Efficient Latent-Variable Grammars: Learning and Inference

Aaron Joseph Dunlop
B.A. Computer Science, Willamette University, 1996
M.S. Computer Science, OHSU, 2008

Presented to the
Center for Spoken Language Understanding
within the Oregon Health & Science University
School of Medicine
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science & Engineering

May 2014

Center for Spoken Language Understanding
School of Medicine
Oregon Health & Science University

CERTIFICATE OF APPROVAL

This is to certify that the Ph.D. dissertation of

Aaron Joseph Dunlop

has been approved.

Dr. Brian Roark, Thesis Advisor
Research Scientist, Google

Dr. Richard Sproat
Research Scientist, Google

Dr. Slav Petrov
Research Scientist, Google

Dr. Izhak Shafran
Associate Professor

Dr. Steven Bedrick
Assistant Professor

Dedication

To Irene

For her constant support and grace.

Acknowledgements

I want to thank my advisor, Dr. Brian Roark, for convincing me to undertake this adventure, for the freedom to explore my research agenda, and for his insight and wisdom along the way. Thanks also to my other committee members, Drs. Richard Sproat, Slav Petrov, Izhak Shafran, and Steven Bedrick for their recommendations, and for their support of and corrections to this work.

Thanks to my colleagues at OHSU, including Nate Bodenstab, Kristy Hollingshead-Seitz, Meg Mitchell, Emily Tucker Prud'hommeaux, Mahsa Yarmohammadi, Masoud Rouhizadeh, and others too numerous to mention, for making CSLU such an enjoyable environment, and for their input into my research.

I would like to acknowledge my parents; for entertaining the kids over many weekends to give me time for research; for volunteering to replace faucets, fix doors, and tackle other household maintenance while I was deep in writing; and most of all, for their constant encouragement and faith in me.

Thanks to my children, Abram and Meika, for the joys of books, hikes, basketball games, and sailing trips together; for the many things they have taught me; and for the smiles and hugs they share so freely.

Finally, and most importantly, thanks to Irene, the greatest blessing in my life. Thanks for being my constant support, through much sweat and many tears; for giving up far too many evenings and weekends to research, and for reminding me regularly that even a Ph.D. will eventually end. I am eternally grateful for you.

Contents

List of Tables

List of Figures

Abstract

Efficient Latent-Variable Grammars: Learning and Inference

Aaron Joseph Dunlop

Doctor of Philosophy
Center for Spoken Language Understanding
within the Oregon Health & Science University
School of Medicine

May 2014
Thesis Advisor: Dr. Brian Roark

Syntactic analysis is important for many natural language processing (NLP) tasks, but constituency parsing is computationally expensive — often prohibitively so. Consumers who would be best served by constituency parsing are often forced by resource constraints to settle for less effective approaches. In this work, we examine the barriers to efficient context-free processing, and present several approaches to increasing throughput and reducing latency.

We describe a matrix grammar representation and demonstrate that its memory- and cache-efficient properties yield considerable speedups in inference. We present an efficient and parallelizable refactoring of the standard dynamic programming algorithm, based on that matrix grammar, which further improves latency.

We introduce several methods of targeting efficiency as an objective during model training, including some which are applicable outside of constituency parsing. We present several methods of incorporating efficiency concerns into the process of training latent-variable grammars, and experimental trials demonstrating the effects of each approach.

We present several methods of text normalization (prior to grammar induction) that reduce the grammar size considerably and improve parse efficiency with minimal accuracy degradation. We explore the characteristics of a grammar that impact efficient inference, and present a regression model predicting inference time from those characteristics. We incorporate the accuracy and efficiency models into latent-variable grammar training, allowing a controlled tradeoff between speed and accuracy, and optimizing the trained grammar for the desired operating point.

Finally, we explore various optimization criteria for chart decoding, and efficient approximations thereof. We replicate prior results on max-rule decoding, and explore other options which have not been well explored in prior work. We present efficient approximations of these methods, capturing some of those gains without severe computational cost. In aggregate, our methods achieve a speedup of approximately $20\times$ for Viterbi decoding, and $3\times$ for alternate decoding methods.

Chapter 1

Introduction

Human languages have long presented a unique challenge for computational systems. Deterministic processing algorithms often map poorly onto the inherent ambiguity in language usage. Nevertheless, the potential benefits of automatic language processing have motivated decades of research in Natural Language Processing (NLP) and its subfields.

Recent years have seen great advances in NLP, and successful systems are becoming widely used. Personal-assistant applications like Apple's Siri™ and Nuance's Nina™ have popularized speech recognition, Natural Language Understanding (NLU) and dialog modeling. Similarly, Machine Translation (MT) — automatic translation from one language to another — has progressed rapidly in recent years, yielding practical and widely-used implementations.

Question-answering (QA) systems process natural-language queries and attempt to produce a meaningful response (sometimes, but not always, also in the form of natural language). Early systems were limited to specific domains — e.g. BASEBALL [78], covering American Baseball, or SHRDLU [181], which handled instructions and queries about an environment of colored blocks. More recent QA systems incorporate data gleaned automatically from text, allowing coverage of a broader range of queries; IBM's Watson system [65], which successfully competed on TV's *Jeopardy* show, is perhaps the best-known example. Broad-domain QA generally involves preprocessing large amounts of free-form text to populate a knowledge-base, from which question answers can be induced.

These recent advances in NLP were fueled by the development of large training corpora and of computational resources sufficient to effectively apply statistical models. For example, the theoretical underpinnings for MT were developed in the 1960's and earlier, and

the influential IBM model system was defined in the 1990's [179]. But practical implementations required computational resources unavailable until relatively recently. Increased computational power has enabled further modeling research, and more sophisticated models in turn required additional resources. In combination, improvements in modeling and computational throughput have yielded the impressive gains of recent years.

However, much work remains, both in effective modeling approaches and in application of those approaches to real-world problems. In particular, as we deploy NLP systems with increasingly large models, efficiency considerations increase in importance. NLP tasks are often limited by throughput constraints or real-time latency requirements. This thesis presents several approaches to improve the speed of syntactic analysis.

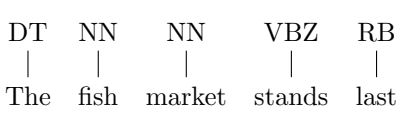
1.1 Hierarchical Syntactic Analysis

Text-processing systems often require annotating free-form sequences with labels describing latent structural elements hidden in the sequence tokens. Example annotation tasks include part-of-speech (POS) tagging, NP-chunking, and constituency and dependency parsing, as demonstrated in Figure 1.1. Subsequent information extraction or other stages make use of the labels to inform their own decisions.

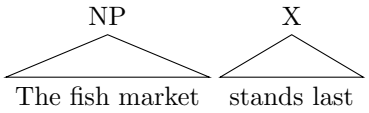
Many NLP systems depend heavily on syntactic analysis. For example, Information Extraction (IE) and Machine Reading systems attempt to extract facts from free-form text. For example, the sentence *Jim's mother Betty is still swimming at 84.* encodes (at least) the following facts:

- Child-of $\langle Jim, Betty \rangle$
- Age $\langle Betty, 84 \rangle$
- Parent-of $\langle Betty, Jim \rangle$

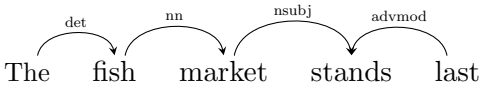
A fact representation of this form can form the knowledge ontologies that drive QA systems, or aid in summarizing the content of long documents. Recovering those facts from text (and avoiding spurious errors, such as inferring that Jim is swimming) will



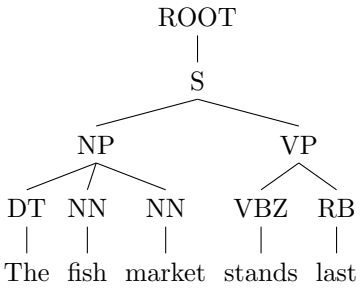
(a) Part-of-speech (POS) tagging applies syntactic labels to each token, but applies no higher-order structures.



(b) Noun-phrase (NP) chunking labels only NPs, leaving all other subsequences unlabeled.



(c) Dependency parsing attaches each token to its syntactic *head*, the word whose function it directly supports. In most dependency-parsing formalisms, the main verb serves as the overall head of the sentence.



(d) Constituency parsing. (Relatively) independent subsequence units are denoted with labels such as NP (noun phrase), VP (verb phrase), and S (clause or sentence), and a hierarchical tree structure denotes the position of each constituent.

Figure 1.1: Examples of several forms of syntactic processing.

almost certainly require a sophisticated syntactic model of the source text. Many other NLP tasks also depend heavily on syntax [16, 9]. MT is often formulated as a special form of parsing, performing hierarchical inference simultaneously in the source and target languages [37]. Other consumers of syntactic analysis include semantic role labeling [148], anaphora and coreference resolution [33], and discourse analysis [67].

Unfortunately, syntactic parsing is one of the more expensive of the common NLP tasks, often orders of magnitude more costly than other processies in a pipeline. Some of the annotations demonstrated in Figure 1.1 can be recovered in linear time (i.e., $O(n)$, where n is the length of the input sequence). POS-tagging and NP-chunking are modeled well by linear sequence models. Quadratic-time and linear greedy search are commonly

applied to dependency parsing [131, 123], but recovering hierarchical constituency structure is usually more expensive. Although greedy search can be applied to constituency structure [161, 95], the resulting accuracy is usually well below the state of the art; the asymptotic complexity of more accurate algorithms is generally at least $O(n^3)$. However, the resulting constituency structure is superior for many downstream tasks [143], and dependency structure can often be extracted most effectively from a constituency parse [30]. Thus, we focus in this thesis on constituency parsing, despite the computational cost. Given the wide range of tasks which depend on syntactic analysis, efficiency gains in constituency analysis are of broad interest to the NLP community.

1.2 Problem Statement

Prior to the introduction of sizable annotated training corpora, linguists would often hand-build grammars for a language or genre of interest; these hand-coded grammars were generally very compact, but usually failed to represent the wide range of constructions observed in real-world language usage. All current wide-coverage parsers deal with this ambiguity by learning grammars from large corpora (as described in Section 2.7). These *treebank* grammars can be quite accurate, and markedly more robust to unseen data, but at a steep efficiency cost, as they encode thousands or millions of individual probabilistic productions. Recent parsing research has greatly improved — and expanded — our statistical models, and classic approaches such as dynamic programming [45, 188, 96] and top-down filtering approaches [118, 61], which were quite effective on smaller models, become less practical with grammars encoding millions of parameters. Most recent research has focused on training increasingly accurate models, and efficiency of inference is often an afterthought; researchers generally refine a pruning model or an inference engine sufficiently to measure accuracy on a test corpus, but rarely is efficient inference a primary goal.

Most modern context-free parsers perform approximate inference, pruning their search space dramatically (and risking search errors) in order to return a result in tractable time (c.f., for example, the Charniak [32], Berkeley [142], and Stanford [100] parsers). Even

with considerable pruning, the speed of most modern parsers ranges from approximately 15–100 words per second [105], orders of magnitude slower than other commonly-used NLP methods.

