Algorithms for Identifying Syntactic Errors and Parsing with Graph Structured Output

Jonathan K. Kummerfeld

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-138
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-138.html

August 5, 2016

Algorithms for Identifying Syntactic Errors and Parsing with Graph Structured Output

by

Jonathan Kay Kummerfeld

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dan Klein, Chair
Professor Marti A. Hearst
Associate Professor Line Mikkelsen

Summer 2016

Algorithms for Identifying Syntactic Errors and Parsing with Graph Structured Output

Abstract

Algorithms for Identifying Syntactic Errors and Parsing with Graph Structured Output

by

Jonathan Kay Kummerfeld

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Dan Klein, Chair

Representation of syntactic structure is a core area of research in Computational Linguistics, disambiguating distinctions in meaning that are crucial for correct interpretation of language. Development of algorithms and statistical models over the past three decades has led to systems that are accurate enough to be deployed in industry, playing a key role in products such as Google Search and Apple Siri. However, syntactic parsers today are usually constrained to tree representations of language, and performance is interpreted through a single metric that conveys no linguistic information regarding remaining errors.

In this dissertation, we present new algorithms for error analysis and parsing. The heart of our approach to error analysis is the use of structural transformations to identify more meaningful classes of errors, and to enable comparisons across formalisms. For parsing, we combine a novel dynamic program with careful choices in syntactic representation to create an efficient parser that produces graph structured output. Together, these developments allowed us to evaluate the outstanding challenges in parsing and to address a key weakness in current work.

First, we present a search algorithm that, given two structures, finds a sequence of modifications leading from one structure to the other. We applied this algorithm to syntactic error analysis, where one structure is the output of a parser, the other is the correct parse, and each modification corresponds to fixing one error. We constructed a tool based on the algorithm and analyzed variations in behavior between parsers, types of text, and languages. Our observations shine light on several assumptions about syntactic errors, showing some to be true and others to be false. For example, prepositional phrase attachment errors are indeed a major issue, while coordination scope errors do not hurt performance as much as expected.

Next, we describe an algorithm that builds a parse in one syntactic representation to match a parse in another representation. Specifically, we build phrase structure parses from Combinatory Categorial Grammar derivations. Our approach follows the philosophy of CCG, defining specific phrase structures for each lexical category and generic rules for combinatory steps. The new parse is built by following the CCG derivation bottom-up, gradually building the corresponding phrase structure parse. This produced significantly more accurate parses than past work, and enabled us to compare performance of several parsers across formalisms.

Finally, we address a weakness we observed in phrase structure parsers: the exclusion of syntactic trace structures for computational convenience. We present an efficient dynamic programming algorithm that constructs the graph structure that has the highest score under an edge-factored scoring function. We define a parse representation compatible with the algorithm, and show how certain linguistic distinctions dramatically impact coverage. We also show various ways to modify the algorithm to improve performance by exploiting properties of observed linguistic structure. This approach to syntactic parsing is the first to cover virtually all structure encoded in the Penn Treebank.

Contents

Contents **i**

List of Algorithms **iv**

List of Figures **v**

List of Tables **vii**

1 Introduction **1**
1.1 Syntax . . . 2
1.2 Error Analysis . . . 4
1.3 Formalism Conversion . . . 4
1.4 Graph Parsing . . . 5
1.5 Contributions of This Dissertation . . . 7

2 Automatic Error Analysis **9**
2.1 Error Classification . . . 9
2.1.1 Transformations . . . 11
2.1.2 Transformation Classification . . . 12
2.2 Results . . . 20
2.2.1 Reranking . . . 24
2.2.2 Out-of-Domain . . . 26
2.3 Chinese Parsing . . . 29
2.3.1 Background . . . 29
2.3.2 Adapting Error Analysis to Chinese . . . 30
2.3.3 Chinese Parsing Errors . . . 30
2.3.4 Chinese-English Comparison . . . 34
2.3.5 Cross-Parser Analysis . . . 34
2.3.6 Tagging Error Impact . . . 37
2.4 Summary . . . 38

3 Formalism Conversion **39**
3.1 Background . . . 39

3.1.1 Combinatory Categorial Grammar (CCG) . . . 39
3.1.2 Clark and Curran (2009) . . . 41
3.1.3 X. Zhang et al. (2012) . . . 41
3.2 Our Approach . . . 42
3.3 Evaluation . . . 45
3.3.1 Parser Comparison . . . 47
3.4 Summary . . . 48

4 Graph Parsing **49**
4.1 Background . . . 49
4.1.1 Why Graph Structures? . . . 50
4.1.2 Previous Algorithms . . . 51
4.2 Overall Algorithm . . . 52
4.3 Sketch of Deduction Rules . . . 53
4.3.1 Notation . . . 54
4.3.2 Item Types . . . 54
4.3.3 Example Derivation . . . 55
4.3.4 Deduction Rules . . . 58
4.4 Comparison with Pitler et al. (2013) . . . 62
4.5 Deduction Rule Definitions and Completeness Proof . . . 63
4.5.1 Notation . . . 63
4.5.2 Item Types . . . 63
4.5.3 Complete Dynamic Program . . . 65
4.5.4 Eisner (1996)'s Algorithm . . . 65
4.5.5 Initialization . . . 67
4.5.6 Interval . . . 67
4.5.7 Exterval . . . 70
4.5.8 Both . . . 71
4.5.9 Left . . . 74
4.5.10 Right . . . 76
4.5.11 Neither . . . 76
4.5.12 Additional Constraints . . . 78
4.6 Algorithm Properties . . . 78
4.6.1 Derivational Ambiguity . . . 78
4.6.2 Complexity . . . 78
4.7 Parse Representation . . . 79
4.7.1 Core Structure . . . 80
4.7.2 Additional Structure . . . 81
4.7.3 Head Rules . . . 83
4.8 Algorithm Extensions . . . 84
4.8.1 Edge Labels and Spines . . . 84
4.8.2 Ensuring the Graph Contains a Structural Tree . . . 84

4.9 Implementation 85
4.9.1 Model 85
4.9.2 Learning 86
4.9.3 Inference 88
4.10 Results 90
4.10.1 Algorithm Coverage 90
4.10.2 Problematic Structures 93
4.10.3 Parsing Performance 94

5 Conclusion **95**

A Resources **97**

Bibliography **99**

List of Algorithms

2.1 Transformation based error classification 11
4.1 General CKY algorithm. 52
4.2 Complete graph parsing dynamic program. 64
4.3 Making I with direct edges. 67
4.4 Making I, case one. 67
4.5 Making I, case two. 68
4.6 Making I, case four. 68
4.7 Making I, cases three and five (part one). 69
4.8 Making I, cases three and five (part two). 69
4.9 Making I, case six. 70
4.10 Making I, case seven. 70
4.11 Making X with direct edges. 71
4.12 Making B with direct edges. 72
4.13 Making B. 73
4.14 Making L with direct edges (part one). 74
4.15 Making L with direct edges (part two). 74
4.16 Making L, cases three through eight. 75
4.17 Making L, cases nine and ten. 75
4.18 Making L, cases one and two. 76
4.19 Making N with direct edges. 77
4.20 Making N, cases three and four. 77
4.21 Making N, cases seven and eight. 78
4.22 Rules that could create crossing arcs. 85
4.23 Online Primal Subgradient with ℓ_1 or ℓ_2 regularization, sparse updates 87
4.24 Full dynamic program with rules unseen in training boxed and colored. 91
4.25 Pruned dynamic program, including only rules observed in the training set, including tighter parent constraints. 92

List of Figures

1.1 Dependency, constituency, and categorial forms of the sentence Ellen enjoys running. . . 2
1.2 Parse representations for graph structures. 7

2.1 Error analysis example: PP attachment. 10
2.2 Error analysis example: Coordination. 13
2.3 Error analysis example: NP attachment. 14
2.4 Error analysis example: Clause attachment. 14
2.5 Error analysis example: VP attachment. 15
2.6 Error analysis example: Adverb and adjective modifier attachment. 16
2.7 Error analysis example: Unary. 16
2.8 Error analysis example: Different label. 17
2.9 Error analysis example: Single word phrase. 17
2.10 Error analysis example: Parenthetical attachment. 18
2.11 Error analysis example: Missing parenthetical. 18
2.12 Error analysis example: NP internal structure. 19
2.13 Error analysis example: NP internal structure (Chinese). 32
2.14 Error analysis example: Adverb and adjective modifier attachment (Chinese). 32
2.15 Error analysis example: Word sense confusion (Chinese). 33
2.16 Error analysis example: Verb taking wrong arguments (Chinese). 33

3.1 An example of CCG and PTB parses with nodes covering crossing spans. 40
3.2 An example function application during CCG to PTB conversion. 42
3.3 Heatmaps comparing gold conversion accuracy, CCG native evaluation, and converted evaluation. 46

4.1 Venn diagram of graph space by properties 51
4.2 An example derivation using our graph parsing deduction rules. 56
4.3 Overall picture of deduction rule definitions. 57
4.4 I deduction rules, case one. 58
4.5 I deduction rules, case two. 58
4.6 I deduction rules, cases three and five. 59
4.7 I deduction rules, case four. 59

4.8 I deduction rules, case seven. 59
4.9 I deduction rules, case six. 59
4.10 Excluded B deduction rule 60
4.11 B structural pattern 60
4.12 B deduction rule. 60
4.13 L deduction rules, cases three through eight. 61
4.14 L deduction rules, cases one and two. 61
4.15 L deduction rules, cases nine and ten. 61
4.16 N deduction rule. 61
4.18 Parse representations for graph structures. 80
4.19 Examples of graph structured syntactic phenomena. 82
4.20 Examples of problematic graph structured syntactic phenomena. 83

A.1 Parse error visualization example. 98

List of Tables

2.1 PARSEVAL results on WSJ section 23 for the parsers we consider. 20
2.2 Breakdown of errors on section 23 for the Charniak parser with self-trained model and reranker. 22
2.3 Average number of bracket errors per sentence due to the top ten error types. 23
2.4 Average number of bracket errors per sentence for a range of K-best list lengths using the Charniak parser with reranking and the self-trained model. 25
2.5 Variation in size and contents of the domains we consider. 27
2.6 Average number of node errors per word for a range of domains. 28
2.7 Breakdown of errors in Chinese parsing. 31
2.8 Error breakdown for a range of parsers on the PCTB. 36
2.9 The most frequently confused POS tag pairs in Chinese parsing. 37

3.1 Example rules from Clark and Curran (2009). 41
3.2 Types of operations in instructions in our CCG to PTB conversion. 42
3.3 Example instructions for our CCG to PTB conversion. 44
3.4 PARSEVAL Precision, Recall, F-Score, and exact sentence match for converted gold CCG derivations. 45
3.5 F-scores on section 23 for PTB parsers and CCG parsers with their output converted by our method. 48

4.1 Number of sentences in the training set that are of each structure type. 92
4.2 Coverage improvements for parts of our graph representation. 93

Acknowledgments

Every doctorate is a long journey, developing new understanding of the universe and yourself. I've had an incredible group of people helping and supporting me on my path.

First, my adviser, Dan Klein. Completing a doctorate is about learning many new skills, and Dan taught me all of them, either explicitly through guidance or implicitly through the example he set of how to do great research, present your work, run a group, teach, and more.

Next, I'd like to recognise the time and effort my committee put into their extremely thoughtful and detailed comments on this thesis. As a result, every chapter explains the work more clearly and fully, and more effectively draws connections with other research.

I have also been fortunate to work with fantastic people. As well as my adviser, the research presented here involved three key collaborators: Daniel Tse, David Hall, and James Curran. Your expertise was crucial, on everything from explaining Chinese syntax to showing how to squeeze as much content into a paper as possible.

Beyond my main collaborators, I have been assisted immensely by conversations with everyone who has been a part of the Berkeley NLP group during my PhD: Jacob Andreas, Mohit Bansal, Taylor Berg-Kirkpatrick, David Burkett, Greg Durrett, Daniel Fried, Dave Golland, David Hall, Nikita Kitaev, Percy Liang, Dominick Ng, Adam Pauls, Max Rabinovich, and Mitchell Stern. In particular, I am grateful to Greg, who I met at the Berkeley visit day, spent most of my degree working (literally) alongside, and have immense respect for as both a researcher and a friend.

Another form of support was provided by the General Sir John Monash Foundation, which connected me to an incredible group of Australians pursuing exciting careers across every field. I was also supported by grants with the Berkeley Security group, which gave me valuable experience working on a multi-disciplinary team with fascinating data.

Yet another form of support came from the circle of friends that I was a part of in Berkeley. In hindsight, I was incredibly lucky when I mailed the rest of the incoming students and asked if anyone wanted to live in a large house together. Not only did you become some of my closest friends, but our home became the centre of a larger group of friends that had many great adventures together. I will miss the community we formed, but am sure that our connection will be just as strong every time we are together again.

Looking back further, I want to acknowledge the people who were role models for me when I chose to follow the academic path. There are too many people to name here, including research advisers on undergraduate projects across computer science, chemistry, and physics, everyone at the JHU workshop, and my friends who started down the doctoral path before me. One group I would like to specifically mention is my family. Each of you inspire me in different ways, and all of you have encouraged and supported me my entire life.

Finally, along the way I found more than just interesting ideas and great friends. I met my now-fiancée, Ellen. You have been supportive, patient, generous, and many more positive adjectives.

Thank you.

Chapter 1

Introduction

Communication is a fundamental part of human society, enabling the transfer of information between people. Textual language in particular is found throughout our daily lives; when asking a question, we type it into a search engine and usually read an answer from either Wikipedia or discussions between other people; to find out what is happening in the world we read newspapers or listen to a newsreader; for entertainment we read stories, listen to podcasts, or watch TV shows (virtually all of which contain dialogue); and for personal communication we write electronic messages in a variety of formats. One of the keys to all of this communication is the use of structure to convey meaning. Part of this structure is visible–by convention you are looking at this page from top to bottom and left to right, treating each connected component of ink (or light) as a character, combining a sequence of characters into words, and so on to progressively larger levels of structure. The focus of this dissertation, syntax, is the mostly invisible structure that exists somewhere between characters and sentences. People are able to identify syntactic structure by drawing on knowledge from a range of sources, and in the process they discard a vast array of alternative interpretations for a given utterance. In the past century, artificial symbolic processing by machines has advanced dramatically, to the point where computers can start to engage in textual communication, both understanding what people have written and writing back. Approaching human-level communication will require systems to resolve syntactic ambiguities in text, either explicitly with an interpretable syntactic structure or implicitly with some other intermediate representation.

Over the past two decades there has been rapid development in systems for the syntactic parsing task, where the input is a single sentence and the output is a structure that encodes syntactic relationships between words in the sentence. This development has largely been driven by new statistical methods for constructing models from resources manually labeled with syntactic structures by linguists. In addition to variations in statistical and engineering approaches, research has explored various syntactic representations. These representations are based on different linguistic theories, each with a body of research going into its design. These theories vary in several ways: what form is used to encode syntax, what distinctions in structure are meaningful or not (and so represented or not), and what distinctions fall within syntax (and so represented here, or in downstream processing).

In this thesis, we present new algorithms that extend the capabilities of various systems related

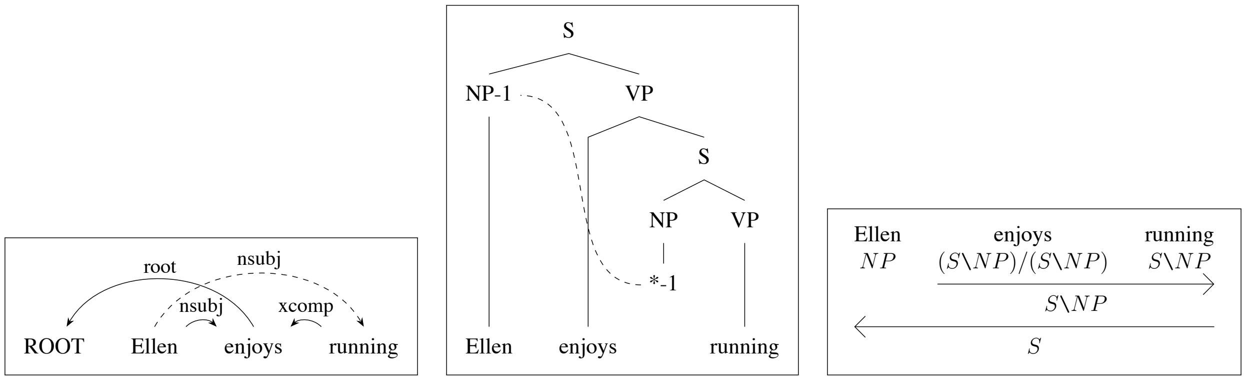


Figure 1.1: Dependency, constituency, and categorial forms of the sentence Ellen enjoys running. Visualizations used here are in the style of Nivre et al. (2016, dependency), Bies et al. (1995, constituency) and Steedman (2000, categorial).

to syntax. Together, these three areas of development show how we can go beyond the standard approach to parsing research, considering new challenges and ways to approach them. The first area of research we consider is how the community evaluates automatic syntactic parsers to understand their strengths and weaknesses. We propose a two-stage algorithm that searches for a set of transformations that will correct a parse, then classifies each transformation into one of several error types. Using the algorithm, we compare parse errors for a range of systems, a range of text domains, and two languages. The second aspect of syntax we consider is conversion between two different linguistic formalisms, which enables comparison of different parsers, and provides greater flexibility for projects where the parser is part of a larger pipeline. Our system uses a bottom-up approach that is more flexible and more effective than previous work. Finally, we explore how to extend parsing algorithms beyond tree structured output. Existing algorithms need the tree constraint to be efficient, but the standard way to apply the constraint has been to discard structure for a range of phenomena, such as wh-movement, passivization, and fronting. We introduce a new algorithm that can efficiently find the max or sum over all possible graph structures for a sentence, scored with an edge-factored model. We also define a new parse representation that is compatible with the algorithm and deterministically convertable into the Penn Treebank (PTB) style. Together, our algorithm and representation are able to produce parses that cover virtually all forms of structure in the Penn Treebank, providing a more complete expression of the structure of sentences for downstream processing.

1.1 Syntax

One of the core research areas in linguistics is syntax, the study of processes that determine word order in sentences. For the purposes of this thesis, it is important to understand a few general properties of the syntactic theories we consider. Each theory encodes relationships between words using structure, but the form of those structures varies considerably. Figure 1.1 shows how the sentence Ellen enjoys running is represented in the three syntactic formalisms we use.

In the leftmost case, which uses dependency grammar (DG; Tesnière 1959), each edge indicates a relationship between two words. The label expresses the type of dependency, out of 40-50 types (Marneffe and Manning 2008; Nivre et al. 2016). The arrow points from the word that is the *dependent* / *child* to the word that is the *head* / *parent*[1]. The dashed edge is often excluded in order to make the edges form a tree (a structure where every word has exactly one parent and the edges from one connected structure).

The middle case, which uses Government and Binding Theory (GB; Chomsky 1981), has a hierarchical phrase structure. Each symbol is a *constituent*, capturing the idea that the set of words beneath it constitute a single functional unit. Constituents are linked together to form larger constituents according to rules. This figure also shows a null element in between enjoys and running, which is used to encode the relationship between Ellen and running. As with the dashed edge in the dependency case, the null element is often removed from this structure to make it a tree.

The rightmost case, categorial grammar (Ajdukiewicz 1935), also has a hierarchical structure, but uses complex lexical categories that are then combined according to a small set of inference rules. This particular example uses Combinatory Categorial Grammar (CCG; Steedman 2000), a variant in which combinatory logic is used to construct both syntactic and semantic forms in parallel. Immediately beneath each word is its lexical category, which can either by an atomic symbol, or a complex structured combination of symbols. A small set of generic combinators define how pairs of adjacent categories can be combined. Along with the syntactic derivation, CCG builds up a lambda expression denoting the semantic structure of the sentence (not shown in this figure). One interesting property of the formalism is that a sentence can have multiple different derivations with the same semantic representation. This ability to have different structures encode the same meaning is a form of derivational or spurious ambiguity[2]. Note that unlike the previous two approaches, here the connection between Ellen and running is retained while keeping the structure a tree. This is possible because the relation has been threaded through the tree via the categories.

While syntactic formalisms are used in a variety of ways, our focus is on the automatic production of syntactic structures by computer programs. These programs take a sentence as input, consider possible structures and return the one that is best according to some scoring model[3]. Individually considering every possible structure for a sentence is infeasible as the number of possible parses is exponential in the length of the sentence. One way to avoid this issue is to perform an approximate search that incrementally builds the parse, maintaining only a few options at any given point. However, if the part of the optimal structure that is built first scores poorly, then it may drop out of the list of options, and so there is no guarantee of finding the optimal parse. Alternatively, we can maintain optimality at the cost of model flexibility, constraining the model so that the score for a parse is the sum of scores for each of its components. This work follows the second approach, which enables dynamic programming methods, where the larger problem (find the optimal parse) is

[1]The direction of the arrow is a convention we are following from linguistics. Note that this is the reverse of the convention in graph theory.

[2]Steedman (ibid.) explores the possibility that this ambiguity could encode variations in prosody.

[3]The sentence is assumed to be grammatical–a parse is always returned.

decomposed into independent sub-problems (find the optimal parse for part of the sentence) whose solutions can be used to solve the original problem (take the best solution from one half of the sentence and combine it with the best solution from the other half). The specific form of dynamic programming applied to the task of finding optimal parses structures is the CKY algorithm (Kasami 1966; Younger 1967; Cocke 1969).

1.2 Error Analysis

The standard resource for parsing research is the Wall Street Journal section of the Penn Treebank (Marcus et al. 1993), a collection of one million words of text from 1989 issues of the Wall Street Journal that have been annotated by experts with syntactic structure in a GB style. The standard measure of constituent parser performance is the F-Score, the harmonic mean of precision[4] and recall[5] on labeled nodes in the parse. Performance on WSJ section 23 has exceeded 90 F_1 (Petrov and Klein 2007), and 92 F_1 when using self-training and reranking (Charniak and Johnson 2005; McClosky et al. 2006a). While these results give a useful measure of overall performance, they provide no information about the nature, or relative importance, of the remaining errors.

Broad investigations of parser errors beyond the PARSEVAL metric (Black et al. 1991) have either focused on specific parsers, e.g., (Collins 2003), or have involved conversion to DG (Carroll et al. 1998; King et al. 2003). In all of these cases, the analysis has not taken into consideration how a set of errors can have a common cause, e.g., a single mis-attachment can create multiple node errors.

In the first part of the thesis, we propose a new method of error classification using tree transformations. Errors in the parse tree are repaired using subtree movement, node creation, and node deletion. Each step in the process is then associated with a linguistically meaningful error type, based on factors such as the node that is moved, its siblings, and parents. Using our method we analyze the output of thirteen constituency parsers on newswire. Some of the frequent error types that we identify are widely recognized as challenging, such as prepositional phrase (PP) attachment. However, other significant types have not received as much attention, such as attachment of clauses, adjective phrases, and adverb phrases. We also investigate where reranking and self-training improve parsing, and where performance decreases when parsing out-of-domain text. Previously, these were all analyzed only in terms of their impact on F-score.

1.3 Formalism Conversion

As shown above, there are many ways of expressing syntactic structure. Extensive work has gone into converting the Penn Treebank to other formalisms, such as HPSG (Miyao et al. 2004), LFG (Cahill et al. 2008), LTAG (Xia 1999), and CCG (Hockenmaier 2003), These conversions are complex processes that render linguistic phenomena in formalism-specific ways. Tools for the reverse

[4] Number of correct nodes in the output structure, divided by the total number of nodes it has.
[5] Number of correct nodes in the output structure, divided by the number of nodes in the gold structure.

process, converting back to the PTB, enable performance comparisons between parsers based on the two formalisms, and provide more options for researchers and developers building pipelines in which parsing is only one step. However, the reversal is difficult, as the original conversion may have lost information or smoothed over inconsistencies in the corpus. Clark and Curran (2009) developed a CCG to PTB auto-conversion tool that treats the CCG derivation as a phrase structure tree and applies hand-crafted rules to every pair of categories that combine in the derivation. Because their approach does not exploit the generalizations inherent in the CCG formalism, they must resort to ad-hoc rules over non-local features of the CCG constituents being combined (when a fixed pair of CCG categories correspond to multiple PTB structures). Even with such rules, they correctly auto-convert only 39.7% of gold CCGbank derivations.

In the second chapter, we describe an auto-conversion method that assigns a set of bracket instructions to each word based on its CCG category, then follows the CCG derivation, applying and combining instructions at each combinatory step to build a phrase structure tree. This requires specific instructions for each category (not all pairs), and generic operations for each combinator. Unlike Clark and Curran (ibid.)'s approach, we require no rules that consider non-local features of constituents, which enables the possibility of simple integration with a CKY-based parser.

Our approach perfectly auto-converts 51.4% of sentences, an 11.7% (absolute) improvement over Clark and Curran (ibid.). On the remaining sentences our auto-conversion generally handles most of the sentence correctly, but makes mistakes on some clause spans and rare spans such as QPs, NXs, and NACs. Many of these errors are inconsistencies in the original PTB annotations that are not recoverable. Applying our tool to the output of several CCG parsers, we are able to compare them with standard PTB parsers, showing that their accuracy falls within the middle of the performance range of well known systems. Our auto-conversion tool is easy to use and more effective than prior work, providing a convenient means of getting PTB-style output from a CCG parser.

1.4 Graph Parsing

In Section 1.1 we saw that while parse structures can be graphs and discontinuous, removing some edges in the dependency parse, or the traces in the Government and Binding structure, can make the structures into projective trees. Whether these edges are included in our grammar impacts the class of formal languages we are generating (Chomsky 1956). The trees are entirely within the context-free class, while the graphs are in the context-sensitive class. There are a range of well known polynomial time algorithms for parsing in the context-free class (Kasami 1966; Younger 1967; Cocke 1969; Earley 1970; Lang 1974), but parsing of context-sensitive grammars is PSPACE complete (Kuroda 1964; Savitch 1970). Fortunately, human language appears to fall into a class somewhere in between these two, which researchers have attempted to characterize using mildly-context-sensitive grammars (Weir and Joshi 1988) and range concatenation grammars (Boullier 1998). In general, work on these intermediate classes has been in conjunction with the development of new formalisms[6].

[6] For example, CCG falls into the mildly-context-sensitive class.

In the Penn Treebank, the traces are distinguished from the core projective tree structure, and are used to represent control structures, wh-movement and more. Traces are indicated using nodes in the parse that do not span any words (null elements) and numbers to indicate a connection with another node in the parse (co-indexation, with the null element getting a reference index and the other node getting an identity index). By varying the null element used, different forms of movement can be indicated. The list below describes all of the null elements used in the treebank, though only the first two can be assigned reference indexes (Bies et al. 1995):

- *T* – Trace of non-argument movement, including parasitic gaps
- (NP *) – Trace of argument movement, arbitrary PRO, and controlled PRO
- 0 – Null complementizer, including null wh-operator
- *U* – Unit
- *?* – Placeholder for ellipsed material
- *NOT* – Anti-placeholder in template gapping
- *RNR* – Pseudo-attach: right node raising
- *ICH* – Pseudo-attach: interpret constituent here
- *EXP* – Pseudo-attach: expletive
- *PPA* – Pseudo-attach: permanent predictable ambiguity

However, most parsers and the standard evaluation metric ignore these edges and all null elements, focusing entirely on the tree structure. By leaving out parts of the structure, they are not explicitly representing all of the relations in the sentence. These aspects of syntax are excluded not because of disagreements regarding theory, but rather because of the computational challenge of including them. Unfortunately, this means that downstream tasks such as question answering have to make do with the more limited structure, e.g., in Who enjoys running? the link between who and running is not present in the tree structure.

While there has been work on capturing some parts of this extra structure, it has generally either been through post-processing on trees (Johnson 2002; Jijkoun 2003; Campbell 2004; Levy and Manning 2004; Gabbard et al. 2006), or has only captured a limited set of phenomena via grammar augmentation (Collins 1997; Dienes and Dubey 2003; Schmid 2006; Cai et al. 2011). In both cases phenomena such as shared argumentation are completely ignored. Similarly, most work on the Abstract Meaning Representation (Banarescu et al. 2015), has removed edges to turn all structures into trees.

In the final chapter, we propose a new parse representation and a new algorithm that can efficiently consider almost all observed syntactic phenomena. Our representation is an extension of TAG-based tree representations (Shen et al. 2007; Carreras et al. 2008), modified to represent

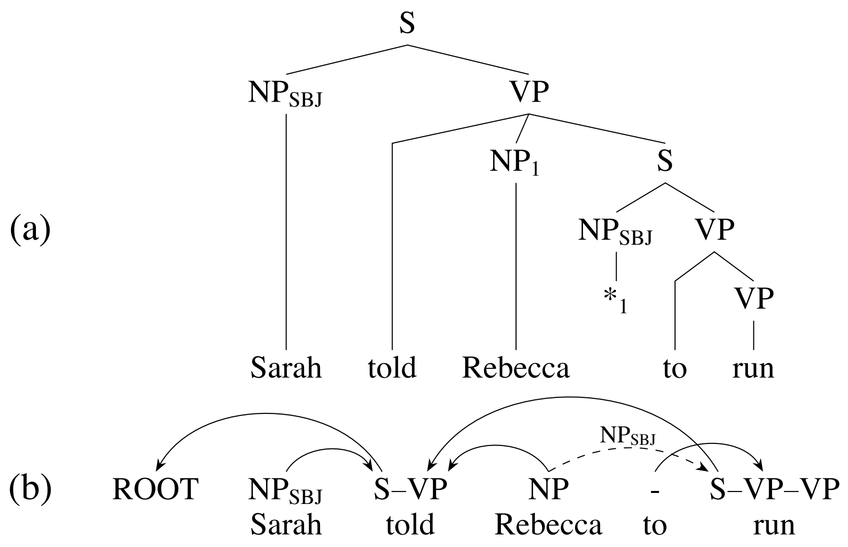


Figure 1.2: Parse representations for graph structures: (a) constituency (b) ours.

graphs and designed to maximize coverage under a new class of graphs. Our algorithm extends a non-projective tree parsing algorithm (Pitler et al. 2013) to graph structures, with improvements to avoid derivational ambiguity.

Our representation, shown in Figure 1.2b, consists of complex tags composed of non-terminals, and edges indicating attachment. In this form, traces can create problematic structures such as directed cycles, but we show how careful choice of head rules can minimize such issues.

Our algorithm runs in time $O(n^4)$ under a first-order model. We also introduce extensions that ensure parses contain a directed projective tree of non-trace edges. We implemented a proof-of-concept parser with a basic first-order model, which scored 88.3 on the standard evaluation metric (F_1 on trees), and recovered a range of trace types. Together, our representation and algorithm form an inference method that can cover 97.7% of sentences, far above the coverage of projective tree algorithms (46.8%).

1.5 Contributions of This Dissertation

Our contributions are a set of novel algorithms and experimental results and analysis using those algorithms. First, we define a new algorithm for error analysis of constituency parsing output, which provides a more intuitive breakdown of error types than previous approaches. We implement the algorithm and use it to give insight into current parsing effectiveness, considering a wide range of systems, multiple domains, and two languages. Second, we present a new algorithm for transforming CCG derivations into GB parses. Our approach is significantly more accurate than previous work and has desirable algorithmic properties. Finally, we describe the first algorithm for inference over the space of GB graph structures. Previous work compromised by leaving out important aspects of the syntactic representation in order to satisfy constraints of their parsing algorithms. We show how to efficiently implement the algorithm, and discuss results and remaining challenges.

Chapter 2

Automatic Error Analysis

Preliminary versions of parts of this chapter appeared as Kummerfeld, D. Hall, et al. (2012) and Kummerfeld, Tse, et al. (2013).

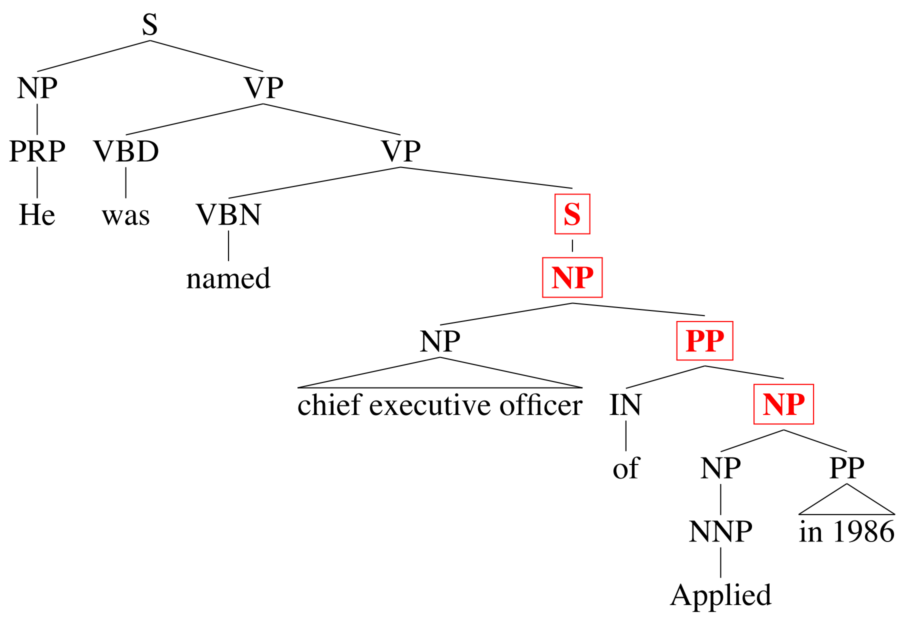
Constituency parser performance is primarily interpreted through a single metric, F-score on WSJ section 23, that conveys no linguistic information regarding the remaining errors. In this chapter, we describe a new error analysis method that classifies errors within a set of linguistically meaningful types using transformations that repair groups of errors together. We use this analysis to answer a range of questions about parser behavior, including what linguistic constructions are difficult for state-of-the-art parsers, what types of errors are being resolved by rerankers, what types are introduced when parsing out-of-domain text, and how the challenges change for Chinese.

2.1 Error Classification

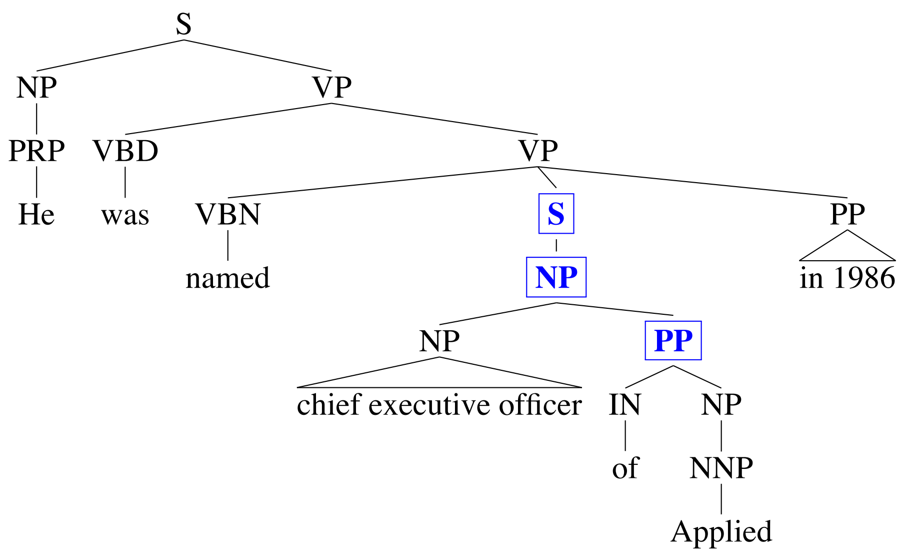
The standard metric for parsing is:

$$\begin{aligned} \mathrm{F}_1 &= \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}} \\ &= \frac{2 * |\text{matching nodes}|}{|\text{nodes in predicted parse}| + |\text{nodes in true parse}|} \end{aligned}$$

This is a robust metric that gives a sense of overall parser performance, but a single number cannot provide linguistically meaningful intuition for the source of remaining errors. Since the metric is defined in terms of nodes, one natural direction for analysis is to break down the errors by node type. Unfortunately, that approach is not particularly informative, as a single attachment error can cause multiple node errors, without a clear link between the two once aggregated. For example, in Figure 2.1 there is a PP attachment error that causes seven bracket errors (extra S, NP, PP, and NP, missing S, NP, and PP). Determining that these correspond to a PP attachment error from just the labels of the missing and extra nodes is difficult. Additionally, once an aggregate count of node



(a) Parser output



(b) Gold parse

Figure 2.1: **Incorrect PP Attachment** *in 1986* is too low. Bold, boxed nodes are either extra (marked in red in the parser output) or missing (marked in blue in the gold parse).

