algorithm for CCG described in Steedman (2000). The CKY algorithm has been modified to merge equivalent derivations caused by ambiguity (which still exists even after Eisner constraints have been enforced) and increases the parsing efficiency of the parser.

Supertagging in the C&C parser involves a back-off model. The parser initially attempts to find a spanning analysis of a sentence using as few CCG tags per token as possible. If parsing fails, the parser requests more CCG tags per token by lowering the beta level of the supertagger and repeats the parsing process. This continues until either the sentence is successfully parsed, the C&C parser enumerates the entire search space, or a user specified threshold is reached. This user specified threshold is most commonly implemented by ending parsing once a specified maximum number of CCG categories have been created during the parsing process.

Through a tight integration between parsing and supertagging Clark and Curran (2007c) use the C&C parser to show that efficient wide-coverage CCG parsing is possible and that state-of-the-art results can be achieved without sacrificing speed.

3.4.1.1 Adaptive Supertagging

As mentioned previously, supertagging increases the speed of CCG parsers substantially. The key, however, is ensuring that the supertagger provides the CCG categories that the parser requires. If the supertagger supplies the wrong tags initially, more supertags are supplied. In many cases this results in the

correct tags being provided to the sentence. In other cases, the supertagger will not supply the correct categories but the parser can still form a spanning analysis.

When more supertags are requested for each word, parsing speed decreases due to the search space increasing due to ambiguity.

Kummerfeld et al. (2010) implement a novel self-training method that allows for interaction between choices the supertagger makes and the C&C parser itself. The supertagger is trained on the CCG categories that the parser returns from the successfully parsed sentences, rather than the CCG categories supplied by human annotators in CCGbank. This results in a supertagger and parser that have much lower ambiguity levels and/or successfully parse the sentence during an earlier pass, resulting in a substantial parsing speed increase. As the supertagger selects the CCG categories that the parser would have used anyway, there is no impact on the accuracy of the parser. This is termed adaptive supertagging.

3.4.2 Hassan et al. Incremental CCG Parser

Hassan et al. (2008) present an incremental, linear time dependency parsed based on a deterministic transform of CCG. To allow for incremental parsing, the shift-reduce algorithm is used. The output of the parser is to be used in machine translation, speech recognition and word prediction systems where speed is a factor.

Whilst their work aimed for high accuracy, they claim the deterministic and linear-time nature of their parser forces results below that of state-of-the-art for CCG parsing. They report a parser speed increase of ten times over the C&C parser on comparable hardware.

3.4.3 The Zhang and Clark Shift-Reduce CCG Parser

The shift-reduce parser implemented in Zhang and Clark (2011) performs full CCG parsing, as opposed to the modified CCG subset used by Hassan et al. (2008). It achieves state-of-the-art results in CCG parsing and compares favourably to the C&C parser on which it is based.

The Zhang and Clark shift-reduce CCG parser uses a number of components from the C&C parser, such as the tagging and evaluation infrastructure. This allows them to provide a detailed error comparison with the C&C parser. As little changes other than the parsing algorithm, this demonstrates the relative advantages and disadvantages of the CKY and shift-reduce algorithms.

3.5 Pruning

Due to both the ambiguity in natural languages and the flexibility of CCG, parsing speed can be impractical for many tasks. Ambiguity must be reduced to improve parsing speeds. This is commonly achieved through pruning of the parser search space.

3.5.1 Lexical Pruning

Lexical pruning uses the words and the textual hints from the input sentence to prune the search space that must be explored. This is possible as many words are commonly used in restricted contexts.

Lexical pruning traditionally occurs before the parser has begun processing the sentence. In C&C parsing, supertagging performs lexical pruning by restricting the possible CCG categories that can be assigned to a word. This is necessary as CCGbank Sections 02-21 contain 1,286 different CCG categories. To naïvely attempt all CCG categories seen in training with each word in the sentence would be disastrous. Even attempting only the CCG categories seen with a particular word would not work. The words as, is and to have been seen with over 100 different CCG categories each in CCGbank Sections 02-21 for example. If each of the 50 categories had to be considered each time one of those words were seen, CCG parsing would be impractical.

An example of this could occur with the word "saw" during CCG parsing. The word could be either a noun (a cutting implement) or a verb (to see). Without additional context, all possible CCG categories that apply to the word would need to be supplied. With additional context, however, it is possible we can reduce the assigned categories without an impact on parsing accuracy. If the preceding word was "the", then it is most likely a noun as few instances of "the saw" have occurred with "saw" as a verb.

If a sentence is not successfully parsed, the parser requests more supertags from the supertagger and repeats the process. In the worst case, it would be possible to repeat this until the supertagger provides all possible supertags ever associated with the word. As such, if the sentence was a rare construction with "the saw" containing "saw" as a verb, then the supertagger would eventually supply the unlikely CCG category.

3.5.2 In-Parser Pruning

Numerous methods have been suggested to improve the speed of CKY chart parsers, both from linguistics and computer science. Some methods stem from graph search algorithms, such as research into best-first probabilistic chart parsing (Caraballo and Charniak, 1996, 1997), beam-search and A^* chart parsing (Klein and Manning, 2003; Pauls and Klein, 2009). Other than A^* chart parsing, none of the other techniques can guarantee that they won't accidentally discard the most probably derivation. This means that the speed increase in these methods comes at a potential accuracy cost.

Methods of chart pruning have also been proposed to improve the speed of CKY parsers. Chart pruning, by indicating words that can start or end multi-word constituents, has met with success in CCG and other formalisms (Roark and Hollingshead, 2008, 2009; Zhang et al., 2010). Coarse-to-fine parsing, where a parser first uses a coarse level grammar to prune the search space for the original grammar, has also been successfully used with the CKY algorithm (Charniak et al., 2006). Unfortunately, generating a coarse level grammar from a fine level grammar is not trivial and has only been done for probabilistic context free grammars (PCFG).

Various practical implementation improvements have also been tried, including parallelisation of the CKY algorithm (Haas, 1987; Grishman and Chitrao, 1988) and a CKY parser using bit-vector operations for both speed and space efficiency (Schmid, 2004). Whilst not linguistically oriented, both these approaches can result in substantial speed-ups as hardware becomes more and more sophisticated.

Even with all of these various speed improvements, CKY parsing is not yet practical on a large scale.

3.6 Summary

In this chapter, we have given a broad background of statistical parsing with a particular focus on parsing CCG. The CKY and shift-reduce algorithm have been described and their relevant merits discussed. The shift-reduce algorithm can parse sentences incrementally but cannot parse sentences efficiently. By using dynamic programming, the CKY algorithm is efficient but cannot parse incrementally.

The graph-structured stack was introduced to allow efficient shift-reduce parsing. Few GSS-based parsers have been implemented, however, and have not been the actively explored since Tomita (1987).

Finally, the traditional parsing pipeline for the C&C parser was described. Currently there is no interaction between the tagging and parsing processes. Tagging happens before parsing and can only use the surface features of language such as the raw text itself. In Chapter 7 we explore the possible accuracy gains from tightly integrating the tagging and parsing processes.

CHAPTER 4

Evaluation

Performing a fair and thorough evaluation on parsers is a difficult task. In this chapter, we describe a number of evaluation metrics used for parsers. We also discuss the difficulty in performing cross-formalism comparisons and problems with the field-standard PARSEVAL measure. These evaluation metrics are vital in understanding how modifications to the parser impact performance, both in relation to speed and accuracy.

4.1 Metrics

4.1.1 Precision, Recall and F-score

Before describing accuracy metrics for parsers specifically, it is helpful to introduce accuracy metrics in a broader context. These will be discussed in terms of classification tasks. Imagine we were attempting to identify spam email. In general terms, the three most common metrics in measuring accuracy are precision, recall and F-score. These would be defined in terms of:

- True Positives: Instances that are correctly predicted as being in class X (real spam)
- False Positives: Instances that were incorrectly predicted as being in class X (normal email classified as spam)
- True Negatives: Instances that were correctly predicted as not being in class X (normal email)
- False Negatives: Instances that were not classified as class X but should have been according to the gold standard (real spam classified as normal email)

For classification tasks, *precision* (Equation 4.1) is the number of true positives over the number of true positives and false positives and *recall* (Equation 4.2) is the true positives over the true positives plus false negatives.

These two metrics are required as they provide different aspects of system accuracy. The spam classification system could achieve a recall of 100% (i.e. successfully identify all spam messages) by assigning spam to all pieces of email. This system could also achieve a precision of 100% (i.e. successfully avoid identifying any real messages as spam) by assigning only the most confident cases as spam. The former would not be a very useful system, whilst the latter may be just as useless if it only identifies 1 in 10,000 spam messages. For identifying spam messages in email, it would be preferable to focus on precision over recall (i.e. identify fewer of the spam messages but avoid classifying normal email as spam), but recall is still a consideration. By providing both of these metrics, proper evaluation can occur on the relative benefits of optimising precision over recall or vice versa.

$$P = \frac{tp}{tp + fp} \tag{4.1}$$

$$R = \frac{tp}{tp + fn} \tag{4.2}$$

$$F_\beta = (1 + \beta^2)\frac{PR}{\beta^2 P + R} \tag{4.3}$$

Finally F_β-score is based upon the effectiveness measure introduced by Rijsbergen (1979) and attempts to provide a metric that represents both the precision and the recall of the system as a single metric. This is accomplished by calculating the weighted harmonic mean of both precision and recall (Equation 4.3). By adjusting β, we can place more emphasis on either precision or recall. In practice, F-score usually refers to F_1-score and provides an even weighting between precision and recall. This allows two systems with differing precision and recall to still be comparable on a single metric. Unless stated otherwise, all reported F-scores in this thesis will refer to calculation with $\beta = 1$.

The use of precision, recall and F-score in evaluating the C&C parser will be discussed more specifically in Section 4.2.2.

4.1.2 Coverage

For parsing, coverage is a measure of the percentage of sentences in a corpus that the parser returns one or more parses for. As there is no confirmation of whether the sentences have been parsed correctly, this is a weak measure that provides an upper-bound on the number of sentences a parser could theoretically

produce correct analyses for. Constituent parsers require a spanning analysis, or an analysis that covers all the words in a sentence, before a sentence is considered successfully parsed.

Coverage issues can arise in two primary areas, either due to the coverage of the grammar or errors during parsing. Due to the complexities of natural language, many grammars are not flexible enough to allow for all possible sentences. When sentences use rules not covered by the grammar, parsing is not possible and the no analysis is returned for the sentence. The other source of coverage issues are errors during parsing. For example, if the correct CCG categories are not provided during CCG parsing then it may be impossible to form a spanning analysis.

It is important to note that fragmentary coverage is also possible. Fragmentary coverage is where a subset of the sentence has been successfully parsed but the full sentence has not been. For dependency parsers, fragmentary coverage is a natural result as a single unsuccessful dependency is not likely to prevent the rest of the parsing process. If a given construction deviates too far from the grammar's coverage, but the parser returns the dependencies from the other sub-structures, then the parser successfully returns a fragmentary analysis of the dependency structure. Fragmentary analysis for constituent parsers is also possible but is more complicated as partial trees need to be combined together.

Finally, it is standard in the field to return accuracy metrics only for the sentences that the parser successfully covers. This can be a problem as sentence complexity is not the same. Parsers are more likely to fail on complex sentences for which they would receive low accuracy. By avoiding complex sentences, the overall parser accuracy may appear to be much higher than it actually is.

4.2 Evaluation against the Wall Street Journal

In the field of NLP, the Wall Street Journal subset of the Penn Treebank is used in the training and evaluation of many parsers. The corpus has been split into 25 sections numbered from 00 to 24 with certain sections commonly used for specific purposes. The canonical configuration is Section 00 for development, Sections 02-21 for training, and Section 23 for final evaluation.

Whilst this enables a direct comparison between different parser implementations and formalisms, it could potentially be leading to over-fitting in the community as all parsers are aiming to improve accuracy on Section 23. This also prevents certain forms of statistical analysis on the results, such as k-fold

cross validation, as only the results for Section 23 are considered directly comparable by others in the community.

4.2.1 PARSEVAL

One of the most common performance metrics used to evaluate parsers on the Penn Treebank and similar derived corpora is the PARSEVAL metric (Black et al., 1991).

PARSEVAL checks label and word-span identity in parser output compared to the original treebank trees, scoring the extent to which the bracketing in the two analyses match (labeled bracketing). It provides no measure of error severity and gives no credit to constituents with slightly incorrect phrase boundaries despite correctly recognised syntactic categories. Bangalore (1997) even questions whether success in the PARSEVAL metric is related to success in parsing due to these numerous limitations.

Rehbein and van Genabith (2007) explore whether parsing less-configurational languages such as German is more difficult than English or whether the lower parsing scores of German are just artifacts of treebank encoding and the PARSEVAL metric. By inserting controlled errors into gold-standard treebank trees, they are able to measure the effects of these errors on the parser's evaluation performance. These experiments provide evidence of fundamental issues with the PARSEVAL metric and show that PARSEVAL performance is highly influenced by the treebank encoding.

Despite severe criticisms, PARSEVAL is still the standard metric for parser evaluation on the Penn Treebank. This makes comparing state-of-the-art results between parsers using different formalisms a difficult task (Clark and Curran, 2009). The underlying issue is that constituent accuracy is not meaningful to all parsers.

4.2.2 CCGbank Dependency Recovery

The PARSEVAL metric makes a particularly poor evaluation metric for CCG parsers. This is specifically as the PARSEVAL metric measures similarity between tree bracketing structures. Due to spurious ambiguity (see Section 2.2.2), a CCG parser can return many different equivalent analyses for a sentence that each have a different tree structure. As the PARSEVAL measure can only consider a single gold standard tree to compare against, this can lead to penalising correct CCG derivations due to differing structure and bracket placement even when two CCG trees are equally correct and semantically equivalent. Additionally, Penn Treebank trees are flat in structure as opposed to binary branching CCG trees.

```
saw_2 ((S[dcl]{_}\NP{Y}<1>){_}/NP{Z}<2>){_} 1 Jack_1 0
stole_5 ((S[dcl]{_}\NP{Y}<1>){_}/NP{Z}<2>){_} 1 Jill_4 0
and_3 conj 1 stole_5 0
and_3 conj 1 saw_2 0
pure_7 (N{Y}/N{Y}<1>){_} 1 gold_8 0
the_6 (NP[nb]{Y}/N{Y}<1>){_} 1 gold_8 0
stole_5 ((S[dcl]{_}\NP{Y}<1>){_}/NP{Z}<2>){_} 2 gold_8 0
saw_2 ((S[dcl]{_}\NP{Y}<1>){_}/NP{Z}<2>){_} 2 gold_8 0
```

FIGURE 4.1. The dependencies produced by the C&C parser for the sentence "Jack saw and Jill stole the pure gold".