A small body of prior work has addressed efficient grammar encodings [97] and binarization strategies [169]. However, to our knowledge, prior work has not explored 1) the interaction between grammar encoding and hardware-level memory representation; 2) the impact of grammar characteristics on the efficiency of inference; 3) the efficiency impact of rare- and unseen-word handling; or 4) the interplay between accuracy and efficiency during grammar learning.

1.3 Goals

The overall goal of this thesis is to reduce the barriers to constituency parsing by examining and improving the efficiency of context-free processing. To accomplish that goal, we will focus on the following specific aims.

Aim 1: Efficient and Parallelizable CYK Chart Parsing. Compactly represent a PCFG and efficiently populate a packed parse forest. Demonstrate the trade-offs between grammar intersection approaches, particularly regarding effective parallelization.

Aim 2: Efficiency Characteristics of PCFGs. Characterize how search efficiency is affected by attributes of the grammar. Train a regression model predicting the efficiency of a given grammar.

Aim 3: Efficient Latent-Variable Grammars. Incorporate predicted efficiency into latent-variable (LV) grammar learning, and allow trading off accuracy and efficiency in a controlled manner.

Aim 4: Accuracy and Efficiency of Advanced Chart Decoding Methods. Compare competing chart decoding methods and examine approximations thereof which improve accuracy over Viterbi search with reasonable impact on efficiency.

In combination, these aims contribute to the field of computer science in several ways:

In our exploration of parallel parsing, we pull together research from the NLP community with that of the parallel programming and supercomputing fields. By exploring grammar properties, we improve understanding of the characteristics which yield effective and efficient grammars, and the potential operating points when trading off accuracy and efficiency. We present several novel methods of incorporating efficiency metrics into the process of training latent-variable grammars. Finally, the infrastructure and results are incorporated into the open-source BUBS parser [20], and are thus available for further research in parsing and in downstream tasks.

1.4 Organization and Contributions of the Thesis

We begin in Chapter 2 with a brief introduction of probabilistic context-free grammars (PCFGs) and the standard approaches to PCFG training of and inference. Of particular interest is the split-merge approach to training latent-variable grammars [142]. We use this training method extensively in chapters 4 and 5, so we encourage any readers not already familiar with it to review the discussion in Section 2.9.

The central contributions of this thesis are concentrated in chapters 3–5. Chapter 3 begins with a discussion of the relationship between matrix operations and CYK parsing, and presents a matrix grammar encoding. We demonstrate a dramatic improvement in memory and cache efficiency, and a commensurate speedup in parsing. We then compare several approaches to the central argmax in the CYK algorithm and present a novel refactoring of CYK, based on the matrix grammar encoding, that reduces the number of expensive grammar-intersection operations from $O(n^3)$ to $O(n^2)$. We show that this approach provides further efficiency gains and parallelizes smoothly across multicore architectures, scaling nearly linearly with the number of available processor cores. This chapter is adapted from work previously published in Dunlop et al. [58] and Dunlop et al. [59].

In chapters 4 and 5, we present methods of learning grammars optimizing for efficient inference, allowing controlled tradeoff between speed and accuracy. Chapter 4 present text-normalization approaches that can produce considerably smaller — and faster — grammars.

In Chapter 5, we examine various attributes of a PCFG and how those characteristics affect the efficiency of inference with that grammar. We present a regression model which predicts parsing time based on the grammar characteristics. We then extend the work of Petrov et al. [142] to incorporate efficiency predictions and measurements into grammar learning. We demonstrate that the tradeoff between efficiency and accuracy is not strictly linear, permitting induction of jointly optimized grammars with markedly improved efficiency at minimal cost in accuracy.

In Chapter 6 we explore chart decoding methods, including Max-Rule [144] and Approximate Minimum-Bayes-Risk [74, 83]. These decoding methods improve accuracy over Viterbi decoding, but often at a severe computational cost. We present efficient approximations which capture some of those accuracy gains with minimal computational overhead, and demonstrate consistent gains across languages and genres.

The combination of these methods yields considerable speedups in common parsing tasks, as demonstrated over a variety of languages and domains. In Chapter 7, we present experiments demonstrating the additive effects of the approaches explored throughout the thesis, resulting in throughput in excess of 5700 words/second, improving by approximately 2.5× on previously published results, and by 20× on earlier approaches.[1] We present a set of grammar-training recommendations and guidelines, aiming to provide others a simple and effective path to incorporate these techniques into their own systems, and suggest applications in areas other than constituency parsing, including dependency analysis and machine translation.

[1] To our knowledge, the best previously-published results are those in Bodenstab [19], which incorporate the grammar encoding from Chapter 3.

Chapter 2

Background and Preliminaries

Many common NLP tasks can be performed effectively by finite-state systems, including regular languages [39] and Hidden Markov Models [10]. For example, POS-tagging and NP-chunking (as demonstrated in Figure 1.1) can be modeled accurately with relatively simple sequence models, such as Hidden Markov Models (HMMs). Further, finite-state inference is $O(n)$ (where n is the length of the sequence in tokens), and is thus tractable even for lengthy sequences. However, finite-state models are unable to capture long-distance dependencies, which are common in linguistic and biological sequences. Recovering long-distance relationships requires a more complex model, incorporating hierarchical structure. Hierarchical language modeling permits many of the constructions and variations we see in everyday language usage. For example, consider the following sentences and 2 simple modifications thereof:

John likes beans. He hates peas.	(Original)
Beans, John likes. Peas, he hates.	(Topicalized)
Beans are liked by John.	(Passivized)

As demonstrated in these simple examples, certain subsequence units can function on their own, and may (in some cases) be moved independently without rendering the sentence ungrammatical. We call these self-contained subsequences *constituents*. In addition to the movements demonstrated above, constituents can often be replaced in their entirety. For example, in the sentence *The fish market stands last*, we might replace the phrase *stands last* with an alternate verb construction:

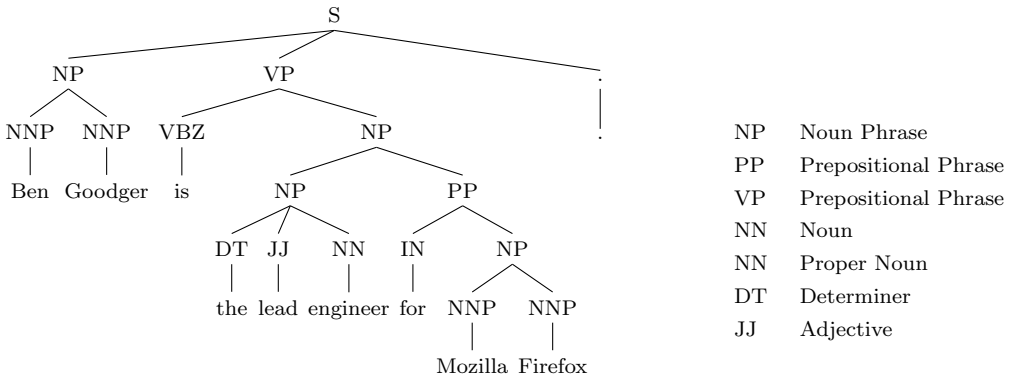


Figure 2.1: A constituency parse tree, labeling one possible hierarchical structure for a simple sentence.

- The fish market *was closed last week*
- The fish market *is on the corner of 8th and main*

Further, these constituents can be of arbitrary complexity:

- The fish market *next to the taco stand, which eventually went bankrupt and was replaced with a ...*

Constituent length is theoretically unbounded, and can be quite long even in everyday usage, so a simple sequence model cannot practically capture the variations observed in real-world language. Moving from a linear sequence model to a hierarchical structure allows us to capture many of the generalities in language usage from limited observations. Many linguistic formalisms incorporate hierarchy and constituent structures; in this chapter, we will discuss several such models, and describe in depth the formalism of the context-free language and its application to linguistic processing.

2.1 Formal Languages and the Chomsky Hierarchy

A formal language $\mathcal{L}$ is defined by a *phrase-structure grammar* [38], a form of *rewriting system* [174]. A grammar G consists of a non-terminal set V, a terminal set T, a special start symbol $S^\dagger \in V$, and a set of rewriting rules, or *productions* P. Thus, we often represent G as the tuple $\langle V, T, S^\dagger, P \rangle$.

Grammar	Language	Production Rules	Complexity of Inference
Type 0	Recursively Enumerable	$\alpha \rightarrow \beta$	Undecidable
Type 1	Context-sensitive	$\alpha A \beta \rightarrow \alpha \gamma \beta$	NP-Complete
Type 2	Context-free	$A \rightarrow \gamma$	$O(n^3)$
Type 3	Regular	$A \rightarrow a$, $A \rightarrow aB$	$O(n)$

Table 2.1: The Chomsky Language Hierarchy. $A, B \in V$ (single non-terminals), $\alpha, \beta \in V^*$ (0 or more non-terminals), $\gamma \in \{V \cup T\}^+$ (a sequence of 1 or more non-terminals and terminals), and $a \in T$ (a single terminal).

Each string of $\mathcal{L}$ is a sequence of terminals (T^*); no non-terminal may also appear in T (i.e., $\{V \cap T\} = \{\}$). In 1956, Noam Chomsky categorized formal languages into the classes in Table 2.1 [39]. Each language type is a subset of its predecessor — e.g., all regular languages are also context-free, but many context-free languages are not regular.[1]

In a linguistic grammar, common non-terminal labels include NP for Noun Phrase, JJ for adjective, and S for clause.[2] The terminal set T consists of the words permitted by the language.

A *derivation* is defined as an application of one or more rewrite rules to $A \in V$, ending with an ordered list of terminals. To 'apply' a rule production, we replace the parent non-terminal with the one or more children that constitute the rule. Repeated rule application is normally denoted with $\stackrel{*}{\Rightarrow}$. For example, the following derivation shows that $S^\dagger \stackrel{*}{\Rightarrow}$ *stands fish market* by the grammar in Table 2.2.

[1] Several other language classes have since been described, fitting between the types 0–3 of Table 2.1. Of those, the class known as *mildly context-sensitive* is the most closely related to this thesis; we will describe it briefly in Section 2.2.1.

[2] The Penn Treebank annotation guidelines define S as a 'simple declarative clause'. S is often used to label complete sentences, presumably the motivation for the choice of the label S, but the same label is used for subordinate clauses as well.