Algorithm 2.1 Transformation based error classification

```
$U$ = initial set of node errors
Sort $U$ by the depth of the error in the parse, deepest first

$G = \emptyset$                                          Transformation stage (Section 2.1.1)
for all errors $e \in U$ do
    $t = \underset{t \in \text{transformations}}{\text{argmax}} \text{ errors repaired by } t$
    $g$ = new error group
    Correct $e$ as specified by $t$
    for all errors $f$ that $t$ corrects do
        Remove $f$ from $U$
        Insert $f$ into $g$
    Add $g$ to $G$

for all groups $g \in G$ do                              Classification stage (Section 2.1.2)
    Classify $g$ based on properties of the group
```

errors is made, different types of errors may contribute to the same count, for example mistakes in PP attachment and coordination could both contribute to the count of incorrect NP nodes. In contrast, the approach we describe below takes into consideration the relations between node errors, grouping them into linguistically meaningful sets.

We classify node errors in two phases. First, we find a set of transformations that convert the output parse into the gold parse (Section 2.1.1). Second, each transformation is classified as one of several error types (Section 2.1.2). Pseudocode for our method is shown in Algorithm 2.1.

2.1.1 Transformations

Our algorithm finds a path from the parser output to the gold parse, where states are parse structures and steps are transformations. We define three general types of transformations:

Create node takes a set of nodes that are siblings and places them under a new node.

Delete node removes a node and re-attaches its children where it was.

Move nodes takes a node and moves it either up or down within the parse, with the constraint that it must not make the structure discontinuous. If this move leads to the creation of a unary production between two non-terminals of the same type, also remove one of them.

The first two of these are straightforward, to demonstrate the third we will describe the move transformation that fixes the errors in Figure 2.1. The PP node spanning *in 1986* is too low in the parser output (top half of the figure). We will make a move transformation, moving the PP up to be a child of the VP. By moving the PP, the extra nodes cover a shorter span of text and three of them now match the three missing nodes (shown in the bottom half of the figure). The remaining extra node is the lowest extra NP, which was originally over *Applied in 1986*. After *in 1986* moves up, an NP to NP unary production is left behind, and so one of the NPs is deleted.

Once a transformation is performed, all of the nodes that were fixed are placed in a single group and information about the nearby parse structure before and after the transformation is recorded. This recording is necessary, since subsequent changes may alter the surrounding structure in ways that impact the classification process described in the next section.

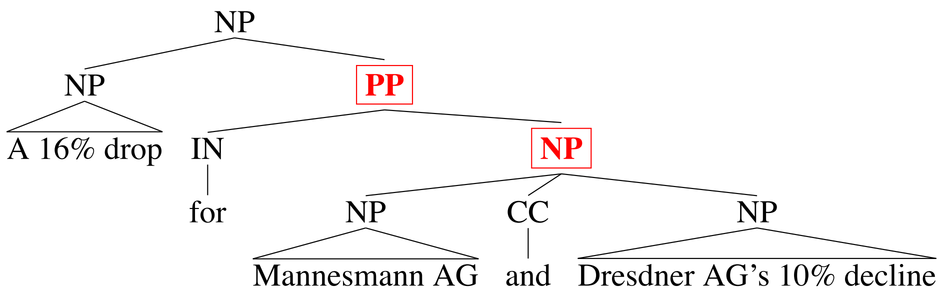
With this definition of states and transformations, a range of standard search algorithms are available to us. We considered search algorithms with guarantees for finding the shortest path, but found they were prohibitively slow. Instead, we find a path by applying a greedy bottom–up approach, iterating through the errors in the parse, deepest first. Fortunately, on the shorter sentences for which finding the optimal path was feasible, we found that the greedy approach had no substantial impact on results. This is the case because in many sentences a single correction will fix the parse and in cases with multiple errors they are often in disjoint sections of the parse.

2.1.2 Transformation Classification

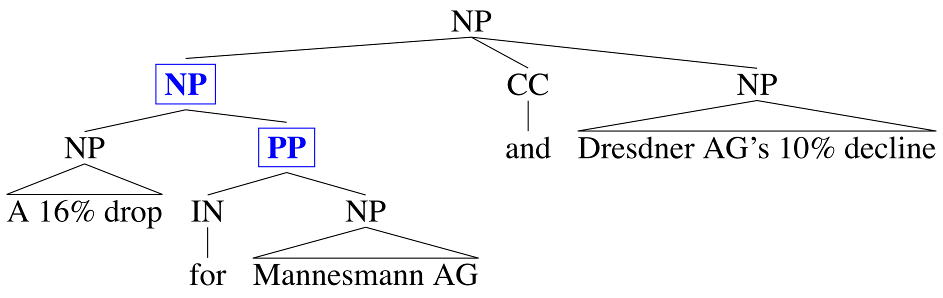
We began with a large set of node errors, in the first stage they were placed into groups, one group per transformation used to get from the automatically generated parse to the gold parse. Now we classify each group as one of the error types below. The ideal set of error classes would consider each group of incorrect constituents and show how their function has been misinterpreted by the parser, but it is difficult to define simple rules for such a classification. In future work it would be interesting to build statistical models for classifying errors, using expert analysis of parse errors to train the system. In the meantime, the approach we took was to construct general rules that classify errors mainly by the category of the nodes being moved and their surrounding context. We developed our rules by running the system with no classification, then progressively adding rules to decrease the number of unclassified errors. In the process of adding rules we also inspected observed classifications to check that they were correct, amending the rules if necessary.

PP Attachment (Figure 2.1)
Any case in which the transformation involved moving a Prepositional Phrase, or the incorrect bracket is over a PP, e.g.,
He was (VP *named chief executive officer of* (NP *Applied* (PP *in 1986*)))
where (PP *in 1986*) should modify the entire VP, rather than just *Applied*.

Coordination (Figure 2.2)
Cases in which a conjunction is an immediate sibling of the nodes being moved, or is the leftmost or rightmost node being moved, e.g.,
(NP *A 16% drop* (PP *for* (NP (NP *Mannesmann AG*) *and* (NP *Dresdner AG's 10% decline*))))



(a) Parser output



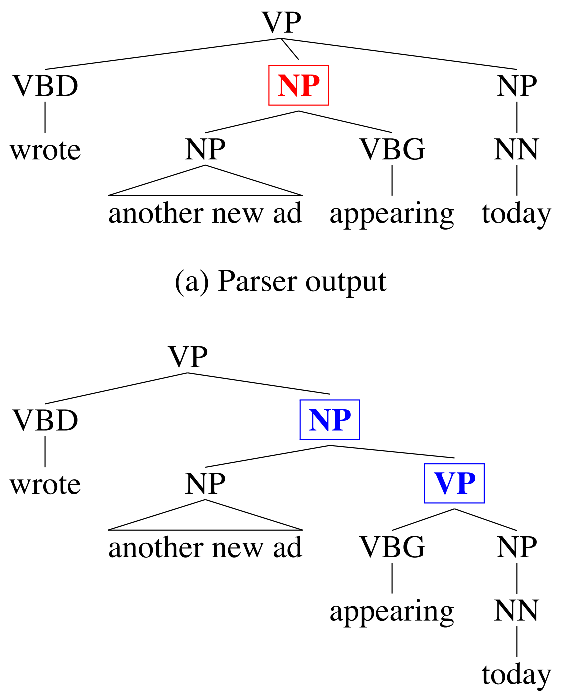
(b) Gold parse

Figure 2.2: **Coordination** *and Dresdner AG's 10% decline* is too low.

where the conjunction and second NP should be at the top level, between *A 16% drop for Mannesmann AG* and *Dresdner AG's 10% decline*.

NP Attachment (Figure 2.3)
Several cases in which NPs had to be moved, particularly for mistakes in appositive constructions and incorrect attachments within a verb phrase, e.g.,
The bonds (VP *go* (PP *on sale* (NP *Oct. 19*)))
where *Oct. 19* should be an argument of *go*.

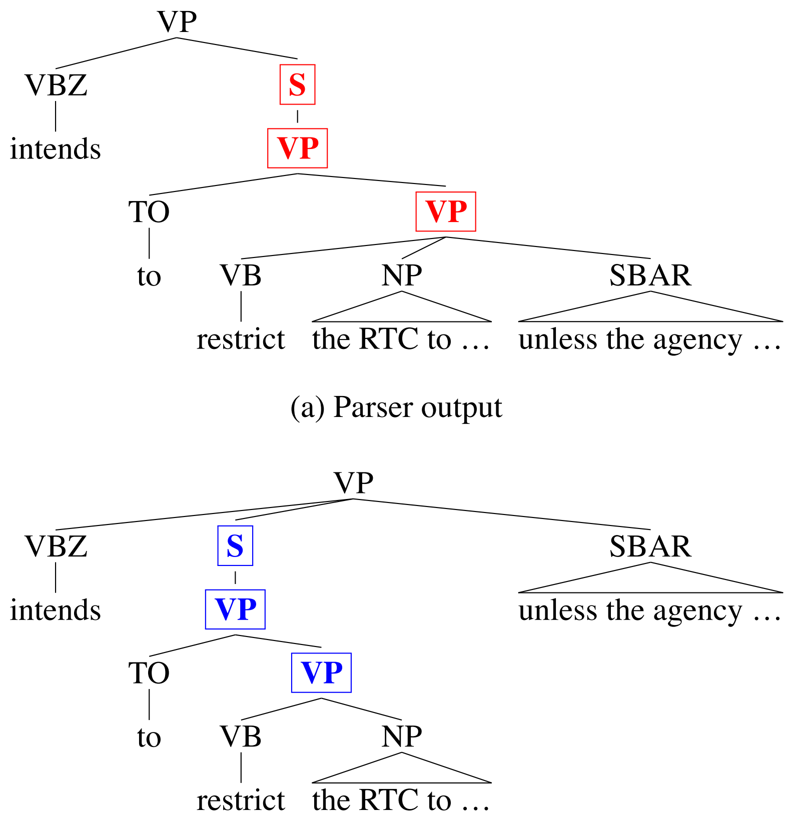
Clause Attachment (Figure 2.4)
Any group that involves movement of some form of S node, e.g.,
intends (S (VP *to* (VP *restrict the RTC to ...* (SBAR *unless the agency ...*))))
where the SBAR should be modifying *intends*, rather than the lower VP.



(a) Parser output

(b) Gold parse

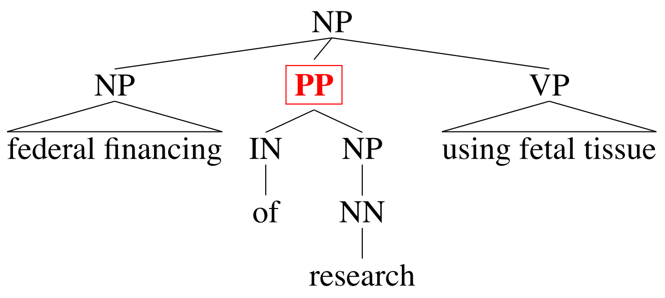
Figure 2.3: **NP Attachment** *today* is too high, it should be the argument of *appearing*, rather than *wrote*.



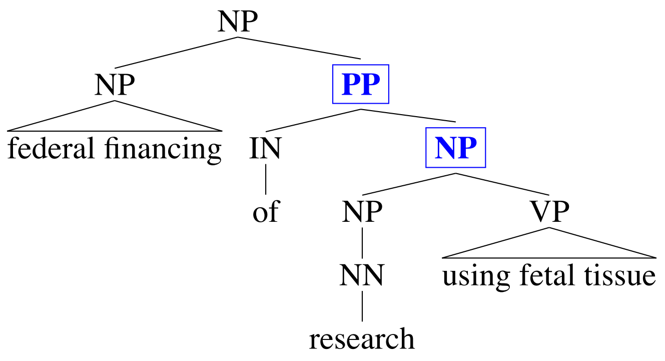
(a) Parser output

(b) Gold parse

Figure 2.4: **Clause Attachment** *unless the agency receives specific congressional authorization* is attaching too low.



(a) Parser output



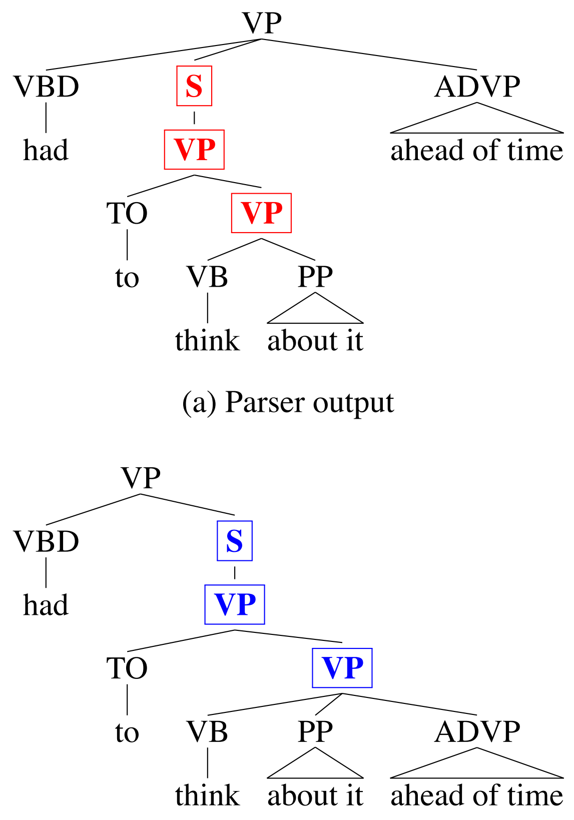
(b) Gold parse

Figure 2.5: **VP Attachment** *using fetal tissue* should modify *research*, not *financing*.

VP Attachment (Figure 2.5)
When the correction involves moving a VP, e.g.,
(NP (NP *federal financing*) (PP *of research*) (VP *using fetal tissue*))
where the VP should modify *research*, not *financing*.

Adverb and Adjective Modifier Attachment (Figure 2.6)
Cases involving incorrectly placed adjectives and adverbs, including errors corrected by mode movement and errors requiring only creation of a node, e.g.,
(NP (ADVP *even more*) *severe setbacks*)
where there should be an extra ADVP node over *even more severe*.

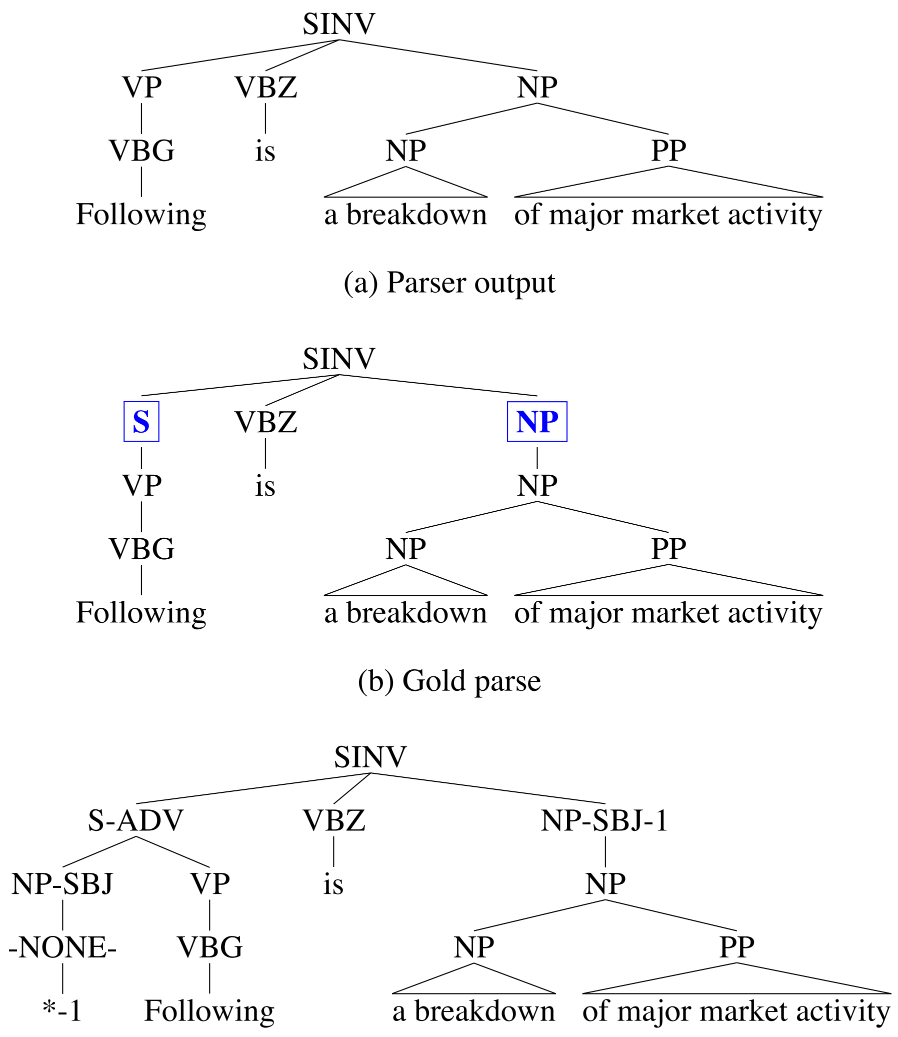
Unary (Figure 2.7)
Mistakes involving unary productions that are not linked to a nearby error such as a matching extra or missing node, e.g.,
(SINV (VP *following*) *is a breakdown of major market activity*)
where the VP should have an S above it. In this particular example we can see how not modeling traces may be hurting performance–the S is intended to span both the VP and a null NP that is co-indexed with *a breakdown of major market activity*.



(a) Parser output

(b) Gold parse

Figure 2.6: **Adverb and Adjective Modifier Attachment** *ahead of time* is too high, it should modify *think*, not *had*.

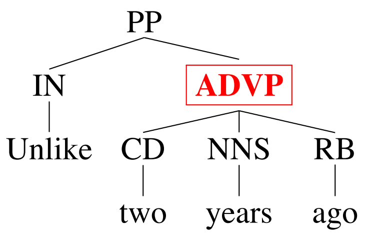


(a) Parser output

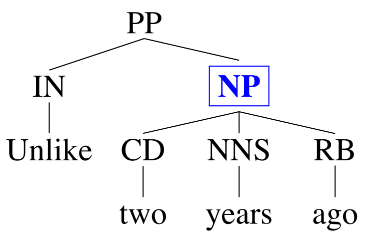
(b) Gold parse

(c) Gold parse with traces and function tags

Figure 2.7: Two **Unary** errors: a missing S and a missing NP. The third parse is the PTB parse before traces and function tags are removed, included here to show the distinction between the two NPs in the production.

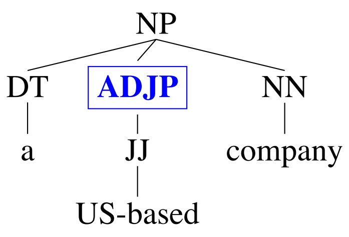


(a) Parser output

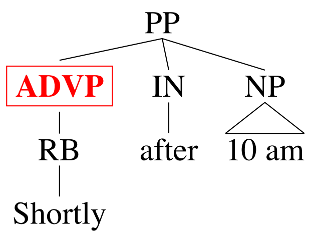


(b) Gold parse

Figure 2.8: **Different Label** *two years ago* should be an NP rather than an ADVP.



(a) Missing node in parser output



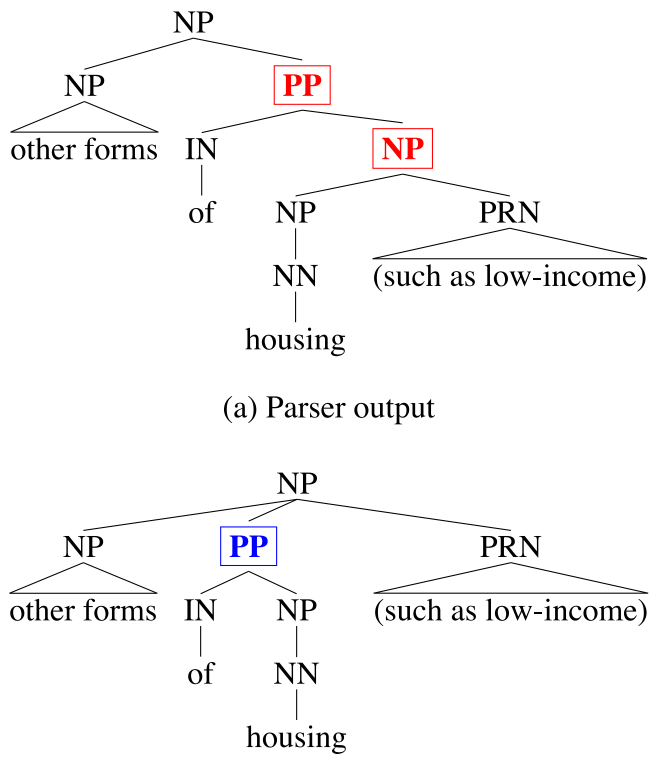
(b) Extra node in parser output

Figure 2.9: **Single Word Phrase** Two examples, one in which the parser missed a node spanning a single word, and one in which it had an extra node.

Different label (Figure 2.8)
In many cases a node is present in the parse that spans the correct set of words, but has the wrong label, in which case we group the two node errors, (one extra, one missing), as a single error, e.g.,
(PP *Unlike* (ADVP *two years ago*))
where the ADVP should be an NP.

Single word phrase (Figure 2.9)
Node errors that span a single word, e.g.,
(PP (ADVP *Shortly*) *after 10 a.m.*)
where the ADVP should not be present. We include checks to ensure this is not linked to another error, such as one part of a set of internal noun phrase errors.

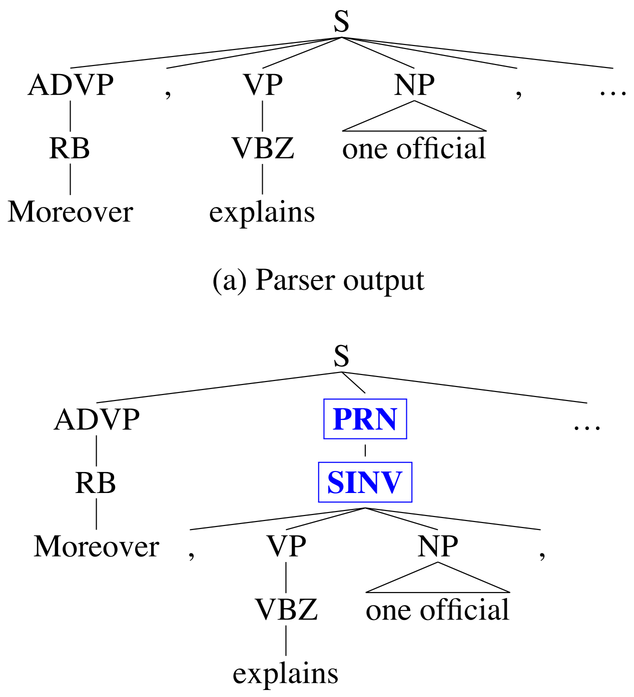
Parenthetical Attachment (Figure 2.10)
When a parenthetical node modifies at the wrong level, e.g.,
(NP *other forms* (PP *of* (NP *housing* (PRN *(such as low-income)*))))
where the parenthetical should be higher.



(a) Parser output

(b) Gold parse

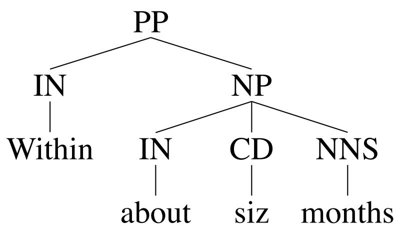
Figure 2.10: **Parenthetical Attachment** The parenthetical should be higher.



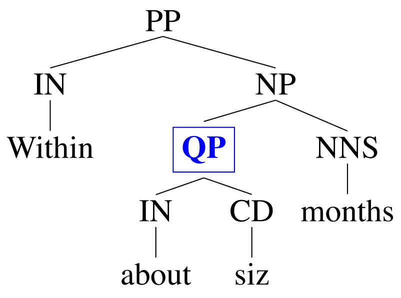
(a) Parser output

(b) Gold parse

Figure 2.11: **Missing Parenthetical** *, explains one official,* should be a parenthetical.



(a) Parser output



(b) Gold parse

Figure 2.12: **NP Internal Structure** *about six* should form a QP.

Missing Parenthetical (Figure 2.11)
When a parenthetical is entirely missing, e.g.,
(S *Moreover, explains one official, ...*)
where there should be a PRN around *, explains one official,*.

NP Internal Structure (Figure 2.12)
While most NP structure is not annotated in the PTB, there is some use of ADJP, NX, NAC and QP nodes. We form a single group for each NP that has one or more errors involving these types of nodes, e.g.,
(PP *Within* (NP *about six months*))
should have a QP over *about six*.

Other There is a long tail of other errors. Some could be placed within the categories above, but would require far more specific rules.

Working from only a raw list of node errors in a parse, it would be difficult to determine errors like those described above. Even for error types that can be measured by counting node errors or rule production errors, our approach has the advantage that we identify groups of errors with a single cause. For example, what appears as a missing unary production may correspond to an extra bracket that contains a node that attached incorrectly. By grouping node errors and classifying each group as a single mistake, we are able to more precisely characterize the mistakes a parser makes.

System	P	R	F	Exact	Speed
Enhanced Training / Systems					
Charniak S & R (SR)	92.44	91.70	92.07	44.87	1.8
Charniak Re-ranking (R)	91.78	91.04	91.41	44.04	1.8
Charniak Self-trained (S)	91.16	90.89	91.02	40.77	1.8
Standard Parsers					
Berkeley	90.30	89.81	90.06	36.59	4.2
Charniak	89.88	89.55	89.71	37.25	1.8
SSN	89.96	88.89	89.42	32.74	1.8
BUBS	88.57	88.43	88.50	31.62	27.6
Bikel	88.23	88.10	88.16	32.33	0.8
Collins-3	87.82	87.50	87.66	32.22	2.0
Collins-2	87.77	87.48	87.62	32.51	2.2
Collins-1	87.29	86.90	87.09	30.35	3.3
Stanford Lexicalized (L)	86.35	86.49	86.42	27.65	0.7
Stanford Unlexicalized (U)	86.48	85.09	85.78	28.35	2.7

Table 2.1: PARSEVAL results on WSJ section 23 for the parsers we consider. The columns are precision, recall, F-score, exact sentence match, and speed (sentences / sec). Coverage was left out as it was above 99.8% for all parsers.

2.2 Results

We used sections 00 and 24 of the WSJ section of the PTB as development data while constructing the transformation and error group classification methods. All of our examples in text come from these sections as well, but for all tables of results we ran our system on section 23. We chose to run our analysis on section 23 as it is the only section we are sure was not used in the development of any of the parsers, either for tuning or feature development. Our evaluation is entirely focused on the errors of the parsers, so unless there is a particular construction that is unusually prevalent in section 23, we are not revealing any information about the test set that could bias future work.

Our evaluation is over a wide range of PTB constituency parsers and their variants from the past twenty years. For all parsers we used the publicly available version, with the standard parameter settings.

Berkeley (Petrov, Barrett, et al. 2006; Petrov and Klein 2007). An unlexicalised parser with a grammar constructed with automatic state splitting.

Bikel (2004) Implementation of Collins (1997).

BUBS (Bodenstab et al. 2011; Dunlop et al. 2011). A 'grammar-agnostic constituent parser,' which uses a Berkeley Parser grammar, but parses with various pruning techniques to improve speed, at the cost of accuracy.

Charniak (2000) A generative parser with a maximum entropy-inspired model. We also use the reranker (-R; Charniak and Johnson 2005), and the self-trained model (-S; McClosky et al. 2006a).

Collins (1997) A generative lexicalized parser, with three models, a base model, a model that uses subcategorization frames for head words, and a model that takes into account traces.

SSN (Henderson 2003, 2004) A statistical left-corner parser, with probabilities estimated by a neural network.

Stanford (Klein and Manning 2003a,b) We consider both the unlexicalised PCFG parser (-U) and the lexicalized factored parser (-L), which combines the PCFG parser with a lexicalized dependency parser.

Table 2.1 shows the standard performance metrics, measured on section 23 of the WSJ, using all sentences. Speeds were measured using a Quad-Core Xeon CPU (2.33GHz 4MB L2 cache) with 16GB of RAM. These results clearly show the variation in parsing performance, but they do not show which constructions are the source of those variations.

Our system enables us to answer questions about parser behavior that could previously only be probed indirectly. We demonstrate its usefulness by applying it to a range of parsers (here), to reranked K-best lists of various lengths (§ 2.2.1), and to output for out-of-domain parsing (§ 2.2.2).

First, in Table 2.2, we focus on a detailed error breakdown for the best system we consider, the Charniak parser with a self-trained model and with reranking of the top fifty candidate parses produced by the parser (Charniak 2000; Charniak and Johnson 2005; McClosky et al. 2006a). For each of our error types, the table shows how many times it occurred in the 2,416 sentences in WSJ section 23 of the PTB (Occurrences), the total number of nodes involved in the groups that were classified as each error type (Nodes Involved), and the ratio between the two (Nodes / Occurrences).

The ratios show that some errors typically cause only a single node error, where as others, such as coordination, generally cause several. This means that considering counts of error groups would over-emphasize some error types, e.g., single word phrase errors are second most important by number of groups, but seventh by total number of node errors (not counting 'Unclassified', which corresponds to many different errors).

In Table 2.3 we consider the breakdown of parser errors on WSJ section 23. The shaded area of each bar indicates the frequency of parse errors (i.e., empty means fewest errors). The area filled in is determined by the average number of node errors per sentence that are attributed to that type of error. The average number of node errors per sentence for a completely full bar is indicated by the bottom row (Worst), and the value for a completely empty bar is indicated by the top row

Error Type	Occurrences	Nodes Involved	Ratio
PP Attachment	846	1455	1.7
Single Word Phrase	490	490	1.0
Clause Attachment	385	913	2.4
Adverb and Adjective Modifier Attachment	383	599	1.6
Different Label	377	754	2.0
Unary	347	349	1.0
NP Attachment	321	597	1.9
NP Internal Structure	299	352	1.2
Coordination	209	557	2.7
Unary Clause Label	185	200	1.1
VP Attachment	64	159	2.5
Parenthetical Attachment	31	74	2.4
Missing Parenthetical	12	17	1.4
Unclassified	655	734	1.1

Table 2.2: Breakdown of errors on section 23 for the Charniak parser with self-trained model and reranker. Errors are sorted by the number of times they occur. Ratio is the average number of node errors caused by each error we identify (i.e., Nodes Involved / Occurrences).

(Best). We use counts of node errors to make the contributions of each error type to F-score more interpretable.

As expected, PP attachment is the largest contributor to errors, across all parsers. Coordination is surprisingly low on the list (6th) given how significant the problem is considered in the community (McClosky et al. 2006b). This appears to indicate that parsers are better at coordination than at PP attachment, but the raw counts do not take into consideration the relative frequency of the two decisions. We can get a sense of how frequent these decisions are by counting CCs and PPs in sections 02–21 of the treebank: 16,844 and 95,581 respectively. These counts are only an indicator of the number of decisions as the nodes can be used in ways that do not involve a decision. Taking into consideration the 6:1 ratio of decisions, the 4:1 ratio of errors is less surprising, and instead supports the belief that coordination scope ambiguity is very difficult. However, it does appear that improvements in coordination accuracy will have a limited impact on our performance metrics. One surprisingly common error involves unary productions. Looking at the breakdown by unary type we found that clause labeling (S, SINV, etc) accounted for a large proportion of the errors.

Parser	F-score	PP Attach	Clause Attach	Diff Label	Mod Attach	NP Attach	Co-ord	1-Word Span	Unary	NP Int.	Other
Best		0.60	0.38	0.31	0.25	0.25	0.23	0.20	0.14	0.14	0.50
Charniak-RS	92.07										
Charniak-R	91.41										
Charniak-S	91.02										
Berkeley	90.06										
Charniak	89.71										
SSN	89.42										
BUBS	88.63										
Bikel	88.16										
Collins-3	87.66										
Collins-2	87.62										
Collins-1	87.09										
Stanford-L	86.42										
Stanford-U	85.78										
Worst		1.12	0.61	0.51	0.39	0.45	0.40	0.42	0.27	0.27	1.13

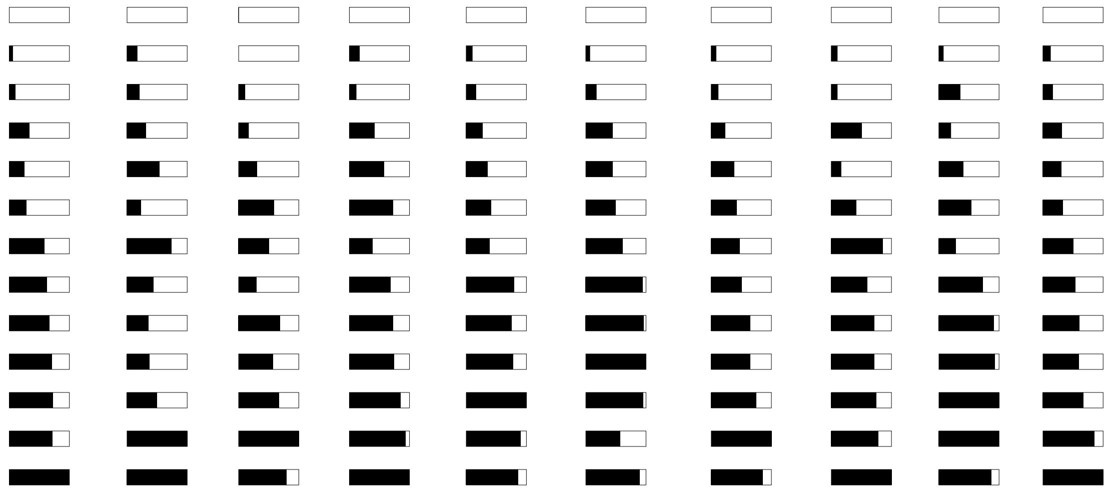


Table 2.3: Average number of bracket errors per sentence due to the top ten error types. For instance, Stanford-U produces output that has, on average, 1.12 bracket errors per sentence that are due to PP attachment. The scale for each column is indicated by the Best and Worst values.

By comparing the performance of the three Collins parsers, which were released in 1997, with more recent systems we can see where performance has and has not changed over the past fifteen years. There has been improvement across the board, but in some cases, e.g., clause attachment errors and different label errors, the change has been more limited (24% and 29% reductions respectively). It is difficult to know why these ambiguities are more difficult, but three possibilities are: (1) annotation errors may be more prevalent, making the potential for improvement vary, (2) they pose fundamental difficulties for the community's overall approach to parsing, (3) there may be biases in our automatic classification scheme. We investigated the breakdown of the different label errors by label, but no particular cases of label confusion stand out, and we found that the most common cases remained the same between Collins and the top results. Overall, it seems that these error types may have been unintentionally neglected in research and could be a productive area for future investigation.

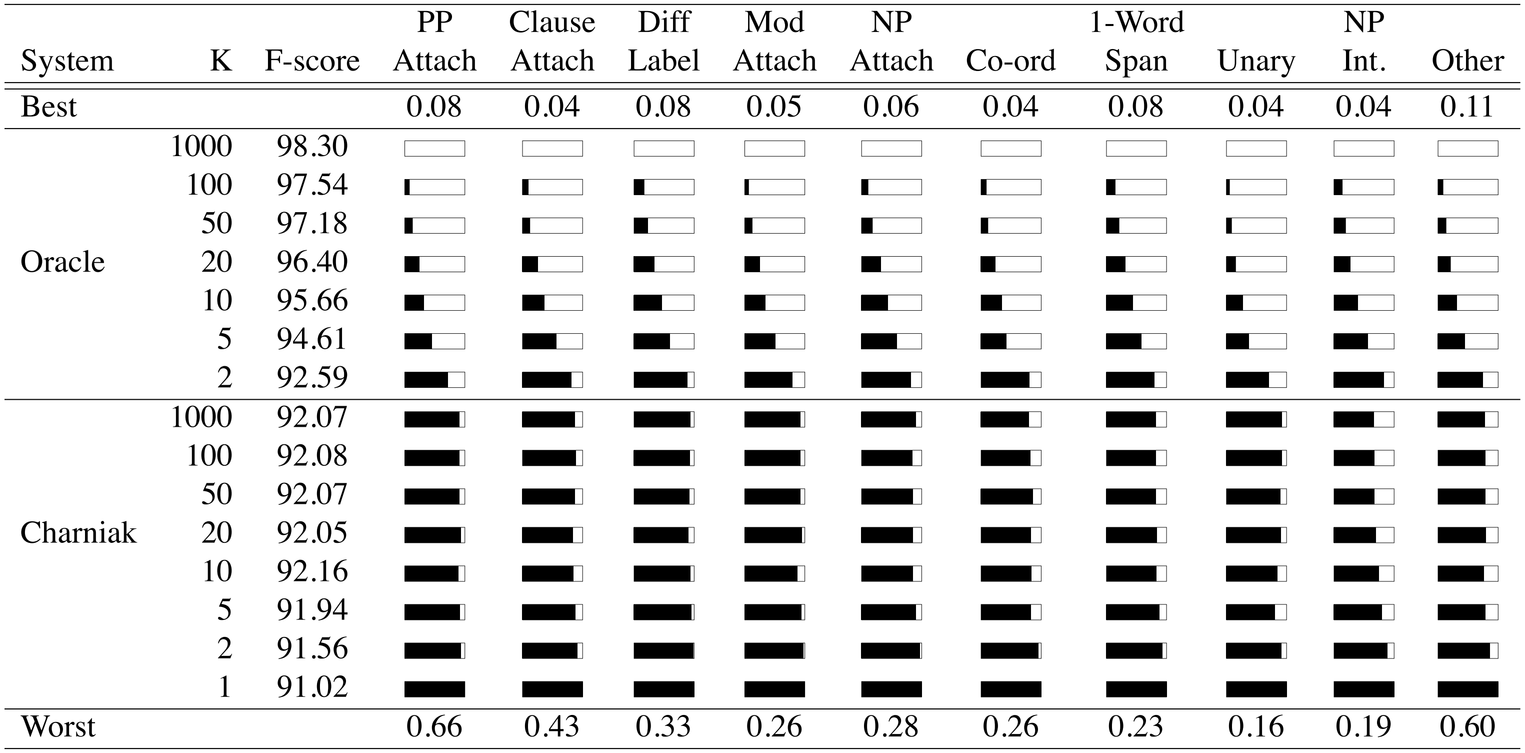
It is also interesting to compare pairs of parsers that share aspects of their architecture. One such pair is the Stanford parser, where the factored parser combines the unlexicalised parser with a lexicalized dependency parser. The main sources of the 0.64 gain in F-score are PP attachment and coordination.