With substantially more brackets in the CCG trees, comparing scores across Penn Treebank based trees and CCG trees is not valid. To counter this, the standard evaluation for most CCG parsers is dependency recovery over a held-out test set from CCGbank.

To understand what dependency recovery looks like in CCG, we can refer to Figure 4.1. These are the dependencies produced when the parser analyses the sentence "Jack saw and Jill stole the pure gold".

Dependency (a) indicates that the word "pure" is modifying "gold". This is indicated by the numbering in the CCG category, $N/N < 1 >$, as it says *pure* modifies whatever is in argument slot *1*. The *1* to the left of *gold* indicates it occupies this argument slot.

(a) pure_7 (N{Y}/N{Y}<1>){_} 1 gold_8 0

This can be extended to more complex CCG categories. Dependency (b_0) indicates that *Jill* was the subject of the verb *stole*, whilst dependency (b_1) indicates that *gold* was the direct object of *stole*. This can be seen by repeating the analysis from the example given in dependency (a) except noting that the CCG category, $(S\backslash NP < 1 >)/NP < 2 >$, has two argument slots that can be filled by noun phrases. The first argument slot, $NP < 1 >$, represents the subject of the verb whilst the second argument slot, $NP < 2 >$, represents the direct object of the verb.

(b_0) stole_5 ((S[dcl]{_}\NP{Y}<1>){_}/NP{Z}<2>){_} 1 Jill_4 0

(b_1) stole_5 ((S[dcl]{_}\NP{Y}<1>){_}/NP{Z}<2>){_} 2 gold_8 0

The evaluation can be calculated over two dependency types: *labeled* dependencies and *unlabeled* dependencies. *Labeled dependencies* take into account the category containing the dependency relation, the argument slot, the word associated with the category, and the head word of the argument. Only if

Parser	Coverage (%)	Labeled F-score (%)	Unlabeled F-score (%)	Speed (sentences/second)
CKY C&C	99.01	86.37	92.56	55.6
CKY C&C Auto	98.90	84.30	91.26	56.2

TABLE 4.1. Evaluation of the CKY C&C parser on Section 00 of CCGbank, containing 1,913 sentences. Auto indicates that automatically assigned POS tags were used instead of gold standard POS tags.

all four are correct is the dependency considered successfully recovered. This can be considered a harsh evaluation metric as the parser is not rewarded for correctly identifying some subset of the required labels. For example, if the parser provided the wrong CCG category for a verb but correctly identified the verb's subject, the dependency is considered entirely incorrect. *Unlabeled dependencies* only require the head and the argument to match, allowing partially correct dependencies like this to be considered correct. The calculation of precision, recall and F-score for dependency recovery in CCGbank is described below and continued in more detail in Clark et al. (2002).

$$P = \frac{\text{\# dependencies correctly recovered by the parser}}{\text{\# dependencies in the parser output}}$$

$$R = \frac{\text{\# dependencies correctly recovered by the parser}}{\text{\# dependencies in the gold standard}}$$

$$F = \frac{2PR}{P+R}$$

In addition to these dependency-based metrics, the C&C parser reports three additional metrics: *sentence accuracy* (the percentage of sentences that have all the dependencies correct), speed (in sentences per second) and *coverage* (the percentage of sentences that actually receive an analysis, see Section 4.1.2 for full detail).

An example of the output produced when evaluating the C&C parser can be seen in Table 4.1.

4.3 Speed Comparisons

Many NLP tasks require the processing of massive amounts of textual data. Since the advent of the World Wide Web, both the size and variety of potential data sources has increased substantially. Such large amounts of data necessarily demand high efficiency parsing tools, as otherwise many tasks in NLP would become impractical due to the processing time involved.

There is a trade-off between parser accuracy, parser efficiency and linguistic detail. There have been some attempts to produce an efficiency metric that is independent of hardware (Roark and Charniak, 2000) but these have not been widely accepted by the research community.

The current method of speed comparison in NLP is simply reporting the time taken to parse a given corpus on an arbitrary hardware instance. This makes fair speed comparisons nearly impossible as hardware instances are difficult to replicate and the source code and data models for many of these experiments are never publicly released.

It is also common to measure parsers in terms of sentences per second, though this too is problematic. As sentences are of variable length and complexity, the measured speed of sentences per second can fluctuate widely. In our evaluation, the parsing speed (as measured in sentences per second) of Section 23 is nearly twice that of Section 00, even though Section 23 has more sentences than Section 00 (2,407 sentences compared to 1,913). Kummerfeld et al. (2010) has attempted to demonstrate parsing speed by grouping and testing on sentences of specific length, but this has not been duplicated by others in the community and there is no official selection of sentences allowing for a fair comparison.

As such, there is no agreed metric for comparing parsing speed and efficiency between parser implementations. Parsers that do not achieve state-of-the-art results are generally met with little enthusiasm, even if the parsing speed has been substantially improved. Efficiency needs to become an important metric in parsing before NLP is more widely used.

4.4 Tagging Accuracy

In our work, tagging is the task of assigning a class to a given word in a sentence. Whilst the evaluation for POS tagging and supertagging are similar for the single tagging case, the evaluation for multi tagging used in supertagging is more involved.

4.4.1 POS Tagging

Accuracy in POS tagging is defined by two metrics: accuracy calculated on a per token basis and accuracy calculated on entire sentences. These are calculated by

$$Accuracy = \frac{\text{\# of correct tags/sentences}}{\text{\# of total tags/sentences}}$$

For tagging, we refer only to accuracy as precision is equivalent to recall because each word must be a ssigned a tag.

Calculation over entire sentences is important to provide an indication of system performance in a real setting. For example, given a POS tagger with a high per token accuracy of 97%, there is one error every 35 classifications. Whilst the average sentence length of section 02-21 of the Penn Treebank is 20.54 words per sentence, many sentences exceed 35 words in length, meaning statistically one or more errors will occur.

An example of the output produced when evaluating the C&C POS tagger can be seen in Table 4.2.

Algorithm	Accuracy (tags)	Accuracy (sentences)
POS Tagging	96.80%	51.76%
Supertagging	93.31%	43.81%

TABLE 4.2. POS tagging and supertagging accuracy on Section 00 of the Penn Treebank, calculated across both individual tags and entire sentences.

4.4.2 Supertagging

For supertagging, the same accuracy metric introduced for POS tagging is used. This results in an accuracy calculated on a per tag basis and accuracy calculated on entire sentences. State-of-the-art supertagging, when only a single supertag is assigned, is around 93%. An example of the single tagging output produced when evaluating the C&C supertagger tagger can be seen in Table 4.2.

As supertagging is less accurate than POS tagging, and a missing CCG category can prevent a successful derivation, a method called multi tagging is employed. The supertagger is allowed to assign more than one possible CCG category per word. If any of the assigned supertags is the correct CCG category, parsing can continue successfully. As such, we provide a lightly modified accuracy metric for multi tagging,

$$Accuracy = \frac{\text{\# of CCG tag sets that contain the correct tag}}{\text{\# of total words}}$$

Unfortunately, assigning more than one supertag slows parsing speed substantially. As such, the aim of supertagging is to use as few tags as necessary. To understand the evaluation for multi tagging, a number of parameters must be described:

- Beta Levels (β): Any tag within a certain fraction, β, of the most likely CCG tag will be included in the tag set for the word. Low β values increase the number of categories per word, and thus the accuracy, but result in a slower parser (see *categories per word*).
- Dictionary cutoffs (k): The dictionary cutoff forces the supertagger to use only the categories the category has been seen with in the training data if the category occurs less than k times in the training data. This is as the lack of context combined with a low β level may lead to extremely large numbers of categories assigned to the word.

As part of multi tagger evaluation, the average number of CCG categories assigned per word is recorded. This is defined as *categories per word* and represents the level of ambiguity introduced by the supertagger. This value should be as small as possible whilst allowing for the highest accuracy possible as it is directly related to parsing speed.

Finally, as POS tags are used as features for the supertagger, any errors that occur during POS tagging have an impact on supertagging accuracy. To identify how much of an impact this has, two accuracy metrics are reported: one uses gold standard POS tags and the other uses automatically (auto) assigned POS tags provided by the POS tagger.

An example of the output produced when evaluating the C&C multi tagger for CCG supertagging can be seen in Table 4.3. Note that the number of categories assigned per word increases as the beta level

β	k	CATS/ ACC	ACC WORD	SENT (GOLD)	CATS/ ACC	ACC WORD	SENT (AUTO)
0.075	20	1.25	97.73	69.84	1.26	97.33	65.82
0.030	20	1.41	98.26	75.34	1.41	97.89	71.37
0.010	20	1.68	98.69	80.50	1.68	98.38	76.53
0.005	20	1.94	98.84	82.09	1.94	98.54	78.12
0.001	150	3.46	99.36	89.46	3.46	99.08	85.03

TABLE 4.3. Multi tagger ambiguity and accuracy on the development section, Section 00. The tag *auto* indicates that automatically assigned POS tags were used.

drops. This has a positive impact on accuracy, but results in higher levels of ambiguity for the parser. This in turn results in lower parsing speeds.

4.5 Summary

In this chapter, the methods used to evaluate changes in performance are described. Parsing performance is commonly described in terms of precision, recall, F-score and coverage. Parsing speed is described in terms of sentences per second. Whilst the metrics for measuring parsing speed are naïve and differ widely between parser implementations and the hardware used, they provide a strong indication as to the practical use of a system at the time of its description.

For CCG parsing performance, dependency recovery on CCGbank is used. The use of dependencies provides a method of evaluation across parsers using different formalisms and so can be considered independent of the linguistic theory used.

CHAPTER 5

Machine Learning for Statistical Parsing

Machine learning algorithms are one of the core tools in statistical natural language processing. In these machine learning problems, an algorithm is provided a set of features representing an instance of the problem. These features are then used to classify the instance into a set of classes or assigned a probability that a particular class is correct. This is most commonly done by providing a weight for each feature, storing them in a weight vector. These classes are generally either a yes or no decision (binary classification) or a set such as POS or CCG categories.

For the training process, the algorithm is provided with a set of training instances. Each training instance is composed of a set of features and a definitive class classification. The training procedure then aims to find a set of feature weights that minimize the classification error over this training data. Some machine learning algorithms guarantee an optimal set of feature weights but others can only guarantee a locally optimal set of feature weights.

5.1 Background

5.1.1 Averaged Perceptron

The perceptron learning algorithm is an online machine learning algorithm that works through gradient descent. The algorithm itself is conceptually simple and only updates weights when an instance is incorrectly classified. As the learning process only requires the current instance, training can be done without access to all possible training instances. This is referred to as online training. It has been shown that for any data set that is linearly separable, the perceptron algorithm is guaranteed to find a solution in a finite number of steps. If the perceptron learning algorithm is run on a problem that is not linearly separable, then the weights may fluctuate wildly as the algorithm re-adjusts the weights for each

Algorithm 2 The prediction step for a binary perceptron classifier

$\theta \leftarrow$ dictionary of feature to feature weights (commonly referred to as the weight vector)
$feats \leftarrow$ the set of features an instance is composed of
$total \leftarrow 0$

for $f \in feats$ **do**
  $total = total + \theta_f$
end for
return true if $total > 0$ else false

incorrect instance. Even with this drawback, perceptron-based algorithms are still commonly used due to their simplicity and ability to perform online training.

An important extension to this concept is the averaged perceptron algorithm (Collins, 2002). By using the arithmetic average of the feature weight values seen during training, the algorithm is more resistant to weight oscillations. This can result in substantially improved performance with no computational overhead during classification.

5.1.2 Maximum Entropy Modeling

Maximum entropy, or log-linear models, are statistical models that can incorporate information from a diverse range of complex and potentially overlapping features. The central idea of maximum entropy modeling is that the chosen model should satisfy all constraints imposed by the training data whilst remaining as unbiased (i.e. most uniform) as possible. To achieve an unbiased model, the entropy of the distribution is maximized whilst obeying the constraints the data provides. By maximizing the

Algorithm 3 A single iteration of the perceptron learning algorithm for a binary classification problem

$\theta \leftarrow$ dictionary of feature to feature weights (commonly referred to as the weight vector)
$T \leftarrow$ training set composed of features ($feats$) and a binary class ($kls \in -1, 1$)
for $(feats, kls) \in T$ **do**
  $guess = predict(\theta, feats)$
  if $guess \neq kls$ **then**
    {The weight vector is only modified if there is an error}
    for $f \in feats$ **do**
      {Each feature is moved one up or down by one depending on the direction of error}
      $\theta_f = \theta_f + kls$
    end for
  end if
end for

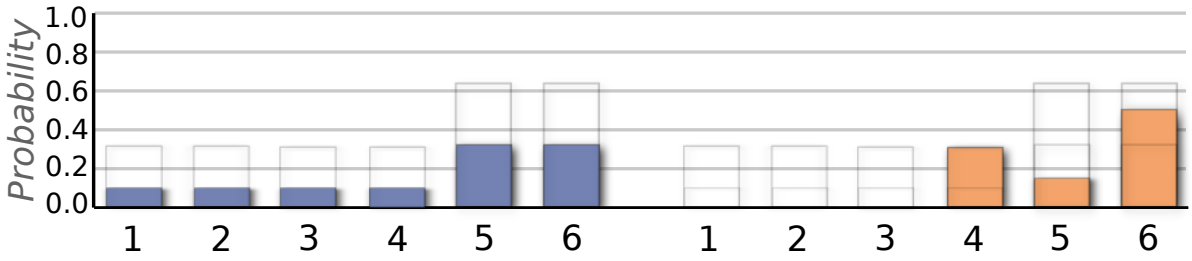


FIGURE 5.1. Presented here is two probability modes for a dice. Both obey two constraints: $p(1) + \cdots + p(6) = 1$ and $p(5) + p(6) = \frac{2}{3}$. The model on the left is the maximum entropy distribution whilst the model on the right contains biases not accounted for by the constraints.

entropy, we minimize the amount of prior information built into the distribution, leading to an unbiased distribution.

A simple and intuitive example of this can be seen through modeling the probability distribution of an unknown six-sided dice, demonstrated graphically in Figure 5.1. We are given only two constraints to generate the probability model for the dice. The first ensures the model is a legal probability distribution by ensuring that $\sum^X p(x) = p(1) + \cdots + p(6) = 1$. The second specifies that two thirds of the time either 5 or 6 are rolled, written as $p(5) + p(6) = \frac{2}{3}$.