Parent		Children	Parent		Children
ROOT	→	S	VP	→	VBD
S	→	NP VP	VP	→	VB @VP
NP	→	DT @NP	@VP	→	NP RB
NP	→	DT NN	DT	→	The
NP	→	NN NNS	NN	→	fish
NP	→	NN NN	VB	→	fish
NP	→	NN RB	NN	→	market
NP	→	NN	VB	→	market
@NP	→	NN @NP	NNS	→	stands
@NP	→	NN NN	VBZ	→	stands
@NP	→	NN NNS	RB	→	last
VP	→	VBZ RB	VBD	→	last

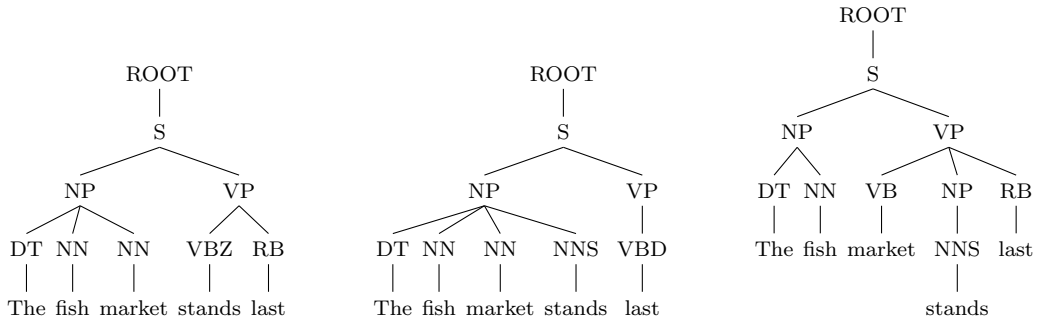
Table 2.2: A simple grammar, intended to accept the example sentence in Figure 2.2.

	Rule Application(s)	State
	ROOT	
1)	ROOT → S	S
2)	S → NP VP	NP VP
3)	NP → NN NNS	NN NNS VP
4)	VP → VB	NN NNS VB
5)	NN → *market*, NN → *stands*, and VB → *fish*	market stands fish

Note that different derivations can produce the same sentence. For example, we can reverse steps 3 and 4, or apply the first rule in step 5 prior to step 4, without changing the final string.

2.2 Context Free Grammars

A context-free grammar G defines a context-free language (Type 2 in Table 2.1). The rewrite rules in P are of the form $A \rightarrow \gamma$, where $A \in V$ and $\gamma \in (V \cup T)^+$. Note that γ includes the empty production *epsilon*, although this form is disallowed in certain stricter

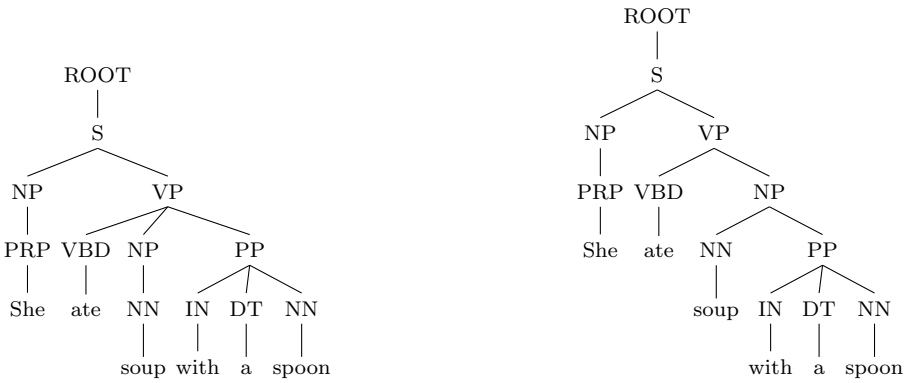


(a) The fish market is the last one standing.

(b) Long-lasting market stands.

(c) Fish doing marketing

Figure 2.2: Three possible parses for the example sentence, 'The fish market stands last', and the interpretations they imply.



(a) Preferred attachment

(b) Dispreferred attachment

Figure 2.3: Example of prepositional phrase (PP) attachment ambiguity. The modifier 'with a spoon' attach to the VP, thus modifying 'ate', or to the object NP, modifying 'soup'. Disambiguating between PP attachment targets can be a very difficult task for syntactic parsers.

formalisms (see below). Table 2.2 demonstrates a simple context-free grammar (CFG), designed to accept the example sentence in Figure 2.2, 'The fish market stands last'.

A language $\mathcal{L}_G$ is considered *context-free* if all sentences in the language can be derived from $S^\dagger$: $\mathcal{L} = \{t \in T^* : S^\dagger \overset{*}{\Rightarrow} T^*\}$. A CFG is defined as *proper* if and only if:

1. All non-terminals $A \in V$ are accessible — $\forall A \in V, \exists \alpha, \beta \in \{V \cup T\}^* : S^\dagger \overset{*}{\Rightarrow} \alpha A \beta$
2. All symbols are productive, generating one or more terminals — $\forall A \in V \, \exists t \in T^* : A \overset{*}{\Rightarrow} t$
3. No epsilon productions ($A \rightarrow \epsilon$) are allowed.

With the exception of simple example grammars, all grammars discussed in this thesis are induced from annotated treebanks (as described in Section 2.7). Each non-terminal in the treebank is observed in a tree rooted at $S^\dagger$ and derives one or more terminal symbols, so these conditions are guaranteed [36].

2.2.1 Strengths and Weaknesses of CFGs

CFGs can represent the vast majority of human language constructs [39]; notable exceptions include cross-serial dependencies in Swiss German and Tagalog [166, 116] and reduplication in Bambara [52]. *Context-sensitive* grammar formalisms address these limitations, but at a very steep computational cost — CFG processing is $O(n^3)$ (as described in Section 2.3), and fully context-sensitive approaches are exponential in the sentence length. Even *mildly context-sensitive* formalisms [93] — of which Tree Adjoining Grammars [94] and Combinatorial Categorial Grammars [42] are well-known examples — are considerably more expensive to process, asymptotically $O(n^4)$ or greater.[3] This thesis targets high-throughput and low-latency parsing; cubic-complexity CFG inference is already problematic for many application domains, and increased complexity is usually impractical. So we will restrict our focus to the context-free formalism.

2.2.2 Weighted and Probabilistic CFGs

The unweighted CFGs we have discussed thus far admit a binary decision on any sequence — is the sequence in or out of the language $\mathcal{L}$? Given the ambiguity encoded in most human languages, we may wish to make finer-grained distinctions, comparing

[3] General-case TAG and CCG algorithms are $O(n^6)$. *Splittable* tree-adjoining grammars can be processed in $O(n^4)$ [29]. Other mildly context-sensitive approaches are at least $O(n^5)$.

Parent		Children	Prob	Parent		Children	Prob
ROOT	→	S	1	VP	→	VBD	1/3
S	→	NP VP	1	VP	→	VB @VP	1
NP	→	DT @NP	1/4	@VP	→	NP RB	1
NP	→	DT NN	1/4	DT	→	The	1
NP	→	NN NNS	1/8	NN	→	fish	2/3
NP	→	NN NN	1/8	VB	→	fish	1/3
NP	→	NN RB	1/8	NN	→	market	2/3
NP	→	NN	1/8	VB	→	market	1/3
@NP	→	NN @NP	1/3	NNS	→	stands	1/2
@NP	→	NN NN	1/3	VBZ	→	stands	1/2
@NP	→	NN NNS	1/3	RB	→	last	2/3
VP	→	VBZ RB	1/2	VBD	→	last	1/3

Table 2.3: The grammar from Table 2.2 with the addition of probabilistic rule scores, allowing comparison of the three parses in Figure 2.2.

derivations to one another. For example, we might wish to rank the candidate parses in Figure 2.2, or those in Figure 2.3. A weighted context-free grammar (WCFG) adds this capability through a mapping ρ from $P \to \mathbb{R}$, assigning a real-valued weight to each rule. Conditionally normalizing the weights on the rule parents yields a probabilistic context-free grammar, or PCFG (e.g. Table 2.3). WCFGs and PCFGs are equally expressive [168], and most of the algorithms discussed in this thesis are applicable to either; however, we can more easily compare scores of disparate nonterminal labels if the grammar is properly normalized, and we can compute the probability of a (sub)tree as the product of the probabilities of all productions incorporated therein.

2.2.3 Binarization

Many algorithms, including those of primary interest to us, require a binarized grammar; that is, we must transform the grammar into a weakly equivalent grammar containing only

rules rewriting to 2 or fewer non-terminals.[4] One possible transform yields a grammar in *Chomsky Normal Form* [84]. CNF permits only productions of the form $A \rightarrow B\ C$ and $A \rightarrow \alpha$ where $A, B, C \in V$ and $\alpha \in T$. Note that CNF does not allow unary productions; we can achieve full CNF by collapsing all unary productions into new non-terminals, but this increases the non-terminal space considerably, so we generally choose to retain unary productions in our models and handle them appropriately during inference.

We binarize a grammar by dividing n-ary productions into multiple rules, creating additional 'factored' non-terminals as required. For example, we can transform the rule $A \rightarrow B\ C\ D$ into 2 rules: $A \rightarrow @A\ D$ and $@A \rightarrow B\ C$. When creating a new binarized rule, we can place the original non-terminal consistently to the left of the factored non-terminal or to the right (left- and right-binarization, respectively), or according to some other rule set (as demonstrated in Figure 2.4).

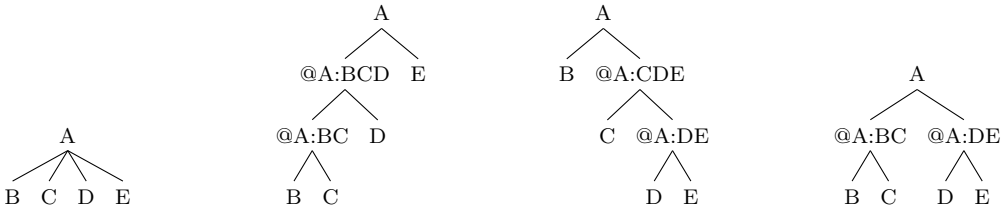
In some cases, we may choose during binarization to also Markovize the grammar [119]. That is, we 'forget' the sibling context beyond some specified limit, thus collapsing some of the newly-introduced factored non-terminals. Figure 2.5 demonstrates Markovization of the same example tree from Figure 2.4. In Figure 2.5b, @A:CDE has been abbreviated to @A:C. Or we may collapse all factored versions of a non-terminal, as in Figure 2.5c; this produces the most compact grammar, but one in which the newly-introduced categories encode no context whatsoever.

2.3 Parsing and the CYK Algorithm

As described in Section 2.1, we can apply a derivation (a sequence of rewrite rules from a grammar G) to produce a string in the language $\mathcal{L}$. Given a sequence from T, we can also produce candidate derivations for that sequence. Inference of this form is commonly called simply 'parsing'.[5] Often, many potential derivations represent the same hierarchical

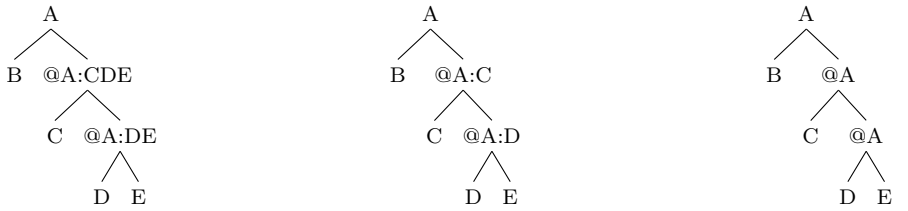
[4] This transformation allows sharing of common substructure, permitting dynamic programming algorithms such as those described in Sections 2.3 and 2.6.