Another interesting pair is the Berkeley parser and the BUBS parser, which uses a Berkeley grammar, but improves speed by pruning. The pruning methods used in BUBS are particularly damaging for PP attachment errors and unary errors.

Various comparisons can be made between Charniak parser variants (Charniak, Charniak-S, Charniak-R, and Charniak-RS). We discuss the reranker below (-R). For the self-trained model (Charniak-S), McClosky et al. (2006a) performed some error analysis, considering variations in F-score depending on the frequency of tags such as PP, IN and CC in sentences. Here we see gains on all error types, though particularly for clause attachment, modifier attachment and coordination, which fits with their observations.

2.2.1 Reranking

The standard dynamic programming approach to parsing limits the range of features that can be employed. One way to deal with this issue is to use a two-stage process. First, a modified version of the parser produces the top K parses (rather than just the 1-best). Then, a model with more sophisticated features chooses the best parse from the K-best list (Collins 2000). While re-ranking has led to gains in overall performance (Charniak and Johnson 2005), there has been limited analysis of how effectively rerankers are using the list of parses they are ranking. Recent work has explored this question in more depth, but focusing on how variation in inference and model parameters impacts performance on standard metrics (Huang 2008; Ng, Honnibal, et al. 2010; Auli and Lopez 2011; Ng and Curran 2012). We will explore the question from the perspective of observed errors, considering the potential for improvement using only the n-best list of parses, and how many errors a re-ranker actually avoids.



System	K	F-score	PP Attach	Clause Attach	Diff Label	Mod Attach	NP Attach	Co-ord	1-Word Span	Unary	NP Int.	Other
Best			0.08	0.04	0.08	0.05	0.06	0.04	0.08	0.04	0.04	0.11
	1000	98.30										
	100	97.54										
	50	97.18										
Oracle	20	96.40										
	10	95.66										
	5	94.61										
	2	92.59										
	1000	92.07										
	100	92.08										
	50	92.07										
Charniak	20	92.05										
	10	92.16										
	5	91.94										
	2	91.56										
	1	91.02										
Worst			0.66	0.43	0.33	0.26	0.28	0.26	0.23	0.16	0.19	0.60

Table 2.4: Average number of bracket errors per sentence for a range of K-best list lengths using the Charniak parser with reranking and the self-trained model. The oracle results are determined by taking the parse in each K-best list with the highest F-score.

In Table 2.4 we present a breakdown over error types for the Charniak parser, using the self-trained model and reranker. For both the top half and bottom half of the table, we first used the Charniak parser to create K-best lists of varying length (indicated by the K column). In the top half, we chose the parse from the list that has the highest F-score (this is an oracle because to do it we use the correct answer, which we have for this data)[1]. In the bottom half, we let the Charniak and Johnson (2005) reranker select the parse from the list, with no gold information. The table has the same columns as Table 2.3, but the ranges on the bars now reflect the min and max for these sets.

While there is improvement on all errors when using the reranker, there is very little additional gain beyond the first 5-10 parses (this may be easier to see by rotating the page containing the table). Even for the oracle results, most of the improvement occurs within the first 5-10 parses. The limited utility of extra parses for the reranker may be due to the importance of the base parser output probability feature (which, by definition, decreases within the K-best list). Another possibility is that there is less useful variation further down the K-best list. The utility is higher for the oracle, but we do not see greater improvement further down the list because those parses will be combinations of a set of variations in the parse that change the model probability only slightly, rather than providing useful variation.

Interestingly, the oracle performance considerably improves across all error types, even at the 2-best level. This indicates that the base parser model is not particularly biased against particular error types, as if it were we would expect that fixing it would require going further down the list of parses given as options. Focusing on the rows for $K = 2$ we can also see two interesting outliers. The PP attachment improvement of the oracle is considerably higher than that of the reranker, particularly compared to the differences for other errors, suggesting that the reranker lacks the features necessary to make the decision better than the parser. The other interesting outlier is NP internal structure, which continues to make improvements for longer lists, unlike the other error types.

2.2.2 Out-of-Domain

Parsing performance drops considerably when shifting outside of the domain a parser was trained on (Gildea 2001). Previously, Clegg and Shepherd (2005) evaluated parsers qualitatively on node types and rule productions, while Bender et al. (2011) designed a Wikipedia test set to evaluate parsers on dependencies representing ten specific linguistic phenomena.

To provide a deeper understanding of the errors arising when parsing outside of the newswire domain, we analyze performance of the Charniak parser with reranker and self-trained model on the eight parts of the Brown corpus (Marcus et al. 1993), and two parts of the Google Web corpus (Petrov and McDonald 2012). Table 2.5 shows statistics for the corpora. The variation in average sentence lengths skew the results for errors per sentence. To handle this we divide by the number of

[1] This result is not the absolute best possible result, as F-score does not factor over sentences and so it may be better to make slightly different choices. However, this is a close approximation.

Corpus	Description	Sentences	Av. Length
WSJ 23	Newswire	2416	23.5
Brown F	Popular	3164	23.4
Brown G	Biographies	3279	25.5
Brown K	General	3881	17.2
Brown L	Mystery	3714	15.7
Brown M	Science	881	16.6
Brown N	Adventure	4415	16.0
Brown P	Romance	3942	17.4
Brown R	Humor	967	22.7
Google Web	Blogs	1016	23.6
Google Web	E-mail	2450	11.9

Table 2.5: Variation in size and contents of the domains we consider. The variation in average sentence lengths skews the results for errors per sentences, and so in Table 2.6 we consider errors per word.

words to determine the results in Table 2.6, rather than by the number of sentences, as in previous figures.

There are several interesting features in the table. First, on the Brown datasets, while the general trend is towards worse performance on all errors, NP internal structure is a notable exception and in some cases PP attachment and unaries are as well.

In the other errors we see similar patterns across the corpora, except humor (Brown R), on which the parser is particularly bad at coordination and clause attachment. This makes sense, as the colloquial nature of the text includes more unusual uses of conjunctions, for example here *no mistake* is a phrase that is unlikely to be in the WSJ, and if it were, would probably be preceded by a verb like *make*:

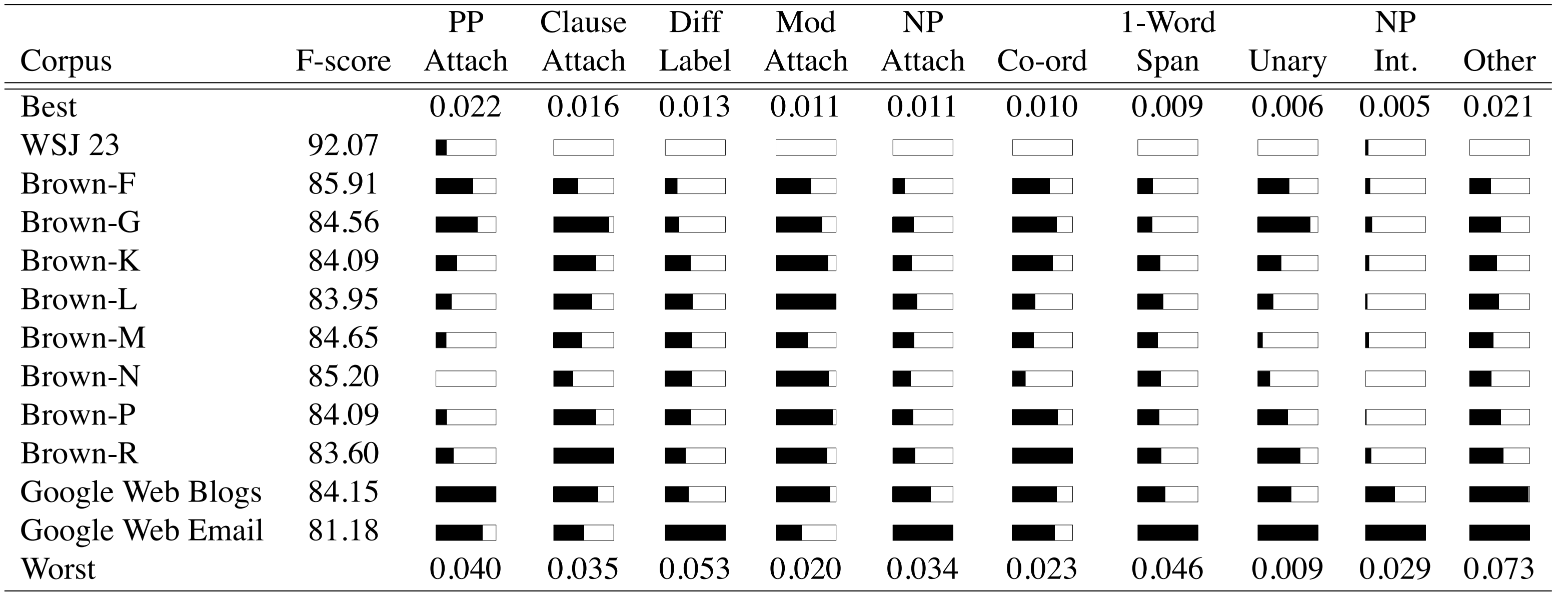
She was a living doll and no mistake – the ...

Comparing the Brown corpora and the Google Web corpora, there are much larger divergences. We see a particularly large decrease in NP internal structure. Looking at some of the instances of this error, it appears to be largely caused by incorrect handling of structures such as URLs and phone numbers, which do not appear in the PTB[2]. There are also some more difficult cases, for example:

... going up for sale in the next month or do .

where *or do* is a QP. The typographical error of *do* instead of *two* is extremely difficult to handle for a parser trained only on well-formed text.

[2]The corpus contains articles from 1989, pre-dating public internet access.



		PP	Clause	Diff	Mod	NP		1-Word		NP	
Corpus	F-score	Attach	Attach	Label	Attach	Attach	Co-ord	Span	Unary	Int.	Other
Best		0.022	0.016	0.013	0.011	0.011	0.010	0.009	0.006	0.005	0.021
WSJ 23	92.07										
Brown-F	85.91										
Brown-G	84.56										
Brown-K	84.09										
Brown-L	83.95										
Brown-M	84.65										
Brown-N	85.20										
Brown-P	84.09										
Brown-R	83.60										
Google Web Blogs	84.15										
Google Web Email	81.18										
Worst		0.040	0.035	0.053	0.020	0.034	0.023	0.046	0.009	0.029	0.073

Table 2.6: Average number of node errors per word for a range of domains using the Charniak parser with reranking and the self-trained model. We use per word error rates here rather than per sentence as there is great variation in average sentence length across the domains, skewing the per sentence results.

For e-mail there is a substantial drop on single word phrases. Breaking the errors down by label we found that the majority of the new errors are missing or extra NPs over single words. Here the main problem appears to be temporal expressions, though there also appear to be a substantial number of errors that are also at the POS level, such as when NNP is assigned to *ta* in this case:

... let you know that I 'm out ta here !

Some of these issues, such as URL handling, could be resolved with suitable training data. Other issues, such as ungrammatical language and unconventional use of words, pose a greater challenge.

2.3 Chinese Parsing

Aspects of Chinese syntax result in a distinctive mix of parsing challenges. However, the contribution of individual sources of error to overall difficulty is not well understood. We conduct a comprehensive automatic analysis of error types made by Chinese parsers, covering a broad range of error types for large sets of sentences, enabling the first empirical ranking of Chinese error types by their performance impact. To accommodate error classes that are absent in English, we augment the system to recognize Chinese-specific parse errors. To understand the impact of part-of-speech tagging errors on different error types, we also performed a part-of-speech ablation experiment, in which particular confusions are introduced in isolation. By analyzing the distribution of errors in the system output with and without gold part-of-speech tags, we are able to isolate and quantify the error types that can be resolved by improvements in POS tagging accuracy. Our analysis shows that improvements in tagging accuracy can only address a subset of the challenges of Chinese syntax. Further improvement in Chinese parsing performance will require research addressing other challenges, in particular, determining coordination scope.

2.3.1 Background

A decade of Chinese parsing research, enabled by the Penn Chinese Treebank (PCTB; Xue et al. 2005), has seen Chinese parsing performance improve from 76.7 F_1 (Bikel and Chiang 2000) to 84.1 F_1 (Qian and Liu 2012). While recent advances have focused on understanding and reducing the errors that occur in segmentation and part-of-speech tagging (Forst and Fang 2009; Jiang et al. 2009; Qian and Liu 2012), a range of substantial issues remain that are purely syntactic.

The closest previous work is the detailed manual analysis performed by Levy and Manning (2003). While their focus was on issues faced by their factored PCFG parser (Klein and Manning 2003b), the error types they identified are general issues presented by Chinese syntax in the PCTB. They presented several Chinese error types that are rare or absent in English, including noun/verb ambiguity, NP-internal structure and coordination ambiguity due to *pro*-drop. They attributed some of these differences to treebank annotation decisions and others to meaningful differences in syntax, suggesting that closing the English-Chinese parsing gap demands techniques beyond those currently used for English. Based on this analysis they considered how to modify their parser to capture the information necessary to model the syntax within the PCTB. However, as noted in their final section, their manual analysis of parse errors in one hundred sentences only covered a portion of a single parser's output, limiting the conclusions they could reach regarding the distribution of errors in Chinese parsing.

2.3.2 Adapting Error Analysis to Chinese

Our analysis builds on the system described in Sections 2.1.2 and 2.1.1, which finds the shortest path from the system output to the gold annotations, then classifies each transformation step into one of several error types. When directly applied to Chinese parser output, the system placed over 27% of the errors in the catch-all 'Other' type. Many of these errors clearly fall into one of a small set of error types, motivating an adaptation to Chinese syntax.

To adapt the system to Chinese, we developed a new version of the second stage of the system, which assigns an error category to each transformation step.

To characterize the errors the original system placed in the 'Other' category, we looked through one hundred sentences, identifying error types generated by Chinese syntax that the existing system did not account for. With these observations we were able to implement new rules to catch the previously missed cases, leading to the set shown in Table 2.7. To ensure the accuracy of our classifications, we alternated between refining the classification code and looking at affected classifications to identify issues. We also periodically changed the sentences from the development set we manually checked, to avoid over-fitting.

Where necessary, we also expanded the information available during classification. For example, we use the structure of the final gold standard parse when classifying errors that are a byproduct of sense disambiguation errors.

2.3.3 Chinese Parsing Errors

Table 2.7 presents the errors made by the Berkeley parser. Below we describe the error types that are either new in this analysis, have had their definition altered, or have an interesting distribution.

In all of our results we follow the same approach as earlier, presenting the number of bracket errors (missing or extra) attributed to each error type. As discussed in Section 2.2, for the purpose of understanding the cause of the remaining performance gap, bracket counts are more informative than a direct count of each error type, because the impact on PARSEVAL F-score varies between errors, e.g., a single attachment error can cause 20 bracket errors, while a unary error causes only one.

NP-Internal Structure (Figure 2.13)
The PCTB annotates more NP-internal structure than the PTB. We assign this error type when a transformation involves words whose parts of speech in the gold parse are one of: CC, CD, DEG, ETC, JJ, NN, NR, NT and OD.

We investigated the errors that fall into the NP-internal category and found that 49% of the errors involved the creation or deletion of a single pre-terminal phrasal bracket. These errors arise when a

Error Type	Brackets	% of total
NP-internal structure*	6019	22.70%
Coordination	2781	10.49%
Verb taking wrong args*	2310	8.71%
Unary	2262	8.53%
Adverb and adjective modifier attachment	1900	7.17%
One word span	1560	5.88%
Different label	1418	5.35%
Unary A-over-A	1208	4.56%
Wrong sense/bad attachment*	1018	3.84%
Noun boundary error*	685	2.58%
VP attachment	626	2.36%
Clause attachment	542	2.04%
PP attachment	514	1.94%
Split verb compound*	232	0.88%
Scope error*	143	0.54%
NP attachment	109	0.41%
Other	3186	12.02%

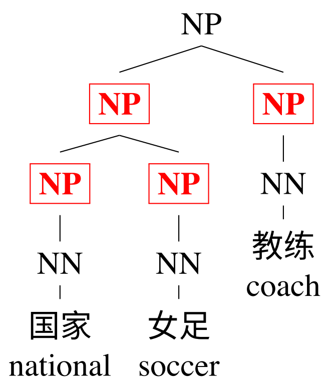
Table 2.7: Errors made when parsing Chinese. Values are the number of bracket errors attributed to that error type. The values shown are for the Berkeley parser, evaluated on the development set. * indicates error types that were added or substantially changed as part of the adaptation to Chinese.

parser proposes a parse in which POS tags (for instance, JJ or NN) occur as siblings of phrasal tags (such as NP), a configuration used by the PCTB bracketing guidelines to indicate complementation as opposed to adjunction (Xue et al. 2005).

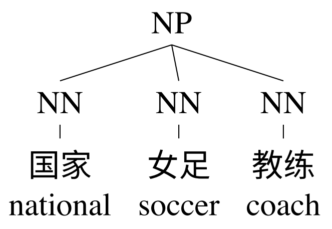
Adverb and Adjective Modifier Attachment (Figure 2.14)
Incorrect modifier scope caused by modifier phrase attachment level. This is less frequent in Chinese than in English: while English VP modifiers occur in pre- and post-verbal positions, Chinese only allows pre-verbal modification.

Wrong Sense / Bad Attachment (Figure 2.15)
This applies when the head word of a phrase receives the wrong POS, leading to an attachment error. This error type is common in Chinese because of POS fluidity, e.g., the well-known Chinese verb/noun ambiguity often causes mis-attachments that are classified as this error type.

In Figure 2.15, the word 投资 *invest* has both noun and verb senses. While the gold standard interpretation is the relative clause *firms that Macau invests in*, the parser returned an NP interpretation *Macau investment firms*.

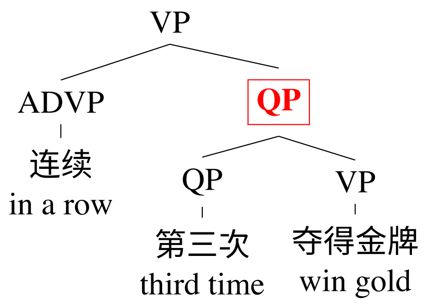


(a) Parser output

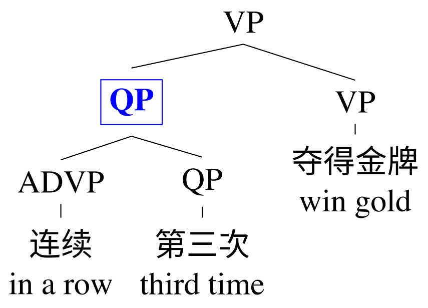


(b) Gold parse

Figure 2.13: **NP Internal Structure** This should be a flat structure.



(a) Parser output



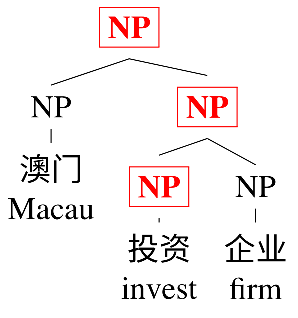
(b) Gold parse

Figure 2.14: **Modifier Attachment** 连续 *in a row* should modify only 第三次 *third time*.

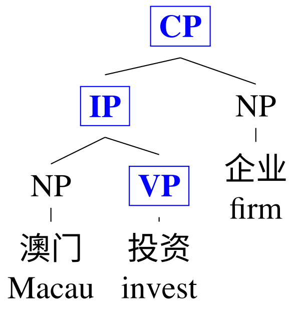
Verb taking wrong args (Figure 2.16)
This error type arises when a verb (e.g., 扭转 *reverse*) is hypothesized to take an incorrect argument (布什 *Bush* instead of 地位 *position*). Note that this also covers some of the errors that were classified as NP Attachment for English, changing the distribution for that type.

Unary
For mis-application of unary rules we separate out instances in which the two brackets in the production have the the same label (A-over-A). This case is created when traces are eliminated, a standard step in evaluation. More than a third of unary errors made by the Berkeley parser are of the A-over-A type. This can be attributed to two factors: (i) the PCTB annotates non-local dependencies using traces, and (ii) Chinese syntax generates more traces than English syntax, such as *pro*-drop, the omission of arguments where the referent is recoverable from discourse (Guo et al. 2007). However, for parsers that do not return traces they are a benign error.

Noun boundary error
In this error type, a span is moved to a position where the POS tags of its new siblings all belong to

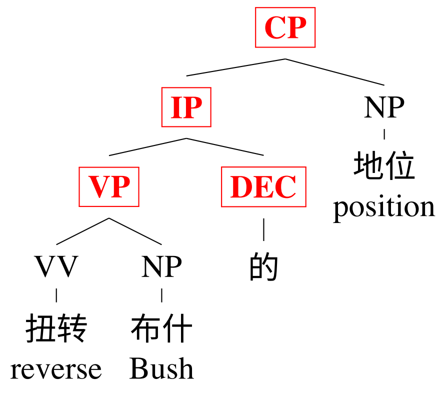


(a) Parser output

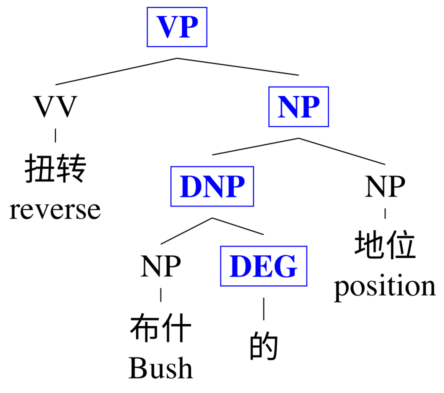


(b) Gold parse

Figure 2.15: **Sense Confusion** By treating 投资 *invest* as a noun, the parser forms an NP about a type of firm, rather than a clause about action by *Macau*.



(a) Parser output



(b) Gold parse

Figure 2.16: **Verb Taking Wrong Arguments** The verb takes the argument 布什 *Bush* too early, before it has been bound with 地位 *position*.

the list of NP-internal structure tags which we identified above, reflecting the inclusion of additional material into an NP.

Split verb compound

The PCTB annotations recognize several Chinese verb compounding strategies, such as the serial verb construction (规划建设 *plan [and] build*) and the resultative construction (煮熟 *cook [until] done*), which join a bare verb to another lexical item. We introduce an error type specific to Chinese, in which such verb compounds are split, with the two halves of the compound placed in different phrases.

Scope error
These are cases in which a new span must be added to more closely bind a modifier phrase (ADVP, ADJP, and PP), e.g.,
(IP (ADVP 仅 *Only*) (NP 去年 *last year,*) (NP 中国银行 *Bank of China*) ...),
where the ADVP should modify the first NP only, which can be indicated by adding an NP to form (NP (ADVP 仅 *Only*) (NP 去年 *last year*)).

PP attachment
This error type is rare in Chinese, as adjunct PPs are pre-verbal. It does occur near coordinated VPs, where ambiguity arises about which of the conjuncts the PP has scope over. Whether this particular case is PP attachment or coordination is debatable; we follow the approach above and label it PP attachment.

2.3.4 Chinese-English Comparison

It is difficult to directly compare error analysis results for Chinese and English parsing because of substantial changes in the classification method, and differences in treebank annotations.

As described in the previous section, the set of error categories considered for Chinese is very different to the set of categories for English. Even for some of the categories that were not substantially changed, errors may be classified differently because of cross-over between two categories (e.g., between Verb Taking Wrong Arguments and NP Attachment).

Differences in treebank annotations also present a challenge for cross-language error comparison. The most common error type in Chinese, NP-internal structure, is rare in the English results, but the datasets are not comparable because the PTB has much more limited NP-internal structure annotated than the PCTB. Further characterization of the impact of annotation differences on errors is beyond the scope of this work.

Three conclusions that can be made are that (i) coordination is a more common issue in Chinese, but remains difficult in both languages, (ii) PP attachment is a much greater problem in English, and (iii) substantial challenges are posed by the higher frequency of syntactic structures generating traces and null-elements in Chinese compared to English.

2.3.5 Cross-Parser Analysis

Table 2.7 showed the error types and their distribution for a single Chinese parser. Here we confirm that these are general trends, by showing that the same pattern is observed for several different parsers on the PCTB 6 dev set.[3] We include results for a range of parsers:

ZPAR (Y. Zhang and Clark 2009). A transition-based parser.

[3] We use the standard data split suggested by the PCTB 6 file manifest. As a result, our results differ from those previously reported on other splits.

Berkeley (Petrov, Barrett, et al. 2006; Petrov and Klein 2007). The same unlexicalised split-merge parser described in Section 2.2, but trained on Chinese instead of English. Here we also consider the product-parser version, which uses the same algorithm, but scoring with the product of multiple grammars trained with different random seeds (Petrov 2010). This variation leads to models that are different enough to combine productively similarly to system combination approaches. We label the product parser as Berk-2, as opposed to Berk-1.

Bikel (Bikel and Chiang 2000). A lexicalized parser.

Stanford (Klein and Manning 2003a,b; Levy and Manning 2003). The same parsers described in Section 2.2, but trained on Chinese instead of English.

These parsers represent a variety of parsing methods, though exclude some recently developed parsers that are not publicly available (Xiong et al. 2005; Qian and Liu 2012).

Comparing the two Stanford parsers in Table 2.8, the factored model provides clear improvements on sense disambiguation, but performs slightly worse on coordination.

We run the Berkeley product parser with only two grammars because we found, in contrast to the English results (Petrov 2010), that further grammars provided limited benefits. Comparing its performance with the standard Berkeley parser, it seems that the diversity in the grammars only assists certain error types. Most of the improvement in score is due to reductions in four categories: NP Internal Structure, Verb Arguments, Modifier Attachment, and Clause Attachment. Meanwhile, there is no improvement in two categories (Unary and Wrong Sense), and a slight decrease in three (One-Word Spans, Different Label, VP Attachment). It is difficult to confidently say why some errors are reduced and others are not, but it does imply that there are systematic biases that cannot be avoided by training slightly different models.

System	F$_1$	NP Int.	Coord	Verb Args	Unary	Mod. Attach	1-Word Span	Diff Label	Wrong Sense	Noun Edge	VP Attach	Clause Attach	PP Attach	Other
Best		1.54	1.25	1.01	0.76	0.72	0.21	0.30	0.05	0.21	0.26	0.22	0.18	1.87
Berk-G	86.8													
Berk-2	81.8													
Berk-1	81.1													
ZPAR	78.1													
Bikel	76.1													
Stan-L	76.0													
Stan-U	70.0													
Worst		3.94	1.75	1.73	1.48	1.68	1.06	1.02	0.88	0.55	0.50	0.44	0.44	4.11

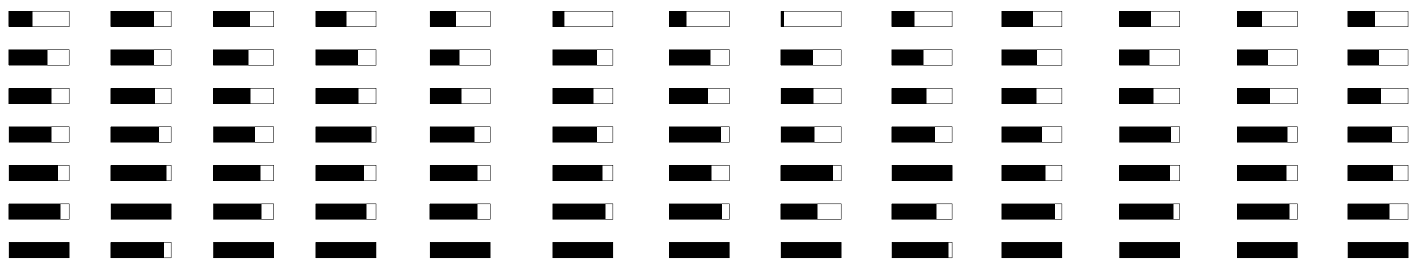


Table 2.8: Error breakdown for the development set of PCTB 6. The area filled in for each bar indicates the average number of bracket errors per sentence attributed to that error type, where an empty bar is no errors and a full bar has the value indicated in the bottom row. The parsers are: the Berkeley parser with gold POS tags as input (Berk-G), the Berkeley parser (Berk-1; Petrov, Barrett, et al. 2006; Petrov and Klein 2007), the Berkeley product parser with two grammars (Berk-2; Petrov 2010), ZPAR Y. Zhang and Clark (2009), the Bikel parser (Bikel and Chiang 2000), the Stanford Factored parser (Stan-L; Klein and Manning 2003b; Levy and Manning 2003), and the Stanford Unlexicalized PCFG parser (Stan-U; Klein and Manning 2003a).

Confused tags		Errors	Δ F_1
VV	NN	1055	-2.72
DEC	DEG	526	-1.72
JJ	NN	297	-0.57
NR	NN	320	-0.05

Table 2.9: The most frequently confused POS tag pairs. Each Δ F_1 is relative to Berk-G.

2.3.6 Tagging Error Impact

The challenge of accurate POS tagging in Chinese has been a major part of several recent papers (Forst and Fang 2009; Jiang et al. 2009; Qian and Liu 2012). The Berk-G row of Table 2.8 shows the performance of the Berkeley parser when given gold POS tags.[4] While the F_1 improvement is unsurprising, for the first time we can clearly show that the gains are only in a subset of the error types. In particular, tagging improvement will not help for two of the most significant challenges: coordination scope errors, and verb argument selection.

To see which tagging confusions contribute to which error reductions, we adapt the POS ablation approach of Tse and Curran (2012). We consider the POS tag pairs shown in Table 2.9. To isolate the effects of each confusion we start from the gold tags and introduce the output of the Stanford tagger whenever it returns one of the two tags being considered.[5] We then feed these "semi-gold" tags to the Berkeley parser, and run the fine-grained error analysis on its output.

VV/NN This confusion has been consistently shown to be a major contributor to parsing errors (Levy and Manning 2003; Qian and Liu 2012; Tse and Curran 2012), and we find a drop of over 2.7 F_1 when the output of the tagger is introduced. We found that while most error types have contributions from a range of POS confusions, verb/noun confusion was responsible for virtually all of the noun boundary errors corrected by using gold tags.

DEG/DEC This confusion between the relativizer and subordinator senses of the particle 的 *de* is the primary source of improvements on modifier attachment when using gold tags.

NR/NN and JJ/NN Despite their frequency, these confusions have little effect on parsing performance. Even within the NP-internal error type their impact is limited, and almost all of the errors do not change the logical form.

[4]We used the Berkeley parser as it was the best of the parsers we considered. Note that the Berkeley parser occasionally prunes all of the parses that use the gold POS tags, and so returns the best available alternative. This leads to a POS accuracy of 99.35%, which is still well above the parser's standard POS accuracy of 93.66%.

[5]We introduce errors to gold tags, rather than removing errors from automatic tags, isolating the effect of a single confusion by eliminating interaction between tagging decisions.

2.4 Summary

The single F-score objective over brackets or dependencies obscures important differences between statistical parsers. For instance, one or many mismatched brackets could be caused by a single attachment error.

In Section 2.1, we presented a novel transformation-based methodology for evaluating parsers that categorizes errors into linguistically meaningful types. Using this approach, we presented the first detailed examination of the errors produced by a wide range of constituency parsers for English and Chinese. We found that PP attachment and clause attachment are the most challenging constructions in English, while coordination turns out to be less problematic than previously thought. We also noted interesting variations in error types for parser variants.

We investigated the errors resolved in reranking, and introduced by changing domains. We found that the Charniak reranker improved most error types, but made little headway on improving PP attachment. Changing domain has an impact on all error types, except NP internal structure.

We also quantified the relative impacts of a comprehensive set of error types in Chinese parsing. Our analysis has shown that while improvements in Chinese POS tagging can make a substantial difference for some error types, it will not address two high-frequency error types: incorrect verb argument attachment and coordination scope. The frequency of these two error types is also unimproved by the use of products of latent variable grammars. These observations suggest that resolving the core challenges of Chinese parsing will require new developments that suit the distinctive properties of Chinese syntax.

We released our system so that future constituent parsers could be evaluated using our methodology (see Appendix A). Our analysis provides new insight into the development of parsers over the past two decades, and the challenges that remain.

Chapter 3

Formalism Conversion

A preliminary version of this chapter appeared as Kummerfeld, Klein, et al. (2012).

In the previous chapter, we explored error analysis of PTB parser output, developing a new tool and applying it to understand the mistakes systems make. However, not all parsers are designed to produce PTB-style phrase structure parses; many have been developed for other grammar formalisms. In our effort to understand the relative performance of different parsers, we turned to the task of automatic parse conversion to enable comparison of systems across formalisms.

In this chapter, we propose an improved, bottom-up method for converting one syntactic representation, CCG, into another, PTB. In contrast with past work (Clark and Curran 2009), which used simple rules based on category pairs, our approach follows the generalizations of CCG, assigning richer rules to individual categories and defining general methods of combining them for each of the CCG combinators. Our conversion preserves more sentences under round-trip conversion (51.1% vs. 39.6%) and is more robust. In particular, unlike past methods, ours does not require ad-hoc rules over non-local features, and so could be integrated into a parser.

3.1 Background

There has been extensive work on converting parser output for evaluation, e.g., Lin (1998) and Briscoe et al. (2002) proposed using underlying dependencies for evaluation. There has also been work on conversion to phrase structure, from dependencies (Xia and Palmer 2001; Xia, Rambow, et al. 2009) and from lexicalized formalisms, e.g., HPSG (Matsuzaki and Tsujii 2008) and TAG (Chiang 2000; Sarkar 2001). Our focus is on CCG to PTB conversion (Clark and Curran 2009).

3.1.1 Combinatory Categorial Grammar (CCG)

The lower half of Figure 3.1 shows a CCG derivation (Steedman 2000) in which each word is assigned a *category*, and *combinatory rules* are applied to adjacent categories until only one remains. Categories can be atomic, e.g., the N assigned to *magistrates*, or complex functions of the form *result / arg*, where *result* and *arg* are categories and the slash indicates the argument's directionality.

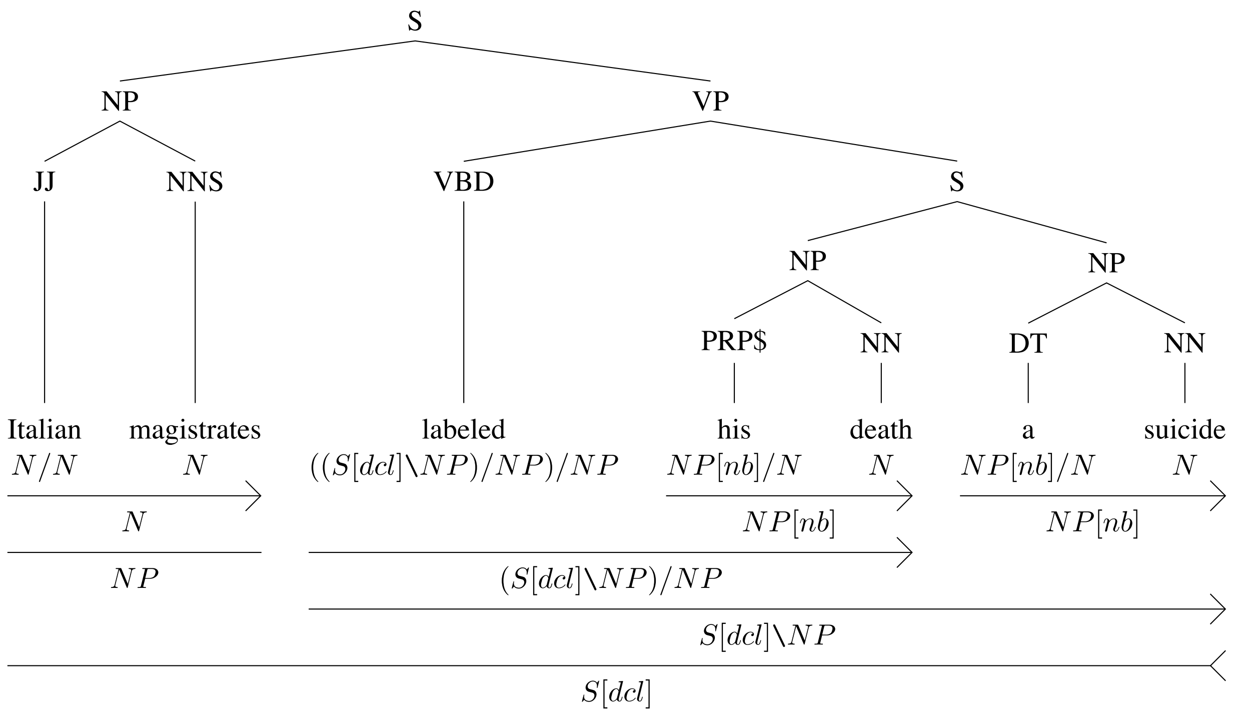


Figure 3.1: An example of CCG and PTB parses with nodes covering crossing spans: *his death a suicide* (PTB) and *labeled his death* (CCGbank).