A human would, with no other information, assume that $p(5) = p(6) = \frac{1}{3}$ and that the four remaining values would occur with equal probability. This is the probability model with maximum entropy, as seen on the left of the example figure. By distributing the probability mass as evenly as possible given the constraints, we make no assumptions in the model beyond what the constraints provide. This is not the only distribution that these constraints allow, however. The supplied constraints allow for an infinite number of different yet valid probability models. The figure on the right of the example figure illustrates one of these possible probability models.

Maximum entropy modeling has been used extensively in obtaining statistical models for NLP. Ratnaparkhi (1996) produced state-of-the-art results for POS tagging on the Wall Street Journal corpus. It was claimed that as maximum entropy modeling makes no assumption on the distribution of provided features that diverse experimentation was easier. This included implementing rich feature sets and providing contextual features that were overlapping or statistically sparse. This work forms the basis of the POS tagger and supertagger found in the C&C framework (Curran and Clark, 2003).

Many machine learning techniques are susceptible to *over-fitting*, where the model sacrifices generality in order to improve accuracy on the training corpus. Rare features can exacerbate over-fitting as the lack of context can result in excessively large positive or negative weights for the features. Traditionally, to prevent over-fitting due to rare features, any feature that occurs less than n times in the training corpus is removed. This is referred to as a frequency cut-off. A frequency cut-off is based on the assumption that if a feature rarely occurs, it is unlikely to be informative or reliable.

Maximum entropy modeling can allow these rare features to be considered without leading to over-fitting by a process called *smoothing*. By using a Gaussian prior as part of the training process, the model penalises large positive or negative weights (Chen and Rosenfeld, 1999). This discourages heavy reliance on any one feature, allowing for rare features to be added to the process. The use of smoothing with a Gaussian prior has been shown to improve accuracy over using a frequency cut-off (Curran and Clark, 2003). These results suggest that the assumption behind applying frequency cut-offs is incorrect and that rare features can be useful in attaining higher accuracy.

As opposed to the perceptron model, maximum entropy modeling requires summations over all of the features in the training corpus and training cannot be done in an incremental fashion. The training process is also computationally intensive as there is no analytic solution for finding the maximum entropy model. Training therefore uses numerical solvers that provide better and better approximations of the optimal model after each iteration. The number of iterations required and the time taken per iteration are highly dependent on the solver and the training corpus supplied.

5.2 Algorithm Implementations in the C&C Parser

5.2.1 Maximum Entropy Modeling

Maximum entropy modeling plays an integral part in the C&C parser. It is used in POS tagging (Curran and Clark, 2003), supertagging (Clark and Curran, 2004b) and in producing the scoring model used to determine the highest-scoring derivation (Clark and Curran, 2007c).

The training and classification components of the maximum entropy classifier are written in *C++*. For training the parser scoring model, the training process can require up to 25 GB of memory. To handle this hardware requirement, a parallelised version of the algorithm exists that can run on a cluster of

machines. For the training of the POS tagger and supertagger models used in this thesis, however, a single machine is sufficient.

5.2.2 Perceptron Algorithm

In an attempt to lower the memory requirement for training the maximum entropy model, Clark and Curran (2007a) investigate training using the perceptron algorithm instead of maximum entropy modeling. Whilst the perceptron training process takes over three times as long as the maximum entropy training process, it uses only 20 MB of memory compared to the 19 GB required by the parser scoring model. The results of the perceptron model are closely comparable to those of the maximum-entropy model, suggesting that either training algorithm could be used depending on the context.

The perceptron algorithm has also been used in adaptive supertagging (Kummerfeld et al., 2010). As mentioned previously, adaptive supertagging is a form of self-training as it uses the parser output as training data. The CCG categories from sentences that have been successfully parsed are used as training data for the supertagger. The perceptron algorithm was chosen as it could be trained online, improving the efficiency of the supertagging model after each new sentence has been parsed.

5.3 Summary

In this chapter, the two primary methods of machine learning used in this work are introduced: maximum entropy modeling and the averaged perceptron classifier.

Maximum entropy modeling is theoretically sound but is an offline algorithm, requiring all the training samples to be provided before training can begin. The perceptron algorithm is conceptually simple and can be implemented efficiently, but does not have the same theoretical guarantees as maximum entropy modeling. The perceptron algorithm has the advantage of being an online algorithm and can be trained incrementally by adding new training data.

The maximum entropy modeling is used as the machine learning algorithm in Chapter 7 for all the tagging results, whilst the perceptron algorithm is implemented in Chapter 8 for use in frontier pruning.

CHAPTER 6

Implementing the Shift-Reduce Algorithm in the C&C CCG Parser

---

The C&C parser has a highly optimised implementation of the CKY algorithm at its core. The CKY algorithm is not incremental however and requires all the words of a sentence to be provided before parsing can begin. For the experiments performed in Sections 7 and 8, incremental parsing is necessary.

The first major contribution of this thesis is the implementation of the shift-reduce algorithm in the C&C parser, allowing for incremental CCG parsing. As naïve shift-reduce parsing results in exponential time complexity, we implement a graph-structured stack for CCG to allow for polynomial time shift-reduce CCG parsing. As the shift-reduce algorithm does not require knowledge of the upcoming tags, this means that the features of the current parse state can be used to more accurately classify the incoming words. This is explored in Chapter 7.

The work in this chapter has been accepted for publication under the title *Frontier Pruning for Shift-Reduce* CCG *Parsing* at the Australasian Language Technology Workshop in December 2011.

6.1 The Shift-Reduce Algorithm for CCG Parsing

The shift-reduce algorithm has recently been extended to handle CCG parsing by both Hassan et al. (2008) and Zhang and Clark (2011). The parser in Hassan et al. (2008) provides incremental shift-reduce parsing by using a deterministic grammar, sacrificing the expressiveness of CCG. Thus, our focus will fall on the parser in Zhang and Clark (2011) which implement a shift-reduce CCG parser with comparable accuracy to the C&C parser. Whilst they do not focus on the incremental nature of the shift-reduce algorithm, it is still beneficial to discuss how shift-reduce parsing has been modified for CCG by Zhang and Clark (2011). The set of actions used by their CCG parser is {*shift*, *reduce*, *unary*}.

The *shift* action has been substantially modified from the traditional definition used by shift-reduce dependency parsers. In shift-reduce dependency parsers, the next word is pushed on to the stack. In

shift-reduce CCG parsing, however, there are multiple CCG categories that could be considered. These CCG categories are supplied by the supertagger. In the worst case hundreds of CCG categories may need to be considered. By only allowing one word to be added, the *shift* action is performing lexical category disambiguation in Zhang and Clark (2011) by rejecting other potentially valid CCG categories.

The *reduce* action closely mirrors that used in traditional shift-reduce parsers. The top two CCG categories on the stack are popped off, combined to form a new CCG category, and then the resulting CCG category is pushed to the top of the stack. In CCG parsing, the *reduce* actions are the CCG combinatory rules.

Finally, the shift-reduce algorithm is extended to include the *unary* action. The *unary* action pops the top CCG category off of the stack, applies a unary rule to form a new CCG category, and then pushes the resulting CCG category on to the top of the stack. This additional action was necessary to allow the unary type-changing and type-raising rules found in the CCG grammar to be parsed using the shift-reduce algorithm.

As shift-reduce parsing is exponential in the worst case, Zhang and Clark (2011) used beam search in the shift-reduce CCG parser. After each parse action, both the n-best partial derivations and the best-scoring completed derivation are retained. This is repeated until no new parse actions can be applied to the partial derivations found in the n-best partial derivation list. At this point, the best-scoring completed derivation found during parsing is returned. If no completed derivation has been found, then the highest scoring of the n-best partial derivations is used.

6.2 Extending the Graph-Structured Stack for CCG Parsing

Whilst Tomita (1988) provides an example of how a graph-structured stack could be applied to Categorial Grammar, a precursor to CCG, it is limited in scope and is provided in the paper as a proof of concept.

The primary step in implementing a graph-structured stack for a new grammar formalism is converting the three base concepts: splitting, combining and local ambiguity packing. Whilst GSS-based uses shift-reduce parsing as a basis, it is interesting to note how divergent the actions for {*shift*, *reduce*, *unary*} are when compared to their use in the Zhang and Clark (2011) CCG parser.

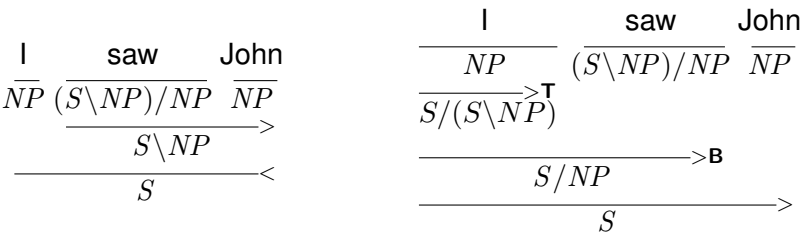


FIGURE 6.1. We will illustrate how the graph-structured stack can parse the short sentence "I saw John", producing these two equivalent derivations.

6.2.1 Splitting

The basic notion of splitting remains the same as the core graph-structured stack implementation. When a shift or reduce operation occurs, the graph-structure stack creates a new head for the CCG category. This results in a non-destructive reduce operation and an efficient shift operation.

For CCG, splitting must be extended to include unary rules such as type-changing and type-raising. In the Zhang and Clark (2011) CCG parser, the unary rules are implemented by an explicit new operation called *unary*. As the graph-structured stack computes and stores the results of all possible operations, the unary operation must be implemented after every possible action.

Whenever a new CCG category is added to the graph-structured stack, whether by a shift or reduce, all possible unary reductions are applied to this new category. Each of the resulting CCG categories are then added as a new head to the structure. This is equivalent to running the *unary* operation on all possible trees.

In the following graph-structured stack examples, we will parse the short sentence "I saw John" as seen in Figure 6.1. In Figure 6.2, only the CCG tag for the word "I", NP , has been shifted on to the graph-structured stack. This tag can be modified by a unary rule, type-raising, to become $S/(S\backslash NP)$. In the

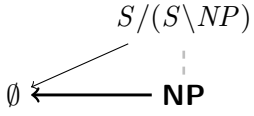


FIGURE 6.2. This is a graph-structured stack (GSS) representing an incomplete parse of the sentence fragment "I" with CCG category NP.

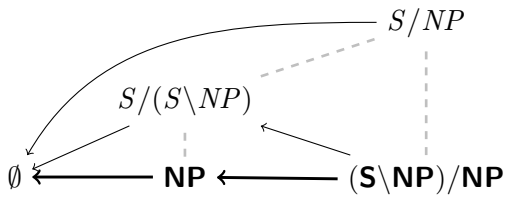


FIGURE 6.3. This is a graph-structured stack (GSS) representing an incomplete parse of the sentence fragment "I saw".

graph-structured stack, splitting occurs and both NP and $S/(S\backslash NP)$ become heads, allowing for non-destructive operations. In a traditional shift-reduce CCG parser, only one of these parsing actions could be represented.

6.2.2 Combining

Combining is equivalent to the *shift* operation in the Zhang and Clark (2011) CCG parser. Whilst shift-reduce dependency parsers only have one possibility for the next node to be added (most commonly the word token itself), in shift-reduce CCG parsing there may be multiple possibilities. These multiple possibilities are dictated by the supertagger. If the supertagger assigns n possible CCG categories to the next word, the CCG parser must select which of those to add to the stack. Thus, the *shift* operation in traditional shift-reduce CCG parsers perform disambiguation. If the disambiguated CCG category is incorrect, the parsing process may fail.

The Zhang and Clark (2011) CCG parser uses beam search to allow for limited ambiguity in these situations. This allows for more than one of the possible CCG categories to be considered as long as the partial derivation remains in the n best-scoring partial derivation list. This improves the probability that the correct CCG category will be shifted, but there is no guarantee this will occur.

The graph-structured stack prevents this issue by processing all possible CCG categories that the supertagger assigns. This is possible as the graph-structured stack can encode this ambiguity efficiently, as opposed to traditional shift-reduce CCG parsers.

In Figure 6.3, we extend the example in Figure 6.2. Incremental parsing continues as we push on the word "saw" with the CCG tag $(S\backslash NP)/NP$. Note that this *combines* with the two previous heads on the stack, NP and $S/(S\backslash NP)$. As such, we only perform one push operation but this impacts multiple stacks.

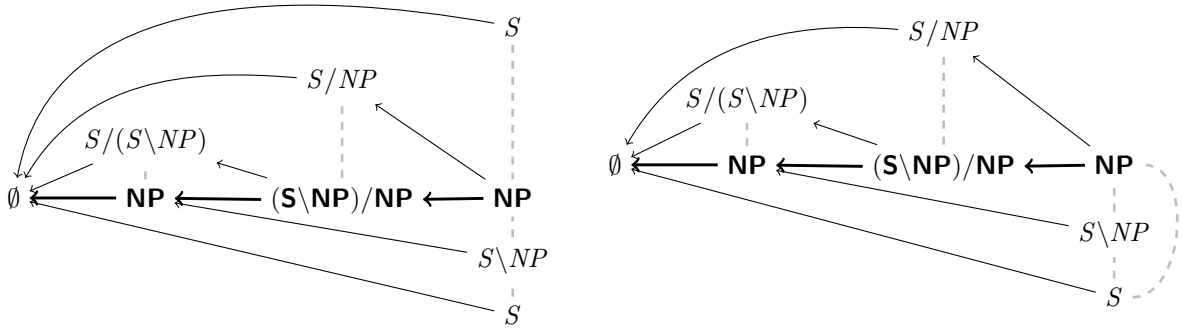


FIGURE 6.4. This is a graph-structured stack (GSS) representing an incomplete parse of the sentence fragment "I saw John". Local ambiguity packing will recognise the two S nodes as semantically equivalent and merge them into a single node.

6.2.3 Local Ambiguity Packing

The core concept of the graph-structured stack is local ambiguity packing. Without it, the tree becomes exponential in size, even with splitting and combining. Local ambiguity packing for Categorial Grammar is only briefly alluded to in Tomita (1988) and is not used in the Zhang and Clark (2011) CCG parser.

To enable CCG parsing using a graph-structured stack, we need to define syntactic equivalence for the CCG categories. Syntactic equivalence occurs when two CCG categories have the same category and will result in the same final derivation.

Checking for equivalence in CCG categories is implemented in the C&C parser as it is necessary for the CKY algorithm. In the C&C parser, two CCG categories are equivalent if they have the same category type, identical heads and identical unfilled dependencies. By ensuring the head and unfilled dependencies are equivalent, the same dependencies will be created in any subsequent parsing. In shift-reduce parsing with the graph-structured stack, we must also check to ensure that the CCG category points to the same previous stack position.

An example of local ambiguity packing can be seen in Figure 6.4. The CCG category S can be generated by two different structures for the sentence "I saw John", one using type-raising and one without. Using local ambiguity packing, these two nodes will be recognised as equivalent and be combined to form a single node. The two derivations shown initially in 6.1 are now represented by the single node S.