[5] The general term 'parsing' can refer to several different algorithms. We are focused primarily on *constituency* parsing; when discussing other forms of parsing, we will denote the distinction explicitly.



(a) N-ary rule (b) Left (c) Right (d) Equal

Figure 2.4: Example binarizations of an n-ary production. For linguistic processing, we often choose either left- or right-binarization, but other binarizations are possible, as demonstrated in (d). In these examples and others, we prefix the newly introduced 'factored' categories with '@'.



(a) Binarization only (b) Markov-order-1 (c) Markov-order-0

Figure 2.5: Markovization during binarization. When binarizing a tree, we can record all of the sibling context into the newly-introduced categories, as in (a), or we can 'forget' sibling context beyond some specified limit — e.g. beyond 1 sibling as in (b), or all context, as in (c).

structure, differing only in the order of rule application; downstream processes generally consider these alternatives as equivalent, so we represent the entire equivalence class as a single tree structure. Parsing is commonly applied in compilers, NLP, bioinformatics, machine translation, and other fields. Unambiguous languages, such as programming languages and XML are parsed effectively by deterministic parsing approaches; natural languages are very ambiguous, permitting multiple interpretations of the same input. For example, consider the various parses of the same sentence in Figure 2.2.

Because of this ambiguity, any robust PCFG will permit a large number of trees $\mathcal{Y}$ for any input sentence $x \in T^*$ (and many spurious ambiguities arising from massive overgeneration). The rule probabilities allow us to score competing (permitted) derivations; we

generally want to recover $\hat{y}$, the most probable tree.

$$\hat{y} = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} P(y|x, G) \tag{2.1}$$

Exhaustively enumerating $|\mathcal{Y}|$ is very expensive; the graph search space is exponential in the sentence length n, so inference is impractical even over relatively short linguistic sequences. However, this problem meets the requirements for dynamic programming [13] — namely: 1) *Optimal substructure* — the optimal tree incorporates locally optimal parses of substrings and 2) *Overlapping subproblems* — Naïve search solves the same subproblems repeatedly; *memoizing* and reusing these partial solutions instead of recomputing them reduces the search space greatly. Algorithm 2.1 shows pseudocode of the widely-used CYK algorithm [45, 96, 188], which parses in $O(n^3)$ time.[6] We store subsolutions in a 2-D chart structure; each *cell* in a CYK chart contains possible non-terminal labels over a particular substring (or *span*) of the entire input. Computation for each cell depends only on those cells covering smaller spans within the same substring, so we can compute potential labels for each span once and memoize them for subsequent reuse while processing higher cells (longer spans). Figures 2.6 and 2.7 demonstrate example chart structures, the first populated with a single tree, and the second demonstrating a *packed parse forest* — a

[6] This algorithm, first presented in the 1960's, is alternatively referred to as 'CYK' and 'CKY'.

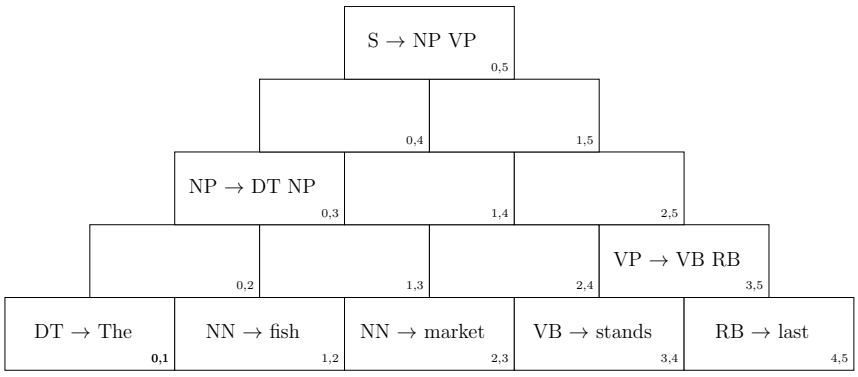


Figure 2.6: A CYK chart populated with labels matching the parse tree in Figure 2.2a.

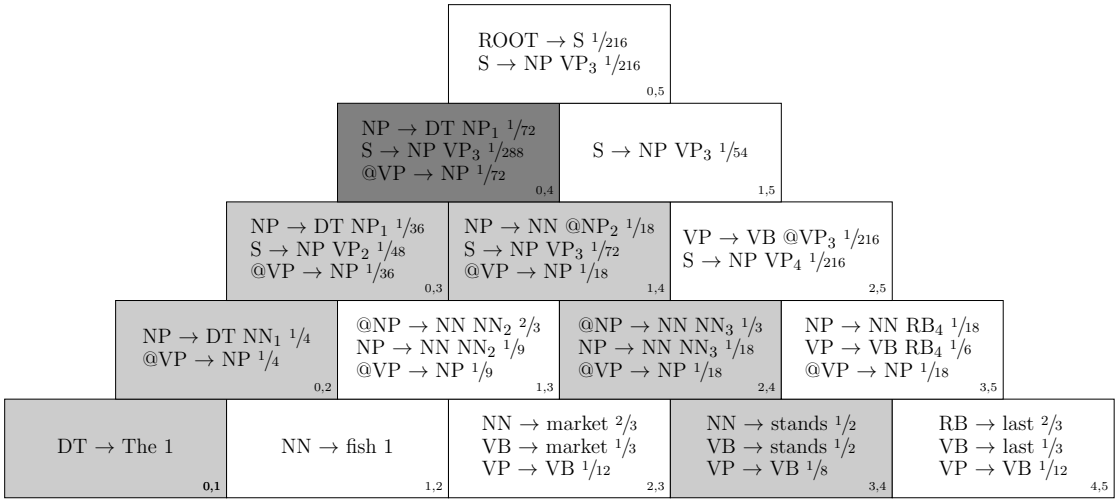


Figure 2.7: A CYK chart populated with a packed parse forest — a compact representation of many unique parse trees. The cell spanning the first 4 words is marked in dark grey, and the potential subconstituents of that cell in lighter grey.

compact representation of a (potentially) exponential number of trees.

We begin by initializing span-1 cells (the bottom row of the chart) with all part-of-speech (POS) tags,[7] and with any phrase-level labels spanning a single word. At higher (longer-span) cells, such as the dark grey cell in Figure 2.7, we build new constituents by combining constituents spanning adjacent substrings, guided by the productions in the grammar. In each cell in the chart, we store the highest score for each entry in V, along with a backpointer to the child non-terminals it produces.[8] We process $O(n^2)$ cells, and examine $O(n)$ midpoints for each cell, yielding $O(n^3 |G|)$ complexity (where $|G|$ is the size of the grammar). Unfortunately, even this (relatively) efficient asymptotic complexity in n is often dominated by the *grammar constant* G, and exhaustive search remains intractable for large grammars. Subsequent chapters present several approaches to this problem.

[7] In most treebanks, parts-of-speech (POS) and phrase-level labels are disjoint sets. Thus, POS labels occur only in conjunction with terminals (words), and are also referred to as *preterminals*.

[8] We may choose to maintain a backpointer containing a midpoint and a rule from P, or we may conserve storage by omitting the midpoint at the cost of additional search effort after chart population when decoding a tree from the packed forest.

Algorithm 2.1 CYK$(w_1 \dots w_n)$, $G = (V, T, S^\dagger, P, \rho))$

PCFG G must be in CNF; α represents the population of the current cell. Notation adapted from Roark and Sproat [157].

1: **for** $b = 1$ to n **do** ▷ Span = 1 (Words/POS tags)
2: **for** $j = 1$ to $|V|$ **do**
3: $\alpha_j(b, 1) \leftarrow \rho(A_j \rightarrow w_b)$
4: **for** $s = 2$ to n **do** ▷ All spans > 1 (rows in the chart)
5: **for** $b = 1$ to $n - 1$ **do** ▷ All start-points (cells in a row)
6: $e \leftarrow b + s - 1$ ▷ End-point for cell
7: **for** $i = 1$ to $|V|$ **do**
8: $\alpha_i(b, e) \leftarrow \max_{j,k \in V,\, b < m \leq e} \rho(A_i \rightarrow A_j A_k)\, \alpha_j(b, m-1)\, \alpha_k(m, e)$
9: $\zeta_i(b, e) \leftarrow \operatorname{argmax}_{j,k \in V,\, b < m \leq e} \rho(A_i \rightarrow A_j A_k)\, \alpha_j(b, m-1)\, \alpha_k(m, e)$

Note that the argmax in Equation 2.1 and in Algorithm 2.1 lines 7–8 maximizes the probability of the derivation, but alternate objective functions are also possible, such as those that maximize measures of expected constituent accuracy. We will present experiments comparing some of those methods in Chapter 6.

2.4 Evaluation

The constituency tree structure recovered by the CYK algorithm can be represented as a multiset of labeled spans T (e.g., a label of NP on the first 4 words of a sentence might be represented as $\text{NP}_{0,4}$). The standard evaluation metric, known as `PARSEVAL` [17] measures labeled precision (LP) and labeled recall (LR) of those spans vs. the gold tree τ (excluding parts-of-speech, but *including* phrase-level nonterminals spanning a single word).

$$LP = \frac{|T \cup \tau|}{|T|} \qquad LR = \frac{|T \cap \tau|}{|T|} \tag{2.2}$$

If a single summary statistic is desired, the common convention in the parsing community is to report the harmonic mean of these two measures, a metric generally labeled as F_1, or equivalently F-score or F-measure. F_1 appears to be reasonably well correlated with the impact of parse accuracy on many downstream tasks. State-of-the-art parsers

achieve $F_1 > 90$ for English newswire text, but performance is considerably lower on most other languages and on non-canonical genres, and even $F_1 > 90$ is still considerably lower than human inter-annotator agreement [23].

2.5 Expectation Maximization

The Expectation Maximization (EM) algorithm [54] learns latent (unobserved) parameters of a statistical model from observed data points. EM training first initializes a random model θ^0, and then iterates between an *expectation* step, which calculates the likelihood of the observations X under the current model θ^t, and a *maximization* step, which updates the model parameters to maximize the likelihood of those observations.

$$L(X|\theta^t) = \prod_{x \in X} p(x|\theta^t) \tag{2.3}$$

$$\theta^{t+1} = \underset{\theta}{\operatorname{argmax}}\ L(X|\theta^t) \tag{2.4}$$

Familiar examples of EM algorithms include the Baum-Welch algorithm for parameterizing HMM models [11], and the *k-means* clustering algorithm [117, 113].

EM guarantees improvement in the training likelihood on each iteration, and eventual convergence to a consistent model [183]. However — like many other iterative optimization techniques — EM is subject to local minima, so the learned model is not guaranteed to be the maximum-likelihood model for the training data. The random initialization state determines the trajectory of the optimization, so training runs with different initialization states may converge to very different models; thus, one common approach trains multiple models and selects a final model based on performance on held-out data.