Combinators define how adjacent categories can combine. Figure 3.1 uses *function application*, where a complex category consumes an adjacent argument to form its result, e.g., $S[dcl]\backslash NP$ combines with the NP to its left to form an $S[dcl]$. More powerful combinators allow categories to combine with greater flexibility.

We cannot form a PTB tree by simply relabeling the categories in a CCG derivation for two reasons. First, the mapping from categories to phrase labels is many-to-many, so it would be difficult to determine the correct label (though other researchers have tried, as discussed in Section 3.1.3). Second, there are some spans that occur only in the CCG derivation and others that occur only in the PTB parse, e.g., in Figure 3.1 there is a node for *his death a suicide* in PTB but not CCG, and vice versa for *labeled his death*. This second issue is particularly difficult because in some cases, including the example given, the differing nodes in the two parses cover spans of the sentence that cross[1]. These differences are the result of conscious decisions made in the construction of CCGbank, in many cases enabling the derivation to encode extra dependencies that are either implicit or expressed via traces in the PTB.

[1]This means a local insertion or deletion of a node cannot make the necessary changes, there would have to be several simultaneous changes to go from one structure to the other.

Categories	Schema
N	create an NP
$((S[dcl]\backslash NP)/NP)/NP$	create a VP
$N/N + N$	place left under right
$NP[nb]/N + N$	place left under right
$((S[dcl]\backslash NP)/NP)/NP + NP$	place right under left
$(S[dcl]\backslash NP)/NP + NP$	place right under left
$NP + S[dcl]\backslash NP$	place both under S

Table 3.1: Example C&C-Conv lexical and rule schemas.

3.1.2 Clark and Curran (2009)

Clark and Curran (ibid.), hereafter C&C-Conv, assign a *schema* to each leaf (lexical category) and rule (pair of combining categories) in the CCG derivation. The PTB tree is constructed from the CCG bottom-up, creating leaves with lexical schemas, then merging/adding sub-trees using rule schemas at each step.

The schemas for Figure 3.1 are shown in Table 3.1. These apply to create NPs over *magistrates*, *death*, and *suicide*, and a VP over *labeled*, and then combine the trees by placing one under the other at each step, and finally create an S node at the root.

C&C-Conv has sparsity problems, requiring schemas for all valid pairs of categories — at a minimum, the 2853 unique category combinations found in CCGbank. Clark and Curran (ibid.) create schemas for only 776 of these, handling the remainder with approximate catch-all rules.

C&C-Conv only specifies one simple schema for each rule (pair of categories). This appears reasonable at first, but frequently causes problems, e.g.,:

$$(N/N)/(N/N) + N/N$$

(1) "more than" + "30"
(2) "relatively" + "small"

Here either a QP bracket (1) or an ADJP bracket (2) should be created. Since both examples involve the same rule schema, C&C-Conv would incorrectly process them in the same way. To combat the most glaring errors, C&C-Conv manipulates the PTB tree with ad-hoc rules based on non-local features over the CCG nodes being combined — an approach that cannot be easily integrated into a parser.

These disadvantages are a consequence of failing to exploit the generalizations that CCG combinators define. We return to this example below to show how our approach does exploit those generalizations and thereby handles both cases correctly.

3.1.3 X. Zhang et al. (2012)

X. Zhang et al. (ibid.) considered a statistical approach to conversion from CCG to PTB. Their system works by classifying each step in the CCG derivation as either being a PTB node or being

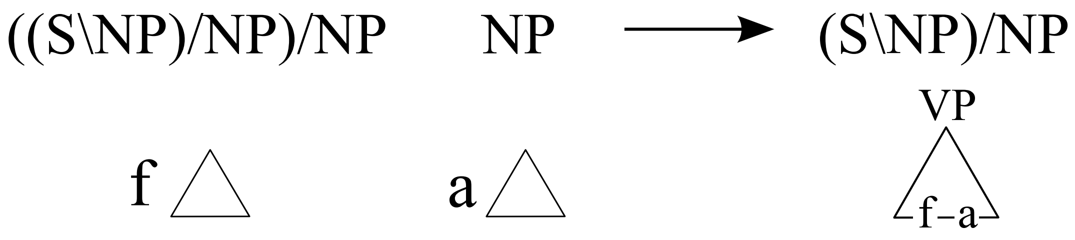


Figure 3.2: An example function application. Top row: CCG rule. Bottom row: applying instruction (VP f a).

Symbol	Meaning	Example Instruction
(X f a)	Add an X bracket around functor and argument	(VP f a)
{ }	Flatten enclosed node	(N f {a})
X*	Use same label as argument or default to X	(S* f {a})
f_i	Place subtrees	(PP f_0 (S $f_{1..k}$ a))

Table 3.2: Types of operations in instructions.

a dummy node, to be flattened in the final structure. This approach has the limitation that it cannot handle sentences where the structure of the CCG derivation is missing nodes that exist in the structure of the PTB parse, such as the example in Figure 3.1. These cases account for 10% of sentences, but the method is still fairly effective because they are able to recover most of the structure for such sentences.

3.2 Our Approach

Our conversion assigns a set of instructions to each lexical category and defines generic operations for each combinator that combine instructions. Figure 3.2 shows an example of a common form of instruction *(VP f a)*, which specifies three things:

- Create a new VP node, indicated by the VP and brackets
- Place the PTB parse for the functor under the new node, on the left, indicated by the position of the f
- Place the PTB parse for the argument under the new node, on the right, indicated by the position of the a

Table 3.2 shows all of the operators we define. For convenience, in this description we will refer to the parses coming from the functor and argument as sub-parses. The first operator is the one described above, though note that it can be more powerful, constructing any set of PTB nodes indicated with standard bracket notation plus the two markers for the functor and argument. The

{ } operator indicates that a flattened version of that sub-parse should be placed, in the example in the table the top node in the sub-parse for the argument would be deleted so its children are placed instead. The * operator takes the label for the new node from the argument sub-parse, e.g., the result of an $S\backslash S_1$ should correspond to the label of argument 1 (SINV, SBAR, etc, which all appear as S in CCG). This is necessary because CCGbank uses additional annotations on categories to distinguish cases that are assigned entirely different labels in the PTB. The final operator enables breaking up children in a sub-parse and placing them at different locations in the structure being formed. This only makes sense when creating multiple nodes, a PP and an S in the example, and uses subscripts to indicate which sub-parse to place where. Finally, complex categories with multiple arguments are assigned a list of instructions, one per argument.

The lists of rules are processed by following the CCG derivation, taking actions depending on the type of combinator used:

Lexical Assignment The initial instruction list for a category is taken from our hand-annotated set.

Function Application The next rule in the functor's list is applied and any remaining rules in its list are retained as the new rule set (any rules remaining in the list for the argument are discarded).

$$\begin{array}{ccc} X/Y & Y & \Rightarrow & X \\ Y & X\backslash Y & \Rightarrow & X \end{array}$$

Function Composition The unapplied instructions of the argument are combined with the remaining steps for the functor.

$$\begin{array}{ccc} X/Y & Y/Z & \Rightarrow & X/Z \\ Y\backslash Z & X\backslash Y & \Rightarrow & X\backslash Z \end{array}$$

Crossed Composition Only backwards crossed composition is used (the lower of the two). The new instruction list is composed of the top rule from the left, followed by all but the top rule from the right.

$$\begin{array}{ccc} X/Y & Y\backslash Z & \Rightarrow & X\backslash Z \\ Y/Z & X\backslash Y & \Rightarrow & X/Z \end{array}$$

Type Raising A new list of instructions is taken from our hand-annotated set. If necessary, the current structure is flattened to prevent a unary production of identical symbols being formed when the next instruction is applied.

$$\begin{array}{cc} X \Rightarrow & T/(T\backslash X) \\ X \Rightarrow & T\backslash(T/X) \end{array}$$

Coordination In CCG, coordination is a single step, but for implementation in parsing it is typically broken into two steps. We follow the derivation, doing the core structural construction in the second step. The instructions retained after the second step are from the right conjunct.

Functional Substitution These combinators occur rarely in the treebank and are not used in current parsers, so we do not implement them.

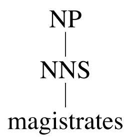
Additionally, we vary the instructions assigned based on the POS tag in 32 cases, and for the word *not*, to recover distinctions not captured by CCGbank categories alone. In 52 cases the later instructions depend on the structure of the argument being picked up, often to capture different handling of adverbial and adjectival phrases. For the non-combinatory binary and unary rules in

Category	Instruction set
N	(NP f)
N/N_1	(NP f {a})
$NP[nb]/N_1$	(NP f {a})
$((S[dcl]\backslash NP_3)/NP_2)/NP_1$	(VP f a)
	(VP {f} a)
	(S a f)

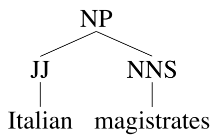
Table 3.3: Instruction sets for the categories in Figure 3.1.

CCGbank we define twenty-eight special instruction sets.

For the example from the previous section we begin by assigning the instructions shown in Table 3.3. Some of these can apply immediately as they do not involve an argument, e.g., *magistrates* has (NP f), producing:

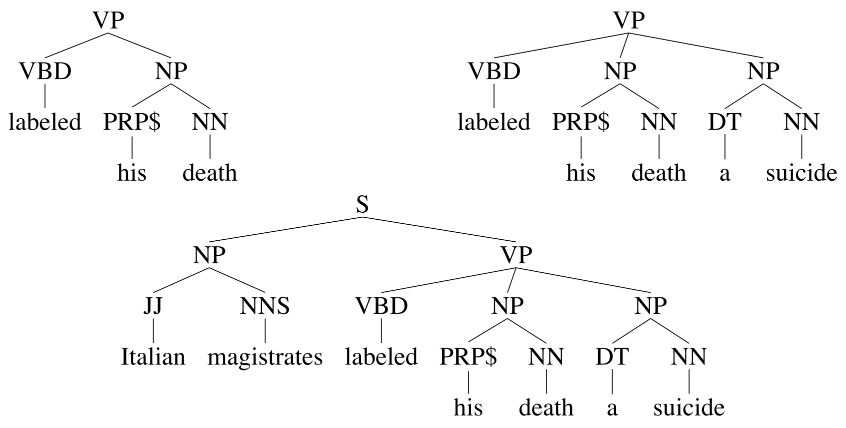


A slightly more complicated instruction is applied to *Italian*: (NP f {a}). This creates a new NP bracket, inserts the functor's tree, and flattens and inserts the argument's tree, producing:



Similar operations apply for the other NPs, followed by a unary rule, mapping N to NP. We handle this by applying one of the special cases for non-combinatory unary operations in CCGbank.

The next three steps all involve instructions from the verb's list. The verb takes three arguments, and so has three instructions applied as follows:



		Length ≤ 40				All lengths			
System	Data	P	R	F	Sent.	P	R	F	Sent.
C&C-Conv	00	94.39	95.85	95.12	42.1	93.67	95.37	94.51	39.6
	23	94.04	95.44	94.73	41.9	93.95	95.33	94.64	39.7
Zhang, et al. (2012)	00	97.09	95.40	96.24	–	96.92	94.82	95.86	–
	23	96.69	94.79	95.73	–	96.67	94.77	95.71	–
This work	00	96.98	96.77	96.87	53.6	96.69	96.58	96.63	51.1
	23	96.57	96.21	96.39	53.8	96.49	96.11	96.30	51.4

Table 3.4: PARSEVAL Precision, Recall, F-Score, and exact sentence match for converted gold CCG derivations.

The final tree in this case is almost correct but omits the S bracket around the two NPs on the right. To fix it we could have modified *labeled*'s second instruction to move *his death* under a new S via the final symbol in Table 3.2. However, for this particular construction the PTB annotations are inconsistent, and so we cannot recover without introducing more errors elsewhere.

Our approach naturally handles our QP vs. ADJP example from the previous section because the two cases have different lexical categories:

than	$((N/N)/(N/N))\backslash(S[adj]\backslash NP)$
relatively	$(N/N)/(N/N)$

This lexical difference means we can assign different instructions to correctly recover the QP and ADJP nodes, whereas C&C-Conv applies the same schema in both cases because after the first function application for *than*, the category becomes the same as the category for *relatively*.

3.3 Evaluation

Using sections 00-21 of the treebanks, we hand-crafted instructions for 527 lexical categories, a process that took under 100 hours, and includes all the categories used by the C&C parser. There are 647 further categories and 35 non-combinatory binary rules in sections 00-21 that we did not annotate. For unannotated categories, we use the instructions of the result category with an added instruction.

Table 3.4 compares our approach with C&C-Conv and X. Zhang et al. (2012) on gold CCG derivations. The results shown are as reported by EVALB (Black et al. 1991) using the Collins (1997) parameters. Our approach leads to increases over C&C-Conv on all metrics of at least 1.1%, and increases exact sentence match by over 11% (both absolute).

Many of the remaining errors relate to missing and extra clause nodes and a range of rare structures, such as QPs, NACs, and NXs. The only other prominent errors are single word spans, e.g., extra or missing ADVPs. Many of these errors are unrecoverable from CCGbank, either because inconsistencies in the PTB have been smoothed over or because they are genuine but rare constructions that were lost.

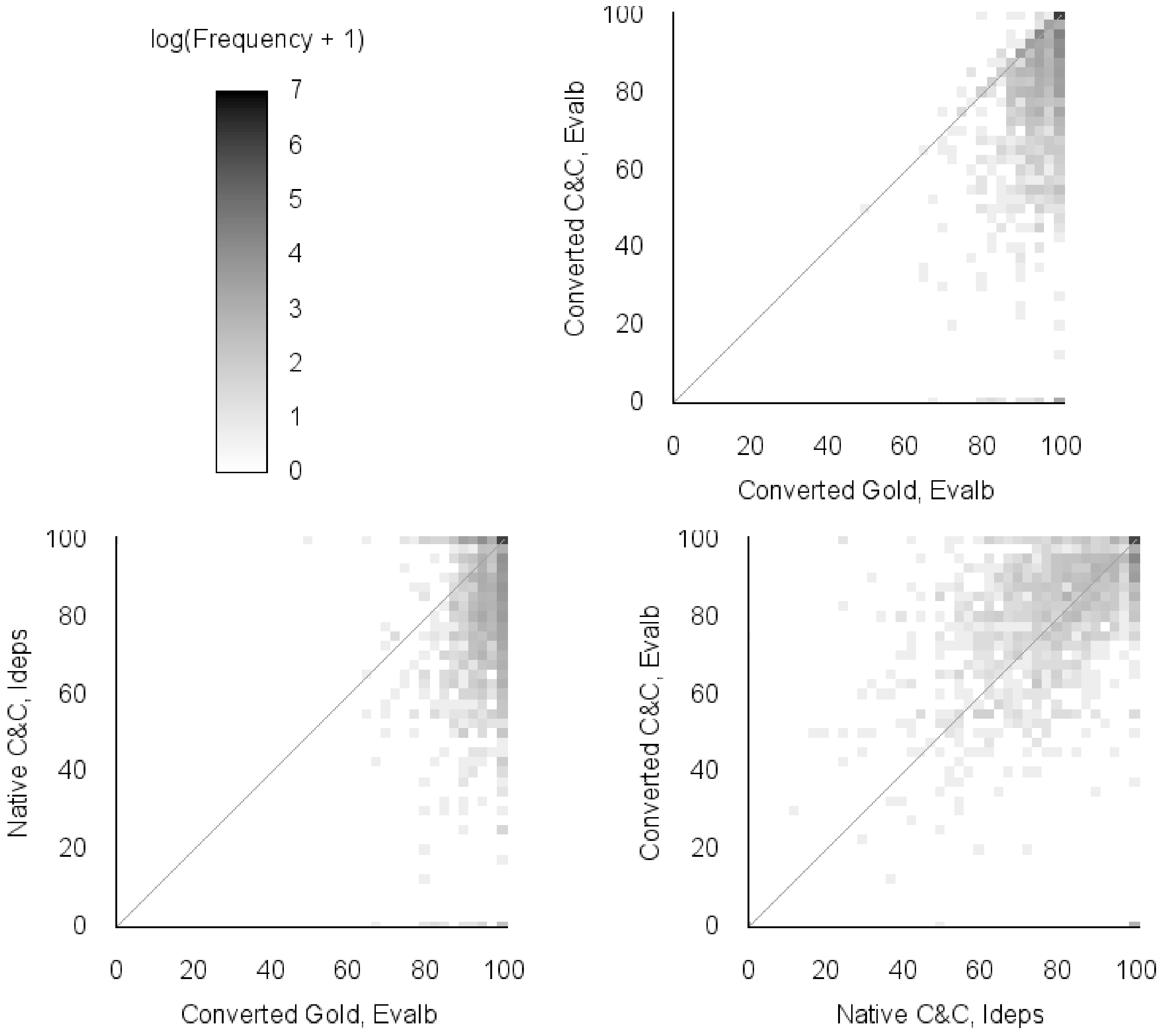


Figure 3.3: For each sentence in the treebank, we plot the converted parser output against gold conversion (top), the original parser evaluation against gold conversion (left), and the converted parser output against the original parser evaluation against (right). A diagonal line indicating $x = y$ is also included. Top: Most points lie below the diagonal, indicating that the quality of converted parser output (y) is upper bounded by the quality of conversion on gold parses (x). Left: No clear correlation is present, indicating that the set of sentences that are converted best (on the far right), are not necessarily easy to parse. Right: In general, accuracy on the native metric is correlated with accuracy after conversion.

3.3.1 Parser Comparison

When we convert the output of a CCG parser, the PTB trees that are produced will contain errors created by our conversion as well as by the parser. In this section we are interested in comparing parsers, so we need to factor out errors created by our conversion.

One way to do this is to calculate a projected score (Proj), as the parser result over the oracle result, but this is a very rough approximation. Another way is to evaluate only on the 51% of sentences for which our conversion from gold CCG derivations is perfect (Clean). However, even on this set our conversion introduces errors, as when the parser output differs from the gold derivation, it may contain categories that are harder to convert.

Parser F-scores are generally higher on Clean, which could mean that this set is easier to parse, or it could mean that these sentences don't contain annotation inconsistencies, and so the parsers aren't incorrect for returning the true parse (as opposed to the one in the PTB). To test this distinction we look for correlation between conversion quality and parse difficulty on another metric. In particular, Figure 3.3 (bottom left) shows CCG labeled dependency performance for the C&C parser vs. CCGbank conversion PARSEVAL scores. The lack of a strong correlation, and the spread on the line $x = 100$, supports the theory that these sentences are not necessarily easier to parse, but rather have fewer annotation inconsistencies.

In the top plot, the y-axis is PARSEVAL on converted C&C parser output. Conversion quality essentially bounds the performance of the parser. The few points above the diagonal are mostly short sentences on which the C&C parser uses categories that lead to one extra correct node (a common case is ADVP v ADJP). The main constructions on which parse errors occur, e.g., PP attachment, are rarely converted incorrectly, and so we expect the number of errors to be cumulative. The bottom right plot shows three noteworthy properties, (1) in general the two evaluation metrics are correlated, (2) the CCG evaluation is slightly harsher, with fewer points below the diagonal line than above it[2], (3) there are many cases on the far right, where the CCG evaluation is perfect, but conversion mistakes mean the PTB score is not perfect. Some sentences are higher in the right plot than the left because there are distinctions in CCG that are not always present in the PTB, e.g., the argument-adjunct distinction.

Table 3.5 presents F-scores for three PTB parsers and three CCG parsers (with their output converted by our method). One interesting comparison is between the PTB parser of Petrov and Klein (2007) and the CCG parser of Fowler and Penn (2010), which use the same underlying parser. The performance gap is partly due to structures in the PTB that are not recoverable from CCGbank, but probably also indicates that the split-merge model is less effective in CCG, which has far more symbols than the PTB.

It is difficult to make conclusive claims about the performance of the parsers. As shown earlier, Clean does not completely factor out the errors introduced by our conversion, as the parser output may be more difficult to convert, and the calculation of Proj only roughly factors out the errors. However, the results do suggest that the performance of the CCG parsers is somewhere between the Stanford and Petrov parsers.

[2]This is consistent with prior work, and the fact that CCG makes some additional distinctions, such as between arguments and adjuncts.

Sentences	Clean	All	Proj
Converted gold CCG			
CCGbank	100.0	96.3	–
Converted CCG			
Clark and Curran (2007)	90.9	85.5	88.8
Fowler and Penn (2010)	90.9	86.0	89.3
Auli and Lopez (2011)	91.7	86.2	89.5
Native PTB			
Klein and Manning (2003a)	89.8	85.8	–
Petrov and Klein (2007)	93.6	90.1	–
Charniak and Johnson (2005)	94.8	91.5	–

Table 3.5: F-scores on section 23 for PTB parsers and CCG parsers with their output converted by our method. Clean is only on sentences that are converted perfectly from gold CCG (51%). All is over all sentences. Proj is a projected F-score (All result / CCGbank All result).

3.4 Summary

By exploiting the generalized combinators of the CCG formalism, we developed a new method of converting CCG derivations into PTB-style trees. Our system is more effective than previous work, increasing exact sentence match by more than 11% (absolute), and can be directly integrated with a CCG parser. The system is available online, see Appendix A.

Chapter 4

Graph Parsing

Parses in the Penn Treebank (Marcus et al. 1993) are graph structured, but parsers are typically restricted to tree structures for efficiency and modeling reasons. All except one of the parsers considered in Chapter 2 produce trees, which is why we did not consider the non-tree components of structure in our analysis. The frequency of this deficiency provides strong motivation for the goal of this chapter: to construct an efficient parser for graph structures.

We propose a new representation and algorithm for a restricted class of graph structures that are flexible enough to cover almost all treebank structures, yet are restricted enough to admit learning and inference that is almost as efficient as for trees. In particular, we consider directed, acyclic, one-endpoint-crossing graph structures, which cover most dislocation, shared argumentation, and similar tree-violating linguistic phenomena. We describe how to convert phrase structure parses, including traces, to our new representation, in a reversible manner. Our dynamic program uniquely decomposes structures, is sound and complete with respect to a subset of the class of one-endpoint-crossing graphs, and covers 97.7% of the Penn English treebank. We also implement a proof-of-concept parser that recovers a range of null elements and trace types.

The algorithm is a new general-purpose parsing approach that could be applied more broadly than the specific use we consider for our experiments. To clearly separate the definition of the algorithm and our specific use of it, we start by defining the algorithm in Sections 4.2 through 4.6, then show how it can be applied to Penn Treebank parsing in Section 4.7.

4.1 Background

Before discussing the motivation for this work and the previous work in this area, we need to mention several concepts from discrete mathematics:

Graph A set of vertices and edges. Here each word is a vertex, plus a root vertex not associated with any word.

Tree A graph in which every pair of vertices are connected by a unique path (ignoring directionality on edges).

Cycle A sequence of edges that begin and end at the same vertex (here we only allow traversal of edges in the direction indicated by the edge).

Projective The linear order of words allows us to define an additional concept: projectivity. A projective graph can be drawn in a two dimensional half-plane with the vertices on the edge of the plane and without any edges crossing.

One-Endpoint Crossing Pitler et al. (2013) defined a space of tree structures named one-endpoint crossing (one-EC). To be in the space, every edge in the tree must obey the one-EC property. The property states that for a given edge e, all the edges that cross e must share an endpoint (treating the edges as undirected).

We generalize Pitler et al. (ibid.)'s definition of one-endpoint crossing trees to graphs. The one-EC property remains the same, we just relax the requirement that the edges form a tree. To enable efficient parsing we will further restrict the space we consider, not covering a few specific one-endpoint-crossing structures. In practice, these structures are rarely observed in the PTB, and when they do occur it is usually due to where punctuation attaches.

Figure 4.1 shows the space of graphs divided up based on these properties. Note that by only using subsets of the rules in our algorithm, we could restrict the space of structures we generate to the projective DAG space, the projective tree space, or the one-EC tree space. Additionally, versions of these spaces with undirected edges could be easily handled with the same approach we apply.

4.1.1 Why Graph Structures?

Non-projectivity is important in syntax for representing many structures, to the point where Chomsky identifies it as a distinguishing feature of syntax, called displacement (Chomsky 2000). For the purpose of phrase structure representations, adding non-projective links means we are also creating graphs, as adding any edge to a tree will create a graph. These examples show some of the ways in which these extra edges are needed to encode the structure of the sentence:

- **Null infinitive subject** *Bob wants to teach*, here *Bob* is both *wanting* and *teaching*, but in a tree *Bob* can only have one parent, and so can't be linked to both.
- **Wh-movement** *When should Bob teach?*, here the tree would not have a connection between *when* and *teach*.
- **Gapping** *I saw Judy in Sydney and Bob in Berkeley*, here the tree would either lack edges indicating the conjunction of *Judy* and *Bob* or lack edges indicating their connections to *saw*.

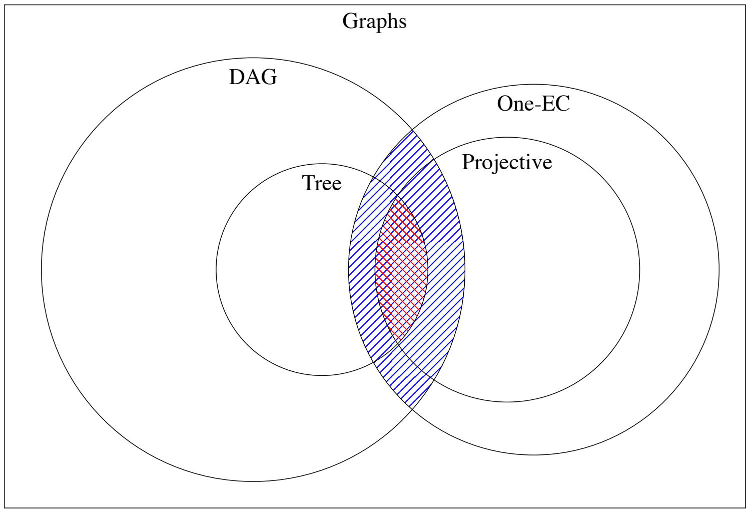


Figure 4.1: Venn diagram showing the space of graphs divided into the properties we consider. DAG stands for Directed Acyclic Graph. Our algorithm is able to produce structures in the blue hatched region while the standard parsing approach only produces structures in the blue and red cross-hatched region. Pitler et al. (2013)'s algorithm produces structures in the overlap of the Tree region and the one-EC region. Regions are not scaled to reflect the actual number of possible structures of each type.

- **Fronting** *Hard though the journey may be*, here the word *hard* will not be linked to *be* in the tree.

Section 1.4 presented the full set of null elements used in the Penn Treebank, which cover a wide range of phenomena. The annotation guide for the Penn Treebank (Bies et al. 1995) provides detailed descriptions of all of these structures as well as examples drawn from the data.

4.1.2 Previous Algorithms

As discussed in Section 1.4, a range of algorithms have been developed for efficient parsing of projective trees. This efficiency comes from the ability to break the projective tree down into independent pieces. Since no edge can be crossed, every edge defines a division of the tree into two parts, inside and outside the edge, i.e., one piece that includes all of the structure linking words between the two ends of the edge, and another piece for all other words.

Inference over the space of all non-projective graph parses is intractable, but in practice almost all parses are covered by well-defined subsets of this space. For dependency parsing, recent work has defined algorithms for inference within various subspaces (Gómez-Rodríguez and Nivre 2010; Pitler et al. 2013). We build upon these algorithms and adapt them to constituency parsing. For constituency parsing, a range of alternative formalisms have been developed, starting with Generalized Phrase Structure Grammar (Gazdar et al. 1985), which used features and slash categories

Algorithm 4.1 General CKY algorithm.

```
Initialize with an empty parse item for each position in the sentence
for each possible width, from 1 to the length of sentence do
    for each span[1] mark of that width do
        Create items by combining existing, smaller, items
        Create items by adding structure to items with this span
```

to encode traces. Further work built on these ideas, forming what was later described as mildly-context sensitive grammar formalisms, including LFG (Kaplan and Bresnan 1982), LTAG (Joshi and Schabes 1997), and CCG (Steedman 2000).

Our representation is similar to LTAG-Spinal (Shen et al. 2007), but has the advantage that it can be converted back into the PTB representation. Recently, dependency parsers have been used to assist in constituency parsing, which has involved varying degrees of representation design, but only for trees (J. Hall, Nivre, and Nilsson 2007; J. Hall and Nivre 2008; Fernández-González and Martins 2015; Kong et al. 2015).

Previous work on parsing traces and other null elements in the PTB has taken two general approaches. The first broadly effective system was Johnson (2002), which post-processed the output of a parser, inserting extra elements. This was effective for some types of structure, such as null complementizers, but had difficulty with long distance dependencies. One challenge for a post-processing system is that it must deal with parser mistakes–Johnson's system F-score was 7 points higher on gold trees than on parser output.

The other common approach has been to expand the grammar, threading a trace through the tree structure on the non-terminal symbols. Collins (1997)'s third model used this approach to recover wh-traces, while Cai et al. (2011) used it to recover null pronouns, and others have used it for a range of movement types (Dienes and Dubey 2003; Schmid 2006). All of these approaches have the disadvantage that each additional trace dramatically expands the grammar.

4.2 Overall Algorithm

Our algorithm falls into the dynamic programming class, and specifically the type defined by Kasami (1966), Younger (1967), and Cocke (1969). Algorithm 4.1 shows a pseudocode outline of the CKY algorithm.

This algorithm works by exploiting the constraint that the larger problem (find the optimal parse) can be decomposed into independent sub-problems (find the optimal parse for part of the sentence) whose solutions can be combined to solve the original problem (combine the best solution from one span with the best solution from another span). To turn this into a specific algorithm we need to specify:

- What types of items can exist

[1]A continuous range, e.g., from 1 to 4 would be a span of width 3

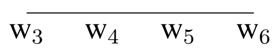
- How items can be combined
- How structure can be added to an item

The second and third of these needs can be described by deduction rules. Most of this chapter will be concerned with the definition of these deduction rules and the properties we are able to achieve by our choices.

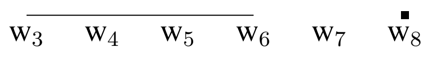
4.3 Sketch of Deduction Rules

In this section we begin with an introduction to build intuition, then define the item types (§ 4.3.2), work through an example (§ 4.3.3), and sketch the deduction rules for our algorithm (§ 4.3.4). In Section 4.4, we discuss how it differs from Pitler et al. (2013)'s tree parsing algorithm, and in Section 4.5 we provide the complete details of the algorithm definition and proof. Eventually we will show how we can support directed, labeled edges, and labeled words, but to simplify the presentation of the algorithm we first focus on deduction rules for undirected, unlabeled edges and ignore word labels.

The intuition for the items is that they are either a continuous span of the sentence, going from one word to another (shown with a line, intentionally without internal structure):



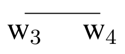
Or a span plus a point outside the span:



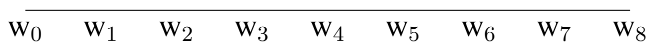
Whether a word in the item has edges, and whether the item can be modified to give it an edge, depends on its location:

		Has	Can get
Location	Example	edges?	edges?
Within a span	w_4, w_5	Yes	No
At the ends of the span	w_3, w_6	Maybe	Yes
Between the span and the exterior point	w_7	No	No
At the exterior point	w_8	Yes	Yes

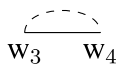
We start out with spans going between each pair of words, with no structure:



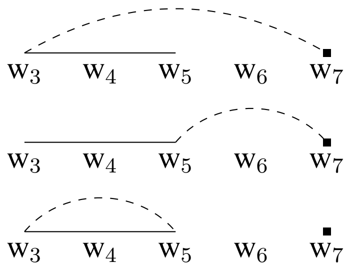
Our goal is to form a span that covers the entire sentence:



For items that are just a span, we can create an edge between the endpoints. An edge added like this is not crossed in the final, complete parse for the sentence.

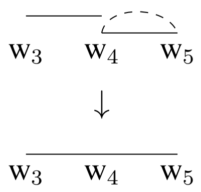


For items with an exterior point we gain the option to create edges going between the exterior point and either end of the span. In the top and middle cases, the edge being added will be crossed at some point later on in the derivation. In the bottom case, the edge being added will cross one or more existing edges.



w_3 w_4 w_5 w_6 w_7
w_3 w_4 w_5 w_6 w_7
w_3 w_4 w_5 w_6 w_7

The intuition for the deduction rules is that they combine two or three items with adjacent spans to create a new item. By carefully deciding which combinations are permitted and which are not, we are able to provide guarantees about the final structure.



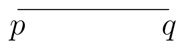
Crucially, our items are defined by a small number of values, such as the start and end points of the span, rather than their full internal structure. During parsing we only need to store the best structure corresponding to each item, not every structure. That means the time complexity of our algorithm depends on the number of possible items (polynomial complexity), rather than the number of structures (exponential complexity).

4.3.1 Notation

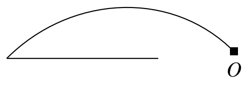
We use p, q, etc to refer to word positions. To indicate ranges we use $[pq]$, $[pq)$, $(pq]$, or (pq), where the bracket variations indicate inclusion, $[]$, or exclusion, $()$, of the endpoint. To indicate an edge we use two points without brackets, e.g., pq. To define a class of edges we either use a point and a set connected by a dash, e.g., o–(pq), or two sets connected by a dash, e.g., (ps)–(sq).

4.3.2 Item Types

As shown above and in Figure 4.2, our items start and end on words, fully covering the gaps in between[2]. We use six item types, differing in the type of edge crossing they contain:

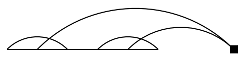


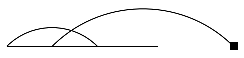
I**, Interval** A span in which all points in (pq) have a parent in $[pq]$, and no edges will exist that go between (pq) and points outside $[pq]$.

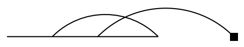


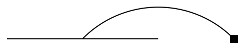
X**, Exterval (an external point + an Interval)** An interval plus a single edge between o and either p or q, where o is outside $[pq]$.

[2] This is the inverse of the conventional constituency parsing approach where items fully cover words and end in the gaps between words. The idea can be traced back to Eisner (1996).


B**, Both** An interval $[pq]$ and a point o. o–(pq) edges may be crossed by p–(pq) or q–(pq) edges, and at least one crossing of each type occurs. o–(pq) edges may not be crossed by (pq)–(pq) edges.


L**, Left** Same as B, but o–(pq) edges may only be crossed by p–(pq) edges.


R**, Right** Same as L, but with edges crossed by q–(pq) edges rather than p-(pq) edges.


N**, Neither** An interval and a point, with at least one o–(pq) edge. o–(pq) edges can only be crossed by pq, not other $[pq]$–$[pq]$ edges.

4.3.3 Example Derivation

Figure 4.2 presents a derivation of a sentence with crossing edges, showing examples of several deduction rules. In the next section, we define the deduction rules from a top-down perspective to easily show how well they cover the space of structures. To build intuition, here we describe the example bottom-up and with more conventional deductive reasoning notation.