6.3 Implementation and Performance Optimisations

Introducing a new parsing algorithm to the C&C parser involves a number of modifications throughout the code-base. The CKY algorithm and GSS-based shift-reduce algorithm should produce almost equivalent output for similar sentences. If the shift-reduce algorithm and graph-structured stack are implemented correctly, accuracy should be near equivalent for both the CKY CCG parser and shift-reduce CCG parser. Thus, accuracy is not an issue if the new algorithm is correctly implemented. The primary issue is the impact on parsing speed this new algorithm has.

As the only parsing algorithm used by the C&C parser has been the CKY algorithm, much of the code-base is tightly integrated with the existing parsing code for efficiency. Whilst the shift-reduce algorithm enables both incremental parsing and an improved feature set (to be discussed in Sections 7 and 8), it is unlikely to be used if parsing speed is substantially reduced. The C&C parser features an myriad of such optimisations in the code-base for the CKY algorithm and only a small number of these could be duplicated for the graph-structure stack implementation. Over time, further improvements are possible, but they are primarily engineering in nature.

6.3.1 High-speed Equivalence Checking

The C&C parser has a highly optimised collection of hash tables to allow for rapid equivalence checking in the cells of the CKY chart.

The same equivalence checking is also used for local ambiguity packing in the graph-structured stack to allow for polynomial time parsing using the shift-reduce algorithm. Unfortunately, the stress on the equivalence hash table substantially larger due to the number of CCG categories that now exist in each of the frontier cells in the graph-structured stack. As seen in Figure 3.9, a graph-structured stack collapses all of the diagonal cells of the CKY chart into a single frontier cell. To improve the performance, engineering tweaks are required such as re-evaluating the optimal size of the hash table. As opposed to standard hash table implementations, it is best for the hash tables in the C&C parser to have extremely low utilisation. The parser is optimising for rapid look-ups instead of low memory footprints, so the tables are sized appropriately in order to avoid the majority of hash collisions.

One advantage of the frontier cells representing large numbers of cells in the CKY chart is that fewer redundant CCG categories are produced. In the CKY chart, two equivalent CCG categories may be produced but are not checked for equivalence until higher up in the chart structure. In the graph-structured stack, these equivalent CCG categories are recognised at each frontier level. As there are only n frontier levels, where n is the number of words, there are fewer places for redundancy. As the C&C parser will only generate a given number of maximum CCG categories before aborting a parse, this results in higher coverage numbers but lower parsing speeds for the shift-reduce parser.

6.3.2 Frontier Equivalence Checking

As part of the equivalence check for each CCG category in the shift-reduce algorithm, the frontiers of both categories are compared. The two CCG categories must have frontiers composed of the same CCG categories to be considered semantically equivalent. Initially each CCG category maintained a list of its relevant nodes in the previous frontier, but this resulted in slow comparisons between CCG categories. By caching these frontier lists and implementing them as a pointer, checking if two frontiers is equivalent is reduced to a single pointer comparison. This caching infrastructure prepares the groundwork for later performance optimisations used in frontier pruning (see Chapter 8).

6.4 Results

As the only thing to change is the core parsing algorithm, both parsers use the same parsing model. The results for the shift-reduce and CKY C&C parsers should be near equivalent. Due to the complexity of the algorithms and code involved, however, slight differences may occur.

During development, the GSS-based shift-reduce parser was tested against the results provided by the base CKY C&C parser. The results from this evaluation are illustrated in Table 6.1 and were used in identifying potential issues with the shift-reduce implementation. Parsing F-score for both labeled and unlabeled dependencies are near equivalent. These results show empirically that the extension of the graph-structured stack to the CCG formalism is practically sound. Major differences in parsing accuracy would indicate an error in implementation.

Finally, the speed of the parsers is significantly different, with the CKY parser outperforming the shift-reduce parser by a considerable margin. Due to the mature state of the CKY algorithm in the C&C parser, the speeds were expected to be significantly different. As the C&C architecture has been tuned with the

Parser	Coverage (%)	Labeled F-score (%)	Unlabeled F-score (%)	Speed (sentences/second)
CKY C&C	99.01	86.37	92.56	55.6
SR C&C	98.90	86.35	92.44	48.6
CKY C&C Auto	98.90	84.30	91.26	56.2
SR C&C Auto	98.85	84.27	91.10	47.5

TABLE 6.1. Comparison of the CKY C&C parser to the SR C&C parser that utilises the graph-structured stack. Auto indicates that automatically assigned part of speech tags were used. All results are against the development dataset, Section 00 of CCGbank, which contains 1,913 sentences.

CKY algorithm in mind, there have been many tweaks and optimisations that are not applicable or have not yet been implemented in the shift-reduce algorithm. Whilst an attempt has been made to reproduce obvious optimisations, reproducing all of them was not feasible in the time available.

Once all changes were tested, final evaluation was performed on Section 23 of CCGbank, illustrated in Table 6.2. Mirroring the results seen on the development dataset, parsing F-score for both labeled and unlabeled dependencies are similar.

In the final evaluation, the shift-reduce parser shows improved coverage over the CKY parser. This is due to more efficient dynamic programming in the graph-structured stack. Complex sentences can produce millions of CCG categories during parsing. To prevent excessive slowdowns, the C&C parser limits the number of CCG categories that can be produced during parsing. As the shift-reduce parser creates fewer CCG categories to parse a given sentence, more complex sentences can be parsed without hitting this limit.

However, this coverage increase comes at the expense of parsing speed however. Higher hash table utilisation and more time spent on complex sentences results in a slower overall parser. The parsing speed of the shift-reduce parser is approximately 34% slower than the finely tuned CKY parser. Further speed comparisons for this shift-reduce CCG parser will be made in Chapter 8, where the new features provided by shift-reduce parsing allow for a novel method of pruning.

It is interesting to note the extreme differences in parsing speed between Section 00 and Section 23 of CCGbank. Although both are of a similar size, the parsing speed (measured in sentences per second) is nearly twice as fast on Section 23. This is due to Section 00 being dominated by a number of large sentences composed of up to a hundred words.

Parser	Coverage (%)	Labeled F-score (%)	Unlabeled F-score (%)	Speed (sentences/second)
CKY C&C	99.34	86.79	92.50	96.3
SR C&C	99.58	86.78	92.41	71.3
CKY C&C Auto	99.25	84.59	91.20	82.0
SR C&C Auto	99.50	84.53	91.09	61.2

TABLE 6.2. Final evaluation of the CKY and SR CCG parsers on Section 23 of CCGbank, containing 2,407 sentences. Auto indicates that automatically assigned part of speech tags were used.

6.5 Summary

Incremental CCG parsing has been previously achieved (Hassan et al., 2008) but this was only possible by converting both CCG and CCGbank to use a deterministic grammar. Whilst this allowed for a ten times increase in parsing speed compared to the C&C parser, parsing accuracy was substantially lower.

This is the first time a graph-structured stack has been implemented in a shift-reduce CCG parser and is the first major contribution of this thesis. We have shown how to apply the graph-structured stack to CCG and have handled any issues caused by the new formalism. By implementing a graph-structured stack, shift-reduce CCG parsers can generate all possible derivations whilst still having comparable speed profiles to CKY parsers. Our results demonstrate that incremental CCG parsing is possible without sacrificing accuracy for speed.

In later chapters, the novel features that the shift-reduce parsing state enables will be shown to substantially improve other tasks such as part-of-speech tagging and CCG supertagging. They will also be shown to be effective as features in a pruning algorithm that allows for increased parsing speeds with little impact on parsing accuracy.

Preliminary results from this chapter comparing the CKY and shift-reduce C&C parser implementations have been accepted for publication under the title *Frontier Pruning for Shift-Reduce* CCG *Parsing* at the Australasian Language Technology Workshop in December 2011.

CHAPTER 7

In-place POS Tagging and Supertagging

In this chapter, the concept of tightly integrating parsing and tagging is explored. Traditionally, there is little interaction between components in the parsing pipeline. In the C&C parser, the only interaction occurs when the parser requests more supertags from the supertagger. This is quite limited in scope but has been used to improve parsing speed (Kummerfeld et al., 2010).

Due to the incremental nature of the shift-reduce algorithm, we can extract features from the parser state and use them as inputs to the tagging processes. By providing a partial understanding of the sentence so far, both the POS tagger and supertagger can make better informed and less ambiguous decisions. This cycle continues after each new word is introduced to the incremental parser.

7.1 Motivation

Tagging is the process of assigning a specific category to a word in a corpus by analysing the word's context. A given word may be assigned different tags depending on the word's function in the sentence, leading to ambiguity in tagging a word. In most taggers, this ambiguity has been clarified by looking at adjacent words in the sentence. As the tagger can only access surface features of the text, it lacks a deeper understanding of how the sentence might be structured. This leads to sub-optimal tagging decisions.

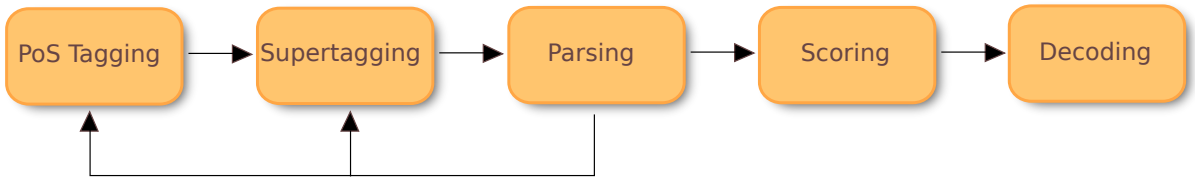


FIGURE 7.1. As opposed to the traditional parsing pipeline used for CCG parsers (Figure 3.10), parsing is used as input to both the POS tagging and supertagging.

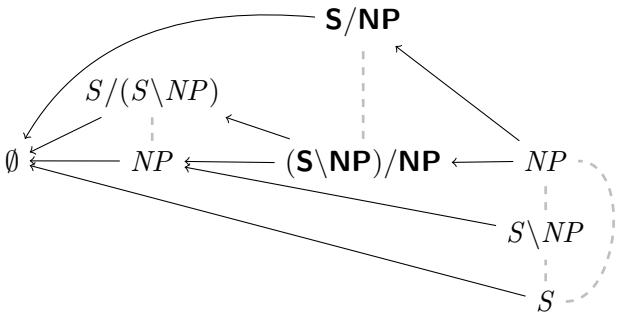


FIGURE 7.2. In this graph-structured stack representing the parse for "I saw John", the second frontier is represented in bold.

We propose using incremental parsing to assist in these tagging decisions. This results in a parsing pipeline that adds integration between the parser and the taggers, illustrated in Figure 7.1.

Many forms of tagging would benefit from understanding of how the sentence might be parsed. As we have developed the incremental CCG parser in Chapter 6, it is now possible for the tagging process to receive an update of how the sentence is being parsed before the next word is considered. This is possible as shift-reduce parsing can process $n-1$ words in the sentence before having to consider the n^{th} word. For the n^{th} word, both the POS tagger and the supertagger could use features derived from parsing the previous $n-1$ words.

The novel features provided by incremental parsing are called *frontier features* and are a representation of the current state of the parser, containing all possible CCG categories that can be created by the words processed so far.

In the graph-structured stack shown in Figure 7.2, the frontier for the verb "saw" is represented in bold. The frontier contains two categories for "saw". One is the category supplied by the supertagger, $(S\backslash NP)/NP$, and has not yet been combined with anything. The other category, S/NP, has been generated by forward composition and has already identified "I" as the subject of "saw". Similar to how humans incrementally parse sentences, an incremental parser expects certain constituents in the upcoming input. Note that both the categories are looking for an NP on the right. This suggests that the upcoming CCG category will either be a NP itself or interact with or produce an NP, such as a determiner (NP/N), an adjective (N/N) or a noun (N) that will later be type changed to a noun phrase (NP). Thus, analysing the frontier can result in less ambiguous and more accurate tagging decisions for both the POS tagger and supertagger.

The addition of frontier features impacts a substantial portion of the parsing pipeline. The incremental parser produces frontier features, representing the understanding of the sentence so far. The POS tagger uses these frontier features to improve POS tagging accuracy. The supertagger then uses both the improved POS tags and the frontier features to improve supertagging accuracy. Finally, the incremental parser uses the POS tags and supertags from the two previous steps to base future decisions on.

This concept is novel and has not yet been attempted in the literature. We believe that this will lead to higher accuracy across the entire parsing pipeline and is an example of the advanced features possible from the incremental parser introduced in Chapter 6.

7.2 Features

This work extends the maximum-entropy taggers described in Curran and Clark (2003). The base features used by the original taggers will be described in addition to the novel frontier features introduced by incremental parsing.

7.2.1 Base Features

The contextual predicates used to generate the features for both the POS tagger and supertagger are illustrated in Table 7.1. If the contextual predicate returns true and the contextual predicate's condition holds, then the relevant feature is added.

These features include the raw text of surrounding words and the previous two assigned tags. The supertagger additionally uses the tags assigned during POS tagging as this has shown to improve the performance of supertagging and results in lower supertag ambiguity (Clark and Curran, 2004b). Finally, for the POS tagger, additional features are activated when a word appears less than five times in the training corpus ($freq(w_i) < 5$). This assists in disambiguating rare words by analysing their orthographic properties.

7.2.2 Frontier Features

As seen in Figure 7.2, frontier features represent the current understanding of the sentence by the parser. They can be used to assist in the tagging process by suggesting what the function upcoming words will serve.

Condition	Contextual predicate
$freq(w_i) \geq 5$	$w_i = X$
$freq(w_i) < 5$ (POS tagger)	X is prefix of w_i, $\|X\| \leq 4$ X is suffix of w_i, $\|X\| \leq 4$ w_i contains a digit w_i contains uppercase char w_i contains a hyphen
$\forall w_i$	$t_{i-1} = X$ $t_{i-2}t_{i-1} = XY$ $w_{i-1} = X$ $w_{i-2} = X$ $w_{i+1} = X$ $w_{i+2} = X$
$\forall w_i$ (supertagger)	$\text{POS}_i = X$ $\text{POS}_{i-1} = X$ $\text{POS}_{i-2} = X$ $\text{POS}_{i+1} = X$ $\text{POS}_{i+2} = X$

TABLE 7.1. Conditions and contextual predicates used in the POS tagger and supertagger for generating features.

The training code of both the POS tagger and supertagger had to be extended to allow the consideration of frontier features. To enable debugging and improve human readability, the frontier features are encoded as a string representation of the CCG category. The frontier in Figure 7.2 would add the new features $(S\backslash NP)/NP$ and S/NP for example.