EM is also prone to overfitting, particularly if run to convergence. Validation on held-out data and careful application of alternate stopping criteria can in some cases help prevent overfitting. In the following section, we will discuss the Inside-Outside algorithm, an application of EM PCFG training. Section 2.9 discusses one approach to avoid local minima and alleviate overfitting in PCFG training, and chapters 4 and 5 present novel methods expanding on that approach.

2.6 Inside-Outside Algorithm

The Baum-Welch algorithm [11] is a form of EM frequently used to train Hidden Markov Models. The E step of Baum-Welch performs inference with the current model, using the Forward-Backward algorithm to sum probabilities over all paths. Forward-backward operates on linear sequences and HMMs; the inside-outside algorithm [7, 108] is the analog over trees and context-free grammars; thus, we can use inside-outside to iteratively re-estimate grammar probabilities.

Inside-outside, a close relative of CYK, computes posterior probabilities of each labeled span. We compute *inside* probabilities of non-terminal span labels as in Algorithm 2.1, but we replace the argmax in line 8 with a summation:

$$\alpha_i(b,e) \leftarrow \sum_{m=b}^{e-1} \sum_{j \in V} \sum_{k \in V} \rho(A_i \rightarrow A_j A_k)\, \alpha_j(b, m-1)\, \alpha_k(m, e) \tag{2.5}$$

Since α is a sum over all possible derivations, we omit the backpointer ζ. Note that $\alpha_{S^\dagger}(0,n) = \mathrm{P}(w_1 \ldots w_n)$, the probability of the sentence according to G (i.e., the sum of the probabilities of all derivations of $w_1 \ldots w_n$).

We then compute the *outside* probability for each label and span; informally, the outside probability is the probability that a label will combine with the words to its left and those to its right to participate in a full parse tree. The label may join with a label on its left or on its right, so the total probability includes both possibilities:

$$\begin{aligned} \beta_i(b,e) \leftarrow & \sum_{b'=1}^{b-1} \sum_{j \in V} \sum_{k \in V} \rho(A_i \rightarrow A_k A_j)\, \beta_i(b', e)\, \alpha_k(b', b-1) + \\ & \sum_{e'=e+1}^{n} \sum_{j \in V} \sum_{k \in V} \rho(A_i \rightarrow A_j A_k)\, \beta_i(b, e')\, \alpha_k(e+1, e') \end{aligned} \tag{2.6}$$

We can now compute the posterior probability of a labeled span as:

$$\gamma_i(b,e) = \frac{\alpha_i(b,e)\, \beta_i(b,e)}{P(w_1 \ldots w_n)} \tag{2.7}$$

And the conditional probability of each rule at a span:

$$\xi_{ijk}(b,e) = \frac{\alpha_{ijk}(b,e)\,\beta_{ijk}(b,e)}{P(w_1 \dots w_n)} \sum_{m=b+1}^{e} \alpha_j(b,m)\,\alpha_k(m,e) \tag{2.8}$$

We can use these values to maximize the conditional probability of a parse tree or to re-estimate the PCFG. In the following section, we will describe induction of a PCFG model from a labeled corpus, and Section 2.9 will incorporate γ and ξ to optimize those models.

2.7 PCFG Induction

We generally induce PCFGs from a manually-annotated corpus of labeled trees, known as a 'treebank.' In recent years, treebanks have been developed in a variety of domains, including newswire text in English, Chinese, and other languages [120, 185, 167], telephone conversations [71], and web text [14]. Treebank-trained grammars generally cover a much wider range of real-world language usage than do hand-coded grammars, so nearly all modern parsers incorporate treebank training.

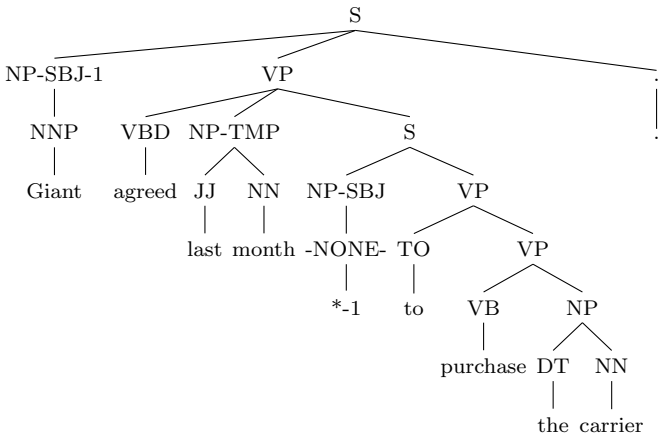
2.7.1 Corpus Transformations

Many treebanks, including those we evaluate on, annotate information beyond context-free trees, such as tense, aspect, gender, predicate-argument structure, and elided or implied clauses (indicated by empty nodes). We often drop this additional information prior to grammar induction; if required by a downstream process, we can often recover these annotations following parsing [90, 55, 25]. In this thesis, we apply the following preprocessing operations to all corpora:

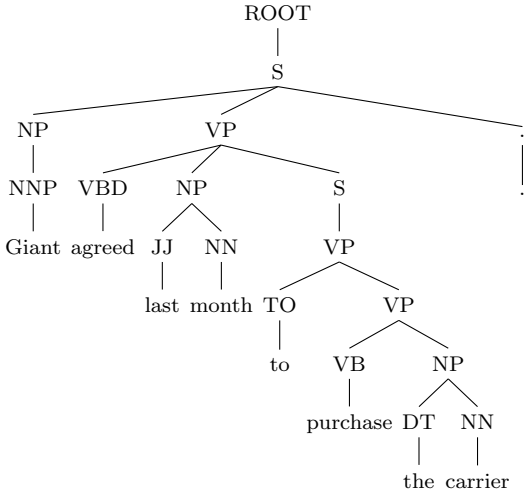
1. Remove empty nodes. We focus on bottom-up inference mechanisms, which do not allow arbitrary insertion, so parameterizations learned from empty nodes are not useful at time of inference.

2. Strip functional tags and cross-referencing annotations.

3. Affix a root unary production to the root symbol of the original tree.



(a) Penn Treebank tree structure



(b) Processed tree structure

Figure 2.8: Preprocessing applied to Penn Treebank annotations

Figure 2.8 demonstrates these transformations as applied to a tree from the Penn Treebank [120]. Preprocessing practices vary somewhat amongst researchers; the three operations we apply are fairly standard; other researchers add some combination of the following transformations as well:[9]

1. Collapse unary chains to a single (possibly composite) unary production [98]. Many inference systems are limited by implementation details to unary chains of a limited

[9]Most of these transformations are not included in the standard accuracy evaluation, so minor differences in preprocessing between researchers are unlikely to greatly affect the comparability of their results.

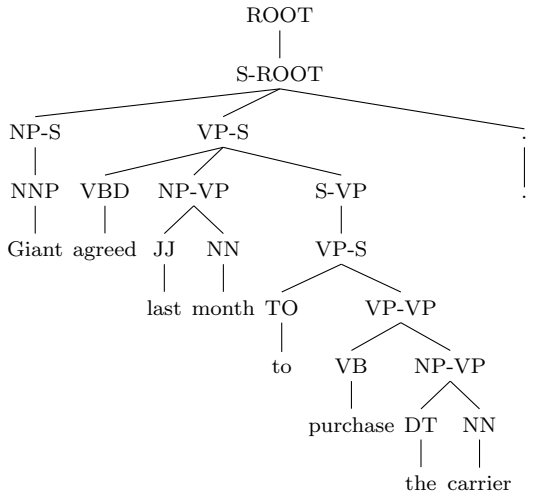


Figure 2.9: Parent-annotated version of the tree in Figure 2.8b.

depth (often only one). This transform allows recovery of the top unary parent, even if intermediate labels are unavailable. If we introduce new labels to represent unary chains, we can recover the entire chain as a post-processing step.

2. Remove X → X unary productions for all non-terminals X.[10]

3. Introduce new categories such as AUX. (Charniak, 1997)

4. Collapse categories such as PRT and ADVP. [46]

5. Remove quotation marks. [18]

2.7.2 State-splitting and Smoothing

The most straightforward grammar induction method simply normalizes over counts of production occurrences in the treebank, producing a maximum-likelihood estimate [31].

$$P(A \rightarrow B\ C) = \frac{count(A \rightarrow B\ C)}{count(A)} \tag{2.9}$$

[10]Many inference systems cannot produce X → X productions, so this transformation may affect reported accuracy, but such productions are relatively rare, so the accuracy increase will be small.

Model	$\|V\|$	$\|P_b\|$	F_1
Markov-order-0	100	3.8k	62.9
Markov-order-2	2.6k	12k	74.3
Parent-Annotated [89]	6k	22k	78.6
Lexical [32]	Implicit	Implicit	89.6
Klein and Manning [100]	15k	46k	85.7
Petrov et al. [142]	1.1k	1.7m	89.7

Table 2.4: Relative sizes and Viterbi-search accuracies of various grammars. V is the non-terminal set. Common entries include NP for Noun Phrase, JJ for adjective, S for sentence, etc. $|P_b|$ is the number of binary rules in the grammar.

On the commonly-used Penn Treebank [120], the resulting grammar achieves a labeled F_1 score of approximately 65. This simple grammar suffers from two important and seemingly contradictory problems: 1) Common treebank productions often occur in multiple contexts, so the resulting rule probabilities conflate distinct grammatical constructions and often cannot disambiguate effectively; 2) Even sizable annotated corpora are unlikely to include examples of all grammatical phenomena needed by a general-purpose parser. E.g., the large n-ary productions found in the training corpus are unlikely to match those needed in a test corpus, even within a similar domain.

We can solve the first problem by further splitting the non-terminal space; for instance, we might choose to annotate each $A \in V$ with its parent non-terminal [89]. For example, an NP descended from an S becomes NP-S, whereas one descended from a VP becomes NP-VP. Thus, the NP state is split into several substates; Figure 2.9 shows a version of the tree from Figure 2.8b annotated in this manner.

Unfortunately, the solution to the problem of disambiguating contexts exacerbates the second problem, that of 'sparse data'. We approach sparse data by smoothing the grammar probabilities; i.e., by reserving some probability mass for unseen productions. In some cases, Markovization (as described in Section 2.2.3) can be an effective form of smoothing. However, Markovization is often too coarse a tool, failing to discriminate between useful and less-useful context. More sophisticated backoff and smoothing methods

generally rely on knowledge of specific linguistic structures [49, 32].

2.7.3 Bilexical Grammars

We can split the non-terminal space further, conditioning each production rule not only on the parent non-terminal, but also on grandparents and other related edges or on nearby lexical items. One popular approach annotates each non-terminals with its lexical 'head' [49, 32], where the head of a label is chosen from the heads of its 2 children via a set of inheritance rules. Naturally, these 'bilexical' grammars suffer even more from problem sparse data. In fact, Bikel [15] found that bilexical probabilities from the training corpus were available for less than 1.5% of all probability calculations during inference.