We initialize with items of width one, placing an item between each pair of words.

$$\emptyset \mapsto I_{0,1}$$

Our main loop from Algorithm 4.1 considers each width and combines items, then adds structure. We start with items of width one, which cannot be formed by combination. We can add edges though, such as the *We–like* edge, and the *like–ROOT* edge. Note, in the second case the edge creates an Exterval, and the edge will eventually be crossed.

$$I_{1,2} \wedge pq_{1,2} \mapsto I_{1,2}$$
$$I_{0,1} \wedge op_{0,2} \mapsto X_{0,1}$$

In the next step we consider items of width two. We can form an item of width two by combining items either side of *to*.

$$I_{2,3} \wedge I_{3,4} \mapsto I_{2,4}$$

From (3) to (4) we are still considering items of width two, but now adding edges, such as *We–run*. This edge would not be part of a tree parse, and will be crossed in our complete parse.

$$I_{2,4} \wedge oq_{1,4} \mapsto X_{2,4}$$

Next, a width four item is formed by combining items either side of *run*. This creates a Neither item, as there is an uncrossed edge from the external point to within the span. Note that we can determine it is an N with only limited information about the internal structure of the items being combined.

$$X_{2,4} \wedge I_{4,6} \mapsto N_{2,6}$$

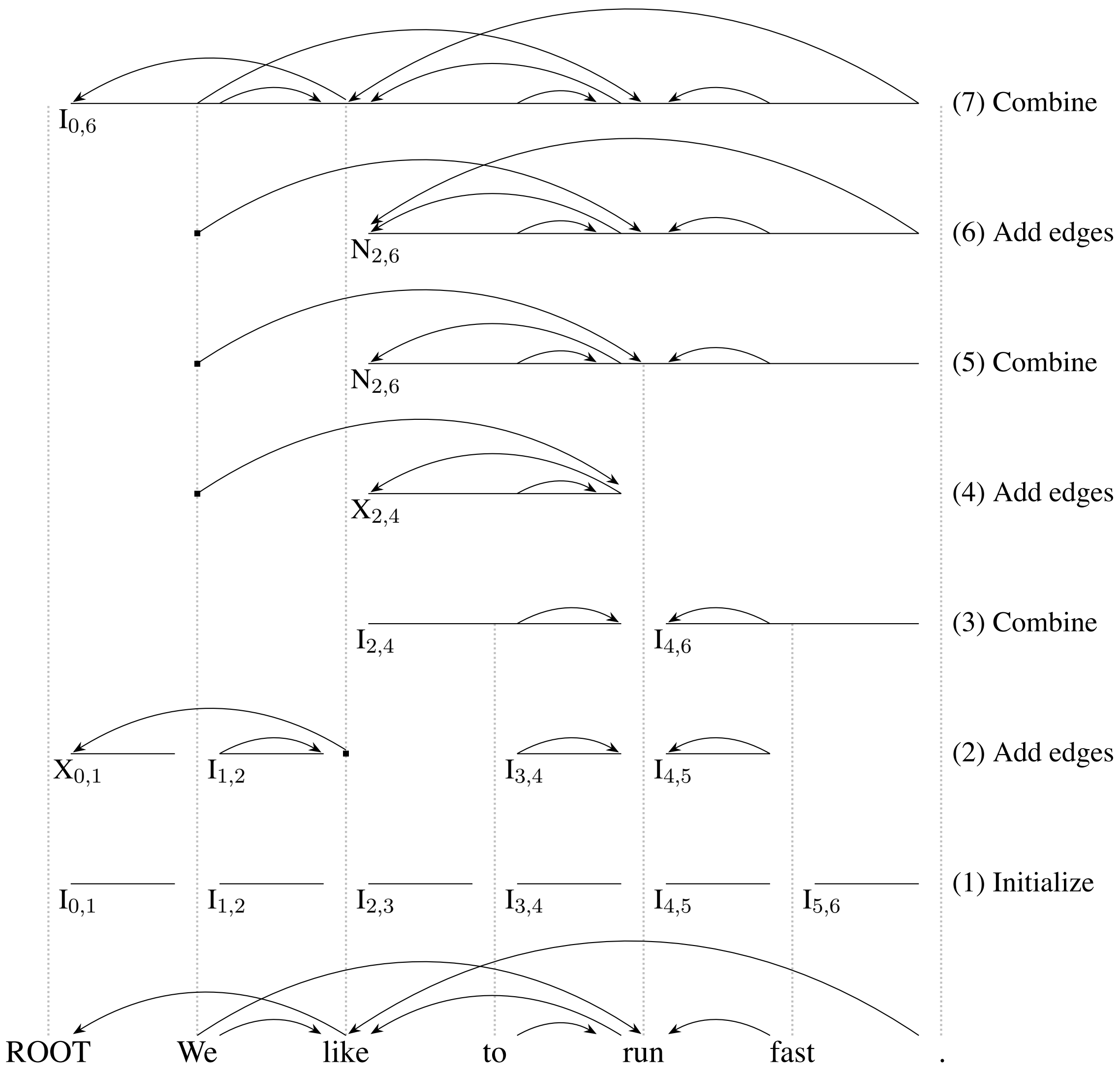


Figure 4.2: An example derivation using our graph parsing deduction rules.

Going from (5) to (6) we introduce our first crossing of edges. The new edge, *.–like*, crosses the edge added two steps ago.

$$N_{2,6} \ \wedge \ qp_{2,6} \ \mapsto \ N_{2,6}$$

Finally, we combine three items to form the complete derivation.

$$X_{0,1} \ \wedge \ I_{1,2} \wedge N_{2,6} \ \mapsto \ I_{0,6}$$

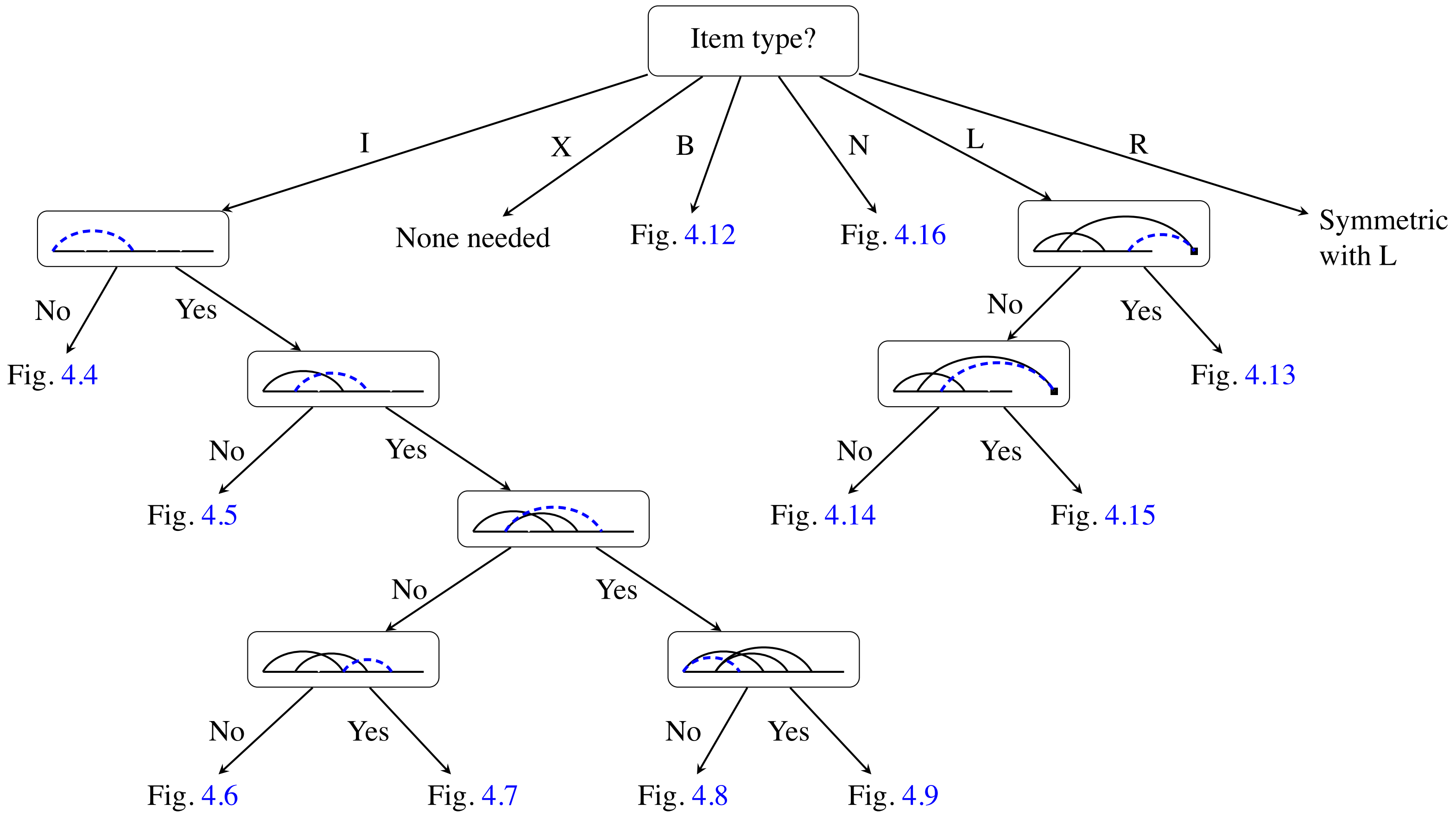


Figure 4.3: Our algorithm defines a unique decomposition for any structure. This chart shows how we define the decompositions that involve multiple items (as opposed to removing a single edge, such as in X cases). Each rectangle poses a questions, using the black structure to express what is known, and the dashed blue edge to ask the question: is such an edge present or not? Leaves of the chart are labeled with the section in the text that gives the decomposition for that case.

4.3.4 Deduction Rules

One set of deduction rules is concerned with removing direct edges between the points p, q, and o in each item. These are straightforward, so we leave them for the full derivation in Section 4.5. The other type of deduction rules, which we sketch below, involve decomposing an item into parts.

To make the deduction rules manageable, we define only some constraints explicitly, and then use code to enumerate all options and enforce additional constraints. Here we describe the explicit constraints and the reasoning behind them.

When parsing, we simply have a list of valid rules and consider any way of combining items that satisfies a rule. To define the rules, we need to consider each of our item types and define a unique decomposition into smaller pieces. Figure 4.3 gives an overview of the distinctions we make in order to define each type of decomposition. For some item types, such as X, every possible structure can be broken down into pieces in the same way. For others, such as L, how the item is broken down depends on some properties of its structure. Below we work through each path in Figure 4.3, specifying the questions more precisely and then providing the decomposition.

Interval

Is there a p–(pq) edge?
No, then split at $p + 1$:

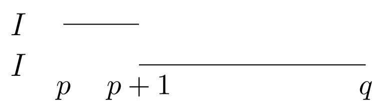


Figure 4.4: I deduction rules, case one.

Yes, then consider ps, the longest p–(pq) edge.
Do any edges cross ps?
No, then split at s:

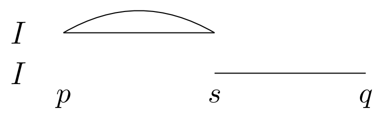


Figure 4.5: I deduction rules, case two.

Yes, then consider the set of edges C, that cross ps. If $|C| > 1$, let t be the common endpoint of all edges in C (they must have a common endpoint to satisfy the one-EC property for ps). If $|C| = 1$, let t be the endpoint outside ps.

Is $s < t$ and are there any s–(tq) edges?
Yes $s < t$, no s–(tq) edges (in this case it is also possible that $t = q$, in which case there is no $[tq]$ item):

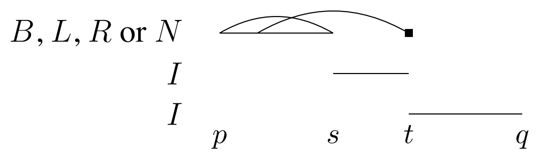


Figure 4.6: I deduction rules, cases three and five.

Yes $s < t$, and yes there are s–(tq) edges:

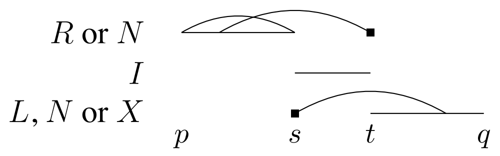


Figure 4.7: I deduction rules, case four.

Now consider $s > t$. In this case $C > 1$ by construction. Are there any p–(ts) edges?
No:

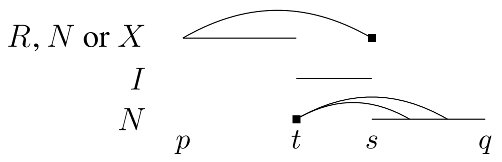


Figure 4.8: I deduction rules, case seven.

Yes:

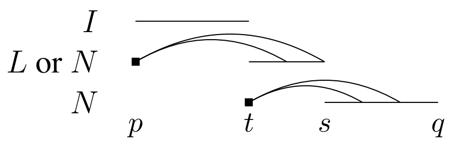


Figure 4.9: I deduction rules, case six.

Exterval

No decomposition rules are needed, as the removal of the op or oq edge leaves behind an Interval.

Both

Consider the case when o is to the right (the case when o is to the left is symmetrical). Consider ps_p, the longest p–(pq) edge.

Does ps_p cross any q–(pq) edges?
Yes:

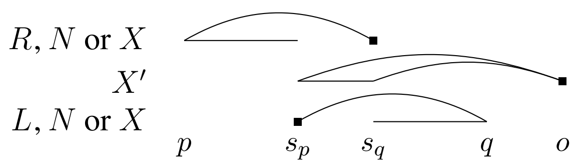


Figure 4.10: Excluded B deduction rule

The extra edges shown must be present because of the definition of B items and the one-EC property. The X' is a slight variation on our X, since it has both s_qo and s_po. We do not include this rule because it would be $O(n^5)$, and this specific structure is almost never observed in the treebank. Now consider the alternative, when ps_p does not cross any q–(pq) edges. In this case the possibilities follow a pattern:

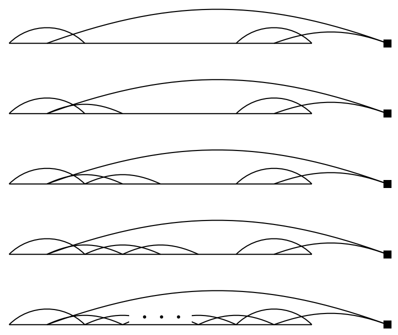


Figure 4.11: B structural pattern

Does the pattern go all the way to the right, as shown in the last case?
Yes:

> We cannot define a simple decomposition of the structure using our item types. By eliminating this case we restrict ourselves to a subset of one-endpoint-crossing graphs. This does not decrease treebank coverage.

No, then use the end of the edge crossing pattern as s and decompose the item as:

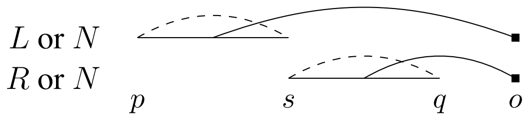


Figure 4.12: B deduction rule.

The dashed edges are required for the N items, but not the L items. Below we describe how to check that the chaining pattern is present in an L, but in practice we avoid it entirely by requiring the L to have the edge ps, with no loss in coverage.

Left and Right

These are defined symmetrically. Consider the o–(pq) edges in an L. Use the edge os with s furthest from p to define the split point.
Is os crossed by any p–(pq) edges?
No:

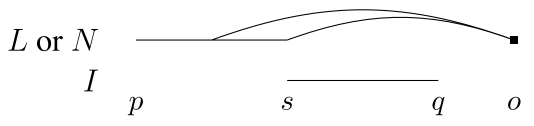


Figure 4.13: L deduction rules, cases three through eight.

Yes, then is os the only o–(pq) edge?
No:

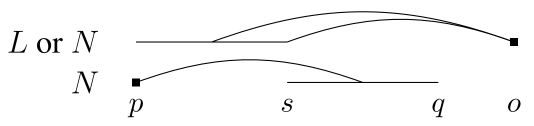


Figure 4.14: L deduction rules, cases one and two.

Yes:

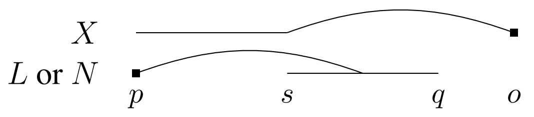


Figure 4.15: L deduction rules, cases nine and ten.

We also have to track whether there are multiple edges or only a single o–(pq) edge, to avoid derivational ambiguity in the I decomposition. The first two cases lead to multiple such edges, while the third case will only have one.

We could also track the chaining property discussed in the previous section. The property is only true in the following cases: when the edge pq is present, when an X is combined with an L that has the property, and when an X is combined with an N that has the pq edge.

Neither

Consider the o–(pq) edges. Use the edge os with s furthest from o to define the split point. The case with o to the left is symmetrical to the case with o on the right, which is:

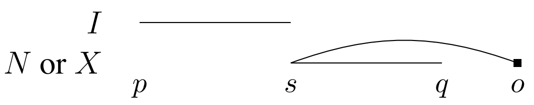


Figure 4.16: N deduction rule.

Again, we need to track whether there are multiple o–pq edges. When a sub-item is an X there is only one o–(pq) edge, and when it is an N there is more than one.

4.4 Comparison with Pitler et al. (2013)

Our algorithm is based on Pitler et al. (ibid.), which had the crucial ideas of one-endpoint crossing, isolated crossing regions, and a complete decomposition. Our changes give several overall benefits:

- We extend to graph structures, which introduces particular challenges for the B item type.
- We avoid derivational ambiguity, where a single parse can be decomposed in multiple ways.
- Changes that give new intuition about how the algorithm works, by being able to give guarantees about whether edges will be crossed at the time they are created.
- A somewhat more concise overall algorithm definition, mainly due to our use of a combination of templates and code to generate the final rules.
- Separation of merging items from edge creation.

Below we provide some specific notes on where changes occurred and why.

Intuition The clean division of cases between edges that will be crossed and those that will not be is partially due to choices we made in the definition of the deduction rules; it does not hold for Pitler et al. (ibid.)'s algorithm.

Items We have one entirely new item type, the Exterval, and we have added requirements for the later items. Specifically, while the allowed crossings are the same, we add the requirement that at least one crossing of the specified type exists (this is also reflected in the renaming of their LR item to B). These changes are necessary to avoid derivational ambiguity when a structure falls into multiple classes. Enforcing these difference involves changes throughout the deduction rules.

Interval One source of ambiguity in their algorithm arises when an edge is crossed by only one other edge. The ambiguity is in the choice of the second split point, t, which could be either end of the crossing edge in their derivation. In that circumstance we require it to be the endpoint further from p (see the discussion after Figure 4.5).

Both The difficult cases we discuss do not come up in their algorithm. This is due to a combination of (1) the tree constraints they apply, (2) the need to form a complete structure, and (3) use of directed edges. For the normal B case, the choice of split point is a source of derivational ambiguity in their algorithm.

Left and Right Our decomposition of this item differs from theirs in several ways because of our decision to separate edge creation and item combination, and because we need to track if there are multiple edges to the external point.

Neither As for L and R, our decomposition differs because of our altered definition of N, and the need to track if there are multiple edges to the external point.

4.5 Deduction Rule Definitions and Completeness Proof

Above we provided only a sketch of the algorithm, here we explain the full derivation. For completeness, some of the content is repeated. When showing the possible deduction rules for each item type we show figures that indicate the required characteristics in each case. In the figures black solid curves indicate edges that must exist, and red dotted curves are edges that are not allowed, either by construction or due to constraints.

4.5.1 Notation

To indicate a range we use $[pq]$, $[pq)$, $(pq]$, or (pq), where the bracket variations indicate inclusion, $[]$, or exclusion, $()$, of the endpoint. We also use $[pq]$ to indicate a span without an external point, and $[pq.o]$ to indicate a span with external point o. To indicate edges we either use two points without brackets, e.g., pq, a point and a set connected by a hyphen, e.g., o–(pq), or two sets connected by a hyphen, e.g., (ps)–(sq).

4.5.2 Item Types

We use six item types, differing in the type of edge crossing they contain. Each consists of a span, $[pq]$ and an optional point outside the span, o, either to the left or right:

I **- Interval** A span in which all points in (pq) have a parent in $[pq]$, and no edges exist that go from outside $[pq]$ to points in (pq).

X **- Exterval (an external point + an Interval)** An interval plus exactly one of the edges op or oq, where o is outside $[pq]$.

N **- Neither** An interval $[pq]$ and a point o, with a least one o–(pq) edge. o–(pq) edges can be crossed by the edge pq, but not by any other $[pq]$–$[pq]$ edge.

L **- Left** Same as N, but o–(pq) edges may be crossed by p–(pq) edges. Also, such a crossing occurs at least once.

R **- Right** Same as L, but with o–(pq) edges crossed by q–(pq) edges rather than p-(pq) edges.

B **- Both** Same as L, but o–(pq) edges may also be crossed by q–(ps) edges, and at least one crossing of each type occurs.

Algorithm 4.2 Complete graph parsing dynamic program.

```
$\utilde{L}[ijx\ .\overline{X}\ .X\ \overline{IJ}] \leftarrow \max$
  $A[x\ j]\quad L[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]$
$\utilde{L}[ijx\ .\overline{X}\ .\overline{X}\ \overline{I}J] \leftarrow \max$
  $A[j\ x]\quad L[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]$
$L[ijx\ J\overline{X}\ \overline{IX}\ \overline{IJ}] \leftarrow \max$
  $A[j\ i]\quad L[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}]$
$L[ijx\ \overline{JX}\ I\overline{X}\ \overline{IJ}] \leftarrow \max$
  $A[i\ j]\quad L[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}]$

$\utilde{R}[ijx\ .X\ .\overline{X}\ \overline{IJ}] \leftarrow \max$
  $A[x\ i]\quad R[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]$
$\utilde{R}[ijx\ .\overline{X}\ .\overline{X}\ I\overline{J}] \leftarrow \max$
  $A[i\ x]\quad R[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]$
$R[ijx\ J\overline{X}\ \overline{IX}\ \overline{IJ}] \leftarrow \max$
  $A[j\ i]\quad R[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}]$
$R[ijx\ \overline{JX}\ I\overline{X}\ \overline{IJ}] \leftarrow \max$
  $A[i\ j]\quad R[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}]$

$I[ij\ F\ I] \leftarrow \max$
  $A[i\ j]\quad I[ij\ F\ \overline{I}]$
$I[ij\ J\ F] \leftarrow \max$
  $A[j\ i]\quad I[ij\ \overline{J}\ F]$

$\utilde{N}[ijx\ .X\ .\overline{X}\ \overline{IJ}] \leftarrow \max$
  $A[x\ i]\quad N[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]$
$\utilde{N}[ijx\ .\overline{X}\ .\overline{X}\ I\overline{J}] \leftarrow \max$
  $A[i\ x]\quad N[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]$
$\utilde{N}[ijx\ .\overline{X}\ .X\ \overline{IJ}] \leftarrow \max$
  $A[x\ j]\quad N[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]$
$\utilde{N}[ijx\ .\overline{X}\ .\overline{X}\ \overline{I}J] \leftarrow \max$
  $A[j\ x]\quad N[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]$
$N[ijx\ J\overline{X}\ \overline{IX}\ \overline{IJ}] \leftarrow \max$
  $A[j\ i]\quad N[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}]$
$N[ijx\ \overline{JX}\ I\overline{X}\ \overline{IJ}] \leftarrow \max$
  $A[i\ j]\quad N[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}]$

$X[ijx\ .X\ .\overline{X}\ F] \leftarrow \max$
  $A[x\ i]\quad I[ij\ .\ .]$
$X[ijx\ .\overline{X}\ .\overline{X}\ I\overline{J}] \leftarrow \max$
  $A[i\ x]\quad I[ij\ .\ .]$
$X[ijx\ .\overline{X}\ .X\ F] \leftarrow \max$
  $A[x\ j]\quad I[ij\ .\ .]$
$X[ijx\ .\overline{X}\ .\overline{X}\ \overline{I}J] \leftarrow \max$
  $A[j\ x]\quad I[ij\ .\ .]$

$B\cdot[ijx\ F\ I\overline{X}\ \overline{IJ}] \leftarrow \max$
  $A[i\ j]\quad B\cdot[ijx\ F\ \overline{IX}\ \overline{IJ}]$
$\cdot B[ijx\ J\overline{X}\ F\ \overline{IJ}] \leftarrow \max$
  $A[j\ i]\quad \cdot B[ijx\ \overline{JX}\ F\ \overline{IJ}]$

$I[ij\ F\ \overline{I}] \leftarrow \max$
  $I[i\ i{+}1\ F\ F]\quad I[i{+}1\ j\ \overline{F}\ F]$
  $\max_{k\in(i,j)}$
    $I[ik\ F\ I]\quad I[kj\ .\ .]$
    $BLRN\cdot[ikj\ F\ I\overline{J}\ \overline{IK}]\quad I[kj\ .\ .]$
    $\max_{l\in(k,j)}$
      $RN\cdot[ikl\ F\ I\overline{L}\ \overline{IK}]\quad I[kl\ .\ .]\quad \cdot LNX[ljk\ .\overline{K}\ .\ \overline{L}.]$
      $BLRN\cdot[ikl\ F\ I\overline{L}\ \overline{IK}]\quad I[kl\ .\ .]\quad I[lj\ .\ .]$
    $\max_{l\in(i,k)}$
      $I[il\ F\ .]\quad \cdot LN[lki\ .\overline{I}\ .I\ F]\quad \cdot\utilde{N}[kjl\ \overline{JL}\ \overline{K}.\ \overline{K}.]$
      $RNX\cdot[ilk\ F\ .\overline{K}\ I\overline{L}]\quad I[lk\ .\ .]\quad \cdot\utilde{LN}[kjl\ .\overline{L}\ .\ \overline{K}.]$
$I[ij\ j\ F] \leftarrow \max_{k\in(i,j)}$
  $I[ik\ \overline{F}\ .]\quad I[kj\ J\ F]$
  $I[ik\ \overline{F}\ .]\quad \cdot BLRN[kji\ J\overline{I}\ F\ \overline{KJ}]$
  $\max_{l\in(i,k)}$
    $RNX\cdot[ilk\ \overline{F}\ .\overline{K}\ .\overline{L}]\quad I[lk\ .\ .]\quad \cdot LN[kjl\ J\overline{L}\ F\ \overline{KJ}]$
    $I[il\ \overline{F}\ .]\quad I[lk\ .\ .]\quad \cdot BLRN[kjl\ J\overline{L}\ F\ \overline{KJ}]$
  $\max_{l\in(k,j)}$
    $\utilde{N}\cdot[ikl\ \overline{KF}\ \overline{IL}\ .\overline{K}]\quad RN\cdot[klj\ .J\ .\overline{J}\ F]\quad I[lj\ .\ F]$
    $\utilde{RN}\cdot[ikl\ \overline{F}\ .\overline{L}\ .\overline{K}]\quad I[kl\ .\ .]\quad \cdot LNX[ljk\ .\overline{K}\ F\ \overline{L}J]$
$\utilde{L}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
  $LN[ikx\ .\overline{X}\ .X\ \overline{IK}]\quad \cdot N[kji\ \overline{JI}\ \overline{KI}\ \overline{KJ}]$
  $LN[ikx\ .\overline{X}\ .\overline{X}\ \overline{I}K]\quad \cdot N[kji\ \overline{JI}\ \overline{KI}\ \overline{KJ}]$
  $L[ikx\ .\overline{X}\ .X\ \overline{IK}]\quad I[kj\ .\ .]$
  $L[ikx\ .\overline{X}\ .\overline{X}\ \overline{I}K]\quad I[kj\ .\ .]$
  $N[ikx\ \overline{KX}\ IX\ \overline{IK}]\quad I[kj\ .\ .]$
  $N[ikx\ \overline{KX}\ I\overline{X}\ \overline{I}K]\quad I[kj\ .\ .]$
  $N[ikx\ K\overline{X}\ \overline{I}X\ \overline{IK}]\quad I[kj\ .\ .]$
  $N[ikx\ K\overline{X}\ \overline{IX}\ \overline{I}K]\quad I[kj\ .\ .]$
$\underset{\cdot}{L}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
  $X[ikx\ .\overline{X}\ .X\ F]\quad \cdot LN[kji\ .\overline{I}\ .\overline{I}\ \overline{KJ}]$
  $X[ikx\ .\overline{X}\ .\overline{X}\ \overline{I}K]\quad \cdot LN[kji\ .\overline{I}\ .\overline{I}\ \overline{KJ}]$

$B\cdot[ijx\ F\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
  $LN\cdot[ikx\ F\ I\overline{X}\ \overline{IK}]\quad R\cdot[kjx\ .\ .\overline{X}\ .\overline{J}]$
  $LN\cdot[ikx\ F\ I\overline{X}\ \overline{IK}]\quad N\cdot[kjx\ \overline{J}.\ K\overline{X}\ .\overline{J}]$
  $LN\cdot[ikx\ F\ I\overline{X}\ \overline{IK}]\quad N\cdot[kjx\ J.\ \overline{KX}\ .\overline{J}]$
$\cdot B[ijx\ \overline{JX}\ F\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
  $\cdot L[ikx\ .\overline{X}\ .\ \overline{I}.]\quad \cdot RN[kjx\ J\overline{X}\ F\ \overline{KJ}]$
  $\cdot N[ikx\ K\overline{X}\ \overline{I}.\ \overline{I}.]\quad \cdot RN[kjx\ J\overline{X}\ F\ \overline{KJ}]$
  $\cdot N[ikx\ \overline{KX}\ I.\ \overline{I}.]\quad \cdot RN[kjx\ J\overline{X}\ F\ \overline{KJ}]$
$\utilde{N}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
  $\cdot N[ikx\ \overline{KX}\ \overline{I}X\ \overline{IK}]\quad I[kj\ .\ .]$
  $\cdot N[ikx\ \overline{KX}\ \overline{IX}\ \overline{I}K]\quad I[kj\ .\ .]$
  $I[ik\ .\ .]\quad N\cdot[kjx\ \overline{J}X\ \overline{KX}\ \overline{KJ}]$
  $I[ik\ .\ .]\quad N\cdot[kjx\ \overline{JX}\ \overline{KX}\ K\overline{J}]$
$\underset{\cdot\cdot}{N}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
  $\cdot X[ikx\ .\overline{X}\ .X\ \overline{IK}]\quad I[kj\ .\ .]$
  $\cdot X[ikx\ .\overline{X}\ .\overline{X}\ \overline{I}K]\quad I[kj\ .\ .]$
  $I[ik\ .\ .]\quad X\cdot[kjx\ .X\ .\overline{X}\ \overline{KJ}]$
  $I[ik\ .\ .]\quad X\cdot[kjx\ .\overline{X}\ .\overline{X}\ K\overline{J}]$
$\utilde{R}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
  $N\cdot[ikj\ \overline{KJ}\ \overline{IJ}\ \overline{IK}]\quad RN[kjx\ .X\ .\overline{X}\ \overline{KJ}]$
  $N\cdot[ikj\ \overline{KJ}\ \overline{IJ}\ \overline{IK}]\quad RN[kjx\ .\overline{X}\ .\overline{X}\ K\overline{J}]$
  $I[ik\ .\ .]\quad R[kjx\ .X\ .\overline{X}\ \overline{KJ}]$
  $I[ik\ .\ .]\quad R[kjx\ .\overline{X}\ .\overline{X}\ K\overline{J}]$
  $I[ik\ .\ .]\quad N[kjx\ JX\ \overline{KX}\ \overline{KJ}]$
  $I[ik\ .\ .]\quad N[kjx\ J\overline{X}\ \overline{KX}\ K\overline{J}]$
  $I[ik\ .\ .]\quad N[kjx\ \overline{J}X\ K\overline{X}\ \overline{KJ}]$
  $I[ik\ .\ .]\quad N[kjx\ \overline{JX}\ K\overline{X}\ K\overline{J}]$
$\underset{\cdot}{R}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
  $RN\cdot[ikj\ .\overline{J}\ .\overline{J}\ \overline{IK}]\quad X[kjx\ .X\ .\overline{X}\ F]$
  $RN\cdot[ikj\ .\overline{J}\ .\overline{J}\ \overline{IK}]\quad X[kjx\ .\overline{X}\ .\overline{X}\ K\overline{J}]$
```

4.5.3 Complete Dynamic Program

Figure 4.2 shows the templates for the complete dynamic program. The notation used is as follows:
Edges are $A[p\ c]$, where p and c are the parent and child positions.
Items are $T[lrx\ pl\ pr\ px]$, where each part is:

- T is the type of item, with multiple letters indicating any one of those types are allowed.
- $\cdot T$ and $T\cdot$ indicates the position of the external point relative to the item's span (left or right respectively).
- $\underset{.}{T}$ indicates that an item has only one edge from the external point into the span, while $\underset{\sim}{T}$ indicates that there is more than one such edge.
- l, r, and x are labels that indicate the position of the left end of the span, the right end, and the external point, respectively.
- pl, pr, and px are expressions that indicate the parents of the left end of the span, the right end of the span, and the external point.
- The parent expressions can be an explicit need for a link, either direct, indirect, or none, e.g., I, i, and F. They can also be the complement of such an expression, e.g., $\overline{I}$, or if they can be any value a . is used.

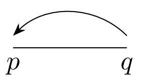
Note, since we are considering graphs, two points could be linked both directly and indirectly. We track parent information to know about parents of items, and existence of specific edges. If both an indirect link and a direct link are present, the state records a direct link.

4.5.4 Eisner (1996)'s Algorithm

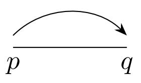
This work and Pitler et al. (2013)'s algorithm both extend upon the ideas first described by Eisner (1996). Eisner's algorithm is an $O(n^3)$ dynamic program for projective dependency tree parsing. If the reader is familiar with Eisner's algorithm, then this definition of it using our notation may help build intuition for our algorithm:

Arc creation rules:

$I[pq\ F\ P] \leftarrow \max$
$A[p\ q] \quad I[pq\ F\ \boxed{F}]$

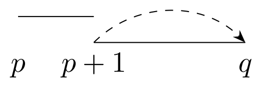


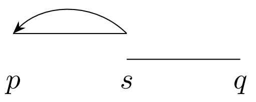
$I[pq\ Q\ F] \leftarrow \max$
$A[q\ p] \quad I[pq\ \boxed{F}\ F]$



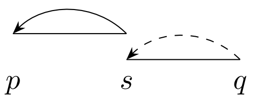
Binary composition rules:

$$I[pq\ F\ F] \leftarrow \max \begin{cases} I[p\ p{+}1\ F\ F] \quad I[p{+}1\ q\ \overline{F}\ F] \\ \\ \max_{s\in(p,q)} \\ \{\ I[ps\ F\ P] \quad I[sq\ \boxed{F}\ F] \end{cases}$$

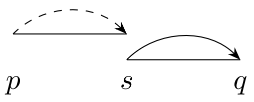




$I[pq\ F\ p] \leftarrow \max_{s\in(p,q)}$
$I[ps\ F\ P] \quad I[sq\ F\ \overline{F}]$

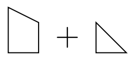


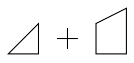
$I[pq\ q\ F] \leftarrow \max_{s\in(p,q)}$
$I[ps\ \overline{F}\ F] \quad I[sq\ Q\ F]$

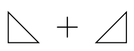


To be specific, this is a version of the algorithm in which the structure is built from the outside in[3], and where arc creation is separated out into an independent step[4]. A projective graph parsing algorithm can be formed by changing the boxed blue parts to be $\overline{P}$ (upper left), $\overline{Q}$ (lower left), and $\bullet$ (right). These rules are all present in our algorithm as well, they correspond to the rules that only use intervals.

Eisner's algorithm has been formulated in several different ways that share the same core idea, but structure the deduction rules slightly differently. One well-known formulation was presented by Eisner and Smith (2005). Their items are slightly different, with their triangle items going from on one word to between two words (whereas ours are always from on one word to on another word). However, we can roughly relate the inference rules as follows, where some of our rules take place in two steps in their rules:

 corresponds to the third binary rule, or half of the second binary rule.

 corresponds to the fourth binary rule, or half of the first binary rule.

 corresponds to half of the first two binary rules, plus either of the arc creation rules.

Another connection to make at this point is with the CFG version of Eisner's algorithm, presented by Johnson (2007). It is probably not possible to perform a similar transformation on the algorithm we propose, though in Section 4.9 we discuss similar challenges in terms of correctly computing inside and outside scores as are raised in Section 4 of Johnson (ibid.).

[3] Alternatives are from the inside out, always left to right, or always right to left.

[4] Arc creation could be combined into the binary step, though for a first-order model there should be an independent search for the best arc over each span, so that complexity is $O(n^3 + an^2)$ rather than $O(an^3)$, where a is the number of arc types.

4.5.5 Initialization

To start, an item is added for every position p in the sentence, except the last one. The items are:

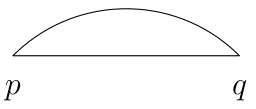
$$I[p\ p{+}1\ F\ F]$$

If any type of word label is being used, an item must be inserted for every pair of possible labels.

4.5.6 Interval

The first way we subdivide into cases is based on whether there is a direct edge linking the two ends of the item.

Direct Edges Present



This covers two possible structures, one with the edge going from p to q and one with the edge going from q to p. In both cases, the other end must have a parent state of F, as otherwise we would have a directed cycle. We can break these cases down into the edge, plus the item beneath it:

Algorithm 4.3 Making I with direct edges.

```
$I[pq\ F\ P] \leftarrow \max A[p\ q] \quad I[pq\ F\ \overline{P}]$
$I[pq\ Q\ F] \leftarrow \max A[q\ p] \quad I[pq\ \overline{Q}\ F]$
```

If we were only considering trees then we would constrain the sub-item to be $I[pq\ F\ F]$.

No Direct Edges Present

First consider the case when p does not have a parent in this item (the case when it does is symmetrical to the case when q has a parent, which is covered by this case).

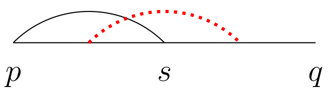
If there are no p–(pq) edges, then this can be broken into two intervals, splitting at $p+1$. The second item has an $\overline{F}$ because $p+1$ must get a parent from somewhere.

Algorithm 4.4 Making I, case one.

```
$I[p\ p{+}1\ F\ F] \quad I[p{+}1\ q\ \overline{F}\ F]$
```

If p–(pq) edges do exist, consider the longest such edge, ps. Consider the set C, of edges that cross ps (i.e., (ps)–(sq)). Depending on the size of the set C we get different decompositions.