7.3 Training

The training data for the POS tagging and supertagging was acquired from Sections 02-21 of CCGbank. Preprocessing of the training data was necessary as the novel frontier features are generated by the parser itself. To generate the frontier features, sentences were provided with gold standard POS and super tags to the shift-reduce CCG parser. These sentences were then parsed and an output file was created that contained the frontier that the parser had constructed for each word.

During preprocessing of the data, 3,189 sentences had to be discarded from the 39,604 sentences of Sections 02-21, resulting in 91.95% coverage of the initial training data. This was due to the parser being unable to reach a spanning analysis due to CCGbank using CCG rules or categories that are incompatible with the CCG representation implemented in the C&C parser. Note that this low coverage only impacts

the training phase of the parsing process, which has strict requirements as to what sentences make good training data, and will not decrease the coverage of the tagger or parser in normal operation.

Finally, it is important to note that we are using gold supertags for generating the frontier lists. This is similar to using gold POS tags in parsing. It is possible accuracy will decrease when in-place POS and supertagging is directly integrated into the parsing pipeline. This work does not contain numbers for automatically assigned frontier features, but in future work the evaluation methodology will be similar to POS tags. Two separate results, one for gold frontier features and another for automatically assigned (auto) frontier features, would be reported.

7.4 Results

For our experiments, we used the development section of CCGbank, Section 00, as our initial evaluation of the in-place POS tagging and supertagging implementation.

Tagging accuracy has been split into two parts: single tagging and multi tagging. For POS tagging, only the single tagging result is given as multi tagging is rarely used in practice.

7.4.1 Single Tagging

For single tagging, the Viterbi algorithm is used to select the sequence of tags. The Viterbi algorithm is a dynamic programming algorithm that finds the most likely sequence of tags, given that certain tags are more likely to transition to other tags. An example of this is in POS tagging, where a noun is likely to follow a determiner, such as “the cat”.

The POS tagging results in Table 7.2 show improvements in POS tagging accuracy when frontier features are added. Whilst per token accuracy increases by 0.21% this is still a substantial contribution considering the high baseline. Most importantly, per sentence accuracy increased by 3.39%. This result has a number of implications. Frontier features, although assisting in per token errors, were most effective in ensuring an entire sentence was correct. This shows that the features incremental parsing provides allows the POS tagger an improved understanding of the sentence structure.

For supertagging, the evaluation becomes more complicated. As the supertagging process uses POS tags as features, the errors produced by the POS tagger referenced in Table 7.2 have an impact on the

Model	Accuracy (tags)	Accuracy (sentences)
Standard	96.80%	51.76%
Frontier	**97.01**%	**55.16**%

TABLE 7.2. POS tagging accuracy on Section 00 of the Penn Treebank, calculated across both individual tags and entire sentences. Training data for POS tagger was Sections 02-21.

accuracy. To explore the impact the improved POS tagging model has on supertagging, we report three different numbers

- Gold POS tags
- Frontier POS tags, which uses the Frontier POS model with improved accuracy from Table 7.2
- Standard POS tags, which uses the standard POS model from Table 7.2

Gold POS tags provide an oracle score, or the theoretical maximum supertagging accuracy we could attain with a perfect POS tagger. The frontier and standard POS models provide a better indication of real world performance. By comparing the three models, we can identify the effect errors in POS tagging have on supertagging accuracy.

In Table 7.3, the evaluation for supertagging is provided. With gold POS tags, the contribution of frontier features can be measured independently. As opposed to POS tagging, frontier features lead to a substantial improvement in both per token and per sentence accuracy. On a per token basis, the frontier features improve supertagging accuracy by 1.30% over the standard supertagging model. On a per sentence basis, the improvement increases to 1.92%. This is not unexpected as supertagging is "almost parsing" and hence would be expected to benefit from being provided partial sentence structure by the incremental parser.

As gold POS tags are the oracle, it is interesting to examine how automatically assigned POS tags impact accuracy. Two POS models are provided: the standard model and the frontier model from Table 7.2.

Even without adding frontier features to the supertagger, supertagging accuracy is improved by using the POS model using frontier features. This can be seen by comparing the standard supertagging model with and without the frontier POS tagging input. Per token accuracy increases from 92.25% to 92.94% (+0.69%) and per sentence accuracy increases 40.25% to 41.92% (+1.67%). This shows that even a small decrease in the number of errors from the previous components in the parsing pipeline can have a substantial impact on future accuracy.

MODEL	ACCURACY (TAGS)	ACCURACY (SENTENCES)
Standard (gold POS)	93.31%	43.81%
Frontier (gold POS)	**94.61**%	**45.74**%
Standard (frontier POS)	92.94%	41.92%
Frontier (frontier POS)	**94.25**%	**43.91**%
Standard (standard POS)	92.25%	40.25%
Frontier (standard POS)	**93.67**%	**42.34**%

TABLE 7.3. Supertagging accuracy on Section 00 of CCGbank, calculated across both individual tags and entire sentences. Frontier and standard POS use the POS tags from the experiment in Table 7.2.

7.4.2 Multi Tagging

The purpose of multi tagging is different to that of single tagging. Multi tagging aims to select any tags which are within a given fraction β of the most probable tag. The Viterbi algorithm used for single tagging can not be extended to select the top n tags per state. To select these most likely tags per state, the forward-backward algorithm is used. The forward-backward algorithm can be used to find the most likely set of states at any point in time, but cannot be used to find the most likely sequence of states. As such, it is less accurate on a sentence level than the Viterbi algorithm, but provides good results for selecting multiple tags.

In Table 7.4, we explore the impact that frontier features have on multi tagging. The results are separated into three different accuracy measures: one across gold POS tags, another over POS tags produced by the frontier POS model and the final across POS tags produced by the standard POS model. The gold POS

FEATURES	β	k	CATS/ WORD	ACC (GOLD)	SENT ACC	CATS/ WORD	ACC (FRTR)	SENT ACC	CATS/ WORD	ACC (STD)	SENT ACC
Standard	0.075	20	1.25	97.73	69.84	1.26	97.33	65.82	1.26	96.84	62.70
	0.030	20	1.41	98.26	75.34	1.41	97.89	71.37	1.41	97.46	68.14
	0.010	20	1.68	98.69	80.50	1.68	98.38	76.53	1.69	98.02	73.58
	0.005	20	1.94	98.84	82.09	1.94	98.54	78.12	1.94	98.22	75.11
	0.001	150	3.46	99.36	89.46	3.46	99.08	85.03	3.47	98.86	82.54
Frontier	0.075	20	1.19	98.04	71.03	1.20	97.65	66.78	1.20	97.30	64.06
	0.030	20	1.32	98.47	76.64	1.32	98.09	72.39	1.33	97.74	69.56
	0.010	20	1.55	98.79	80.90	1.55	98.48	76.93	1.56	98.19	74.43
	0.005	20	1.76	98.90	82.31	1.76	98.60	78.57	1.77	98.38	76.47
	0.001	150	2.96	99.39	89.51	2.96	99.12	85.37	3.00	98.95	83.28

TABLE 7.4. Multi tagger ambiguity and accuracy on the development section, Section 00.

tags provide an optimistic upper-bound on parsing accuracy. As our focus is on real world application, we will primarily discuss the supertagging accuracy attained using the frontier POS tags.

For the same beta (β) levels, the frontier multi tagging model has higher accuracy and lower ambiguity than the standard multi tagging model currently used by the C&C parser. For example, at $\beta = 0.0075$ we find that the frontier model achieves a per token accuracy of 97.33% whilst the standard model achieves an accuracy of 97.65%. This is achieved with less ambiguity by the frontier model, however, as can be seen by the smaller number of categories assigned per word compared to the standard model (1.20 by the frontier model compared to 1.26 by the standard model).

Most impressively, for an extremely low beta level of 0.001, the ambiguity is decreased from 3.46 categories per word to 2.96 categories per word for nearly equivalent accuracy. As parsing speed is directly related to the number of CCG categories assigned per word, this suggests that improved frontier pruning would substantially improve the parsing speed of the C&C parser without an impact on parsing accuracy.

7.5 Final Results

As the results in the previous section were attained whilst improving the in-place POS tagging and supertagging model, it is possible that over-fitting occurred. For this reason, a final evaluation is performed against a data set that has not previously been used for the purposes of development. All experiments in this section will be on the final evaluation section of CCGbank, Section 23.

7.5.1 Single Tagging

In Table 7.5, the final evaluation for POS tagging accuracy is shown. The frontier model is shown to improve per token POS tagging accuracy by 0.28%, in line with the 0.21% improvement seen during evaluation. For sentence level POS tagging, the frontier model is shown to improve accuracy by 3.01%. This is a small decrease (3.39% improvement down to 3.01%) from the improvement seen in the development evaluation, but again suggests that the small per token accuracy tagging increase achieves substantial improvements on a per sentence level.

The final single tag evaluation for supertagging is provided in Table 7.6. As before, the gold POS tags allow for evaluating the impact that the frontier features alone have on the supertagging model. In the

MODEL	ACCURACY (TAGS)	ACCURACY (SENTENCES)
Standard	97.20%	57.86%
Frontier	**97.48**%	**60.87**%

TABLE 7.5. POS tagging accuracy on Section 23 of the Penn Treebank, calculated across both individual tags and entire sentences. Training data for POS tagger was Sections 02-21.

MODEL	ACCURACY (TAGS)	ACCURACY (SENTENCES)
Standard (gold POS)	93.72%	46.28%
Frontier (gold POS)	**95.00**%	**47.65**%
Standard (frontier POS)	93.14%	44.12%
Frontier (frontier POS)	**94.41**%	**45.12**%
Standard (standard POS)	92.53%	42.63%
Frontier (standard POS)	**93.88**%	**43.50**%

TABLE 7.6. Supertagging accuracy on Section 23 of CCGbank, calculated across both individual tags and entire sentences.

final evaluation, frontier features by themselves are shown to improve per token accuracy by 1.28% compared to 1.30% in development. This suggests a low variation in per token supertagging accuracy between data sets. Sentence level accuracy does not experience this low variation, however. Frontier features improve sentence level accuracy by 1.37% compared to 1.92% in development. This suggests that sentence level accuracy is closely tied to the complexity and length of the sentences processed. Further investigations into the impact that sentence complexity and length have are planned.

This final single tag evaluation convincingly shows that POS tagging and supertagging with the addition of frontier features results in substantially improved accuracy on both a per token and per sentence basis.

7.5.2 Multi Tagging

The final evaluation for multi tagging, seen in Table 7.7, mirrors the results seen during the development evaluation. For the same beta levels, the frontier multi tagging model outperforms the standard supertagging model across all metrics. Most importantly, the accuracy is equivalent or higher for a lower level of tag ambiguity. For the lowest beta level (0.001), ambiguity is decreased from 3.40 categories per word to 2.91 categories per word. This shows little variance to the development data set, where the ambiguity is decreased from 3.46 categories per word to 2.96 categories per word.

FEATURES	β	k	CATS/ WORD	ACC (GOLD)	SENT ACC	CATS/ WORD	ACC (FRTR)	SENT ACC	CATS/ WORD	ACC (STD)	SENT ACC
Standard	0.075	20	1.25	97.88	71.72	1.25	97.40	67.67	1.26	96.86	63.85
	0.030	20	1.41	98.41	77.80	1.41	98.02	74.11	1.56	97.53	70.46
	0.010	20	1.68	98.77	81.04	1.69	98.42	77.53	1.70	98.02	74.43
	0.005	20	1.93	98.90	82.40	1.95	98.61	79.42	1.95	98.28	76.45
	0.001	150	3.40	99.43	90.14	3.43	99.18	86.72	3.44	98.94	83.57
Frontier	0.075	20	1.19	98.12	72.04	1.19	97.73	68.44	1.20	97.28	64.84
	0.030	20	1.31	98.53	77.67	1.32	98.20	73.89	1.33	97.82	70.46
	0.010	20	1.54	98.87	81.27	1.55	98.58	77.76	1.56	98.27	75.01
	0.005	20	1.75	98.98	82.76	1.76	98.74	79.84	1.77	98.49	77.58
	0.001	150	2.91	99.47	90.63	2.94	99.23	87.03	2.97	99.06	85.05

TABLE 7.7. Multi tagger ambiguity and accuracy on Section 23.

This definitively shows that frontier features provide useful information for the purpose of supertagging. By producing the equivalent or higher accuracy for a lower level of tag ambiguity, parsing speed and accuracy will be substantially improved.

7.6 Summary

In this chapter, we have extended the C&C POS tagger and supertagger with novel frontier features, motivated by the capabilities of the shift-reduce CCG parser developed in Chapter 6. Due to the incremental nature of the shift-reduce CCG parser, it is now possible for the tagging and parsing process to interact by passing features from the parser back into the tagger.

The features generated from this interaction improve accuracy and ambiguity levels for both POS tagging and supertagging on Sections 00 and 23 in CCGbank substantially. Whilst POS tagging shows only a 0.21% variation, there is a 3.39% jump in sentence level accuracy. This indicates that when the frontier features are used, they help define and improve the structure of the sentence. This shows that even a small decrease in the number of errors from the previous components in the parsing pipeline can have a substantial impact on future accuracy.

These same features could be used in other components to improve accuracy and decrease ambiguity. In the next chapter, we explore whether the frontier features used to improve supertagging accuracy can also be used as the basis of a novel method of pruning.

CHAPTER 8

Frontier Pruning

The purpose of frontier pruning is to cut down the search space of the parser by only considering partial derivations that are likely to form part of the highest-scoring derivation. Like adaptive supertagging, it exploits the idea that the only partial derivations the parser needs to generate are those used by the highest-scoring derivation. The model is trained using the parser's initial unpruned output and aims to distinguish between partial derivations that are necessary and those that are not. By eliminating a large number of those unnecessary partial derivations, parsing ambiguity is significantly decreased.

This approach is similar to beam search as frontier pruning removes partial derivations once it is likely they will not be used in the highest-scoring derivation. For certain instances, such as n-best re-ranking, beam search would be preferred as derivations without the highest score are still useful in the parsing process. For one best parsing, however, the parser may waste time generating these additional derivations when it could be known in advanced that they will not be used. This could occur during attachment ambiguity where, although the parser is guaranteed to select one attachment, the other attachment may be constructed as it is valid and still competitive when considered by beam search's criteria.

8.1 Pruning Overview

In this chapter, we introduce a novel form of pruning based upon classifier-based pruning. As previously mentioned, there are millions of possible derivations that a parser needs to explore. Our method of pruning is based upon a binary classifier. After each possible step in the parser, the binary classifier is run. The binary classifier evaluates the current branch of the search. If the branch appears promising, parsing is allowed to continue. Otherwise, parsing on this branch is aborted and other potential branches are explored.