Bilexical grammars (with appropriate smoothing) can be very accurate (c.f. Table 2.4). Unfortunately, they may also be impractically large. Under reasonable memory constraints, we cannot fully enumerate wide-coverage grammars conditioned on bilexical probabilities. Parsing with such grammars requires computing backoff probabilities on-the-fly, a sizable computational cost. And the computational complexity is worse as well — the naive inference approach is $O(n^5)$; more sophisticated methods reduce that somewhat [62], but the cost of lexicalization is still considerable. So solving the problems of disambiguation and sparse data yields an accurate grammar, but at the cost of expensive inference. The following sections describe one approach to this problem, and chapters 4 and 5 present several methods of training compact, accurate, and efficient grammars.

2.8 High-Accuracy Unlexicalized Grammars

One solution to the high cost of inference with lexicalized grammars is to return to the smaller state-space of unlexicalized grammars and to address generalizability by clustering non-terminals. That is, to subdivide the non-terminal space with annotations not directly encoding lexical items or relations directly extracted from the training trees. For example, a lexicalized grammar would annotate the DT (determiner) symbols with the associated word (DT-THE, DT-A, DT-AN, and so on). However, noting that 'the' is the most common determiner, and that 'a' and 'an' operate in similar contexts, we might instead choose to

split the DT symbol in 3 — DT_0 for 'the', DT_1 for 'a' and 'an', and DT_2 for all other usages. This limited state-splitting allows specialized non-terminals when appropriate, without the massive explosion of the state space of full lexicalization. These unlexicalized PCFGs can be much more compact than bilexical grammars, both in non-terminal space and rule count. We are particularly interested in such enumerable grammars, which permit exhaustive search, efficient memory representations, and analysis of a grammar's efficiency characteristics.

One advantage of the Markovized or lexicalized grammars discussed earlier is that training such grammars is typically quite straightforward, consisting primarily of accumulating occurrence counts over the training corpus (as in Equation 2.9). In contrast, training an unlexicalized grammar is more complex.

Klein and Manning [100] observed that parent annotation does not exhaust the possibilities for effective state-splitting. They present a set of linguistically-motivated manual annotations which further split the state space. For example, they divide the IN tag into 6, including specific tags for noun-modifying prepositions (e.g., *of*), verb-modifying forms (*as*), etc. Each of these annotations provides a minor gain, and the combination thereof yields an F_1 of 85.7 on WSJ text, competitive with the state-of-the-art at the time. Moreover, their final grammar was relatively compact, with a sizable vocabulary, but a ruleset only about twice the size of the parent-annotated grammar in Table 2.4 (46k vs. 22k).

To our knowledge, this manual annotation approach has not (yet) been extended enough to compete with current state-of-the-art parsers, but it is certainly not unimaginable that additional linguistic annotations could further improve accuracy, while maintaining a compact and enumerable (and thus very efficient) grammar. However, this approach requires linguistic insight and observations of the likely parse errors, observations which are unlikely to generalize beyond a specific target domain. Klein and Manning's grammar performs very well on WSJ text, but their process of exploration and manual annotation would probably need to be repeated when moving from newswire to other English genres; it would certainly not apply to other languages.

One possible goal of unlexicalized grammar training is to learn effective grammars directly and automatically from training data, without manual intervention in the training

process. An automatic approach simplifies domain adaptation (e.g., by mixing training data from multiple domains) and has the potential to generalize across languages.

2.9 Split-merge Training of Latent Variable Grammars

Instead of predefining a method of splitting the state space (parent-annotation, lexicalization, etc.), we want to learn state-splits directly from the training corpus (ideally, splits which can both disambiguate *and* generalize). Matsuzaki et al. [122] demonstrated one such approach, which was further refined by Petrov et al. [142]. They subdivided each non-terminal into n separate state splits, annotating each with an arbitrary index — e.g. DT_0, NP_2, etc. Splitting the grammar in this way multiplies the number of productions in a predictable manner; each lexical rule is split into 2 (e.g., DT $\rightarrow$ *the* in the baseline grammar becomes DT_0 $\rightarrow$ *the* and DT_1 $\rightarrow$ *the*). Each unary production is divided into 4, and each binary production into 8. They use EM to parameterize this expanded grammar — that is, to learn appropriate probabilities for each production.

They titled this approach *PCFG with latent annotations*, or *PCFG-LA*. 'Latent', in this instance, refers to the fact that the non-terminal annotations (split indices) are not directly observed in the training data, and that the production probabilities instead arise indirectly through iterated optimization. The algorithm is fully described in those publications, but the summary here is sufficient to describe the modifications we will present in Chapter 5

1. Induce a simple Markov-order-0 binarized grammar. When trained on the Penn Treebank, this grammar will consist of approximately 100 non-terminals, 3800 binary productions, 100 unary productions, and 55,000 lexical productions.

2. Annotate each category with a latent index of 0. 'Latent', in this instance, refers to the fact that the non-terminal annotations are not directly observed in the training data (after state-splitting in Step 3), and that the production probabilities instead arise indirectly through iterated optimization.

3. Split each category into 2 sub-states, labeled 0 and 1 respectively (DT_0, DT_1, NP_0, NP_1, etc.). This split divides each lexical production into two, each unary

production into 4, and each binary production into 8.

4. Randomize each production probability slightly to break symmetry.

5. Iterate over the training corpus, performing expectation maximization (as described in Section 2.5), to estimate probabilities for each grammar production.

   - The expectation step computes γ, the posterior probabilities of each labeled span, as described in Section 2.6. Since these posteriors can be constrained by the gold trees, this operation is $O(n)$ [139].
   - Maximization re-estimates rule probabilities using ξ from the previous calculation.

$$\rho(A_i \rightarrow A_j A_k) = \frac{\sum_{\tau, n \in \tau} \gamma_i \rho(A_i \rightarrow A_j A_k) \alpha_j \alpha_k}{\sum_{\tau, n \in \tau, j', k'} \gamma_i \rho(A_i \rightarrow A_{j'} A_{k'}) \alpha_{j'} \alpha_{k'}} \tag{2.10}$$

6. Estimate the cost of re-merging each split pair (e.g., the cost of re-combining NP_0 and NP_1 into a single NP_0 category).

7. Re-merge the least costly 50% of the splits.[11]

8. (Optional) Smooth the post-merge production probabilities.

9. Repeat the split-EM-merge-smooth process (steps 3–7) several times — 5–7 cycles seems to work well.

Matsuzaki et al. [122] and Dreyer and Eisner [57] demonstrated the effectiveness of state-splitting and EM parameter estimation. The addition of the re-merge operation [142] offers two related benefits:

1. It reduces greatly the number of parameters we must learn and encode into the grammar. Performing 4 split cycles without any merging could produce as many as 15

[11] Note: we may perform additional EM iterations after the merge and smoothing steps. This can further improve the training-corpus likelihood, but does not alter the overall structure of the algorithm.

million binary productions (3840×2^4), and 6 cycles could exceed 1 billion.[12] Grammars that large are impractical on current hardware, and even a sizable treebank does not encode enough information to accurately estimate the rule probabilities.

2. An effective merge criteria removes many of the less-helpful state-splits. For example, splitting the comma non-terminal into ,_0, ,_1, ... is likely to produce highly specific productions which fail to generalize.

This framework produces very accurate grammars, exceeding the results reported in similar work on latent-variable grammar learning [122, 57], and approximately matching the accuracies obtained with lexicalized grammars and discriminative reranking [34].

Further, the method is robust across languages and genres, requiring little manual intervention to adapt to new languages and genres (c.f. Petrov et al. [142], Petrov [141], Le Roux et al. [109]).[13] However, while the vocabulary of the resulting grammars is quite compact, the ruleset is generally very large — usually multiple millions of productions — and inference is costly. In Chapter 5, we will describe limited changes to the training algorithm, primarily to the merge ranking criteria in step 6, aiming to improve the efficiency of the final grammar.

2.10 Evaluation of Latent-Variable Learning Algorithms

Randomization (step 4 in the learning algorithm) plays an important role in training latent-variable grammars. Petrov [141] observed significant differences between grammars trained with different random seeds. Varying the random seed affects the distribution of state-splits retained through the split-merge process. The general patterns remain clear regardless of seed — e.g., NP, VP, NNP, JJ are retained the most, and $ and UH are nearly always re-merged. But the variance in retained annotations is considerable, especially for the most-common classes.

[12] Note that EM would reduce the probability of some — perhaps many — of those productions to 0, but even so, the size of the grammar would be intractable.

[13] Adapting to a new language or genre will often require modifications to unknown-word processing, but the remainder of the system remains unchanged.

These differences (and, presumably the differences in production probabilities learned) have noticeable impact on parse accuracy. At a coarse level, some grammars perform measurably better than others on standard PARSEVAL scoring; when evaluated more finely, varying the random seed often moves performance on recovery of different labels (NP, PP, QP, etc.) and on various higher-level subtree evaluations such as PP-attachment and coordination [141].

Any modification to the learning algorithm will impact the choice of which state-splits to re-merge, and thus affect the final learned grammar. Given the inherent variance in the algorithm, we cannot confidently evaluate a modification to the learning approach based on a single learned grammar. In all trials presented in this thesis, we train multiple grammars with each approach (each with a different random seed) and evaluate the performance of the learning approach based on the average performance of those grammars.

2.11 Hardware Background

Low-level hardware issues are rarely discussed in the NLP literature, so we present here a greatly simplified description of processor architectures and memory hierarchies. Our analysis will not extend to quantitative modeling of memory access times, but our qualitative discussion refers often to cache behavior, so we include a brief introduction here.

Semiconductor designers over the past few decades have been very successful at scaling processor transistor counts. Until quite recently, they have also been able to consistently increase clock speeds. Unfortunately, memory access speeds have not kept pace with the increases in processing power. The execution time of most algorithms is now dominated by memory latency [51]. To alleviate this inefficiency, modern CPUs are isolated from main memory by two or more layers of faster cache memory; memory requests are first checked against a Level 1 cache (*L1*) of 16-32 KB, and then against a larger, but slower, Level 2 cache (*L2*).[14] Recently-accessed memory items are kept in cache, and older items are evicted. Each successive layer (L1, L2, and main memory) is roughly 5–10× slower

[14] Many architectures, particularly multicore designs, add a third layer of cache as well; in such cases, the hierarchical model is unchanged, so this condition does not affect our analysis.

than the last. Thus, frequent reuse of nearby memory can dramatically improve runtimes vs. an uneven access pattern.