$|C| = 0$

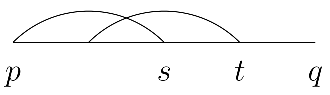


In this case there are no p–$(sq]$ edges, because of how s was chosen. There are no (ps)–$(sq]$ edges, because in this case $|C| = 0$. Therefore we can break this case down into two intervals. The left interval contains a direct edge from the left end to the right end. The right interval can have a range of options for parent sets, which we leave undefined in the template, to be constrained later using the general rules.

Algorithm 4.5 Making I, case two.

$I[ps\ F\ P] \quad I[sq\ .\ .]$

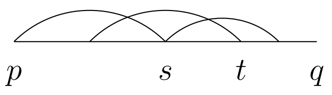
$|C| = 1$



Here we have made a choice, placing t outside of ps. Placing it inside would also lead to a valid set of deduction rules, but allowing both creates an ambiguity.

In this case there are no (ps)–(st) edges by construction, and there are no (st)–$(tq]$ edges, as they would violate the one-EC property for the edge ending at t. We subdivide this into two cases, depending on whether there is an s–(tq) edge.

$|C| = 1$
Edge present



In this case we have an item $[ps.t]$, because $|C| = 1$ means there are no further $[ps)$–(sq) edges. The edge with an endpoint at t could be crossed by an s–(ps) edge, but not by a p–(ps) edge, as that would violate the one-EC property. This means $[ps.t]$ is either an N or and R, and in either case will have the edge ps.

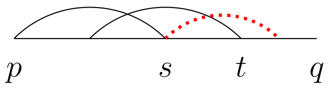
We also have an interval, $[st]$, because any (st)–(tq) edges would violate the one-EC property for the edge with an endpoint at t, and there are no (st)–(ps) edges by construction.

The third item is $[tq.s]$, for reasons already given (in terms of what edges are allowed). This cannot be an R or B because that would violate the one-EC property for the s–(tq) edge. If there is only one s–$(tq]$ edge, and it is sq, then this item is an X. Otherwise it will be either an L or N, depending on whether the s–(tq) edge is crossed by a t–(tq) edge. If st or ts exists, we attribute it to the middle sub-item (it could come from any of them, so this avoids a spurious ambiguity).

Algorithm 4.6 Making I, case four.

$RN\cdot[pst\ F\ P\overline{T}\ \overline{PS}] \quad I[st\ .\ .] \quad \cdot LNX[tqs\ .\overline{S}\ .\ \overline{T}.]$

$|C| = 1$
No edge

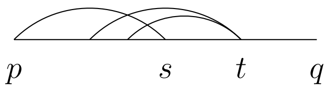


By the same argument as above, we have an item $[ps.t]$, and no (st)–(tq) edges. Since we don't have any s–(tq) edges, the third item is now an interval, $[tq]$. $[st]$ is still an interval, by the same argument as above. The first item is less restricted than before, it could be am L, R, or N. It cannot be a B because there is only one (ps)–t edge. In this case it's also possible that $t = q$, in which case there is no third item (it would have width 0).

Algorithm 4.7 Making I, cases three and five (part one).

```
$LRN\cdot [psq\ F\ P\overline{Q}\ \overline{PS}]$    $I[sq\ ..]$
$LRN\cdot [pst\ F\ P\overline{T}\ \overline{PS}]$    $I[st\ ..]$    $I[tq\ ..]$
```

$|C| > 1$
Outside

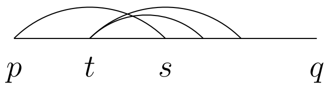


This is the first of two cases where $|C| > 1$, distinguished by the position of t, either inside or outside of ps. Following an argument similar to the one above, we get the same derivation as shown in Algorithms 4.6 and 4.7, the only differences is that $[ps.t]$ must contain at least two t–(ps) edges, and as a result the left item could be a B now.

Algorithm 4.8 Making I, cases three and five (part two).

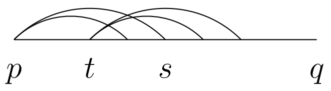
```
$B\cdot [psq\ F\ P\overline{Q}\ \overline{PS}]$    $I[sq\ ..]$
$B\cdot [pst\ F\ P\overline{T}\ \overline{PS}]$    $I[st\ ..]$    $I[tq\ ..]$
```

$|C| > 1$
Inside



In this case there are no (pt)–(ts) edges, as they would violate the one-EC property for the edges in C. There are also no (pt)–(sq) edges, as they would violate the one-EC property for ps. As in the previous case there are two decomposition options, this time depending on whether there is a p–(ts) edge.

$|C| > 1$
Inside
Edge present

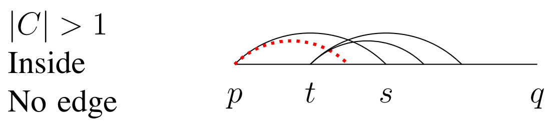


In this case there are no (pt)–$(ts]$ edges, and no (ts)–$(sq]$ edges, because they would violate the one-EC property for the edges in C. There are also no (pt)–$[sq]$ edges, because they would violate the one-EC property for the p–(ts) edges. There are no p–$(sq]$ edges by construction. This means the left item is $[pt]$, the middle item is $[ts.p]$, and the right item is $[sq.t]$. The left item is an I. The middle item can be an L or N, but an R would violate the one-EC property for the p–(ts) edge. The right item must be an N with multiple edges from the external point to the interior of the span. Any other option would violate the one-EC property for edges in C.

Additional constraints are added to ensure direct ts and pt edges can occur in only one of the items. Also, sq cannot occur, as it would violate the one-EC property for edges in C.

Algorithm 4.9 Making I, case six.

$I[pt\ F\ .] \quad \cdot LN[tsp\ .\overline{P}\ .P\ F] \quad \cdot \utilde{N}[sqt\ \overline{QT}\ \overline{S}.\ \overline{S}.]$



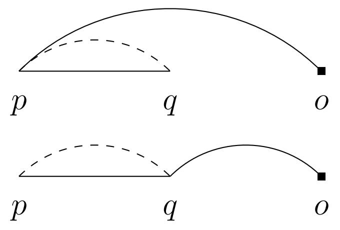
This case is similar to the case above, with most of the argument still holding. The difference is that with the constraint of no edges as shown, $[ts]$ is an I. Meanwhile, there can be s–(pt) edges. Again, we add extra constraints to ensure the direct ts edge can only occur in one item.

Algorithm 4.10 Making I, case seven.

$RNX\cdot [pts\ F\ .\overline{S}\ P\overline{T}] \quad I[ts\ .\ .] \quad \cdot \utilde{LN}[sqt\ .\overline{T}\ .\ \overline{S}.]$

4.5.7 Exterval

In this case we simply need to remove the edge from o to the span and we will have an interval. Let the external point be to the right of the span (having it on the left is symmetrical). The two general structures are as follows, with the dashed lines indicating that a pq edge may or may not be present, and the small square indicating the external point:



The way the other deduction rules are structured, we never encounter the case where both of the solid edges are present. We always remove the solid edges first, leaving the dashed edge (if present) for the interval decomposition. Since this is the only edge with an endpoint at O we also know there are no indirect connections between o and p or q. There are four possibilities in total, depending on which direction the edge goes in:

Algorithm 4.11 Making X with direct edges.

```
$X[pqo\ .O\ .\overline{O}\ F] \leftarrow \max A[o\ p] \quad I[pq\ .\ .]$
$X[pqo\ .\overline{O}\ .\overline{O}\ P\overline{Q}] \leftarrow \max A[p\ o] \quad I[pq\ .\ .]$
$X[pqo\ .\overline{O}\ .O\ F] \leftarrow \max A[o\ q] \quad I[pq\ .\ .]$
$X[pqo\ .\overline{O}\ .\overline{O}\ \overline{P}Q] \leftarrow \max A[q\ o] \quad I[pq\ .\ .]$
```

4.5.8 Both

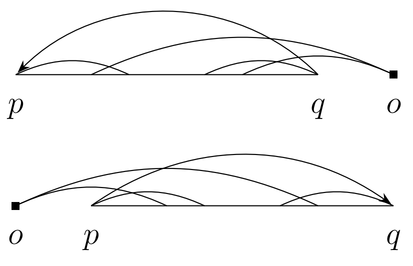
The way we decomposed the Interval case has implications for the range of possible Both structures. Specifically, it can only come up in two ways:

$$B\cdot\,[psq\ F\ P\overline{Q}\ \overline{PS}]$$
$$\cdot\,B[pqo\ Q\overline{O}\ F\ \overline{PQ}]$$

The B item is not involved in any further decompositions, so these are the only cases we need to consider. As in the Interval case, we start by separating out cases with direct edges and those without.

Direct Edges Present

In this case we have two possible structures, with additional edges shown as required by the definition of a B item:



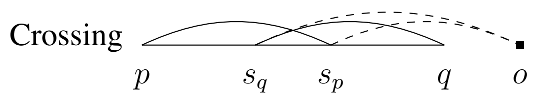
In each case we remove the p–q edge.

Algorithm 4.12 Making B with direct edges.

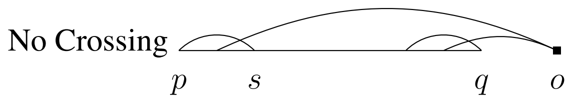
```
$B\cdot\,[pqo\ F\ P\overline{O}\ \overline{PQ}] \leftarrow \max A[p\ q] \quad B\cdot\,[pqo\ F\ \overline{PO}\ \overline{PQ}]$
$\cdot\,B[pqo\ Q\overline{O}\ F\ \overline{PQ}] \leftarrow \max A[q\ p] \quad \cdot\,B[pqo\ \overline{QO}\ F\ \overline{PQ}]$
```

No Direct Edges Present

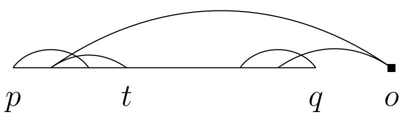
Consider the case when o is to the right of the span (the case when o is to the left of the span is symmetrical). Consider the longest p–(pq) edge. There are two general cases, depending on whether this edge crosses any q–(pq) edges.



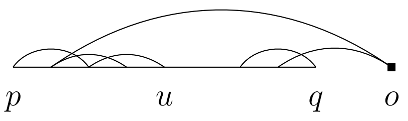
The dashed edges must be present because this is a B. They must end at s_p and s_q because otherwise they would violate the one-EC property for either ps_p or qs_q. Once those edges are set, the one-EC property blocks the following edge types: o-(pq), q-(ps_q), p-(s_pq), (ps_q)-(s_pq), $[ps_q)$-(s_qs_p), (s_qs_p)-$(s_pq]$. This means it can be decomposed into three items, $[ps_q.s_p]$, $[s_qs_p.o]$, $[s_pq.s_q]$. The left of these cannot be an L, as that would violate one-EC for s_p–(ps_q) edges, but it can be an R or N. Similarly, the right item cannot be an R, but can be an L or N. The middle item would be an X, but with a modification to our definition to allow edges from both s_p and s_q to o.

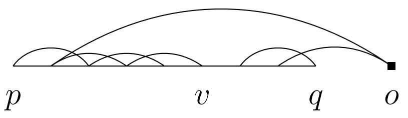


The way we choose the split point in this case will mean we need to track an extra bit of information for L items. If no (ps)–(sq) edge exists, then s is the split point. If an edge does exist, the one-EC property will mean there is only one o–(ps) edge and these two edges share an endpoint:

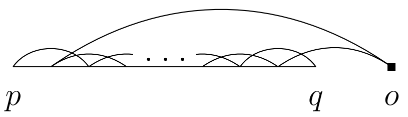


Again, there will be two possibilities, either there are no (pt)–(tq) edges, in which case we use t as the split point, or there are such edges, in which case the pattern continues:





Eventually this chain of crossing edges will either stop in the middle, or cross the edge shown between q and (pq). In the second case it must have the following structure, or otherwise it would have a one-EC violation for reasons symmetrical to the constraints that started the chain:



In this final case, we cannot define a simple decomposition of the structure using the item types we consider. By not covering this case we lose completeness for the space of one-endpoint-crossing graphs. However, as in the problematic $O(n^5)$ case, this is rarely observed in the treebank.

Returning to the case where the chain of crossing edges ends earlier, we use that as the split point and have two items, $[ps.o]$ and $[sq.o]$. The left item must have at least one crossing of an edge o-(ps) and an edge p-(ps), but there cannot be a crossing between o-(ps) and any edge (ps)-(ps) as that would violate the definition of a B. Therefore the left item is either an L or an N with an edge ps. Similarly, the right item is either an N with an edge sq, or an R. In order to avoid a spurious ambiguity, we also need to ensure that the left item has the chaining property discussed above. If the left item is an N, it does, as the edge ps provides a suitable split point. For an L we need to track whether the property is true or not. In practice we avoid the chaining pattern entirely, using the constraint that L must also contain the edge ps, with no loss in coverage. For completeness, we show how to track the chaining property in the following section.

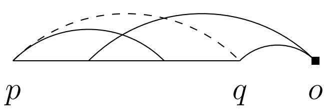
Algorithm 4.13 Making B.

```
$B\cdot\,[pqo\ F\ \overline{PO}\ \overline{PQ}] \leftarrow \max_{s\in(p,q)}$
$LN\cdot\,[pso\ F\ P\overline{O}\ \overline{PS}]$    $R\cdot\,[sqo\ .\ .\overline{O}\ .\overline{Q}]$
$LN\cdot\,[pso\ F\ P\overline{O}\ \overline{PS}]$    $N\cdot\,[sqo\ \overline{Q}.\ S\overline{O}\ .\overline{Q}]$
$LN\cdot\,[pso\ F\ P\overline{O}\ \overline{PS}]$    $N\cdot\,[sqo\ Q.\ \overline{SO}\ .\overline{Q}]$

$\cdot\,B[pqo\ \overline{QO}\ F\ \overline{PQ}] \leftarrow \max_{s\in(p,q)}$
$\cdot\,L[pso\ .\overline{O}\ .\ \overline{P}.]$    $\cdot\,RN[sqo\ Q\overline{O}\ F\ \overline{SQ}]$
$\cdot\,N[pso\ S\overline{O}\ \overline{P}.\ \overline{P}.]$    $\cdot\,RN[sqo\ Q\overline{O}\ F\ \overline{SQ}]$
$\cdot\,N[pso\ \overline{SO}\ P.\ \overline{P}.]$    $\cdot\,RN[sqo\ Q\overline{O}\ F\ \overline{SQ}]$
```

4.5.9 Left

The other decompositions, and the way we will structure this one, enable us to constrain the possible direct edges for L items. Specifically, there is never a po edge. To make this the case there will be instances later in this section where the decomposition provides a choice where an edge could come from one of two sub-items. We will make that choice to ensure this property (no po edges) remains true.



First we have a rule to remove the qo edge. For L items we also have to track whether there are multiple o–(pq) edges. Adding this edge guarantees that there are multiple such edges.

Algorithm 4.14 Making L with direct edges (part one).

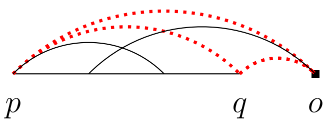
```
$\underset{\sim}{L}[pqo\ .\overline{O}\ .O\ \overline{PQ}] \leftarrow \max A[o\ q]\quad L[pqo\ .\overline{O}\ .\overline{O}\ \overline{PQ}]$
$\underset{\sim}{L}[pqo\ .\overline{O}\ .\overline{O}\ \overline{P}Q] \leftarrow \max A[q\ o]\quad L[pqo\ .\overline{O}\ .\overline{O}\ \overline{PQ}]$
```

Once that edge is removed we are either left with an L that has no direct edges, or one with the dashed edge above. Note, if the dashed edge is present, then the chaining property from the previous section is true. Next are rules to remove that edge:

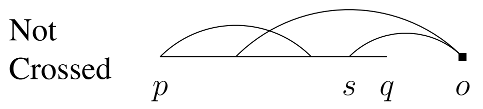
Algorithm 4.15 Making L with direct edges (part two).

```
$L[pqo\ Q\overline{O}\ \overline{PO}\ \overline{PQ}] \leftarrow \max A[q\ p]\quad L[pqo\ \overline{QO}\ \overline{PO}\ \overline{PQ}]$
$L[pqo\ \overline{QO}\ P\overline{O}\ \overline{PQ}] \leftarrow \max A[p\ q]\quad L[pqo\ \overline{QO}\ \overline{PO}\ \overline{PQ}]$
```

Now we are considering an L with no direct edges (as indicated by the red dotted lines):



We will be dividing it into two pieces. To determine the split point, consider the o–(pq) edges. For these edges, let s be the endpoint furthest from p. Using this definition we have that there are no edges o–$(sq]$, There are also no edges (ps)–(sq) by the definition of L. Therefore we can divide the item into $[ps.o]$ and $[sq.p]$. To determine the type of these two regions, consider whether or not os is crossed.

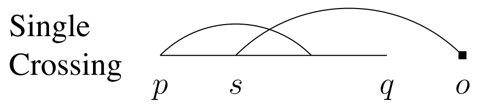


By construction, there are no $[ps)$–$(sq]$ edges, and so $[sq.p]$ is actually the interval $[sq]$. For the left part of the item, consider whether there is a crossing between p–(ps) edges and o–(ps) edges. If such a crossing exists, $[ps.o]$ is an L. Otherwise, it is an N. The N case occurs when the only edge crossing the o–(pq) edges is an edge ps, which ends inside (pq), making the complete item an L, but does not once we split at s.

Algorithm 4.16 Making L, cases three through eight.

```
$L[pso\ .\overline{O}\ .O\ \overline{PS}] \quad I[sq\ .\ .]$
$L[pso\ .\overline{O}\ .\overline{O}\ \overline{P}S] \quad I[sq\ .\ .]$
$N[pso\ \overline{SO}\ PO\ \overline{PS}] \quad I[sq\ .\ .]$
$N[pso\ \overline{SO}\ P\overline{O}\ \overline{P}S] \quad I[sq\ .\ .]$
$N[pso\ S\overline{O}\ \overline{PO}\ \overline{PS}] \quad I[sq\ .\ .]$
$N[pso\ S\overline{O}\ \overline{PO}\ \overline{P}S] \quad I[sq\ .\ .]$
```

In all six of these rules, the chaining property from the previous section is not present. We know this because there is no crossing between edges in the I items and edges in the N or L items, and so no edge ending at q can be part of a chain of crossing edges starting at p.

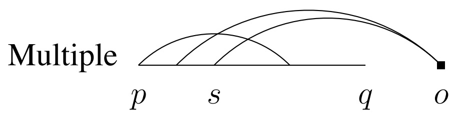


Consider the case when os is the only o–(pq) edge. Here, the first sub-item is an X, $[ps.o]$. For the second sub-item, all p–$(s, q]$ edges will be crossed by os and so cannot be crossed by q–(sq) edges. If there is an s–(sq) edge that crosses a p–(sq) edge then this sub-item is an L, otherwise it is an N. Also, the item being split in this case can be marked as containing only one edge from the external point into the span.

Algorithm 4.17 Making L, cases nine and ten.

```
$X[pso\ .\overline{O}\ .O\ F] \quad \cdot LN[sqp\ .\overline{P}\ .\overline{P}\ \overline{SQ}]$
$X[pso\ .\overline{O}\ .\overline{O}\ \overline{P}S] \quad \cdot LN[sqp\ .\overline{P}\ .\overline{P}\ \overline{SQ}]$
```

Both of these rules could produce items with the chaining property. If the right item is an L and it has the chaining property, then the final item will have it too. This is because we know the chain must start at s, and so the X effectively adds one link to the chain. In the case of an N item, the chaining property will be true if the N contains the pq edge, and false otherwise.



Consider the case when os is not the only o–(pq) edge. If there is an o–(ps) edge that is crossed by a p–(ps) edge then $ps.o$ is an L. Otherwise, $[ps.o]$ is an N (being an R is not possible as by definition, no (pq)–(pq) edge can cross a (pq)–o edge). For the second half, all p–(sq) edges are going to cross two or more o–$(ps]$ edges, and so the one-EC property means they cannot be crossed by any $[sq]$–$[sq]$ edges. Therefore, $[sq.p]$ is an N.

Algorithm 4.18 Making L, cases one and two.

```
$LN[pso\ .\overline{O}\ .O\ \overline{PS}]$    $\cdot N[sqp\ \overline{QP}\ \overline{SP}\ \overline{SQ}]$
$LN[pso\ .\overline{O}\ .\overline{O}\ \overline{P}S]$    $\cdot N[sqp\ \overline{QP}\ \overline{SP}\ \overline{SQ}]$
```

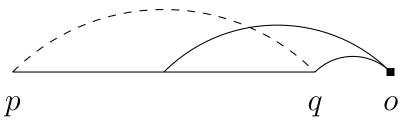
In these cases the chaining property is false. The chain cannot exist because the multiple (pq)–o edges limit the crossing that can occur within the span (any edge crossing the longest p–(pq) edge would violate the one-EC property).

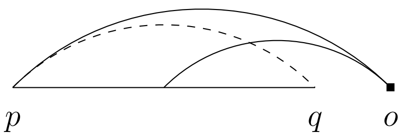
4.5.10 Right

Defined symmetrically to L.

4.5.11 Neither

Here we have the most flexibility in direct edges, though the structure of the decompositions means that we never have the case of both op and oq at the same time.

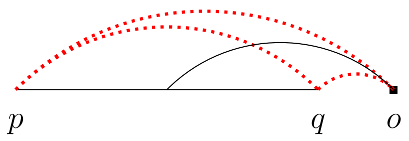




As in the L case, we remove the op or oq edge first, then the pq edge. The definition of an N item also means that any N that has an op or oq edge also meets the criteria of having two o–$[pq]$ edges.

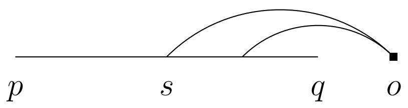
Algorithm 4.19 Making N with direct edges.

```
$\utilde{N}[pqo\ .O\ .\overline{O}\ \overline{PQ}] \leftarrow \max A[x\ i]\ \ N[pqo\ .\overline{O}\ .\overline{O}\ \overline{PQ}]$
$\utilde{N}[pqo\ .\overline{O}\ .\overline{O}\ P\overline{Q}] \leftarrow \max A[i\ x]\ \ N[pqo\ .\overline{O}\ .\overline{O}\ \overline{PQ}]$
$\utilde{N}[pqo\ .\overline{O}\ .O\ \overline{PQ}] \leftarrow \max A[x\ j]\ \ N[pqo\ .\overline{O}\ .\overline{O}\ \overline{PQ}]$
$\utilde{N}[pqo\ .\overline{O}\ .\overline{O}\ \overline{P}Q] \leftarrow \max A[j\ x]\ \ N[pqo\ .\overline{O}\ .\overline{O}\ \overline{PQ}]$
$N[pqo\ Q\overline{O}\ \overline{PO}\ \overline{PQ}] \leftarrow \max A[j\ i]\ \ N[pqo\ \overline{QO}\ \overline{PO}\ \overline{PQ}]$
$N[pqo\ \overline{QO}\ P\overline{O}\ \overline{PQ}] \leftarrow \max A[i\ j]\ \ N[pqo\ \overline{QO}\ \overline{PO}\ \overline{PQ}]$
```



Now we can consider the case with no direct edges. As for L, we will define the split point based on the o–(pq) edges. The split point will be the endpoint furthest from o (i.e., leftmost or rightmost in (pq), depending on which side o is on)[5]. The two sides are symmetrical, so we will only show the case with o to the right here. There are two cases, depending on how many edges are in the o–(pq) set.

$|o–(pq)| > 1$

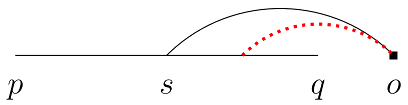


There are no (ps)–(sq) edges by definition, and no o–$[ps)$ edges by construction. Therefore $[ps]$ is an I. There are two o–$[sq)$ edges, and neither is crossed by a (pq)–(pq) edge by definition, therefore $[sq.o]$ is an N.

Algorithm 4.20 Making N, cases three and four.

```
$I[ps\ .\ .]\quad N\cdot[sqo\ \overline{Q}O\ \overline{SO}\ \overline{SQ}]$
$I[ps\ .\ .]\quad N\cdot[sqo\ \overline{QO}\ \overline{SO}\ S\overline{Q}]$
```

$|o–(pq)| = 1$



[5] Note that we could have used the point closest to o and also formed a valid set of deduction rules, though slightly different from the ones shown here. Choices like this make impact parsing efficiency by determining which structures are competing on which beams, but should not change accuracy. Characterizing the space of algorithms similar to this one is an interesting question beyond the scope of this work.

The reasoning for $[ps]$ is the same as above. For $[sq.o]$ we only have one o–$[sq]$ edge, namely os, and so this is an X.

Algorithm 4.21 Making N, cases seven and eight.

```
$I[ps\ .\ .]$    $X\cdot [sqo\ .O\ .\overline{O}\ \overline{SQ}]$
$I[ps\ .\ .]$    $X\cdot [sqo\ .\overline{O}\ .\overline{O}\ S\overline{Q}]$
```

4.5.12 Additional Constraints

The description above gives the complete decomposition, additional general rules are used to determine the specific rule set. They are enforced by the code that converts the templates into all valid deduction rules:

- Items cannot contain cycles. This also implies that every item must have at least one point without a parent.
- When combining two items, any points that will end up in the middle of the new span must have a parent in one of the items being combined.

Both of these can be enforced in the rule creation because we have the complete set of parent connections present in the definition of what combinations are allowed.

4.6 Algorithm Properties

4.6.1 Derivational Ambiguity

One key difference between our algorithm and the original one-endpoint-crossing algorithm presented by Pitler et al. (2013) is that we avoid spurious ambiguity in the ways a parse can be decomposed. Most of the cases in which this ambiguity arises in Pitler et al. (ibid.)'s algorithm are due to symmetry that is not explicitly broken. For example, the choice of t in the description of Intervals. We resolved these issues by breaking such cases of symmetry, and altering the definitions of the items to ensure they cannot represent equivalent structures. By avoiding derivational ambiguity we reduce the search space, and enable efficient summing over the space.

4.6.2 Complexity

We will define the complexity in terms of:

- n, the number of words in a sentence.
- e, the number of edge types in the model.
- s, the number of word labels, e.g., Part-of-Speech tags, in the model.

Unlabeled Structure We can determine the complexity by considering how many different ways the deduction rules can apply. For the ternary rules, the left end could be one of $O(n)$ words, the right end could be one of $O(n)$ words, and each of the two split points could be one of $O(n)$ words, giving $O(n^4)$ overall. The precise number is smaller, since if we choose word p as the left end then we have $n - p$ choices for the first split point, and similarly for the other two points, but the asymptotic complexity remains the same. Similarly, for binary rules with an external point there are four positions (left, right, split, external), and so the complexity is $O(n^4)$. The other rules have fewer possible instantiations, and so do not impact the overall complexity. This means the overall complexity is $O(n^4)$.

Edge labels Since our algorithm separates edge creation and item combination, these only impact the edge creation step. There are $O(n^3)$ possible items (left end, right end, external point), each of which could have up to six edges (two directions, between any pair of three points). However, that over counts drastically, since the same edge can appear in many different items. Instead we can compute all edge scores for each pair of words and store the best option. That gives a complexity of $O(en^2)$, where there are n possible left endpoints, n possible right endpoints, and e possible labels. Together with the complexity of combinations from above, we get $O(n^4 + en^2)$.

Word labels Following the same logic as above, we can update the two terms in our complexity. In both cases the same logic follows, except now there are sn options for each position, rather than n, this is because we need to consider word–label combinations rather than just words. This changes the complexity to $O(s^4n^4 + es^2n^2)$

Depending on the number of edge types, the number of word labels, and the average length of sentences, either the first or second term could dominate the complexity. We will see in the next section that to formulate constituency parsing in a way compatible with our algorithm we will end up with e and s both around a thousand[6], making full enumeration of the dynamic program too slow in practice. We resolve this issue through several types of pruning, described in Section 4.9.3.

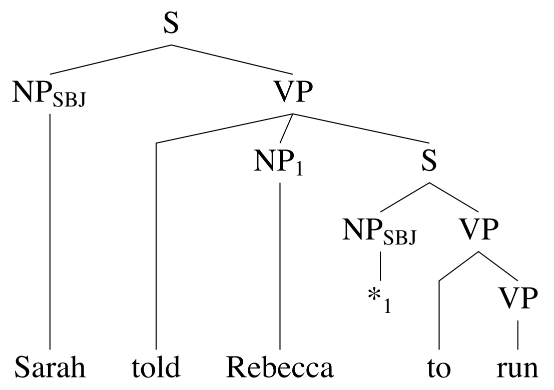
Finally, it is also worth noting that there is an important constant in the complexity, relating to the number of rules. For any set of split points there are multiple items that could be combined. Once our templates are expanded to show the possible state combinations there are 49,292 rules. However, at the end of Section 4.9.3 we will also see how the number of rules can be drastically reduced, to 158, with almost no loss in coverage.

4.7 Parse Representation

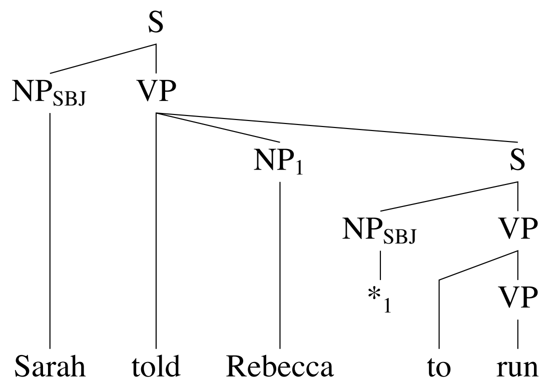
Our algorithm relies on the assumption that we can process the dependents to the left and right of a word independently, then combine the two halves. This means we need lexicalized structures,

[6] While there are less than fifty Part-of-Speech tags in the Penn Treebank, we will be using word labels to do much more in our representation.

(a) Standard PTB structure.



(b) Lexicalized PTB structure.



(c) Our representation.

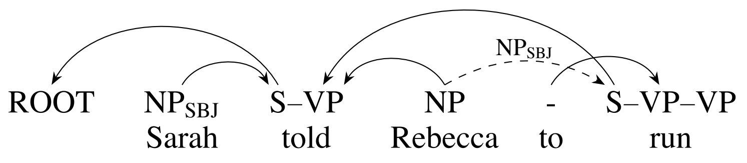


Figure 4.18: Parse representations for graph structures.

which the PTB does not provide. To address this issue, we define a new representation in which each non-terminal symbol is associated with a specific word (the head). Unlike dependency parsing, we retain all the information required to reconstruct the constituency parse.

Our representation is based on Carreras et al. (2008)'s tree representation, with three key differences: (1) we encode all non-terminals explicitly, rather than implicitly through adjunction operations, which can cause ambiguity in their structure, (2) we add representations of null elements and co-indexation, (3) we modify the head rules to avoid problematic structures.

4.7.1 Core Structure

Figure 4.18 shows a comparison of the standard PTB representation, a lexicalized version of the PTB, and our representation. The changes we make are motivated by constraints imposed by our algorithm. Specifically, the algorithm only allows two types of structure to be built: word labels, and edges between pairs of words. In contrast, the standard PTB representation has structural components that do not fit directly into either labels or edges, for example, the S at the top in (a) is not associated with any particular word (and so cannot be a label) and is not associated with a particular pair of words either (and so cannot be an edge).

We address this issue by lexicalizing the PTB, as shown in (b). Going from (a) to (b), all we have done is shift some of the non-terminal symbols left or right; the connectivity is the same, and so the structure is the same. These shifts associate each non-terminal symbol with a specific word, it's *head*. For example, the S mentioned above is associated with *told*, indicated by the fact that it is now directly above *told*.

Going from (b) to (c), we are pushing non-terminals down to be directly above their head, and adjusting the lines to suit that change. Diagonal lines in (b) become curved lines in (c); these will be the edges in our algorithm. The direction of the curved lines indicates which way was up in (b). Vertical lines in (b) become horizontal lines in (c); together with the non-terminals these become special symbols called *spines*. We will discuss how the null element is handled in Section 4.7.2.

Aside from lexicalization, which adds information (the choice of head), this process has only pushed around notation without changing the structure. However, we now have the structure formulated as two components. Every word is assigned a *spine*, and every word is the child of one non-trace edge. Together, these form the base tree structure of the parse.

Spines Shown in black immediately above each word, a spine is the ordered set of non-terminals that the word is the head for, e.g., S-NP for *told*. If a symbol occurs more than once in a spine, we use indices to distinguish between instances.

Edges A link between two words, with a label indicating: (1) the top of the child's spine, and (2) the symbol that it connects to in the parent's spine[7]. In our figures the arcs show their label by starting and ending at the appropriate symbols.

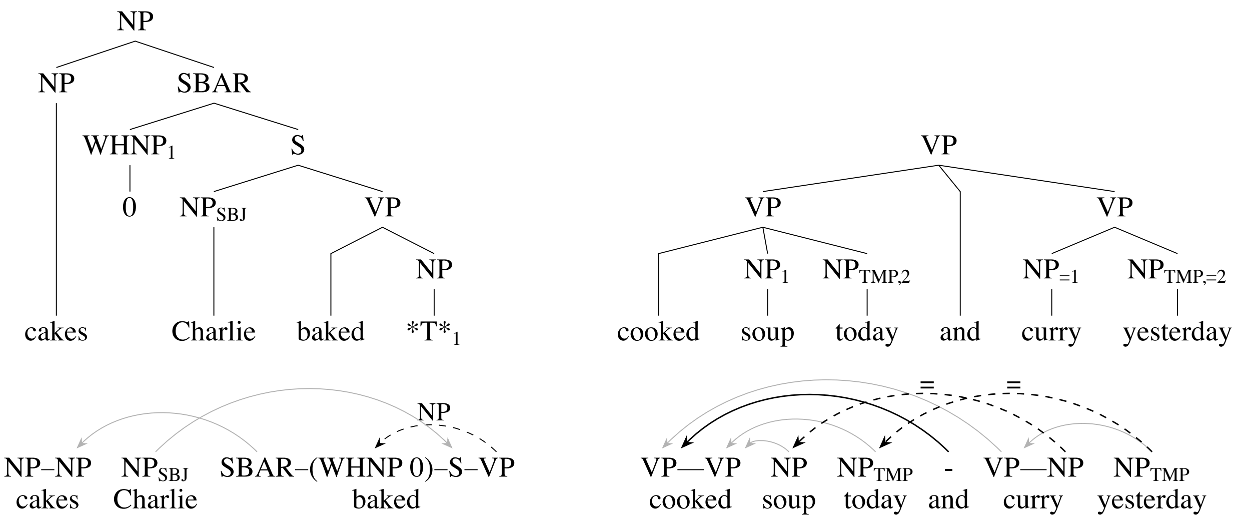
Avoiding Adjunction Ambiguity

Carreras et al. (2008) use r-adjunction to add additional non-terminals to spines. This introduces ambiguity, because edges modifying the same spine from different sides may not have a unique order of application. We resolve this issue by using more articulated spines that have the complete set of non-terminals. The potential drawback of our approach is that coverage of spines is more limited by the training set than their approach. In practice, coverage is high, with 99.93% of spines in the development set observed in the training set.

4.7.2 Additional Structure

The base tree representation described above does not cover null elements or co-indexation, and so only fully represents 26.6% of sentences.

[7] In practice we also indicate the sibling symbol, i.e., the symbol preceding the parent in the parent's spine. This is redundant for representing the structure, but is helpful for modeling.



(a) Null to null (b) Gapping

Figure 4.19: Examples of syntactic phenomena. Dashed edges are traces, solid edges are structural. Some edges are fainter to more clearly show the key edges for each case.

Null Elements The Penn Treebank contains a range of null elements, indicating structures such as the trace of movement, PRO, and null complementizers. These elements do not span any words, so do not have a head word. For example, in Figure 4.18 there is a null element that represents the missing subject of the infinitive.

We represent null elements in our structure in two ways. If the null element has a reference index, shown as a subscript in our figures, we represent the null element as part of an edge, as described below. Otherwise, we insert it into a spine, as shown for the null element and its parent WHNP in Figure 4.19a.

Co-indexation The treebank represents movement with index pairs on null elements and non-terminals, e.g., $*_1$ and NP_1 in Figure 4.20a. To represent co-indexation we create extra edges, one for each index, going from the null element to the non-terminal. The edge is labeled to indicate the type of trace and null element. There are three special cases of co-indexation:

(1) The treebank uses a chain of indexes to represent the case of a non-terminal that links to multiple null elements. We represent this case with multiple edges, all starting at the non-terminal and ending at each of the different null positions.

(2) It is possible for a trace edge to have the same start and end points as a non-trace edge. We restrict this situation to allow at most one base edge, one trace edge, and one chain edge at the same time. This decreases edge coverage in the training set by 0.012%.