If the binary classifier has low recall, then it is likely that the correct derivation will be removed accidentally. If the binary classifier has low precision, then parsing reverts to traditional parsing as all nodes are considered. Parsing speed is likely to suffer in this case, however, as pruning would require computational overhead but result in no speed gains.

As the C&C parser has not operated using this binary classifier based pruning previously, it was necessary to implement the binary perceptron classifier and explore the features that could be used to guide classifications.

8.2 Training and Processing

We train a binary averaged perceptron model (Collins, 2002) on parser output generated by the SR C&C parser using the standard parsing model. Once the base parser has successfully processed a sentence, all partial derivations that lead to the highest-scoring derivation are marked. For each partial derivation in the GSS, the perceptron model attempts to classify whether it was part of the marked set. If the classification is incorrect, the perceptron model updates the weights appropriately.

During processing, pruning occurs as each frontier is developed. For each partial derivation, the perceptron model classifies whether the partial derivation is likely to be used in the highest-scoring derivation. If not, the partial derivation is removed from the frontier, eliminating any paths that the partial derivation would have generated. Perfect frontier pruning would allow only a single derivation, specifically the highest-scoring one, to develop.

8.2.1 Potential Gains

To establish bounds on the potential search space reduction, the size of the marked set compared to the total tree size was tracked over all sentences in the training data. This represents the size of the tree after optimal pruning occurs. Two figures are presented, one with gold supertags and the other with supertags applied by the C&C supertagger. Gold supertags only provide one CCG category per word and represents an optimal supertagger with nearly no ambiguity.

As can be seen in Table 8.1, the size of the marked set is 10 times smaller for gold supertags and 15 times smaller for automatically supplied supertags. This places an upper-bound on the potential speed improvement the parser may see due to aggressive frontier pruning assuming frontier pruning adds no

PRUNING TYPE	REDUCED SIZE
Average pruned size (gold supertags)	9.6%
Average pruned size	6.7%

TABLE 8.1. Recall of the marked set from the frontier pruning algorithm across all trees and the size of the pruned tree compared to the original tree.

overhead. Note that the core parsing algorithm is only one step of many in the parser pipeline. Other components, such as POS tagging, CCG supertagging, scoring and decoding together take substantial time as well.

8.3 Improved Features for Frontier Pruning

For frontier pruning to be effective, the model must be able to accurately distinguish between partial derivations that will be used in the highest-scoring derivation and those that shall not. As the features of the C&C parser dictate the highest-scoring derivation, the features used for frontier pruning have been chosen to be similar. For a full description of the features used in the C&C parser, refer to Clark and Curran (2007c).

Each partial derivation is given a base set of features derived from the current category. The initial features include a NULL which all categories receive, the CCG category itself and whether the category was assigned by the supertagger. There are also features that encode rule instantiation, including whether

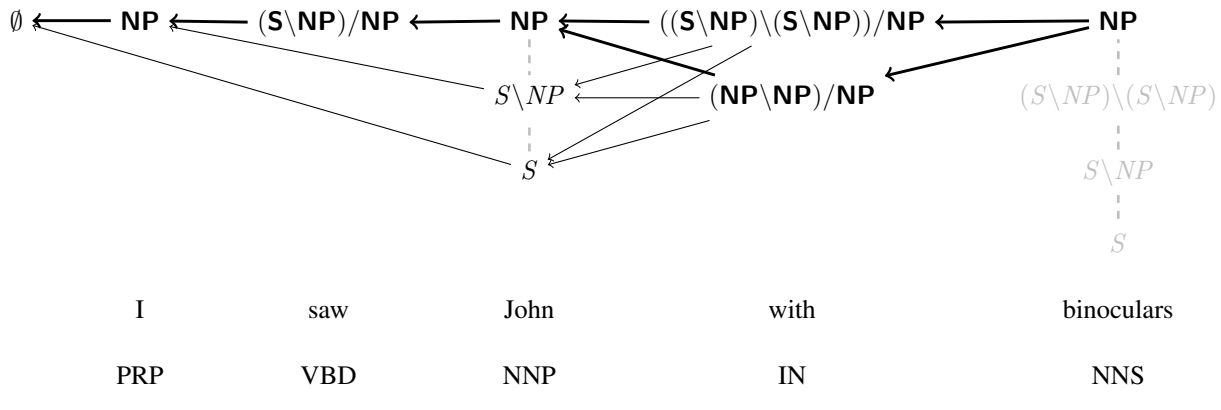


FIGURE 8.1. A graph-structured stack (GSS) representing an incomplete parse of the sentence "I saw John with binoculars". The nodes and lines in bold were provided by the supertagger, whilst the non-bold nodes and lines have been created during parsing. The light gray lines represent what reduce operation created that lexical category.

Feature Type	Example
Category	$S\backslash NP$
Binary Composition	$(S\backslash NP)/NP$ and NP
Forward Application	True
Head Word	*saw*
Head POS	VBD
Previous Frontier	NP
Next Frontier	$((S\backslash NP)\backslash(S\backslash NP))/NP$
Next Frontier	$(NP\backslash NP)/NP$

TABLE 8.2. Example features extracted from $S\backslash NP$ in the third frontier of Figure 8.1. For the frontier features, bold represents the highest-scoring feature selected for contribution to the classification decision.

the category was created by type raising, a lexical rule, or any CCG combinatory rule. If the category was created by a CCG combinatory rule, the type of rule (such as forward/backward application and so on) is included as a feature. These features allow for a naïve representation of the local composition of the CCG tree and allow the machine learning algorithm to generalise over combinatory rules and categories.

Features representing the past decisions the parser has made are also included. Note that *current* represents the current category and *left*/*right* is the current category's left or right child respectively. For categories created by unary rules, a tuple of [current,current→left] is included as a feature. For categories created by binary rules, a tuple of [current→left, current, current→right] is included. If a category is a leaf, then two features [current, word] and [current, POS] are included. Features representing the root category of the partial derivation are also included, encoding the category head's word and POS tag.

Finally, additional features are added that represent the possible future parsing decisions. This is achieved by adding information about the remaining partial derivations on the stack (the past frontier) and the future incoming partial derivations (the next frontier). These do not exist in the C&C parser and are only possible due to the implementation of the GSS. For each category in the previous frontier, a feature is added of the type [previous, current]. For the next frontier, which is only composed of supertags at this point, the feature is [current, next]. These features allow the pruning classifier to determine whether the current category is likely to be active in any other reductions in future parsing work. As we only want to score the optimal path using the previous and next features, only the highest weighted of these features are selected. The rest of the previous and next features are discarded and do not contribute to the classification.

An example of this can be seen in Table 8.2, where the features for the partial derivation of $S\backslash NP$ is enumerated from the context of Figure 8.1.

These features differ to the traditional features used by shift-reduce parsers due to the addition of the GSS. As traditional shift-reduce parsing only considers a single derivation at a time, it is trivial to include history further back than the current category's previous frontier. As GSS-based shift-reduce parsing encodes an exponential number of states, however, the overhead of unpacking these states into a feature representation is substantial. Our approximation of selecting the highest weighted previous and next frontier features approximates the non-deterministic shift-reduce solution.

8.4 Balancing Pruning Features and Speed

For frontier pruning to produce a speed gain, enough of the search space must be pruned in order to not just compensate for the additional computational overhead of the pruning step itself but also reduce ambiguity in the parsing process. This is a challenge as the C&C parser is written in *C++* with a focus on efficiency and already features substantial lexical pruning due to the use of supertagging.

For this reason, there were instances where expressive features needed to be traded for simpler features in the frontier pruning process. Whilst these simpler features may not prune as effectively, their computational overhead would offset the speed gains associated with the reduced search space. The complexity of the frontier pruning features may be dictated by the speed of the core parser itself, with more expressive features being possible if the core parser is slower.

The implementation of these features also had to focus on efficiency. To decrease hash table stress and improve memory locality of the hash table storing the feature weights, only a subset of features were stored. This feature subset was obtained from the gold standard training data as it contains far less ambiguity than the same training data which uses the larger set of lexical categories supplied by the supertagger.

Hash tables were used for storing the relevant feature weights. Simple hash based feature representation were used for associating features with weights to reduce the complexity of equivalence checking. The hash values of features that were to be reused were also cached to prevent recalculation, substantially decreasing the computational overhead of feature calculation.

8.4.1 Hash-Based Feature Representations

The speed of frontier pruning is highly related to the overhead of generating the features of the current parse context. The first version of the frontier pruning algorithm used features based upon strings. For a prototype, these string features were human readable and assisted with both debugging and logic checking. Even with aggressive caching, however, these string based features caused frontier pruning to result in a speed loss.

One possibility to decrease the computational overhead of feature calculation was to use the hash values of features as the features themselves. During parsing, these hashes have already been generated by the parser for checking whether two objects are equivalent. Re-using these hashes only results in a memory look-up rather than any expensive calculations.

Using the feature hashes has been considered previously in the literature in two variations, termed feature mixing and hash kernels. Feature mixing does not handle collisions and has no theoretical guarantees on the error it produces. Hash kernels provide both a theoretical guarantee on the distortions introduced and provide a method to reduce these distortions through using multiple hashing.

8.4.1.1 Feature Mixing

Feature mixing projects a high dimensional feature vector into a lower dimensional feature hash. The original feature vector is discarded and the feature weight is stored at the relevant feature hash location in a hash table. As the size of the hash table can be user specified, this allows for a trade-off between memory use and application accuracy. Feature mixing allow expressive statistical NLP systems on resource constrained devices without a substantial loss in accuracy. Whilst this results in collisions, the memory savings can be immense, especially if the memory use of the features themselves are large.

Collisions in feature mixing are handled by the machine learning algorithm. If feature f_a and feature f_b collide in a specific position n of the hash table, the feature weight at n is representative of $f_a \vee f_b$. If the undistorted feature weights of f_a or f_b differ substantially (i.e. $w(f_a) \gg w(f_b)$) then such a collision may have a large impact on accuracy.

Even with this substantial drawback, feature mixing has been used successfully in a number of applications. In Ganchev and Dredze (2008), four machine learning algorithms are tested on four different NLP

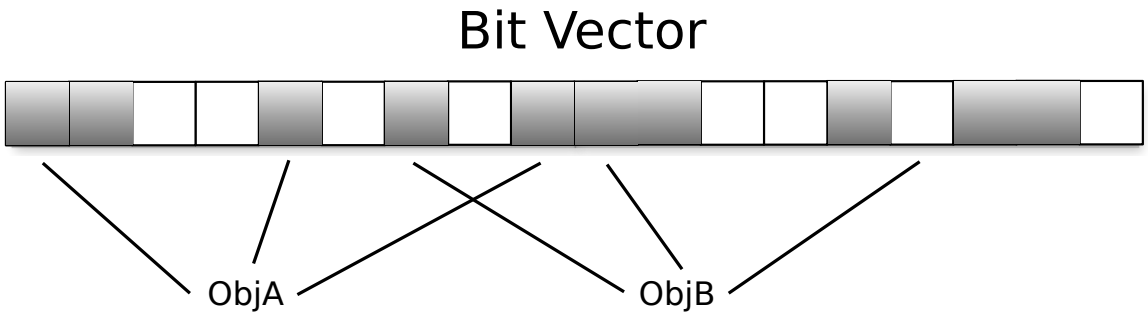


FIGURE 8.2. An example of a Bloom filter with $k = 3$. Object A is considered in the Bloom filter whilst Object B is not. Note that there is a possibility Object A is a false positive. This is the basis of hash kernels.

problem domains. They show that the memory use of the machine learning model can be reduced by 70% without a significant loss in performance.

8.4.1.2 Hash Kernels

Hash kernels (Shi et al., 2009; Weinberger et al., 2009) are an extension of the feature mixing method that allow for theoretical bounds on the impact that dimensionality reduction will have on the feature weights. This is achieved through multiple hashing which alleviates the problems caused by collisions, as seen in feature mixing.

The technique of multiple hashing in hash kernels is similar to a Bloom filter. A Bloom filter is a space-efficient probabilistic set data-structure that does not allow object removal. The data-structure is based around a single large bit-vector, with all bits initially set to an inactive state. To insert an object into a Bloom filter, an object is hashed k times and each of the k relevant Bloom filter locations are activated. To check if an object is contained in a Bloom filter, each of the k relevant Bloom filter locations are checked to ensure they are active. If any one location is inactive, the object was never inserted into the Bloom filter. Note that this may return false positives, but not false negatives, and the more objects inserted into the Bloom filter the higher the false positive rate. An example of this can be seen in Figure 8.2.

This same technique is used by hash kernels and is an extension on the feature mixing method. The hash kernel takes a high dimensional feature vector and converts it into multiple lower dimensional feature hashes. Whilst the number of collisions is increased, each feature now has multiple feature weights

that it can modify. This allows for a better approximation of the original undistorted feature weight by varying each feature weight by a smaller amount than single hashing would require.

Using this method, feature spaces previously inconceivable have been processed. Weinberger et al. (2009) train a model for email spam prediction with data obtained from Yahoo. For features, each word obtained from the emails of the 400,000 test users were split into two features – the basic word $word$ and a user-tagged word $word_{user}$ that allows for personalised spam filtering. As the email corpus contained a vocabulary of 40 million unique tokens, the feature space would explode to 16 trillion possible personalised features, which is far too many for traditional machine learning models. Using hash kernels, however, this large set of features can be processed and they show an error rate reduction of 30%. The error rate reduction is possible only due to the immense feature space and is processed with a finite memory guarantee due to the use of hash kernels.

For high-speed classification applications, there is a major drawback to hash kernels. For each feature you traditionally retrieve a single feature weight, incurring a single memory look-up. With the hash kernel method, each feature incurs k memory look-ups for the k feature weights that are required. In practice, this can slow down the classification process substantially.

8.4.1.3 Feature Hashing Implementation in Frontier Pruning

For our case, the overhead of calculating k hashes prevented the efficient use of hash kernels. As the parser only calculates and stores one hash, each object would require $k-1$ additional hashes. Calculating additional hashes would remove any advantage of using the stored hash. Indeed, whilst the frontier pruning feature space is large, it is not large enough to warrant the additional code complexity required by the hash kernel.

For this reason, a modified version of feature mixing was employed. Feature mixing aims to allow for expressive statistical models to be used on resource constrained devices. The aim of feature mixing in our case is different, focusing on preventing the overhead of additional feature generation and improving the efficiency of the classification process.

Feature mixing in our case is represented by either a unary hash or a binary hash. All features, both unary and binary, are represented by two 64 bit hashes. The hashing function is computationally minimal and involves one shift and two xors for each feature. For the feature f composed of 64 bit unsigned numbers f_a and f_b, the hash function is:

$$h(\{f_a, f_b\}) = (f_a \oplus f_a >> 32) \oplus f_b$$

The shift is required to differentiate $h(\{f_a, f_b\})$ from $h(\{f_b, f_a\})$.