Further, most processors implement cache prediction and pre-fetch logic, which attempts to anticipate upcoming memory requests. For example, in-order iteration through an array is easily predictable, and the CPU is able to fetch subsequent memory locations into cache *before* the program requests them (*prefetching*), providing cache hit rates above 95% for many workloads. Other CPU technologies, such as branch prediction, register renaming, and speculative execution, can allow processing to continue during a memory read, avoiding some latency-induced pipeline stalls. But even the most advanced processor technology cannot completely overcome the impedance mismatch between processor clocks and memory latency, so the execution pattern of many programs is: process (very quickly) for a few clock cycles, wait many clock cycles for data, process for a few cycles, wait for data, repeat... Thus, throughput on most workloads is nowhere near the chip's theoretical peak. In fact, execution time often increases super-linearly as a program's working set expands beyond the CPU's cache size, so expanding an algorithm's memory footprint can dramatically increase processing time. [53, 97, 56]

We apply some of these considerations to the algorithms and implementation described in Chapter 3, resulting in greatly improved parsing throughput. We refer interested readers to the discussions of memory hierarchies and latency in Hennessy and Patterson [82] and Drepper [56].

Graphics processors (GPUs) differ considerably from CPUs, devoting most of their available silicon to arithmetic logic units, rather than to latency alleviation. For example, the NVIDIA GT200 GPU contains 240 individual cores, and the GK104 series up to 1536; each handles thousands of simultaneous threads.[15] On certain problems, GPUs can achieve tremendous speedups; some dense matrix operations are accelerated by 100× or more. For our purposes, it is sufficient to understand that GPUs offer a considerable advantage on algorithms which 1) scale to large thread counts with limited interaction between threads; and 2) access memory in a largely sequential pattern.

[15] Contrast this to current Intel and AMD CPUs, which contain 4–8 cores and process at most 16 simultaneous threads

2.12 BUBS Parser

We perform most of our experimental trials with the open-source BUBS parser, to our knowledge currently the fastest publicly available high-accuracy constituency parser. BUBS implements several state-of-the-art pruning methods (described more fully in Appendix A). However, even the impressive performance of these pruning methods can be improved with better memory layout and hardware utilization (as demonstrated in Chapter 3).

In combination with the matrix grammar encoding described in Chapter 3, these pruning approaches yield the highest reported speeds on a variety of domains [19]. Further, the pruning models train rapidly for arbitrary grammars, allowing us to apply them during grammar training, yielding further improvements in both speed and accuracy as reported in Chapter 5.

A brief note about implementation choices is appropriate here. BUBS is implemented in Java, a language often (and we believe unfairly) viewed as inherently inefficient. This perception became established when Java runtime environments were interpreted rather than compiled. However, recent advances in virtual machine technology have largely eliminated the differential between Java and statically-compiled languages such as Fortran and C.

Java's dynamic just-in-time compiler (JIT) uses runtime statistics (information unavailable to a static compiler) to choose the most effective optimization approaches. Collecting the required statistics imposes some overhead during the first few loop iterations, but the advantages often outweigh that overhead for a long-running process [102, 132]. In addition to dynamic JIT technology, garbage collection [43], range check elimination [44, 184], and array access [172] have all advanced dramatically in recent years, and the performance of Java code is now comparable to — and in some cases superior to — similarly tuned C code [3].

2.13 Evaluation Criteria

In this section, we will describe the criteria by which we evaluate all the algorithms presented in this thesis. In Chapter 3, we evaluate grammar intersection approaches and

parallelization on the Penn Treebank. When evaluating learning algorithms in chapters 4 and 5, we include several other evaluation corpora, chosen to represent a variety of languages and genres:

Penn Treebank [121]. Approximately 1 million words of newswire text. Sections 02-21 (950k words) are traditionally used for training, sections 22 and 24 (73k words) for development, and section 23 (57k words) for testing.

Penn Chinese Treebank [185]. Approximately 540k words of Chinese newswire text. The PCTB training corpus is sizable, at approximately 525k words. However, the standard development and test sets (of 7k and 8k words respectively) are quite small. Thus, accuracy results on the PCTB corpus are likely to be quite noisy, but efficiency trials should be unaffected.

Switchboard Treebank [71]. 1 million words of transcribed telephone conversations.

English Web Treebank [14]. Approximately 250k words of English text, taken from weblogs, newsgroups, email, reviews, and question-answers. This corpus includes development and test sets (of approximately equal size) from each genre, but does not include training data. We report cross-domain evaluations on each of the 5 genres for models trained on WSJ and Switchboard text.

These corpora include both written and oral communication, and cover a wide range of interesting characteristics — language, genre, sentence length, and grammaticality. Table 2.5 reports statistics about each corpus. Trials across this wide range will expose differences between our methods as they apply to particular genres; methods that show benefits across a range of genres should be expected to generalize well to other data as well.

Accuracy: We report the standard `PARSEVAL` scores, as described in Black et al. [17]. We report cross-domain generalization for English models, evaluating models trained on WSJ text on telephone conversations and vice-versa, and evaluating all English models on web text from the English Web Treebank [14]. We also report label-specific accuracy on a subset of the label-set, chosen to represent interesting linguistic phenomena (e.g. the

Treebank	Training words	Test words	Words / Sentence Mean	Med.	Max	Dev-set UNKs
WSJ	950,028	56,684	23.8	23	141	2.8%
Chinese	493,708	8008	27.1	24	240	9.6%
English Web	–	150,644	16.7	14	180	–
Switchboard	852,758	60,999	9.4	7	114	1.1%

Table 2.5: Statistics about each of the evaluation corpora, including total word count of training and test sets, averages of test-set sentence lengths, and the percentage of tokens in the development set unobserved in the training set. Note: The English Web Treebank has no development set, providing instead a large collection of unlabeled in-domain data intended to facilitate domain adaptation.

'edited' label in the Switchboard corpus).[16]

2.13.1 Measuring Efficiency

For all inference trials, we report the average throughput in words per second, evaluated over a sizable corpus. This measure is common in the dependency parsing community, and in other work on efficient constituency parsing (c.f Sagae and Lavie [162], Bangalore et al. [8], Bodenstab [19]). As noted in Section 2.10, we report accuracy and efficiency metrics on split-merge-trained grammars as averages over a large number of training runs with differing random seeds.

Although we make every attempt to isolate the systems executing efficiency trials, operating system overhead and unrelated processes may in some cases distort timings. This distortion will by definition always decrease the measured throughput, so the ideal measure of algorithmic efficiency would be the maximum speed obtained over an infinite number of trials. Note that we don't know the expected distribution of speed measurements over these trials apriori; we expect it to be negatively skewed, with the median and mode

[16] Note that the label-sets differ somewhat between treebanks, so some of these label-specific tests are not applicable for cross-domain trials.

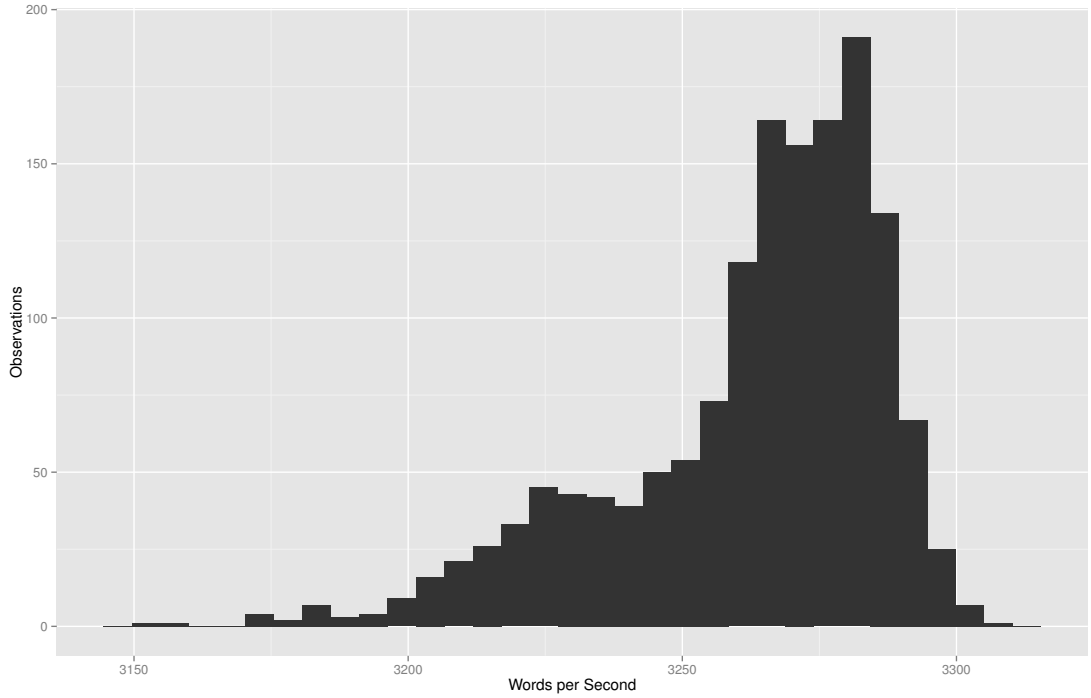


Figure 2.10: Observed distribution of speeds obtained in 1500 pruned inference trials on WSJ section 22.

Percentage of Trials	**Observed Variance from Maximum**
95%	1.04%
99%	1.26%
99.9%	1.65%
99.95%	1.78%
99.99%	2.31%

Table 2.6: Variance observed from population maximum speed in 5-trial samples from that population. E.g., in 95% of all 5-trial samples, the maximum speed observed was within 1.04% of the maximum observed in the 1500-trial 'population'.

near the maximum, and a lengthy left tail (of slower trials, which suffered some form of interference). Since an infinite — or even very large — number of trials is infeasible, we need to estimate the number of trials necessary to produce an efficiency estimate of reasonable accuracy. To quantify the aforementioned distribution, we executed 1500 pruned parsing

runs on a development set (WSJ section 22); Figure 2.10 displays the speeds obtained in the form of a histogram. The observed distribution matches our intuition fairly closely, with the median within 1.1% of the maximum.

For the moment, we will treat this set of 1500 trials as the 'true' distribution. While we could certainly find a probability distribution — or perhaps several — which could be parameterized to fit the empirical observations, we have no theoretical basis to choose one distribution over another. Instead, we consider a nonparametric approach, sampling from the 1500 observations to simulate a practical number of trials for a given experiment. We are interested in the expected difference between the true (distribution) maximum speed and the maximum we will observe in a limited number of trials. We extracted 10,000 random samples of 5 trials each, and computed the difference between the maximum speed observed in the 5-trial sample and the maximum from the full distribution. The variance observed in any 5-trial sample may be considerable, but as demonstrated in Table 2.6, 5 trials is generally sufficient to obtain at least one trial very close to the true maximum from the entire population. Thus, for the remainder of this thesis, we will execute all pruned efficiency experiments 5× and report the maximum speed observed in those trials.[17]

2.14 Summary

In this chapter, we described the algorithmic background on which the methods and experiments presented in subsequent chapters depend. We described existing work in grammar learning, and the potential for extension of that work to incorporate efficiency as a direct objective. Finally, we presented the experimental methodology and evaluation criteria we will use throughout the remainder of this thesis.

[17] Exhaustive inference is much slower, so small OS variances are less detrimental. We generally execute exhaustive trials 2 times, rather than the 5 we use for pruned trials.