(3) In some cases the non-terminal does not span any words, but instead contains another null element, e.g., the WHNP in Figure 4.19a. For these we generate an edge, but reverse the direction. This reversal is necessary to avoid creating a loop in the structure. In Figure 4.19a we show a

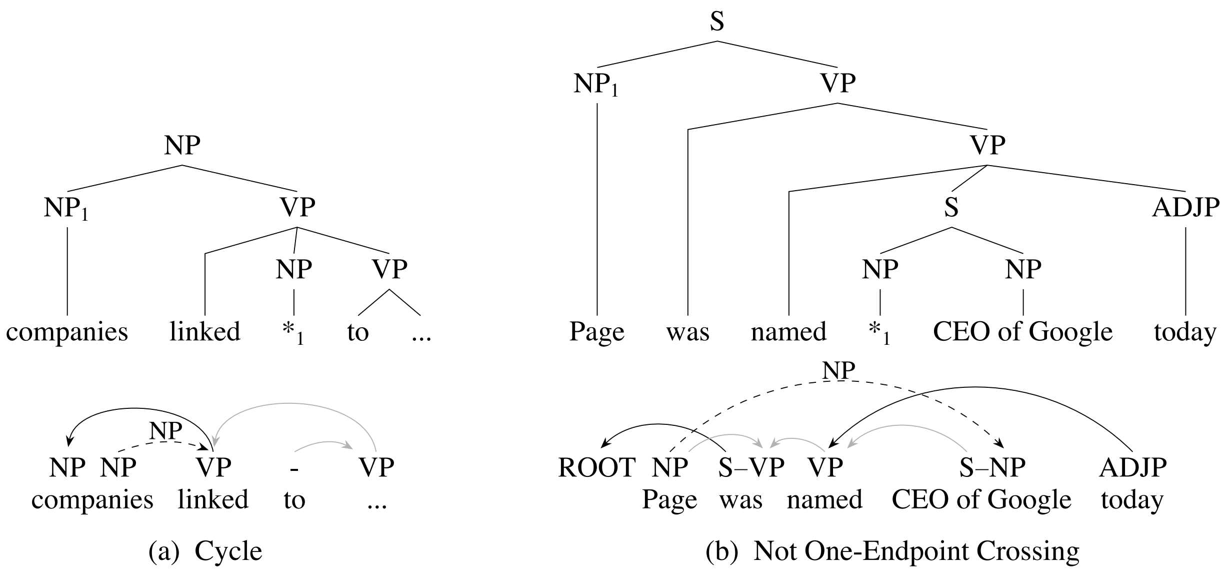


Figure 4.20: Examples of problematic structures. Dashed edges are traces, solid edges are structural. The edges that are problematic are shown in a darker color.

special case where the trace edge links two positions in the same spine. We represent this as part of the spine, rather than as an edge.

Gapping For parallel constructions the treebank co-indexes arguments that are fulfilling the same roles, as shown in Figure 4.19b. These are distinct from the previous cases because neither index is on a null element. We considered two approaches for these cases: (1) add edges from the repetition to the first occurrence (shown in Figure 4.19b), (2) add edges from the repetition to the parent of the first occurrence, e.g., the left VP in Figure 4.19b. The second approach produces more structures that fall within the graph space we consider and explicitly represents all predicates, but only implicitly captures the original treebank structure.

4.7.3 Head Rules

To construct the spines we use head rules that consider the type of a non-terminal and its children. In many cases different head word options represent more syntactic or semantic aspects of the span. When parsing trees, any set of head rules will generate a valid structure. For graphs, the head rules can have a major impact on properties of the final structure. Two particular properties that are relevant to our algorithm are whether cycles are present, and whether the edges obey the one-endpoint crossing property.

Cycles One example of cycles was mentioned in the previous section, but they can occur whenever a trace edge is added, e.g., between *companies* and *linked* in Figure 4.20a. Starting from Carreras et al. (2008)'s head rules, we made modifications to avoid most cycles. For example, in a subordinate clause consisting of a Wh-noun phrase (WHNP) and a declarative clause (S), we switched the head to be the S. Often the WHNP has a trace to within the S, so making it the head would create a cycle.

One-Endpoint Crossing The dark edges in Figure 4.20b show an example of how head rules can impact this property. The trace edge from *Page* to *CEO of Google* does not satisfy the one-EC property because it is crossed by two edges with no endpoints in common. By switching the head rule for VPs to use a child VP rather than an auxiliary, we can resolve this case.

4.8 Algorithm Extensions

To handle our representation, we extend the core algorithm described above to support labels on edges and labels on words (spines). Additionally, we can constrain the search space to structures containing a projective tree of non-trace edges.

4.8.1 Edge Labels and Spines

Edge labels can be added by calculating either the sum or max over edge types when adding each edge, and recording the chosen edge in the back reference for the state. Spines must be added to the state definition, specifying a label for each visible word (p, q and o). This state expansion is necessary to ensure agreement when combining items. Carreras et al. (2008)'s projective tree parser uses a similar approach, while Pitler et al. (2013)'s algorithm is for dependency parsing and so does not consider spines.

4.8.2 Ensuring the Graph Contains a Structural Tree

The algorithm above constrains the space of graph structures, but does not say anything about the edge types being combined. In practice, we are only interested in parses that are composed of trace edges plus a projective tree of structural edges. Since prior work focused on trees, not graphs, support for this constraint has not previously been explored.

To ensure every point gets one and only one structural parent, we add booleans to the state, indicating whether p, q and o have structural parents. When adding edges, a structural edge can not be added if a point already has a structural parent. When combining items, no point can receive more than one structural parent, and points that will end up in the middle of the span must have exactly one. Together, these constrains ensure we have a tree.

To ensure the tree is projective we need to prevent structural edges from crossing. Crossing edges are introduced in two ways, and in both we can avoid structural edges crossing by tracking whether there are structural o–$[pq]$ edges.

When creating edges

Every time we add a pq edge in the N, L, R and B items we create a crossing with all o–(pq) edges. We do not create a crossing with edges oq or op, but our ordering of edge creation means these are definitely not present when we add a pq edge, so tracking structural o–$[pq]$ edges gives us the information we need.

We set the boolean for structural o–$[pq]$ edges to true if it is currently true or we are creating a structural op or oq edge, and false otherwise.

Algorithm 4.22 Rules that could create crossing arcs.

$RN\cdot[ikl\ F\ I\overline{L}\ \overline{IK}]\ \ I[kl\ .\ .]\ \ \cdot LNX[ljk\ .\overline{K}\ .\ \overline{L}.]$	$RNX\cdot[ilk\ \overline{F}\ .\overline{K}\ .\overline{L}]\ \ I[lk\ .\ .]\ \ \cdot LN[kjl\ J\overline{L}\ F\ \overline{KJ}]$
$I[il\ F\ .]\ \ \cdot LN[lki\ .\overline{I}\ .I\ F]\ \ \cdot \utilde{N}[kjl\ \overline{JL}\ \overline{K}.\ \overline{K}.]$	$\utilde{N}\cdot[ikl\ \overline{KF}\ \overline{IL}\ .\overline{K}]\ \ RN\cdot[klj\ .J\ .\overline{J}\ F]\ \ I[lj\ .\ F]$
$RNX\cdot[ilk\ F\ .\overline{K}\ I\overline{L}]\ \ I[lk\ .\ .]\ \ \cdot \utilde{LN}[kjl\ .\overline{L}\ .\ \overline{K}.]$	$\utilde{RN}\cdot[ikl\ \overline{F}\ .\overline{L}\ .\overline{K}]\ \ I[kl\ .\ .]\ \ \cdot LNX[ljk\ .\overline{K}\ F\ \overline{L}J]$
$LN[ikx\ .\overline{X}\ .X\ \overline{IK}]\ \ \cdot N[kji\ \overline{JI}\ \overline{KI}\ \overline{KJ}]$	$N\cdot[ikj\ \overline{KJ}\ \overline{IJ}\ \overline{IK}]\ \ RN[kjx\ .X\ .\overline{X}\ \overline{KJ}]$
$LN[ikx\ .\overline{X}\ .\overline{X}\ \overline{I}K]\ \ \cdot N[kji\ \overline{JI}\ \overline{KI}\ \overline{KJ}]$	$N\cdot[ikj\ \overline{KJ}\ \overline{IJ}\ \overline{IK}]\ \ RN[kjx\ .\overline{X}\ .\overline{X}\ K\overline{J}]$
$X[ikx\ .\overline{X}\ .X\ F]\ \ \cdot LN[kji\ .\overline{I}\ .\overline{I}\ \overline{KJ}]$	$RN\cdot[ikj\ .\overline{J}\ .\overline{J}\ \overline{IK}]\ \ X[kjx\ .X\ .\overline{X}\ F]$
$X[ikx\ .\overline{X}\ .\overline{X}\ \overline{I}K]\ \ \cdot LN[kji\ .\overline{I}\ .\overline{I}\ \overline{KJ}]$	$RN\cdot[ikj\ .\overline{J}\ .\overline{J}\ \overline{IK}]\ \ X[kjx\ .\overline{X}\ .\overline{X}\ K\overline{J}]$

When combining items

We never introduce a crossing when making a B, or in any rule that combines a set of items with only one non-I. That leaves the rules shown in Figure 4.22.

While in general there is the potential that an o–$[pq]$ boolean might be true because of an op or oq edge that does not participate in a crossing, the way we have chosen to set the parent constraints above means this is not the case. Instead, all o–$[pq]$ edges in each pair of items will cross, and so knowing whether any o–$[pq]$ edge is structural is sufficient to determine whether a structural crossing is occurring.

4.9 Implementation

While the core algorithm is fully described in the previous sections, and has desirable asymptotic speed properties, a simple implementation would be slow. Here we describe the additional design decisions needed to enable efficient parsing and modern training methods. Each section below describes one component of the system, its optimizations and the motivation behind them.

4.9.1 Model

We use a discriminative model, which assigns scores to parses using a linear combination of weights. Each weight corresponds to some feature of the parse and input data, e.g., the spine being added is NP and the word to the left is the. Our features are based on the set defined by McDonald et al. (2005).

4.9.2 Learning

We train with an online primal subgradient approach (Ratliff et al. 2007) as described in Kummerfeld, Berg-Kirkpatrick, et al. (2015). We compute the subgradient of the margin objective on each instance by performing a structured loss-augmented decode, then uses these instance-wise subgradients to optimize the global objective using AdaGrad (Duchi et al. 2011) with either L_1 or L_2 regularization. To improve speed, we use sparse updates and batch processing, as described below.

Batches

Instead of updating the model with subgradients calculated for single sentences, we consider the sum over a small number of sentences (a batch). In each pass through the training data we make fewer updates to our model, but each update is based on a more accurate subgradient. This has the advantage that we can parse all of the sentences in the batch in parallel. Since most of our time is spent in parsing, this can produce improvements proportional to the number of CPU cores available. In practice, memory is also a constraint, and CPU utilization is limited by memory bandwidth and possibly memory management by the Java Virtual Machine.

Sparse Updates

Not every weight contributes to the score of every parse, but the simplest implementation of AdaGrad modifies every weight in the model when doing an update. To save time, we distinguish between two different types of update. When the subgradient for a weight is nonzero, we apply the usual update. When the subgradient for a weight is zero, we apply a numerically equivalent update later, at the next time the weight is queried. This saves time, as we only touch the weights corresponding to the (usually sparse) nonzero directions in the current batch's subgradient. The update that occurs later saves time overall because we can combine more than one update together in a simple closed form calculation. Algorithm 4.23 gives pseudocode for our implementation.

Sparse updates do add some memory and computation costs. First, when accessing weights and applying the delayed updates, we need to use synchronization to ensure an exact update. Second, we need to use memory to track the last time each weight was updated and to provide a lock for each weight (used for synchronization).

If we are willing to give up exactness we can avoid the synchronization delay by applying a lock-free approach similar to Recht et al. (2011). Our approach is shown in lines 30 to 33 of Algorithm 4.23. To perform an update we first read all the relevant variables (weight, gradient sum, update time), then calculate the new weight. Before saving the new weight, we check to see if the time of the last update (u_f) is now equal to the current time (n). If it does, we do nothing and continue. If it doesn't, then we set the new update time, then update the weight (that order is important). If multiple threads are applying an update simultaneously then the race to do the update can play out in a few ways. Since the update time is changed before the weight is changed, and all threads are updating to the same new time, we can guarantee that any thread that gets through the time check will be doing the correct update (the worry is that the thread read an inconsistent set of values). If the time has changed then either another thread already did the work, in which case all

Algorithm 4.23 Online Primal Subgradient with ℓ_1 or ℓ_2 regularization, sparse updates

```
 1: Parameter: iters                                   Number of iterations
 2: Parameter: C                    Regularization constant ($10^{-1}$ to $10^{-8}$)
 3: Parameter: $\eta$                       Learning rate ($10^{0}$ to $10^{-4}$)
 4: Parameter: $\delta$                          Initializer for $q$ ($10^{-6}$)
 5: $\mathbf{w} = \mathbf{0}$                                        Weight vector
 6: $\mathbf{q} = \boldsymbol{\delta}$                      Cumulative squared gradient
 7: $\mathbf{u} = \mathbf{0}$        Time of last update for each weight [sparse updates only]
 8: $n = 0$                   Number of updates so far [sparse updates only]
 9: for iter $\in [1, \text{iters}]$ do
10:     for batch $\in$ data do
11:         $\mathbf{g} = \mathbf{0}$       Sum of gradients from loss-augmented decodes
12:         for $(x_i, y_i) \in$ batch do
13:             $y = \text{argmax}_{y' \in Y(x_i)}[\text{SCORE}(y') + \mathbf{L}(y', y_i)]$    Loss-augmented decode
14:             $\mathbf{g} \mathrel{+}= (\mathbf{f}(y) - \mathbf{f}(y_i))$          Update the active features
15:         $\mathbf{q} \mathrel{+}= \mathbf{g}^2$      Add the element-wise square of the subgradient
16:         $n \mathrel{+}= 1$
17:         for $f \in \mathbf{g}$ where $g_f \neq 0$ do    Over all features if not doing sparse updates
18:             $w_f = \text{UPDATE-ACTIVE}(w_f, g_f, q_f)$
19:             $u_f = n$
20: function UPDATE-ACTIVE$(w, g, q)$                           The AdaGrad update
21:     [$\ell_2$] return $\frac{w\sqrt{q} - \eta g}{\eta \mathrm{C} + \sqrt{q}}$
22:     [$\ell_1$] d $= |w - \frac{\eta}{\sqrt{q}} g| - \frac{\eta}{\sqrt{q}} \mathrm{C}$
23:     [$\ell_1$] return sign$(w - \frac{\eta}{\sqrt{q}} g) \cdot \max(0, \mathrm{d})$
24: function UPDATE-CATCHUP$(w, q, t)$     A single update equivalent to a series of AdaGrad
25:     [$\ell_2$] return $w \left( \frac{\sqrt{q}}{\eta \mathrm{C} + \sqrt{q}} \right)^t$      updates where the weight's subgradient was zero
26:     [$\ell_1$] return sign$(w) \cdot \max(0, |w| - \frac{\eta \mathrm{C}}{\sqrt{q}} t$
27: function SCORE$(y')$           Compute $\mathbf{w}^\top \mathbf{f}(y')$, but if doing sparse updates, then
28:     $s = 0$                    for each weight, apply an update to catch up on the
29:     for $f \in \mathbf{f}(y')$ do        steps in which the gradient for that weight was zero
30:         $nw = \text{UPDATE-CATCHUP}(w_f, q_f, n - u_f)$
31:         if $u_f \neq n$ then         For parallel decoding, this enables updates without
32:             $u_f = n$                locking. In some cases, an old weight will be used,
33:             $w_f = nw$                      but $w_f$ will not be updated incorrectly.
34:         $s \mathrel{+}= w_f$
35:     return $s$
```

Note: To implement without the sparse update, use SCORE $= \mathbf{w}^\top \mathbf{f}(y')$, and run the update loop over all features. Also, for comparison, to implement the perceptron, remove the sparse update and use UPDATE-ACTIVE $=$ **return** $w + g$.

is well, or another thread has set u_f, but not w_f, in which case we may continue with an old version of the weight. That slight possibility of using an old weight turns out to not be an issue. We found that avoiding locks did not impact accuracy, but also did not substantially impact speed.

Loss Function

In loss-augmented decoding, we find the parse with the highest score when adding the loss of the parse to the model score. In this context, loss is a measure of the difference between a given parse and the correct parse. To efficiently find the top scoring parse in this case we need the loss function to decompose in a way that matches our dynamic program.

The performance metric we want to optimize, F-score, cannot be decomposed. The simplest alternative would be to use the number of incorrect arcs, as we can adjust the score when adding an arc based on whether the arc is correct or not. For parsing without traces that would be fine, as we are making a fixed number of decisions: one arc and spine per word (i.e., the denominator of the metric we are optimizing is constant). However, when parsing with traces, if there is supposed to be a trace and we leave it out, the mistake is not penalized by the number-of-incorrect-arcs metric.

Instead we use hamming distance, the number of incorrect arcs plus the number of missing arcs. As above, incorrect arcs are easy to account for. For missing arcs, we must be careful to count each mistake exactly once. We can be certain an arc will not be created when a deduction rule is applied that leaves one of the ends in the middle of a span (e.g., when two items are combined the middle point is now in the middle of the span produced). To avoid counting twice (once for each end) we have a few options:

- When combining two halves that would lead to double counting, subtract off to avoid it (at the point of combination we have all the information needed to do so)
- Assign half to each end
- Only count on one end (options include the left end, the right end, the parent, or the child)

In our system we use the first approach.

4.9.3 Inference

The core contributions of this entire chapter is the inference algorithm presented in Section 4.5. The core idea behind the algorithm is to constrain the space to consider so that it can be explored efficiently, while still covering the structures observed in language. Here we describe modifications at various scales that improve speed by pruning the space further.

State beams

In each cell of the chart we use a beam, discarding items based on their Viterbi inside score. We ensure diversity by dividing each beam into a collection of sub-beams. In all three passes, the sub-beams separate items based on their type (N, L, etc), and the parents of each position in the item.

This subdivision enables us to avoid considering most incompatible items. The pass with spines also includes one of the spines in the sub-beam definition for the same reason. We tuned the beam size to prevent the gold structure being pruned in training, with values falling between 100 and 2000.

Cube pruning

We apply the standard cube pruning approach when doing binary and ternary compositions (Chiang 2007). Since we are using sub-beams to determine which items are compatible, we use a heap of sub-cubes during composition. Using fine sub-beams to avoid comparing incompatible items means that there are many of these sub-cubes, and so we also prune entire sub-cubes based on the score of their top item. Again, we tuned how far through the cube we go when combining items by increasing the value until gold structures were not being pruned in training, with the value ranging from 500 to 8000.

Coarse to Fine Pruning

Rather than parsing immediately with the full model we use several passes with progressively richer structure (Goodman 1997):

1. Projective parser without traces or spines, and simultaneously a trace classifier
2. Non-projective parser without spines, and simultaneously a spine classifier
3. Full structure

Each pass prunes using max-marginals from the preceding pass and scores from the preceding classifier. The third pass also prunes spines that are not consistent with at least one unpruned edge from the second pass.

Inside–Outside Calculations

There are two general algorithm classes for parsing that our algorithm can be used within: Viterbi and Inside–Outside. The first of these finds the optimal structure for a sentence under a given model and in the process determines the optimal substructure for every span of the sentence. While that is sufficient for parsing, it does not provide the information we need for pruning. Instead we use the inside–outside algorithm, which computes for every item either the sum or max over all parses that include that item.

When using our algorithm with a model that only places weights on the edges, calculation of the scores is straightforward. Each edge exists in only one place in the derivation, between the item without it and the item with it. Once spines are introduced the situation changes because we would like to score them in all of the items they appear in. This scoring is important for the beams and cube pruning to be effective; if we only scored spines in one of the items they appear in there would be many ties.

For the projective algorithm case each spine is introduced in exactly two items in the derivation, and so we can simply assign half the score to each. For the non-projective version the spine may appear in more locations because it needs to be introduced when we add external points. To correctly calculate the score, and also have effective pruning, we add the complete score every time the spine is introduced and then subtract the score when two items with a spine in common are combined.

Algorithm rule pruning

Many structures that can be generated by our dynamic program are not seen in the data we consider. To improve speed, we leave out all rules that are not used in the derivation of sentences in the training set[8]. Of the 49,292 specific rules in the algorithm, only 158 are needed to generate all sentences in the training set. Narrowing down to these does have implications for coverage, but looking at the development set we found only one rule in one parse that was not in the set of 158.

Figure 4.24 shows the complete dynamic program from Figure 4.2, but with rules that can be completely eliminated boxed and colored blue. We can see several properties:

- B items are never created.
- L and R items are always immediately combined with other items to create an I.
- The external point can be a parent or child when an X is created, but additional edges always have it as the parent, and are only added in N items, not L or R items.

Pruning the rules in this way further constrains the space of structures that can be formed to some subset of one-endpoint cross graphs. This subset more closely describes the structures observed in language. Unfortunately, it is difficult to characterize this space. Figure 4.25 shows the remaining rules, with further constraints on parents based on the observed rules (these were hard to show in Figure 4.24). One description of the space is that it is the set of structures generated by the deduction rules in Figure 4.25, but that is not particularly satisfying. In particular, it may be the case that this set of rules is impacted by variations such as the choice of head rules.

While this pruning opens new questions about the space of structures in parsing, for our purposes it is primarily a way of improving speed. The drastic reduction in rules leads to a substantial improvement in the time required for parsing.

4.10 Results

4.10.1 Algorithm Coverage

Table 4.1 divides sentences in the training set of the treebank by structure type and whether a directed cycle is present or not. Structures that are recoverable using our algorithm are bolded.

[8] Note, this pruning occurs after the parses have been modified to be one-EC structures, i.e., trace edges that created cycles or made non-one-EC crossings have been removed.

Algorithm 4.24 Full dynamic program with rules unseen in training boxed and colored.

```
$\boxed{\begin{array}{l}\utilde{L}[ijx\ .\overline{X}\ .X\ \overline{IJ}] \leftarrow \max\\ \quad A[x\ j]\quad L[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]\end{array}}$
$\boxed{\begin{array}{l}\utilde{L}[ijx\ .\overline{X}\ .\overline{X}\ \overline{I}J] \leftarrow \max\\ \quad A[j\ x]\quad L[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]\end{array}}$
$L[ijx\ J\overline{X}\ \overline{IX}\ \overline{IJ}] \leftarrow \max$
    $A[j\ i]\quad L[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}]$
$L[ijx\ \overline{JX}\ I\overline{X}\ \overline{IJ}] \leftarrow \max$
    $A[i\ j]\quad L[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}]$

$\boxed{\begin{array}{l}\utilde{R}[ijx\ .X\ .\overline{X}\ \overline{IJ}] \leftarrow \max\\ \quad A[x\ i]\quad R[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]\end{array}}$
$\boxed{\begin{array}{l}\utilde{R}[ijx\ .\overline{X}\ .\overline{X}\ I\overline{J}] \leftarrow \max\\ \quad A[i\ x]\quad R[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]\end{array}}$
$R[ijx\ J\overline{X}\ \overline{IX}\ \overline{IJ}] \leftarrow \max$
    $A[j\ i]\quad R[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}]$
$R[ijx\ \overline{JX}\ I\overline{X}\ \overline{IJ}] \leftarrow \max$
    $A[i\ j]\quad R[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}]$

$I[ij\ F\ I] \leftarrow \max$
    $A[i\ j]\quad I[ij\ F\ \overline{I}]$
$I[ij\ J\ F] \leftarrow \max$
    $A[j\ i]\quad I[ij\ \overline{J}\ F]$

$\utilde{N}[ijx\ .X\ .\overline{X}\ \overline{IJ}] \leftarrow \max$
    $A[x\ i]\quad N[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]$
$\boxed{\begin{array}{l}\utilde{N}[ijx\ .\overline{X}\ .\overline{X}\ I\overline{J}] \leftarrow \max\\ \quad A[i\ x]\quad N[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]\end{array}}$
$\utilde{N}[ijx\ .\overline{X}\ .X\ \overline{IJ}] \leftarrow \max$
    $A[x\ j]\quad N[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]$
$\boxed{\begin{array}{l}\utilde{N}[ijx\ .\overline{X}\ .\overline{X}\ \overline{I}J] \leftarrow \max\\ \quad A[j\ x]\quad N[ijx\ .\overline{X}\ .\overline{X}\ \overline{IJ}]\end{array}}$
$N[ijx\ J\overline{X}\ \overline{IX}\ \overline{IJ}] \leftarrow \max$
    $A[j\ i]\quad N[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}]$
$N[ijx\ \overline{JX}\ I\overline{X}\ \overline{IJ}] \leftarrow \max$
    $A[i\ j]\quad N[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}]$

$X[ijx\ .X\ .\overline{X}\ F] \leftarrow \max$
    $A[x\ i]\quad I[ij\ .\ .]$
$X[ijx\ .\overline{X}\ .\overline{X}\ I\overline{J}] \leftarrow \max$
    $A[i\ x]\quad I[ij\ .\ .]$
$X[ijx\ .\overline{X}\ .X\ F] \leftarrow \max$
    $A[x\ j]\quad I[ij\ .\ .]$
$X[ijx\ .\overline{X}\ .\overline{X}\ \overline{I}J] \leftarrow \max$
    $A[j\ x]\quad I[ij\ .\ .]$

$\boxed{\begin{array}{l}B\cdot[ijx\ F\ I\overline{X}\ \overline{IJ}] \leftarrow \max\\ \quad A[i\ j]\quad B\cdot[ijx\ F\ \overline{IX}\ \overline{IJ}]\end{array}}$
$\boxed{\begin{array}{l}\cdot B[ijx\ J\overline{X}\ F\ \overline{IJ}] \leftarrow \max\\ \quad A[j\ i]\quad \cdot B[ijx\ \overline{JX}\ F\ \overline{IJ}]\end{array}}$

$I[ij\ F\ \overline{I}] \leftarrow \max$
    $I[i\ i{+}1\ F\ F]\quad I[i{+}1\ j\ \overline{F}\ F]$
    $\max_{k\in(i,j)}$
        $I[ik\ F\ I]\quad I[kj\ .\ .]$
        $\boxed{B}LRN\cdot[ikj\ F\ I\overline{J}\ \overline{IK}]\quad I[kj\ .\ .]$
        $\max_{l\in(k,j)}$
            $RN\cdot[ikl\ F\ I\overline{L}\ \overline{IK}]\quad I[kl\ .\ .]\quad \cdot\boxed{LX}N[ljk\ .\overline{K}\ .\ \overline{L}.]$
            $\boxed{B}LRN\cdot[ikl\ F\ I\overline{L}\ \overline{IK}]\quad I[kl\ .\ .]\quad I[lj\ .\ .]$
        $\max_{l\in(i,k)}$
            $\boxed{I[il\ F\ .]\quad \cdot LN[lki\ .\overline{I}\ .I\ F]\quad \cdot\utilde{N}[kjl\ \overline{JL}\ \overline{K}.\ \overline{K}.]}$
            $\boxed{RN}X\cdot[ilk\ F\ .\overline{K}\ I\overline{L}]\quad I[lk\ .\ .]\quad \cdot\utilde{\boxed{L}N}[kjl\ .\overline{L}\ .\ \overline{K}.]$
$I[ij\ j\ F] \leftarrow \max_{k\in(i,j)}$
    $I[ik\ \overline{F}\ .]\quad I[kj\ J\ F]$
    $I[ik\ \overline{F}\ .]\quad \cdot\boxed{B}LRN[kji\ J\overline{I}\ F\ \overline{KJ}]$
    $\max_{l\in(i,k)}$
        $RNX\cdot[ilk\ \overline{F}\ .\overline{K}\ .\overline{L}]\quad I[lk\ .\ .]\quad \cdot LN[kjl\ J\overline{L}\ F\ \overline{KJ}]$
        $I[il\ \overline{F}\ .]\quad I[lk\ .\ .]\quad \cdot\boxed{BR}LN[kjl\ J\overline{L}\ F\ \overline{KJ}]$
    $\max_{l\in(k,j)}$
        $\boxed{\utilde{N}\cdot[ikl\ \overline{KF}\ \overline{IL}\ .\overline{K}]\quad RN\cdot[klj\ .J\ .\overline{J}\ F]\quad I[lj\ .\ F]}$
        $\boxed{\utilde{RN}\cdot[ikl\ \overline{F}\ .\overline{L}\ .\overline{K}]\quad I[kl\ .\ .]\quad \cdot LNX[ljk\ .\overline{K}\ F\ \overline{L}J]}$

$\boxed{\begin{array}{l}\utilde{L}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}\\ \quad LN[ikx\ .\overline{X}\ .X\ \overline{IK}]\quad \cdot N[kji\ \overline{JI}\ \overline{KI}\ \overline{KJ}]\\ \quad LN[ikx\ .\overline{X}\ .\overline{X}\ \overline{I}K]\quad \cdot N[kji\ \overline{JI}\ \overline{KI}\ \overline{KJ}]\\ \quad L[ikx\ .\overline{X}\ .X\ \overline{IK}]\quad I[kj\ .\ .]\\ \quad L[ikx\ .\overline{X}\ .\overline{X}\ \overline{I}K]\quad I[kj\ .\ .]\\ \quad N[ikx\ \overline{KX}\ IX\ \overline{IK}]\quad I[kj\ .\ .]\\ \quad N[ikx\ \overline{KX}\ I\overline{X}\ \overline{I}K]\quad I[kj\ .\ .]\\ \quad N[ikx\ K\overline{X}\ \overline{IX}\ \overline{IK}]\quad I[kj\ .\ .]\\ \quad N[ikx\ K\overline{X}\ \overline{IX}\ \overline{I}K]\quad I[kj\ .\ .]\end{array}}$

$\underset{\cdot}{L}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
    $X[ikx\ .\overline{X}\ .X\ F]\quad \cdot\boxed{L}N[kji\ .\overline{I}\ .\overline{I}\ \overline{KJ}]$
    $X[ikx\ .\overline{X}\ .\overline{X}\ \overline{I}K]\quad \cdot\boxed{L}N[kji\ .\overline{I}\ .\overline{I}\ \overline{KJ}]$

$\boxed{\begin{array}{l}B\cdot[ijx\ F\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}\\ \quad LN\cdot[ikx\ F\ I\overline{X}\ \overline{IK}]\quad R\cdot[kjx\ .\ .\overline{X}\ .\overline{J}]\\ \quad LN\cdot[ikx\ F\ I\overline{X}\ \overline{IK}]\quad N\cdot[kjx\ \overline{J}.\ K\overline{X}\ .\overline{J}]\\ \quad LN\cdot[ikx\ F\ I\overline{X}\ \overline{IK}]\quad N\cdot[kjx\ J.\ \overline{KX}\ .\overline{J}]\end{array}}$

$\boxed{\begin{array}{l}\cdot B[ijx\ \overline{JX}\ F\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}\\ \quad \cdot L[ikx\ .\overline{X}\ .\ \overline{I}.]\quad \cdot RN[kjx\ J\overline{X}\ F\ \overline{KJ}]\\ \quad \cdot N[ikx\ K\overline{X}\ \overline{I}.\ \overline{I}.]\quad \cdot RN[kjx\ J\overline{X}\ F\ \overline{KJ}]\\ \quad \cdot N[ikx\ \overline{KX}\ I.\ \overline{I}.]\quad \cdot RN[kjx\ J\overline{X}\ F\ \overline{KJ}]\end{array}}$

$\utilde{N}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
    $\cdot N[ikx\ \overline{KX}\ \overline{I}X\ \overline{IK}]\quad I[kj\ .\ .]$
    $\boxed{\cdot N[ikx\ \overline{KX}\ \overline{IX}\ \overline{I}K]\quad I[kj\ .\ .]}$
    $I[ik\ .\ .]\quad N\cdot[kjx\ \overline{J}X\ \overline{KX}\ \overline{KJ}]$
    $\boxed{I[ik\ .\ .]\quad N\cdot[kjx\ \overline{JX}\ \overline{KX}\ K\overline{J}]}$

$\underset{\cdot\cdot}{N}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
    $\cdot X[ikx\ .\overline{X}\ .X\ \overline{IK}]\quad I[kj\ .\ .]$
    $\cdot X[ikx\ .\overline{X}\ .\overline{X}\ \overline{I}K]\quad I[kj\ .\ .]$
    $I[ik\ .\ .]\quad X\cdot[kjx\ .X\ .\overline{X}\ \overline{KJ}]$
    $I[ik\ .\ .]\quad X\cdot[kjx\ .\overline{X}\ .\overline{X}\ K\overline{J}]$

$\boxed{\begin{array}{l}\utilde{R}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}\\ \quad N\cdot[ikj\ \overline{KJ}\ \overline{IJ}\ \overline{IK}]\quad RN[kjx\ .X\ .\overline{X}\ \overline{KJ}]\\ \quad N\cdot[ikj\ \overline{KJ}\ \overline{IJ}\ \overline{IK}]\quad RN[kjx\ .\overline{X}\ .\overline{X}\ K\overline{J}]\\ \quad I[ik\ .\ .]\quad R[kjx\ .X\ .\overline{X}\ \overline{KJ}]\\ \quad I[ik\ .\ .]\quad R[kjx\ .\overline{X}\ .\overline{X}\ K\overline{J}]\\ \quad I[ik\ .\ .]\quad N[kjx\ JX\ \overline{KX}\ \overline{KJ}]\\ \quad I[ik\ .\ .]\quad N[kjx\ J\overline{X}\ \overline{KX}\ K\overline{J}]\\ \quad I[ik\ .\ .]\quad N[kjx\ \overline{J}X\ K\overline{X}\ \overline{KJ}]\\ \quad I[ik\ .\ .]\quad N[kjx\ \overline{JX}\ K\overline{X}\ K\overline{J}]\end{array}}$

$\underset{\cdot}{R}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
    $\boxed{R}N\cdot[ikj\ .\overline{J}\ .\overline{J}\ \overline{IK}]\quad X[kjx\ .X\ .\overline{X}\ F]$
    $\boxed{R}N\cdot[ikj\ .\overline{J}\ .\overline{J}\ \overline{IK}]\quad X[kjx\ .\overline{X}\ .\overline{X}\ K\overline{J}]$
```

Algorithm 4.25 Pruned dynamic program, including only rules observed in the training set, including tighter parent constraints.

```
$L[ijx\ JF\ F\ \overline{IJ}] \leftarrow \max$
    $A[j\ i]\quad L[ijx\ F\ \overline{I}F\ \overline{IJ}]$
$L[ijx\ F\ IF\ \overline{IJ}] \leftarrow \max$
    $A[i\ j]\quad L[ijx\ F\ \overline{I}F\ \overline{IJ}]$
$R[ijx\ JF\ F\ \overline{IJ}] \leftarrow \max$
    $A[j\ i]\quad R[ijx\ \overline{J}F\ F\ \overline{IJ}]$
$R[ijx\ F\ IF\ \overline{IJ}] \leftarrow \max$
    $A[i\ j]\quad R[ijx\ F\ \overline{I}F\ \overline{IJ}]$

$\utilde{N}[ijx\ .X\ .\overline{X}\ \overline{IJ}] \leftarrow \max$
    $A[x\ i]\quad N[ijx\ F\overline{X}\ F\overline{X}\ F]$
$\utilde{N}[ijx\ .\overline{X}\ .X\ \overline{IJ}] \leftarrow \max$
    $A[x\ j]\quad N[ijx\ F\overline{X}\ F\overline{X}\ F]$
$N[ijx\ J\overline{X}\ \overline{IX}\ \overline{IJ}] \leftarrow \max$
    $A[j\ i]\quad N[ijx\ \overline{J}F\ F\ \overline{IJ}]$
$N[ijx\ \overline{JX}\ I\overline{X}\ \overline{IJ}] \leftarrow \max$
    $A[i\ j]\quad N[ijx\ F\ \overline{IX}\ \overline{IJ}]$

$X[ijx\ .X\ .\overline{X}\ F] \leftarrow \max$
    $A[x\ i]\quad I[ij\ .\ .]$
$X[ijx\ .\overline{X}\ .\overline{X}\ I\overline{J}] \leftarrow \max$
    $A[i\ x]\quad I[ij\ .\ .]$
$X[ijx\ .\overline{X}\ .X\ F] \leftarrow \max$
    $A[x\ j]\quad I[ij\ .\ .]$
$X[ijx\ .\overline{X}\ .\overline{X}\ \overline{I}J] \leftarrow \max$
    $A[j\ x]\quad I[ij\ .\ .]$

$I[ij\ F\ I] \leftarrow \max$
    $A[i\ j]\quad I[ij\ F\ \overline{I}]$
$I[ij\ J\ F] \leftarrow \max$
    $A[j\ i]\quad I[ij\ \overline{J}\ F]$

$I[ij\ F\ \overline{I}] \leftarrow \max$
    $I[i\ i{+}1\ F\ F]\quad I[i{+}1\ j\ \overline{F}\ F]$
    $\max_{k\in(i,j)}$
        $I[ik\ F\ I]\quad I[kj\ .\ .]$
        $LRN{\cdot}\,[ikj\ F\ I\overline{J}\ \overline{I}F]\quad I[kj\ .\ .]$
        $\max_{l\in(k,j)}$
            $RN{\cdot}\,[ikl\ F\ IF\ \overline{IK}]\quad I[kl\ F\ .]\quad \cdot N[ljk\ F\overline{K}\ F\ F]$
            $LRN{\cdot}\,[ikl\ F\ IF\ \overline{IK}]\quad I[kl\ F\ .]\quad I[lj\ .\ F]$
        $\max_{l\in(i,k)}$
            $X{\cdot}\,[ilk\ F\ .F\ IF]\quad I[lk\ F\ .]\quad \cdot\utilde{N}[kjl\ F\ Fl\ F]$
$I[ij\ j\ F] \leftarrow \max_{k\in(i,j)}$
    $I[ik\ \overline{F}\ F]\quad I[kj\ J\ F]$
    $I[ik\ \overline{F}\ F]\quad \cdot LRN[kji\ JF\ F\ \overline{KJ}]$
    $\max_{l\in(i,k)}$
        $RNX{\cdot}\,[ilk\ F\overline{F}\ .\overline{K}\ F]\quad I[lk\ .\ .]\quad \cdot LN[kjl\ JF\ F\ \overline{KJ}]$
        $I[il\ \overline{F}\ F]\quad I[lk\ .\ F]\quad \cdot LN[kjl\ JF\ F\ \overline{KJ}]$

$\underset{\cdot}{L}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
    $X[ikx\ F\ .X\ F]\quad \cdot N[kji\ .\overline{I}\ F\ F\overline{J}]$
    $X[ikx\ F\ .\overline{X}\ \overline{I}K]\quad \cdot N[kji\ .\overline{I}\ F\ F\overline{J}]$
$\underset{\cdot}{R}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
    $N{\cdot}\,[ikj\ F\ .\overline{J}\ \overline{I}F]\quad X[kjx\ .X\ F\ F]$
    $N{\cdot}\,[ikj\ F\ .\overline{J}\ \overline{I}F]\quad X[kjx\ .\overline{X}\ F\ K\overline{J}]$
$\utilde{N}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
    $\cdot N[ikx\ F\overline{X}\ \overline{I}X\ F]\quad I[kj\ F\ .]$
    $I[ik\ .\ F]\quad N{\cdot}\,[kjx\ \overline{J}X\ F\overline{X}\ F]$
$\underset{\cdot\cdot}{N}[ijx\ \overline{JX}\ \overline{IX}\ \overline{IJ}] \leftarrow \max_{k\in(i,j)}$
    $\cdot X[ikx\ .\overline{X}\ .X\ \overline{IK}]\quad I[kj\ .\ .]$
    $\cdot X[ikx\ .\overline{X}\ .\overline{X}\ \overline{I}K]\quad I[kj\ .\ .]$
    $I[ik\ .\ .]\quad X{\cdot}\,[kjx\ .X\ .\overline{X}\ \overline{KJ}]$
    $I[ik\ .\ .]\quad X{\cdot}\,[kjx\ .\overline{X}\ .\overline{X}\ K\overline{J}]$
```

Structure Type	Acyclic	Has a cycle
Projective Tree	**46.74%**	-
Projective Graph	**26.37%**	0.87%
One-Endpoint Crossing	**24.06%**	0.37%
Other Graph	1.57%	0.04%

Table 4.1: Number of sentences in the training set that are of each structure type. Structures that are recoverable using our algorithm are in bold. The acyclic / contains a cycle distinction is disjoint, but the four types of structures are not. The values shown are the percentage of parses that fall into that class and not the classes above, i.e., the total for Projective Graphs is 73.11% but 26.37% is the value shown as that is the number of Projective Graphs that are not also Projective Trees.