A unary hash represents a single piece of information combined with a feature type. Imagine representing the unary feature that stated the current node was a CCG category with the label NP/N. First, the hash of the CCG category NP/N would be taken, producing (for the sake of the example) *0x9F8E*. Second, the arbitrarily specified feature identifier hash for "unary ccg category" is retrieved, producing *0xCCA0*. This would produce the feature $\{0xCCA0, 0x9F8E\}$ which could then be used in the hashing function defined above. The arbitrarily specified feature identifier hash is necessary as it allows multiple features of the form $\{\_, 0x9F8E\}$.

A binary hash represents two pieces of information combined with the feature type. Imagine we were creating the binary feature stating the current node was created by combining $(S\backslash NP)/NP$ and NP. Again, we retrieve the hash for these two CCG categories, with $(S\backslash NP)/NP$ = *0x4473* and NP = *0x8018*. This produces the non-unique feature $\{0x4473, 0x8018\}$. To label the feature with the feature type, we retrieve the arbitrarily specified feature identifier hash for "binary combinatory forming" which is *0xF6A2*. Finally, we combine this with the first piece of information through xoring. This produces the unique feature $\{(0x4473 \oplus 0xF6A2), 0x8018\}$.

The core hash table implementation used is the Google Sparsehash library. The Google Sparsehash library contains hash table implementations optimised for both space and speed. In our work, only the hash table implementation optimised for speed was used. This comes at the price of a higher memory overhead, but for the C&C parser trading memory use for speed is a reasonable decision.

8.4.2 Memoisation of Frontiers

Many of the binary feature operations involve re-using hashes from the previous and next frontiers. Features of the style $h(\{f_{const}, f_{current}\})$ are constantly recreated although f_{const} never changes. To minimize repeated work, once a previous or next frontier features set has been computed, they are cached in the style $h(\{f_{const}, \_\})$ When the frontier feature list is to be used with a new $f_{current}$, the second value of the cached feature is replaced with $f_{current}$. This ensures fast look-up and no redundant computations. Hence, a frontier is only traversed once to create the hash features that are required.

8.4.3 Restricting Feature Set Size

As part of frontier pruning, it is important to keep the feature set size as small as possible for two primary reasons. First, the feature set size is related to the number of possible collisions that will occur due to feature mixing. Second, the fewer feature weights that need to be stored, the more efficient feature look-up will be in the hash table.

To minimize the size of the feature set, only the features seen in the gold supertag training data were used. By providing gold supertags, both the features and the search space are substantially smaller than in full parsing. This is due to the lower ambiguity levels provided by gold supertags compared to the multiple supertags supplied by the supertagger. As can be seen in Table 8.1, even given gold supertags the parser still generates many CCG categories that will never be needed in the final derivation.

8.5 Improving Recall in Frontier Pruning

Compared to the unmarked set, the marked set of partial derivations used to create the highest-scoring derivation is small. Due to how strict the CCG formalism is, the recall of this marked set must be 100%. If a single CCG category from the marked set is pruned accidentally, the accuracy will be negatively impacted. The loss of a single category may even mean it is impossible to form a spanning analysis.

To prevent this loss of accuracy and coverage, the recall of the marked set must be improved whilst still pruning as many unnecessary categories from the parsing process as possible. We explore modifying the threshold of the perceptron algorithm to allow for a trade-off between precision and recall. For more details about the perceptron algorithm, refer to Chapter 5.

Traditionally, a binary perceptron classifier returns true if $w \cdot x > 0$, else false, with w being a vector of weights for each feature and x being a binary vector indicating whether a feature was active. If we wanted to improve recall, we can decrease the perceptron threshold to a level lower than zero. This artificially boosts the instances which were just below the perceptron threshold level of zero, allowing them to be considered as a positive class. As such, we can increase the recall of the true decisions in frontier pruning by allowing through borderline decisions. In our experiments, the perceptron threshold level will be referred to as λ and results in a slightly modified classifier of the form $w \cdot x > \lambda$.

Identifying the optimal threshold value is an important factor for frontier pruning. Too high a recall value would prevent pruning any parts of the parse tree whilst too low a threshold reverts back to traditional unpruned parsing. This value is determined experimentally using the development dataset.

8.6 Results

For frontier pruning, we used the development section of CCGbank, Section 00, as the testbed for the frontier pruning implementation.

8.6.1 Tuning the Perceptron Threshold Level

In Table 8.3, we performed testing on the perceptron threshold level λ. Our experiments revealed there was no substantial improvement in either labeled or unlabeled F-score due to decreasing λ. Further investigation revealed that frontier pruning is successful for the same reason the PARSEVAL measure is not (see Section 4.2.1). For many derivations in CCG, there are multiple structurally different CCG trees exist that have the same semantics. For PARSEVAL this represented a major issue as equivalent but structurally different derivations were penalised. For frontier pruning, this is a major benefit of CCG. By providing multiple structurally different CCG trees that are equally correct, a mistake in frontier pruning does not necessarily prevent the correct output.

Decreasing λ did result in slower parsing speeds, as expected. Unexpectedly however, the frontier pruning parsers with $\lambda = -1$ and $\lambda = -3$ have the highest coverage of all parsers, including the CKY and shift-reduce parsers introduced previously (see Chapter 6). This was not expected as mistakes in frontier pruning, specifically aggressive pruning, were likely to prevent successful analyses rather than assist in producing them. This suggests that for some sentences, aggressive pruning was necessary as the sentence was too large to successfully parse otherwise. By performing aggressive frontier pruning, the correct derivation could develop before the maximum number of categories were produced. Once the maximum number of categories are produced, the C&C parser aborts parsing to prevent excessive slow-downs.

From this, we have shown that ineffective frontier pruning can be as slow as traditional parsing and that modifying the perceptron threshold does not improve accuracy or speed for this task. Thus, all experiments from this point forward will use $\lambda = 0$.

MODEL	COVERAGE (%)	LABELED F-SCORE (%)	UNLABELED F-SCORE (%)	SPEED (sents/sec)
CKY C&C	99.01	**86.37**	**92.56**	55.6
SR C&C	98.90	86.35	92.44	48.6
FP $\lambda = 0$	99.01	86.11	92.25	**61.1**
FP $\lambda = -1$	**99.06**	86.16	92.23	56.4
FP $\lambda = -2$	99.01	86.13	92.19	53.9
FP $\lambda = -3$	**99.06**	86.15	92.21	49.0
CKY C&C Auto	**98.90**	**84.30**	**91.26**	56.2
SR C&C Auto	98.85	84.27	91.10	47.5
FP $\lambda = 0$ Auto	98.80	84.09	90.97	**60.0**

TABLE 8.3. Comparison to baseline parsers and analysis of the impact of threshold levels on frontier pruning (FP). The perceptron threshold level is referred to as λ. All results are against the development dataset, Section 00 of CCGbank, which contains 1,913 sentences.

8.6.2 Frontier Pruning Results

Table 8.3 presents a comparison between the accuracy and speed of three C&C parser variations. The base CKY parser, the shift-reduce parser developed in Chapter 6 and the frontier pruning assisted parser developed in this chapter. Frontier pruning runs on a modified version of the shift-reduce parser, so all speed improvements will use it as the base parsing speed.

In the table, we see that the frontier pruning parser with $\lambda = 0$ achieves the fastest parsing speeds of all parser variations. Unfortunately, frontier pruning reduces parsing accuracy by 0.30% on average when compared against the CKY C&C parser. This is not an entirely negative result, however, as it is possible the accuracy improvements provided by performing in-place POS tagging and supertagging with incremental parsing (see Chapter 7) may mitigate these accuracy losses. As the time to perform this work was limited, these experiments have not yet been performed.

From Chapter 6, the shift-reduce parser is found to be approximately 34% slower than the CKY parser on which it is based. These initial experiments suggest that frontier pruning can improve parsing speed by at least 26%, bringing the parsing speed of shift-reduce parser near to the levels of the CKY algorithm. It may be possible to produce a CCG parser with all of the benefits of incremental parsing whilst operating at speeds near to that of traditional non-incremental CKY parsing.

MODEL	COVERAGE (%)	LABELED F-SCORE (%)	UNLABELED F-SCORE (%)	SPEED (sents/sec)
CKY C&C	99.34	**86.79**	**92.50**	**96.3**
SR C&C	**99.58**	86.78	92.41	71.3
FP $\lambda = 0$	99.38	86.51	92.25	95.4
CKY C&C Auto	99.25	**84.59**	**91.20**	82.0
SR C&C Auto	**99.50**	84.53	91.09	61.2
FP $\lambda = 0$ Auto	99.29	84.29	90.88	**84.9**

TABLE 8.4. Final evaluation on Section 23 of CCGbank for the top performing models from Table 8.3, containing 2,407 sentences.

8.7 Final Results

The final evaluation for this work uses the evaluation section of CCGbank, Section 23. This allows for a fair evaluation as it is possible over-fitting occurred on the development dataset, especially when parameters such as λ or feature set size and expressiveness were optimised based upon the results.

Table 8.4 reports the final speed and accuracy evaluated over Section 23 of CCGbank. The accuracy numbers reported for final evaluation mirror those seen in the development evaluation. Frontier pruning caused on average a 0.30% decrease in both labeled and unlabeled dependency F-score compared to the C&C CKY parser. In contrast to the evaluation, the coverage of both the shift-reduce parser, both with and without frontier pruning, is higher than the C&C CKY parser. Compared to the coverage from the shift-reduce parser, however, there is 0.20% decrease. This suggests that frontier pruning is preventing the parser finding spanning analyses due to aggressive pruning.

Finally, frontier pruning has improved the speed of the shift-reduce C&C parser by 39%, an improvement over the speed increase seen during evaluation. Longer sentences seem to have a higher impact on the speed of the frontier pruning algorithm due to the increased computational complexity of feature generation. This indicates that implementing a form of beam search may be beneficial, retaining only the top k scoring states in a frontier. Currently all partial derivations that are greater than the perceptron threshold level λ are kept.

8.8 Discussion

As the C&C parser is already highly tuned and thus extremely fast, the optimal balance between feature expressiveness and accurate pruning is difficult to achieve. It is important to note that this suggests

that frontier pruning may be more effective when implemented on slower parsers than the C&C parser. More work needs to be done on reducing the number of computationally intensive feature look-ups and calculations. Even when using the gold-standard subset of the features, the feature look-up process accounts for the majority of the slow-down that the frontier pruning algorithm causes.

The C&C code has been highly optimised to suit CKY parsing. It should be possible to improve the GSS parser to be directly competitive with the CKY implementation. The frontier pruning provides speed increases for the GSS parser, allowing it to be competitive with the original CKY parser, but with an improved GSS parser, we could expect further improvements over the original CKY parser.

We have also shown that whilst pruning is occurring at the lexical level due to supertagging, substantial speed-ups are still possible by performing pruning during the parsing process itself.

8.9 Summary

In this chapter, we present a novel form of pruning that takes advantage of the features generated by the shift-reduce parser from Chapter 6. Though the shift-reduce CCG parser is 34% slower than the CKY parser on which it is based, we show that by performing frontier pruning on the GSS, the speed of the parser can be improved by 39% whilst only incurring a small accuracy penalty. This method shows it is possible to attain competitive speeds using a shift-reduce parser even when exploring the full search space.

Further improvements to either the base GSS implementation or the pruning algorithm are likely and will be complimentary. Modification of the perceptron threshold level (λ) was shown to be an ineffective method of weighting the classifier. We are likely to explore other methods of weighting that will be more effective and theoretically better sound. With additional optimisation of both the model and the feature set, we believe the speed and accuracy of the frontier pruned shift-reduce CCG parser can be further increased.

Preliminary frontier pruning results from this chapter have been accepted for publication under the title *Frontier Pruning for Shift-Reduce* CCG *Parsing* at the Australasian Language Technology Workshop 2011.

CHAPTER 9

Conclusion

In this chapter, we describe the future directions for the work described in this thesis and summarise the conclusions that we draw from our work.

9.1 Future Work

The graph-structured stack based incremental parser described in Chapter 6 is not fully optimised compared to the CKY algorithm that exists in the base C&C parser. Our next step of development is implementing the missing optimisations used by the CKY algorithm in the shift-reduce algorithm and graph-structured stack. We have shown that incremental parsing can be competitive with non-incremental methods. Theoretically, there should be no substantial difference in parsing speed between them. Further improvements will minimise the negative impact on speed whilst allowing for the novel features that incremental parsing provides. This will allow for the 39% speed increase to be on top of the base parsing speed.

We have applied the novel features produced during incremental parsing to only two components, the POS tagger and the supertagger. Both have shown substantial accuracy improvements given the small room for further gains. Many other applications would likely see a benefit from integrating these same features produced by the incremental parser. Exploring the impact that these features have on other parsing components and possible even separate NLP tasks would be a promising avenue of research.

Frontier pruning, as described in Chapter 8, is effective at improving parsing speed by removing portions of the search space not likely needed in the parsing process. We believe the parsing speed can be further improved by a form of self-training. By training the frontier pruning model on output from the current frontier pruning model, pruning can be even more aggressive without causing further impacts on

accuracy. This concept was explored by Kummerfeld et al. (2010) and has the potential to be used to improve the frontier pruning model.

The core idea of tightly integrating the parser and other components can be extended significantly. Although our work begins shows accuracy improvements due to our integration, even tighter integration is possible by passing features between all major components. This would likely decrease the severity of errors in the parsing pipeline and would be expected to significantly improve per component accuracy and overall parsing accuracy.

9.2 Contributions

Our core contributions have addressed a number of outstanding questions in the field, specifically focused on incremental parsing and integration in the parsing pipeline.

We have extended the high-speed state-of-the-art C&C parser to support incremental parsing. As this shift-reduce algorithm has a worst case exponential time complexity, we enabled practical shift-reduce parsing by implementing the first graph-structured stack for CCG parsing in the literature. During evaluation, we found our incremental parser produces output of near equivalent accuracy whilst only incurring a 34% speed penalty. We conclude that with further engineering optimisations, the incremental CCG parser could be directly competitive against the traditional C&C parser on both speed and accuracy.

Using the new capabilities our incremental parser, we tightly integrated parsing and tagging to allow for improved accuracy. We have also explored the accuracy impact of errors early in the parsing pipeline We find that, by providing the partial understanding of the sentence that the incremental parser has generated as novel features, we can improve the accuracy of POS tagging and supertagging substantially. With these novel features, we improve sentence level POS tagging accuracy by 3.40% and per token supertagging accuracy by 2.00%.

We have also implemented a form of pruning that improves parsing speed by using the features produced by the incremental parser. Frontier pruning allowed for a 39% speed improvement for the incremental CCG parser with little impact on accuracy. This negated the 34% speed loss caused by replacing the CKY algorithm with the shift-reduce algorithm in the incremental parser.

Thus, our work delivers a high speed state-of-the-art incremental CCG parser that enables improved accuracy across other components in the parsing pipeline.

Preliminary work on the shift-reduce algorithm, the graph-structured stack and frontier pruning is to be published under the title *Frontier Pruning for Shift-Reduce* CCG *Parsing* at the Australasian Language Technology Workshop in December 2011.

Bibliography

Alred V. Aho and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling. Volume I: Parsing*. Prentice-Hall.

Srinivas Bangalore. 1997. *Complexity of Lexical Descriptions and its Relevance to Partial Parsing*. Ph.D. thesis, University of Pennsylvania, Philadelphia, Pennsylvania, USA.

Srinivas Bangalore and Aravind K. Joshi. 1999. Supertagging: An Approach to Almost Parsing. *Computational Linguistics*, 25(2):237–265.

Michele Banko and Eric Brill. 2001. Scaling to Very Very Large Corpora for Natural Language Disambiguation. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics (ACL-01)*, pages 26–33. Toulouse, France.

Yehoshua Bar-Hillel. 1953. A Quasi-Arithmetical Notation for Syntactic Description. *Language*, 29(1):47–58.

Douglas Biber. 1993. Using Register-Diversified Corpora for General Language Studies. *Computational Linguistics*, 19(2):219–241.

Ezra W. Black, Steven P. Abney, Daniel P. Flickenger, Claudia Gdaniec, Ralph Grishman, Philip Harrison, Donald Hindle, Robert J. P. Ingria, Frederick Jelinek, Judith L. Klavans, Mark Y. Liberman, Mitchell P. Marcus, Salim Roukos, Beatrice Santorini, and Tomek Strzalkowski. 1991. A Procedure for Quantitatively Comparing the Syntactic Coverage of English Grammars. In *Proceedings of the Fourth DARPA Speech and Natural Language Workshop*, pages 306–311. Pacific Grove, California, USA.

Ted Briscoe and John Carroll. 2006. Evaluating the Accuracy of an Unlexicalized Statistical Parser on the PARC DepBank. In *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*, pages 41–48. Sydney, Australia.

Ted Briscoe, John Carroll, and Rebecca Watson. 2006. The Second Release of the RASP System. In *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*, pages 77–80. Sydney, Australia.

Aoife Cahill, Michael Burke, Ruth O'Donovan, Stefan Riezler, Josef van Genabith, and Andy Way. 2008. Wide-Coverage Deep Statistical Parsing using Automatic Dependency Structure Annotation. *Computational Linguistics*, 34(1):81–124.

Sharon A. Caraballo and Eugene Charniak. 1996. Figures of Merit for Best-First Probabilistic Chart Parsing. *Computational Linguistics*, 24:275–298.

Sharon A. Caraballo and Eugene Charniak. 1997. New Figures of Merit for Best-First Probabilistic Chart Parsing. *Computational Linguistics*, 24:275–298.

Eugene Charniak, Mark Johnson, Micha Elsner, Joseph Austerweil, David Ellis, Isaac Haxton, Catherine Hill, R. Shrivaths, Jeremy Moore, Michael Pozar, and Theresa Vu. 2006. Multilevel Coarse-to-fine PCFG Parsing. In *Proceedings of the Main Conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, June, pages 168–175. Morristown, NJ, USA.

Eugene Charniak, Kevin Knight, and Kenji Yamada. 2003. Syntax-based Language Models for Statistical Machine Translation. In *MT Summit IX*, pages 40–46.

John Chen and Owen Rambow. 2003. Use of Deep Linguistic Features for the Recognition and Labeling of Semantic Arguments. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing (EMNLP-03)*, pages 41–48.

Stanley F. Chen and Ronald Rosenfeld. 1999. A Gaussian Prior for Smoothing Maximum Entropy Models. Technical Report February, Carnegie Mellon University.

Noam Chomsky. 1957. *Syntactic Structures*. Mouton de Gruyter.

Stephen Clark and James Curran. 2007a. Perceptron Training for a Wide-Coverage Lexicalized-Grammar Parser. In *Proceedings of the Deep Linguistic Processing Workshop at ACL-07*, pages 9–16. Prague, Czech Republic.

Stephen Clark and James R. Curran. 2004a. Parsing the WSJ Using CCG and Log-Linear Models. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pages 103–110. Barcelona, Spain.

Stephen Clark and James R. Curran. 2004b. The Importance of Supertagging for Wide-Coverage CCG Parsing. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING-04)*, pages 282–288. Geneva, Switzerland.

Stephen Clark and James R. Curran. 2007b. Formalism-Independent Parser Evaluation with CCG and DepBank. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL-07)*, pages 248–255. Prague, Czech Republic.

Stephen Clark and James R. Curran. 2007c. Wide-Coverage Efficient Statistical Parsing with CCG and Log-Linear Models. *Computational Linguistics*, 33(4):493–552.

Stephen Clark and James R. Curran. 2009. Comparing the Accuracy of CCG and Penn Treebank Parsers. In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers (ACL-IJCNLP '09)*, pages 53–56. Singapore.

Stephen Clark, Julia Hockenmaier, and Mark Steedman. 2002. Building Deep Dependency Structures using a Wide-Coverage CCG Parser. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL-02)*, pages 327–334. Philadelphia, Pennsylvania, USA.

Michael Collins. 2002. Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP-03)*, pages 1–8.

Michael Collins, Philipp Koehn, and Ivona Kucerova. 2005. Clause Restructuring for Statistical Machine Translation. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL-05)*, pages 531–540. Ann Arbor, Michigan.

Michael A. Covington. 2001. A Fundamental Algorithm for Dependency Parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102.

James R. Curran and Stephen Clark. 2003. Investigating GIS and Smoothing for Maximum Entropy Taggers. In *Proceedings of the Tenth Conference on European Chapter of the Association for Computational Linguistics (EACL-03)*, pages 91–98.

Haskell B Curry and Robert Feys. 1958. *Combinatory Logic: Volume I*. Studies in Logics and the Foundations of Mathmatics.

Jason Eisner. 1996. Efficient Normal-Form Parsing for Combinatory Categorial Grammar. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics (ACL-96)*, pages 79–86. Santa Cruz, California, USA.

Kuzman Ganchev and Mark Dredze. 2008. Small Statistical Models by Random Feature Mixing. In *Proceedings of the ACL Workshop on Mobile Language Processing*.

Daniel Gildea. 2001. Corpus Variation and Parser Performance. In *Proceedings of the 2001 Conference on Empirical Methods in Natural Language Processing (EMNLP-01)*, pages 167–202. Pittsburgh, Pennsylvania, USA.

Daniel Gildea and Martha Palmer. 2002. The Necessity of Parsing for Predicate Argument Recognition. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics (ACL-02)*, pages 239–246. Philadelphia, Pennsylvania, USA.

Ralph Grishman and Mahesh Chitrao. 1988. Evaluation Of A Parallel Chart Parser. In *Proceedings of the 2nd Conference on Applied Natural Language Processing*, pages 71–76. Austin, Texas, USA.

Andrew Haas. 1987. Parallel Parsing for Unification Grammars. In *Proceedings of the 6th International Joint Conference on AI (IJCAI-87)*, pages 615–618.

Hany Hassan, Khalil Sima'an, and Andy Way. 2008. A Syntactic Language Model Based on Incremental CCG Parsing. In *Proceedings of the Workshop on Spoken Language Technology (SLT-08)*, pages 205–208.

Julia Hockenmaier. 2003. *Data and Models for Statistical Parsing with Combinatory Categorial Grammar*. Ph.D. thesis, School of Informatics, University of Edinburgh, Edinburgh, United Kingdom.

Julia Hockenmaier and Mark Steedman. 2007. CCGbank: A Corpus of CCG Derivations and Dependency Structures Extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL-10)*, pages 1077–1086. Uppsala, Sweden.

Tadao Kasami. 1965. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

Tracy Holloway King, Richard Crouch, Stefan Riezler, Mary Dalrymple, and Ronald M. Kaplan. 2003. The PARC 700 Dependency Bank. In *Proceedings of the 4th International Workshop on Linguistically Interpreted Corpora*, pages 1–8. Budapest, Hungary.

Paul Kingsbury and Martha Palmer. 2002. From TreeBank to PropBank. In *Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC-2002)*, pages 1989–1993.

Dan Klein and Christopher D. Manning. 2003. A* Parsing: Fast Exact Viterbi Parse Selection. In *Proceedings of the Human Language Technology Conference and the North American Association for Computational Linguistics (HLT-NAACL-03)*, volume 3, pages 119–126.

Donald E. Knuth. 1965. On the translation of languages from left to right. *Information and Control*, 8(6):607–639.

Jonathan K. Kummerfeld, Jessika Roesner, Tim Dawborn, James Haggerty, James R. Curran, and Stephen Clark. 2010. Faster Parsing by Supertagger Adaptation. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL-10)*, pages 345–355. Uppsala, Sweden.

Joachim Lambek. 1958. The Mathematics of Sentence Structure. *The American Mathematical Monthly*, 65(3):154–170.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Takuya Matsuzaki and Jun'ichi Tsujii. 2008. Comparative Parser Performance Analysis across Grammar Frameworks through Automatic Tree Conversion using Synchronous Grammars. In *Proceedings of the 22nd International Conference on Computational Linguistics (COLING-08)*, pages 545–552. Manchester, UK.

Eleni Miltsakaki, Rashmi Prasad, Aravind Joshi, and Bonnie Webber. 2004. The Penn Discourse TreeBank. In *Proceedings of the 4th International Conference on Language Resources and Evaluation (LREC-2004)*. Lisbon, Portugal.

Joakim Nivre and Mario Scholz. 2004. Deterministic Dependency Parsing of English Text. In *Proceedings of the 18nd International Conference on Computational Linguistics (COLING-04)*, pages 64–70. Geneva, Switzerland.

Adam Pauls and Dan Klein. 2009. K-Best A* Parsing. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 958–966. Singapore.

Martin J. Pickering. 1999. Sentence comprehension. *Language Processing*, 2:123.

Adwait Ratnaparkhi. 1996. A Maximum Entropy Model for Part-of-Speech Tagging. In *Proceedings of the 1996 Conference on Empirical Methods in Natural Language Processing (EMNLP-96)*, pages 133–142. Philadelphia, Pennsylvania, USA.

Ines Rehbein and Josef van Genabith. 2007. Treebank Annotation Schemes and Parser Evaluation for German. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 630–639. Prague, Czech Republic.

C J Van Rijsbergen. 1979. *Information Retrieval, 2nd Edition*. Dept. of Computer Science, University of Glasgow.

Laura Rimell, Stephen Clark, and Mark Steedman. 2009. Unbounded Dependency Recovery for Parser Evaluation. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing (EMNLP-09)*, pages 813–821. Singapore.

Brian Roark and Eugene Charniak. 2000. Measuring Efficiency in High-Accuracy, Broad-Coverage

Statistical Parsing. In *Proceedings of the Workshop on Efficiency in Large-scale Parsing Systems at COLING-00*, pages 29–36.

Brian Roark and Kristy Hollingshead. 2008. Classifying Chart Cells for Quadratic Complexity Context-Free Inference. In *Proceedings of the 22nd International Conference on Computational Linguistics (COLING-08)*, pages 745–752. Manchester, UK.

Brian Roark and Kristy Hollingshead. 2009. Linear Complexity Context-Free Parsing Pipelines via Chart Constraints. In *Proceedings of 2009 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL-09)*, pages 647–655. Boulder, Colorado.

Helmut Schmid. 2004. Efficient Parsing of Highly Ambiguous Context-Free Grammars with Bit Vectors. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING-04)*, pages 162–168. Geneva, Switzerland.

Vijay K. Shanker and David J. Weir. 1994. The Equivalence of Four Extensions of Context-Free Grammars. *Mathematical Systems Theory*, 27(6):511–546.

Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, and S. V. N. Vishwanathan. 2009. Hash kernels for structured data.

Mark Steedman. 2000. *The Syntactic Process*. MIT Press, Cambridge, Massachusetts, USA.

Michael K. Tanenhaus and Sarah Brown-Schmidt. 2008. Language Processing in the Natural World. *Philosophical Transactions of the Royal Society of London*, 363(1493):1105–1122.

Masaru Tomita. 1987. An Efficient Augmented-Context-Free Parsing Algorithm. *Computational Linguistics*, 13(1-2):31–46.

Masaru Tomita. 1988. Graph-structured Stack and Natural Language Parsing. In *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*, pages 249–257. Buffalo, New York, USA.

Masaru Tomita. 1990. The Generalized LR Parser/Compiler V8-4: A Software Package for Practical NL Projects. In *Proceedings of the 13th International Symposium on Computational Linguistics*, pages 59–63.

Daniel Tse and James R. Curran. 2007. Extending CCGbank with Quotes and Multi-Modal CCG. In *Proceedings of the 2007 Australasian Language Technology Workshop (ALTW-07)*, pages 149–151. Melbourne, Australia.

David Vadas and James R. Curran. 2007. Adding Noun Phrase Structure to the Penn Treebank. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL-07)*, pages 240–247. Prague, Czech Republic.

David Vadas and James R. Curran. 2008. Parsing Noun Phrase Structure with CCG. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-08:HLT)*, pages 335–343. Columbus, Ohio, USA.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature Hashing for Large Scale Multitask Learning. *Proceedings of the 26th Annual International Conference on Machine Learning (ICML-09)*, pages 1–8.

Kent Wittenburg. 1986. *Natural Language Parsing with Combinatory Categorial Grammars in a Graph-Unification-Based Formalism*. Ph.D. thesis, University of Texas at Austin.

Kent Wittenburg. 1987. Predictive Combinators: A Method for Efficient Processing of Combinatory Categorial Grammars. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics (ACL-87)*, pages 73–80. Stanford, California, USA.

Mary McGee Wood. 1993. *Categorial Grammars*. Taylor & Francis.

Daniel H. Younger. 1967. Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control*, 10(2):189–208.

Yue Zhang, Byung-Gyu Ahn, Stephen Clark, Curt Van Wyk, James R. Curran, and Laura Rimell. 2010. Chart Pruning for Fast Lexicalised-Grammar Parsing. In *Proceedings of the COLING 2010 Poster Sessions*, pages 1471–1479. Beijing, China.

Yue Zhang and Stephen Clark. 2011. Shift-Reduce CCG Parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-11:HLT)*, pages 683–692. Portland, Oregon, USA.