Approach	Coverage (%) Sentences	Edges
Projective trees without null elements	26.59	96.27
Core representation (4.7.1)	76.69	98.47
+ Head rule changes (4.7.3)	95.76	99.53
+ Null reversal (4.7.2)	97.17	99.59
+ Gapping shift (4.7.2)	97.66	99.60

Table 4.2: Coverage improvements for parts of our representation. Core uses the representation proposed in this work, with the head rules from Carreras et al. (2008). Edge results are when removing only the edges necessary to make a parse representable (e.g., removing one edge to break a cycle).

The structures we consider cover almost all sentences, while projective trees, the standard output of parsers, account for less than half of sentences.

In Table 4.2 we show the impact of design decisions for our representation. The percentages indicate how many sentences in the training set are completely recoverable by our algorithm. Each row shows the outcome of an addition to the previous row, starting from no traces at all, going to our representation with the head rules of Carreras et al. (2008), then changing the head rules, reversing null-null edges, and changing the target of edges in gapping. The largest gain comes from changing the head rules, which is unsurprising since Carreras et al. (ibid.)'s rules were designed for trees, where any set of rules produce valid structures.

4.10.2 Problematic Structures

To understand what structures are still not covered by our approach we manually inspected twenty examples that contained a cycle and twenty examples where the structure did not satisfy the one-endpoint-crossing property. The reason these are a problem is that our algorithm can only generate graphs that do not contain cycles and that satisfy the one-EC property. Adapting to cycles may be possible by adjusting the algorithm, further separating the enforcement of parent relations and the enforcement of connectivity and crossing properties, but this is beyond the scope of this work. Meanwhile, the one-EC property is a fundamental assumption that we use to construct the algorithm, in the same way that the tree property is an assumption enabling Eisner (1996)'s algorithm.

For the cycles, eleven of the cases related to sentences containing variations of NP *said* interposed between two parts of a single quote. A cycle was present because the top node of the parse was co-indexed with a null argument of *said* while *said* was an argument of the head word of the quote, together these edges create a cycle. The remaining cases were all instances of pseudo-attachment, which the treebank uses to show that non-adjacent constituents are related (Bies et al. 1995). These cases were split between use of Expletive (5) and Interpret Constituent Here (4) traces[9].

[9] For Expletive: "When a clausal subject is postposed, expletive it appears in the structural subject position. Char-

For the cases where the parse structure does not satisfy the one-endpoint-crossing property, it was more difficult to determine trends. The same three cases, Expletive, Interpret Constituent Here, and NP *said* accounted for half of the issues. Of the rest, most involved a set of crossing arcs with no clear way to avoid the crossings by adjusting head rules.

4.10.3 Parsing Performance

We implemented a proof-of-concept system to get preliminary results using this algorithm and representation. We used only first-order features, i.e., features do not consider pairs of edges. Code for both the algorithm and conversion to and from our representation are available (see Appendix A).

First we considered the standard parsing metric for trees. After one training pass through sections 2–21 of the PTB on sentences up to length 40, we get an F-score of 88.26 on section 22. This is lower than other systems, including the Carreras et al. (2008) parser, which scores 92.0 on all sentences. However, our result does show that even with simple features and limited training our algorithm can parse at a non-trivial level of accuracy.

Speed Pruning thresholds for the arc pass and the trace pruner were tuned to retain 99% of the gold non-trace edges. With that setting, the first pass prunes all but 0.302% of possible edges, and 53.0% of chart cells. The trace pruner prunes all but 6.0% of traces. The threshold for the spine pruner is tuned to retain 99.5% of the gold spines, at which level it prunes all but 6.5% of all spines. With these settings, for sentences up to length 40, it took 4.3 seconds per sentence.

Accuracy For full graph parsing we considered sentences up to length 40 (92.3% of the treebank). On section 22, for unlabeled trace edges, we obtain a precision of 88.3% and a recall of 50.4%. Using Johnson (2002)'s metric, which requires the label and span of the source and target nodes in the parse to be correct, we get precision 64% and recall 48%. This is lower than Johnson (ibid.)'s results (73 and 63 on all sentences in section 23). However, given the non-local properties considered by Johnson's patterns, it would be difficult for our model to do as well. One potential future direction is to use a forest reranker to incorporate such features, an option that is feasible for our approach and would avoid the constrained parser output Johnson's approach relies on. However, we have shown that our algorithm can recover trace edges and expect that it can improve with feature development and longer training.

acteristic of it-extraposition is that the final clause can replace it." – Section 17 of the PTB annotation guidelines (Bies et al. 1995).
For Interpret Constituent Here: "Used to indicate a relationship of constituency between elements separated by intervening material. For instance, *ICH*-attach is used in 'heavy shift' constructions when the movement results in a configuration in which it is impossible to attach the constituent to the phrase it belongs with." – Section 5.4 of the PTB annotation guidelines (ibid.).

Chapter 5

Conclusion

The overall goal of this research has been to push the boundaries of the conventional parsing research methodology that has dominated the field for the two decades. The standard set up, using the Penn Treebank as training and evaluation data, tree structures as the basis of algorithms, and PARSEVAL as the evaluation metric, has led to substantial progress. However, this dissertation has shown how new algorithms for manipulating syntactic structure can go further. Most significantly, in Chapter 4 we explored parsing with aspects of syntax that are typically ignored for computational reasons. Our algorithm is the first to incorporate virtually all aspects of structure in the PTB into a single inference method that is efficient.

An alternative approach to handling these aspects of syntax, explored in prior work, has been to use representations based on other syntactic theories. Unfortunately, it is difficult to compare performance of systems producing parses with different representations, and to provide consistent output across systems for downstream applications. Our novel method of converting from CCG into PTB-style parses, described in Chapter 3 explores one way to bridge this gap, and improved significantly over prior work.

In Chapter 2, we described our work on going beyond measurements of performance to more nuanced analysis of mistakes. Our error analysis method gives a summary of the types of errors in system output, with categories that are easily interpretable. We applied the technique to a range of parsers, writing styles, and languages, confirming previously anecdotal claims such as the prevalence of prepositional phrase attachment errors.

There are a range of possible extensions for this work. Both the error analysis and conversion systems are designed around trees, and could be extended to cover graph structures. Doing so would require breaking assumptions built-in to the algorithms, but would be particularly illuminating regarding the challenge of long-distance dependencies. There are a several interesting possibilities for the graph parsing algorithm. In the structure of the chapter presenting the algorithm, the parsing algorithm is kept separate from the definition of the syntactic representation we use. That choice was intentional, as the algorithm is a general parsing approach that could also be applied to exciting new resources like the Abstract Meaning Representation. Having non-local structure available also opens new opportunities for downstream applications. Finally, in the implementation of the algorithm we found that the vast majority of the deduction rules are not needed to cover the structures seen in language. This suggests that there may be a property other than one-endpoint crossing, which could more tightly constrain the space of possible structures while capturing all observed structures. Characterizing such a space remains a fascinating open question, and our work on one-endpoint crossing graph parsing provides a great starting point.

Appendix A

Resources

All of the systems described in this work have been publicly released under open source licenses:

Error Analysis (Chapter 2)
`https://github.com/jkkummerfeld/berkeley-parser-analyser`
Two versions of the tool were released. The initial version was the code submitted with the original paper, which only covered English. The second version supported Chinese and also cleaned up the codebase and simplified the search process by using more general operations.

The tool also includes a special output format designed to efficiently show parse errors in a plain-text terminal. Figure A.1 shows an example containing a part-of-speech error and a clause attachment error (`while talking tough` should attach higher, modifying `trying`).

Formalism Conversion (Chapter 3)
`https://github.com/jkkummerfeld/berkeley-ccg2pst`

Graph Parsing (Chapter 4)
`https://github.com/jkkummerfeld/graph-parser`
This repository is being actively used for development and has the entire history of the codebase.

```
12 Bracket errors
1 Clause Attachment error
(ROOT
   (S
      (ADVP
         (IN RB So)
         (RB far))
      (NP Mr. Hahn)
      (VP
         (VBZ is)
         (VP
            (VBG trying)
            (S (VP (S
               (VP
                  (TO to)
                  (VP (VP
                     (VB entice)
                     (NP Nekoosa)
                     (PP (PP
                        (IN into)
                        (S
                           (VP
                              (VP
                                 (S
                                    (VBG negotiating)
                                    (NP a friendly surrender) PP) VP) S) VP)))
                              (SBAR while talking tough))))))))))))
```

Figure A.1: Visualization of errors in a text based output format using color. Red indicates extra nodes in the tree, blue indicates missing nodes, and cyan is used for missing nodes that cross red nodes.

Bibliography

Ajdukiewicz, Kazimierz (1935). “Die syntaktische Konnexität”. In: *Studia Philosophica* 1, pp. 1–27 (cit. on p. 3).

Auli, Michael and Adam Lopez (2011). “A Comparison of Loopy Belief Propagation and Dual Decomposition for Integrated CCG Supertagging and Parsing”. In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT)*. url: http://aclweb.org/anthology/P11-1048 (cit. on pp. 24, 48).

Banarescu, Laura, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider (2015). *Abstract Meaning Representation (AMR) 1.2.2 Specification*. Tech. rep. url: https://github.com/amrisi/amr-guidelines/blob/master/amr.md (cit. on p. 6).

Bender, Emily M., Dan Flickinger, Stephan Oepen, and Yi Zhang (2011). “Parser evaluation over local and non-local deep dependencies in a large corpus”. In: *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. url: http://aclweb.org/anthology/D11-1037 (cit. on p. 26).

Bies, Ann, Mark Ferguson, Karen Katz, Robert MacIntyre, Victoria Tredinnick, Grace Kim, Mary Ann Marcinkiewicz, and Britta Schasberger (1995). *Bracketing Guidelines for Treebank 2 Style Penn Treebank Project*. Tech. rep. url: http://cs.jhu.edu/~jason/465/hw-parse/treebank-manual.pdf (cit. on pp. 2, 6, 51, 93, 94).

Bikel, Daniel M. (2004). “Intricacies of Collins’ Parsing Model”. In: *Computational Linguistics* 30.4, pp. 479–511. url: http://aclweb.org/anthology/J04-4004 (cit. on p. 20).

Bikel, Daniel M. and David Chiang (2000). “Two Statistical Parsing Models Applied to the Chinese Treebank”. In: *Proceedings of the Second Chinese Language Processing Workshop*. url: http://aclweb.org/anthology/W00-1201 (cit. on pp. 29, 35, 36).

Black, E., S. Abney, D. Flickenger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B. Santorini, and T. Strzalkowski (1991). “Procedure for quantitatively comparing the syntactic coverage of English grammars”. In: *Proceedings of the workshop on Speech and Natural Language*. url: http://aclweb.org/anthology/H91-1060 (cit. on pp. 4, 45).

Bodenstab, Nathan, Aaron Dunlop, Keith Hall, and Brian Roark (2011). “Beam-Width Prediction for Efficient Context-Free Parsing”. In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT)*. url: http://aclweb.org/anthology/P11-1045 (cit. on p. 21).

Boullier, Pierre (1998). *Proposal for a Natural Language Processing Syntactic Backbone*. Tech. rep. INRIA Rocquencourt. url: http://hal.archives-ouvertes.fr/docs/00/07/33/47/PDF/RR-3342.pdf (cit. on p. 5).

Briscoe, Ted, John Carroll, Jonathan Graham, and Ann Copestake (2002). “Relational Evaluation Schemes”. In: *Proceedings of the Beyond PARSEVAL Workshop at the 3rd International Conference on Language Resources and Evaluation*, pp. 4–8. url: http://users.sussex.ac.uk/~johnca/papers/beyond02.pdf (cit. on p. 39).

Cahill, Aoife, Michael Burke, Ruth O’Donovan, Stefan Riezler, Josef van Genabith, and Andy Way (2008). “Wide-Coverage Deep Statistical Parsing Using Automatic Dependency Structure Annotation”. In: *Computational Linguistics* 34.1, pp. 81–124. url: http://aclweb.org/anthology/J08-1003 (cit. on p. 4).

Cai, Shu, David Chiang, and Yoav Goldberg (2011). “Language-Independent Parsing with Empty Elements”. In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT)*. url: http://aclweb.org/anthology/P11-2037 (cit. on pp. 6, 52).

Campbell, Richard (2004). “Using Linguistic Principles to Recover Empty Categories”. In: *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P04-1082 (cit. on p. 6).

Carreras, Xavier, Michael Collins, and Terry Koo (2008). “TAG, Dynamic Programming, and the Perceptron for Efficient, Feature-rich Parsing”. In: *Proceedings of the Twelfth Conference on Computational Natural Language Learning (CoNLL)*. url: http://aclweb.org/anthology/W08-2102 (cit. on pp. 6, 80, 81, 83, 84, 93, 94).

Carroll, John, Ted Briscoe, and Antonio Sanfilippo (1998). “Parser Evaluation: a Survey and a New Proposal”. In: *Proceedings of the 1st International Conference on Language Resources and Evaluation (LREC)*. url: http://users.sussex.ac.uk/~johnca/papers/lre98.pdf (cit. on p. 4).

Charniak, Eugene (2000). “A Maximum-Entropy-Inspired Parser”. In: *Proceedings of the 1st Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL)*. url: http://aclweb.org/anthology/A00-2018 (cit. on p. 21).

Charniak, Eugene and Mark Johnson (2005). “Coarse-to-Fine n-Best Parsing and MaxEnt Discriminative Reranking”. In: *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P05-1022 (cit. on pp. 4, 21, 24, 26, 48).

Chiang, David (2000). “Statistical Parsing with an Automatically-Extracted Tree Adjoining Grammar”. In: *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P00-1058 (cit. on p. 39).

— (2007). “Hierarchical Phrase-Based Translation”. In: *Computational Linguistics* 33.2, pp. 201–228. url: http://aclweb.org/anthology/J07-2003 (cit. on p. 89).

Chomsky, Noam (1956). “Three models for the description of language”. In: *IRE Transactions on Information Theory* 2.3, pp. 113–124. url: https://chomsky.info/wp-content/uploads/195609-.pdf (cit. on p. 5).

— (1981). *Lectures on government and binding: The Pisa lectures*. Walter de Gruyter (cit. on p. 3).

— (2000). *New horizons in the study of language and mind*. Cambridge University Press (cit. on p. 50).

Clark, Stephen and James R. Curran (2007). “Wide-Coverage Efficient Statistical Parsing with CCG and Log-Linear Models”. In: *Computational Linguistics* 33.4, pp. 493–552. url: http://aclweb.org/anthology/J07-4004 (cit. on p. 48).

— (2009). “Comparing the Accuracy of CCG and Penn Treebank Parsers”. 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 (ACL)*. url: http://aclweb.org/anthology/P09-2014 (cit. on pp. 5, 39, 41).

Clegg, Andrew B. and Adrian J. Shepherd (2005). “Evaluating and integrating treebank parsers on a biomedical corpus”. In: *Proceedings of the ACL Workshop on Software*. url: http://aclweb.org/anthology/W05-1102 (cit. on p. 26).

Cocke, John (1969). *Programming Languages and Their Compilers: Preliminary Notes*. Tech. rep. Courant Institute of Mathematical Sciences, New York University. url: http://www.softwarepreservation.org/projects/FORTRAN/CockeSchwartz_ProgLangCompilers.pdf (cit. on pp. 4, 5, 52).

Collins, Michael (1997). “Three Generative, Lexicalised Models for Statistical Parsing”. In: *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P97-1003 (cit. on pp. 6, 20, 21, 45, 52).

— (2000). “Discriminative Reranking for Natural Language Parsing”. In: *Proceedings of the Seventeenth International Conference on Machine Learning (ICML)*. url: http://www.cs.columbia.edu/~mcollins/papers/rerank.ps (cit. on p. 24).

— (2003). “Head-Driven Statistical Models for Natural Language Parsing”. In: *Computational Linguistics* 29.4, pp. 589–637. url: http://aclweb.org/anthology/J03-4003 (cit. on p. 4).

Dienes, Pétr and Amit Dubey (2003). “Deep Syntactic Processing by Combining Shallow Methods”. In: *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P03-1055 (cit. on pp. 6, 52).

Duchi, John, Elad Hazan, and Yoram Singer (2011). “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *Journal of Machine Learning Research (JMLR)* 12, pp. 2121–2159. url: http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf (cit. on p. 86).

Dunlop, Aaron, Nathan Bodenstab, and Brian Roark (2011). "Efficient Matrix-Encoded Grammars and Low Latency Parallelization Strategies for CYK". In: *Proceedings of the 12th International Conference on Parsing Technologies (IWPT)*. url: http://aclweb.org/anthology/W11-2920 (cit. on p. 21).

Earley, Jay (1970). "An Efficient Context-free Parsing Algorithm". In: *Communications of the ACM* 13.2, pp. 94–102. url: http://doi.acm.org/10.1145/362007.362035 (cit. on p. 5).

Eisner, Jason (1996). "Three New Probabilistic Models for Dependency Parsing: An Exploration". In: *Proceedings of the 16th International Conference on Computational Linguistics (CoLing)*. url: http://www.aclweb.org/anthology/C96-1058 (cit. on pp. 54, 65, 93).

Eisner, Jason and Noah A. Smith (2005). "Parsing with Soft and Hard Constraints on Dependency Length". In: *Proceedings of the Ninth International Workshop on Parsing Technology (IWPT)*. url: http://aclweb.org/anthology/W05-1504 (cit. on p. 66).

Fernández-González, Daniel and André F. T. Martins (2015). "Parsing as Reduction". In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (ACL-IJCNLP)*. url: http://www.aclweb.org/anthology/P15-1147 (cit. on p. 52).

Forst, Martin and Ji Fang (2009). "TBL-Improved Non-Deterministic Segmentation and POS Tagging for a Chinese Parser". In: *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL)*. url: http://aclweb.org/anthology/E09-1031 (cit. on pp. 29, 37).

Fowler, Timothy A. D. and Gerald Penn (2010). "Accurate Context-Free Parsing with Combinatory Categorial Grammar". In: *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P10-1035 (cit. on pp. 47, 48).

Gabbard, Ryan, Mitchell Marcus, and Seth Kulick (2006). "Fully Parsing the Penn Treebank". In: *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference (NAACL-HLT)*. url: http://aclweb.org/anthology/N06-1024 (cit. on p. 6).

Gazdar, Gerald, Ewan Klein, Geoffrey K. Pullum, and Ivan A. Sag (1985). *Generalized Phrase Structure Grammar*. Harvard University Press (cit. on p. 51).

Gildea, Daniel (2001). "Corpus Variation and Parser Performance". In: *Proceedings of the 2001 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. url: http://aclweb.org/anthology/W01-0521 (cit. on p. 26).

Gómez-Rodríguez, Carlos and Joakim Nivre (2010). "A Transition-based Parser for 2-planar Dependency Structures". In: *Proceedings of the 48th Annual Meeting of the Association for Com-*

putational Linguistics (ACL). url: http://aclweb.org/anthology/P10-1151 (cit. on p. 51).

Goodman, Joshua (1997). “Global Thresholding and Multiple-Pass Parsing”. In: *Second Conference on Empirical Methods in Natural Language Processing (EMNLP)*. url: http://aclweb.org/anthology/W97-0302 (cit. on p. 89).

Guo, Yuqing, Haifeng Wang, and Josef van Genabith (2007). “Recovering Non-Local Dependencies for Chinese”. In: *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. url: http://aclweb.org/anthology/D07-1027 (cit. on p. 32).

Hall, Johan and Joakim Nivre (2008). “A Dependency-Driven Parser for German Dependency and Constituency Representations”. In: *Proceedings of the Workshop on Parsing German*. url: http://www.aclweb.org/anthology/W/W08/W08-1007 (cit. on p. 52).

Hall, Johan, Joakim Nivre, and Jens Nilsson (2007). “A Hybrid Constituency-Dependency Parser for Swedish”. In: *Proceedings of the 16th Nordic Conference of Computational Linguistics (NODALIDA)*, pp. 284–287. url: http://dspace.utlib.ee/dspace/bitstream/10062/2590/1/post-Hall-6.pdf (cit. on p. 52).

Henderson, James (2003). “Inducing History Representations for Broad Coverage Statistical Parsing”. In: *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. url: http://aclweb.org/anthology/N03-1014 (cit. on p. 21).

— (2004). “Discriminative Training of a Neural Network Statistical Parser”. In: *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P04-1013 (cit. on p. 21).

Hockenmaier, Julia (2003). “Data and Models for Statistical Parsing with Combinatory Categorial Grammar”. PhD thesis. School of Informatics, The University of Edinburgh. url: https://www.era.lib.ed.ac.uk/handle/1842/320 (cit. on p. 4).

Huang, Liang (2008). “Forest Reranking: Discriminative Parsing with Non-Local Features”. In: *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT)*. url: http://aclweb.org/anthology/P08-1067 (cit. on p. 24).

Jiang, Wenbin, Liang Huang, and Qun Liu (2009). “Automatic Adaptation of Annotation Standards: Chinese Word Segmentation and POS Tagging – A Case Study”. 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 (ACL-IJCNLP)*. url: http://aclweb.org/anthology/P09-1059 (cit. on pp. 29, 37).

Jijkoun, Valentin (2003). “Finding Non-local Dependencies: Beyond Pattern Matching”. In: *The Companion Volume to the Proceedings of 41st Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P03-2006 (cit. on p. 6).

Johnson, Mark (2002). “A Simple Pattern-matching Algorithm for Recovering Empty Nodes and Their Antecedents”. In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. url: http://aclweb.org/anthology/P02-1018 (cit. on pp. 6, 52, 94).

— (2007). “Transforming Projective Bilexical Dependency Grammars into efficiently-parsable CFGs with Unfold-Fold”. In: *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*. url: http://aclweb.org/anthology/P07-1022 (cit. on p. 66).

Joshi, Aravind K. and Yves Schabes (1997). “Handbook of Formal Languages: Volume 3 Beyond Words”. In: ed. by Grzegorz Rozenberg and Arto Salomaa. Springer Berlin Heidelberg. Chap. Tree-Adjoining Grammars, pp. 69–123 (cit. on p. 52).

Kaplan, R. M. and J. Bresnan (1982). “Lexical-Functional Grammar: A Formal System for Grammatical Representation”. In: *The Mental Representation of Grammatical Relations*. Ed. by J. Bresnan. MIT Press, pp. 173–281. url: http://www2.parc.com/isl/groups/nltt/papers/kb82-95.pdf (cit. on p. 52).

Kasami, Tadao (1966). *An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages*. Tech. rep. University of Illinois at Urbana-Champaign. url: http://hdl.handle.net/2142/74304 (cit. on pp. 4, 5, 52).

King, Tracy H., 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 at EACL*. url: http://aclweb.org/anthology/W03-2401 (cit. on p. 4).

Klein, Dan and Christopher D. Manning (2003a). “Accurate Unlexicalized Parsing”. In: *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P03-1054 (cit. on pp. 21, 35, 36, 48).

— (2003b). “Fast Exact Inference with a Factored Model for Natural Language Parsing”. In: *Advances in Neural Information Processing Systems 15 (NIPS)*. url: http://papers.nips.cc/paper/2325-fast-exact-inference-with-a-factored-model-for-natural-language-parsing.pdf (cit. on pp. 21, 29, 35, 36).

Kong, Lingpeng, Alexander M. Rush, and Noah A. Smith (2015). “Transforming Dependencies into Phrase Structures”. In: *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT)*. url: http://www.aclweb.org/anthology/N15-1080 (cit. on p. 52).

Kummerfeld, Jonathan K., Taylor Berg-Kirkpatrick, and Dan Klein (2015). “An Empirical Analysis of Optimization for Max-Margin NLP”. In: *Proceedings of the 2015 Conference on Empirical*

Methods in Natural Language Processing (EMNLP). url: http://aclweb.org/anthology/D15-1032 (cit. on p. 86).

Kummerfeld, Jonathan K., David Hall, James R. Curran, and Dan Klein (2012). "Parser Showdown at the Wall Street Corral: An Empirical Investigation of Error Types in Parser Output". In: *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. url: http://aclweb.org/anthology/D12-1096 (cit. on p. 9).

Kummerfeld, Jonathan K., Dan Klein, and James R. Curran (2012). "Robust Conversion of CCG Derivations to Phrase Structure Trees". In: *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P12-2021 (cit. on p. 39).

Kummerfeld, Jonathan K., Daniel Tse, James R. Curran, and Dan Klein (2013). "An Empirical Examination of Challenges in Chinese Parsing". In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P13-2018 (cit. on p. 9).

Kuroda, S.-Y. (1964). "Classes of languages and linear-bounded automata". In: *Information and Control* 7.2, pp. 207–223. url: http://www.sciencedirect.com/science/article/pii/S0019995864901202 (cit. on p. 5).

Lang, Bernard (1974). "Deterministic techniques for efficient non-deterministic parsers". In: *Automata, Languages and Programming: 2nd Colloquium, University of Saarbrücken July 29–August 2, 1974*. Ed. by J. Loeckx. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 255–269. url: http://dx.doi.org/10.1007/3-540-06841-4_65 (cit. on p. 5).

Levy, Roger and Christopher D. Manning (2003). "Is it harder to parse Chinese, or the Chinese Treebank?" In: *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P03-1056 (cit. on pp. 29, 35–37).

— (2004). "Deep Dependencies from Context-free Statistical Parsers: Correcting the Surface Dependency Approximation". In: *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*. url: http://aclweb.org/anthology/P04-1042 (cit. on p. 6).

Lin, Dekang (1998). "A dependency-based method for evaluating broad-coverage parsers". In: *Natural Language Engineering* 4.2, pp. 97–114 (cit. on p. 39).

Marcus, Mitchell P., Beatrice Santorini, and Mary Ann Marcinkiewicz (1993). "Building a Large Annotated Corpus of English: The Penn Treebank". In: *Computational Linguistics* 19.2, pp. 313–330. url: http://aclweb.org/anthology/J93-2004 (cit. on pp. 4, 26, 49).

Marneffe, Marie-Catherine de and Christopher D. Manning (2008). "The Stanford Typed Dependencies Representation". In: *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*. url: http://aclweb.org/anthology/W08-1301 (cit. on p. 3).

Matsuzaki, Takuya 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)*. url: http://aclweb.org/anthology/C08-1069 (cit. on p. 39).

McClosky, David, Eugene Charniak, and Mark Johnson (2006a). "Effective self-training for parsing". In: *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*. url: http://aclweb.org/anthology/N06-1020 (cit. on pp. 4, 21, 24).

— (2006b). "Effective Self-Training for Parsing". In: *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*. url: http://aclweb.org/anthology/N06-1020 (cit. on p. 22).

McDonald, Ryan, Koby Crammer, and Fernando Pereira (2005). "Online Large-Margin Training of Dependency Parsers". In: *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P05-1012 (cit. on p. 85).

Miyao, Yusuke, Takashi Ninomiya, and Jun'ichi Tsujii (2004). "Corpus-oriented grammar development for acquiring a head-driven phrase structure grammar from the Penn Treebank". In: *Proceedings of the First international joint conference on Natural Language Processing (IJCNLP)* (cit. on p. 4).

Ng, Dominick and James R. Curran (2012). "Dependency Hashing for n-best CCG Parsing". In: *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P12-1052 (cit. on p. 24).

Ng, Dominick, Matthew Honnibal, and James R. Curran (2010). "Reranking a wide-coverage CCG parser". In: *Proceedings of the Australasian Language Technology Association Workshop (ALTA)*. url: http://aclweb.org/anthology/U10-1014 (cit. on p. 24).

Nivre, Joakim, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D. Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman (2016). "Universal Dependencies v1: A Multilingual Treebank Collection". In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*. url: http://www.lrec-conf.org/proceedings/lrec2016/pdf/348_Paper.pdf (cit. on pp. 2, 3).

Petrov, Slav (2010). "Products of Random Latent Variable Grammars". In: *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*. url: http://aclweb.org/anthology/N10-1003 (cit. on pp. 35, 36).

Petrov, Slav, Leon Barrett, Romain Thibaux, and Dan Klein (2006). "Learning Accurate, Compact, and Interpretable Tree Annotation". In: *Proceedings of the 21st International Conference* *on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P06-1055 (cit. on pp. 20, 35, 36).

Petrov, Slav and Dan Klein (2007). "Improved Inference for Unlexicalized Parsing". In: *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference (NAACL-HLT)*. url: http://aclweb.org/anthology/N07-1051 (cit. on pp. 4, 20, 35, 36, 47, 48).

Petrov, Slav and Ryan McDonald (2012). *SANCL Shared Task (data from the Google Web Treebank)*. LDC2012E43 / LDC2012T13. Linguistic Data Consortium. url: https://catalog.ldc.upenn.edu/LDC2012T13 (cit. on p. 26).

Pitler, Emily, Sampath Kannan, and Mitchell Marcus (2013). "Finding Optimal 1-Endpoint-Crossing Trees". In: *Transactions of the Association for Computational Linguistics* 1, pp. 13–24. url: http://aclweb.org/anthology/Q13-1002 (cit. on pp. 7, 50, 51, 53, 62, 65, 78, 84).

Qian, Xian and Yang Liu (2012). "Joint Chinese Word Segmentation, POS Tagging and Parsing". In: *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. url: http://aclweb.org/anthology/D12-1046 (cit. on pp. 29, 35, 37).

Ratliff, Nathan, J. Andrew (Drew) Bagnell, and Martin Zinkevich (2007). "(Online) Subgradient Methods for Structured Prediction". In: *Eleventh International Conference on Artificial Intelligence and Statistics (AIStats)*. url: http://www.ri.cmu.edu/pub_files/pub4/ratliff_nathan_2007_3/ratliff_nathan_2007_3.pdf (cit. on p. 86).

Recht, Benjamin, Christopher Re, Stephen Wright, and Feng Niu (2011). "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems 24 (NIPS)*. url: https://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf (cit. on p. 86).

Sarkar, Anoop (2001). "Applying Co-Training Methods to Statistical Parsing". In: *Proceedings of the Second Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL)*. url: http://aclweb.org/anthology/N01-1023 (cit. on p. 39).

Savitch, Walter J. (1970). "Relationships between nondeterministic and deterministic tape complexities". In: *Journal of Computer and System Sciences* 4.2, pp. 177–192. url: http://www.sciencedirect.com/science/article/pii/S002200007080006X (cit. on p. 5).

Schmid, Helmut (2006). "Trace Prediction and Recovery with Unlexicalized PCFGs and Slash Features". In: *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (ACL)*. url: http://aclweb.org/anthology/P06-1023 (cit. on pp. 6, 52).

Shen, Libin, Lucas Champollion, and Aravind K. Joshi (2007). "LTAG-spinal and the Treebank". In: *Language Resources and Evaluation* 42.1, pp. 1–19. url: http://link.springer.com/article/10.1007/s10579-007-9043-7 (cit. on pp. 6, 52).

Steedman, Mark (2000). *The Syntactic Process*. MIT Press (cit. on pp. 2, 3, 39, 52).

Tesnière, Lucien (1959). *Éléments de syntaxe structurale*. Klincksieck (cit. on p. 3).

Tse, Daniel and James R. Curran (2012). "The Challenges of Parsing Chinese with Combinatory Categorial Grammar". In: *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. url: http://aclweb.org/anthology/N12-1030 (cit. on p. 37).

Weir, David J. and Aravind K. Joshi (1988). "Combinatory Categorial Grammars: Generative Power and Relationship to Linear Context-Free Rewriting Systems". In: *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*. url: http://www.aclweb.org/anthology/P88-1034 (cit. on p. 5).

Xia, Fei (1999). "Extracting Tree Adjoining Grammars from Bracketed Corpora". In: *Proceedings of the Natural Language Processing Pacific Rim Symposium* (cit. on p. 4).

Xia, Fei and Martha Palmer (2001). "Converting Dependency Structures to Phrase Structures". In: *Proceedings of the First International Conference on Human Language Technology Research (HLT)*. url: http://aclweb.org/anthology/H01-1014 (cit. on p. 39).

Xia, Fei, Owen Rambow, Rajesh Bhatt, Martha Palmer, and Dipti Misra Sharma (2009). "Towards a Multi-Representational Treebank". In: *Proceedings of the 7th International Workshop on Treebanks and Linguistic Theories* (cit. on p. 39).

Xiong, Deyi, Shuanglong Li, Qun Liu, Shouxun Lin, and Yueliang Qian (2005). "Parsing the Penn Chinese Treebank with Semantic Knowledge". In: *Proceedings of the Second International Joint Conference on Natural Language Processing (IJCNLP)*. url: http://aclweb.org/anthology/I05-1007 (cit. on p. 35).

Xue, Nianwen, Fei Xia, Fu-Dong Chiou, and Martha Palmer (2005). "The Penn Chinese TreeBank: Phrase structure annotation of a large corpus". In: *Natural Language Engineering* 11.2, pp. 207–238 (cit. on pp. 29, 31).

Younger, Daniel H. (1967). "Recognition and parsing of context-free languages in time n^3". In: *Information and Control* 10.2, pp. 189–208. url: http://www.sciencedirect.com/science/article/pii/S001999586780007X (cit. on pp. 4, 5, 52).

Zhang, Xiaotian, Hai Zhao, and Cong Hui (2012). "A Machine Learning Approach to Convert CCGbank to Penn Treebank". In: *Proceedings of COLING 2012: Demonstration Papers*. url: http://www.aclweb.org/anthology/C12-3067 (cit. on pp. 41, 45).

Zhang, Yue and Stephen Clark (2009). “Transition-Based Parsing of the Chinese Treebank using a Global Discriminative Model”. In: *Proceedings of the 11th International Conference on Parsing Technologies (IWPT)*. url: `http://aclweb.org/anthology/W09-3825` (cit. on pp. 34, 36).