Chapter 3

Grammar Encoding, Intersection Methods, and Parallelism

Natural language sequences are in general of moderate length, so processing time of NLP algorithms — including CYK parsing — is in some cases dominated by the size of a complex model as much as by the asymptotic complexity of the algorithm itself. An efficient model representation may permit novel algorithms, and in some cases can provide a real-world speedup comparable to an improvement in algorithmic complexity. The inner loop of the CYK algorithm (lines 7–8 in Algorithm 2.1) computes an *argmax* for each constituent span by intersecting the set of observed child categories spanning adjacent substrings with the set of rule productions in the grammar. This *grammar intersection* operation is the most computationally intensive component of the algorithm, and prior work has shown that the cost depends greatly on the grammar representation and access methods [22]. Klein and Manning [98] investigated and modeled the causes of observed super-cubic behavior. They found considerable efficiencies by encoding the PCFG as a finite-state automaton. Moore [125] similarly demonstrated that model encoding can lead to significant efficiency gains, and Penn and Munteanu [138] demonstrated substantial gains through algorithmic refactorings targeted at reducing low-level CPU operations.[1]

We begin this chapter with a matrix grammar encoding, motivated by the hardware-level memory access considerations discussed in Section 2.11. We compare that representation to the baseline implemented in BUBS, and find 13-45% improvements in L2

[1] All grammar encodings discussed, including our own, alter only efficiency; search is exact, so accuracy remains unchanged.

cache hit rate, and an overall increase in inference speed of 2–7×. We examine various approaches to performing the central *argmax* in CYK, and again we find that an appropriate choice of intersection technique yields considerable efficiency gains. In Sections 3.5 and 3.6, we present a matrix-vector grammar intersection method, and compare it to the other methods described previously.

We conclude the chapter with an examination of parallel parsing. Discussions of efficient parsing usually concentrate on *throughput*, the aggregate number of words parsed per second on a particular machine. However, for some applications, response time is of equal or greater interest — e.g., real-time speech recognition and machine translation, or mobile QA systems. Thus, we should also consider *latency*, the time to parse a single sentence, as a primary objective. The recent adoption of multicore CPU architectures and massively-parallel graphics processors allows finer-grained parallelism in parsing, potentially allowing lower latencies than those of serial approaches. We compare several approaches to parallel inference, including very fine-grained methods permitted by matrix-vector grammar intersection.

3.1 Parsing and Matrix Operations

Matrix manipulations are inherently parallel, so we are especially interested in relationships between CYK parsing and matrix operations, as these relationships may lead to practical parallelization strategies. Valiant showed that CYK recognition can be represented as iterated boolean matrix multiplication [177].[2] In theory, this transformation opens the door to algorithms with asymptotic complexity better than $O(n^3)$. However, the matrix multiplication algorithms with sub-cubic complexity involve impractical constant factors [170, 50], and the matrix-element multiplication operator in Valiant's formalism is quite expensive to compute on a sizable grammar. To our knowledge, no implementation of Valiant's algorithm has been reported on a high-accuracy grammar.

Much later, Lee [110] proved the converse — that boolean matrix multiplication can

[2] Church demonstrated finite-state operations using a similar matrix representation, and referenced Valiant's earlier work, but did not extend his own implementation to context-free grammars. [40]

	NP,VP	DT,NP	DT,NN	NN,NN	NN,@NP	NN,RB	VB,RB	⋯
S	1	-	-	-	-	-	-	
NP	-	¼	¼	⅙	⅙	⅙	-	
@NP	-		-	1	-	-	-	
VP	-		-	-	-	-	½	
⋯				⋯				

Figure 3.1: A matrix representation of a portion of the simple PCFG from Table 2.3. Binary parents are represented as rows, and child pairs as columns. The matrix is $|V| \times |V|^2$ (where $|V|$ is the number of non-terminals in the PCFG), but is generally very sparse.

be transformed into context-free parsing and that a practical parser asymptotically faster than $O(n^3)$ would also be practical for fast boolean matrix multiplication (BMM). BMM has been explored even more extensively than CYK, and fast BMM — i.e., much faster than $O(n^3)$ — is generally believed to be impossible.

The combination of these results leads us to believe that parsing exhaustively in less than cubic time is probably not practical, and we will proceed on that assumption. We find the relationship with matrix multiplication encouraging, but the theoretical literature does not provide us with a path to efficiency gains or to practical parallelization. For the moment, we will concern ourselves with a smaller problem, that of performing the grammar intersection within a cell while populating a CYK chart in the normal fashion.

3.2 Matrix Grammar Encoding

In this section we present a matrix encoding that can encode very large grammars to maximize inference efficiency. This matrix grammar encoding is very compact and cache-efficient, improving serial performance on common CPU architectures (as demonstrated in Section 3.3), and it enables the refactoring of the CYK algorithm presented in Section 3.5.

We begin with a binarized PCFG, as described in Section 2.2. We will consider binary rules (those with 2 children) and unary rules (those with only a single child) separately, and we denote those two sets P_b and P_u, respectively. We will focus most of our attention

on P_b, as it is generally much larger, and inference time is disproportionately concentrated on binary processing.

We encode P_b in matrix form, where the rows of the matrix $1..|V|$ represent a production's left-hand-side non-terminal, and the columns represent a tuple of all possible right-hand-side non-terminals (pairs in a binarized grammar). This forms a matrix of $|V|$ rows and $|V|^2$ columns. Figure 3.1 shows a simple grammar represented in this format.[3]

In theory, this matrix could contain $|V|^3$ entries, but most grammars of interest are incredibly sparse, populating only a small fraction of the possible matrix cells. For example, the Berkeley parser's default latent-variable grammar [142] defines 1134 non-terminals, so a fully populated binary rule matrix would contain 1.49 billion rules, but the grammar only populates 1.73 million. The Markov-0 grammar is comparably sparse (4240 of 970k cells populated), and the other grammars we consider, with their larger non-terminal space, are even sparser. Dense matrices map straightforwardly into memory structures, and are generally stored and accessed in linear order, but storing and accessing a sparse matrix efficiently can be more challenging. These representation choices greatly affect overall parsing efficiency, so we will briefly describe our implementation. We use a *compressed sparse column* (CSC) sparse matrix representation [173]. CSC stores a matrix three one-dimensional arrays — the first holding all non-zero matrix entries, a second (parallel to the first) the row indices of each entry, and the third offsets (into the first two) the first entry of each column.

The sparse matrix representation stores productions very compactly — the storage proportional to the number of non-zero entries, and we store the binary rules of the Berkeley grammar in approximately 10.5 MB of memory. However, the column offset array is still $O(|V|^2)$; since many possible left-child / right-child combinations are never observed, we can achieve further gains by storing those column offsets more compactly. We map left- and right-child non-terminal pairs to matrix columns with a perfect hash of the form $h(l, r) \rightarrow [m]$. Since a perfect hash function ensures no collisions, this function is reversible ($h^{-1}([m]) \rightarrow (l, r)$), allowing recovery of the left and right children from their

[3] We store unary rules in a similar, but smaller, matrix.

hashed representation. We construct $|V|$ separate hash functions, mapping $h_l(r) \rightarrow [m_l]$. We store the data structures for these functions adjacently in memory. Thus, iterating over the entire range of $\mathbf{c}$ accesses memory in roughly linear order, with the concomitant advantages thereof, as described in Section 2.11. Global optimization of a perfect hash is an NP-complete problem, so we instead use a displacement heuristic [171] to pack the hash efficiently, achieving 50-80% occupancy for most grammars. We elected not to use a minimal perfect hash, since the decrease in storage space comes at the cost of additional instructions and increased memory access. The column offsets and grammar rules are both stored contiguously in memory and in order of access, so the grammar intersection operation is very cache-efficient.

3.3 Cache Effects

We examined the effects of cache on parsing efficiency by comparing implementations of identical algorithms. We chose the **Grammar Loop** and **Left Child Loop** algorithms as representative examples of the algorithms described in Section 3.4. As demonstrated in Table 3.1, the default implementations suffer greatly from cache misses, and their throughput is quite slow. The matrix grammar encoding from Section 3.2 delivers speedups of 8×, emphasizing the importance of memory representation and implementation details for efficient inference.[4] The critical implementation differences were: 1) representing the grammar as parallel arrays of primitives (thus in a compact and contiguous block of memory) in place of Java objects, which are more memory-intensive and may be located non-contiguously throughout the heap; and 2) sorting those arrays such that rules involving the same left (or right) children are located adjacent to one another in memory.

As shown in Table 3.1, inference with the matrix-encoded grammar requires many fewer requests to lower-level memory caches, resulting in much greater parse throughput. The lower memory footprint generally also results in improved cache hit rates, and even the small reduction in L3 hit-rate for the grammar-loop algorithm is overcome by the

[4] Unlike most other experiments reported in this thesis, these trials were performed on an Intel Core i7 CPU (Model 3520M) under Windows 7 using Java 1.7.0_21.

Algorithm / Encoding	L2 Cache Req	L2 Cache Hits	L3 Cache Req	L3 Cache Hits	Speed (w/s)
Grammar-loop (Alg. 3.1)					
Baseline	7653m	35.6%	4924m	27.3%	.25
Matrix	1866m	80.7%	352m	19.6%	1.2
Left-child loop (Alg. 3.2)					
Baseline	6872m	57.8%	2907m	16.8%	.50
Matrix	584m	71.1%	189m	57.5%	8.4

Table 3.1: L2 and L3 cache accesses and hit rates for matrix-encoded grammar vs. BUBS baseline grammar representation. Evaluated on an Intel Core i7 (3520M) CPU; inference trials performed on the first 25 sentences of WSJ section 22 using the standard Berkeley 6-cycle grammar.

nearly 14× reduction in L3 accesses. In short, the matrix grammar encoding provides a large increase in speed without any reduction in accuracy.

In light of these large improvements, we performed all subsequent trials in this thesis using variations of this grammar encoding, differing only in sort order (for different binarizations) and matrix storage format (compressed-sparse-column in most cases, but compressed-sparse-row for inner-loop implementations which require a parent-oriented rule ordering). This efficient representation and grammar access not only improves parsing throughput in subsequent trials, but provides an efficient and stable baseline for learning efficient grammars in chapters 4 and 5.

3.4 Grammar Intersection Methods

We begin by pointing to Algorithm 2.1, the standard CYK algorithm. The *argmax* on lines 7–8 intersects the set of observed child categories spanning adjacent substrings (stored in chart cells) with the set of rule productions found in the grammar. Algorithms 3.1 and 3.2 show two possible grammar intersection methods, one which loops over productions in the grammar (Alg. 3.1) and one which loops over left-children prior to looking for grammar

Algorithm 3.1 Grammar intersection via full grammar loop (backpointer storage omitted). $\alpha(b, e)$ represents the population of the cell spanning words b to e.

$\alpha(b, e) \leftarrow 0$
for $m = b + 1$ to $e - 1$ **do** ▷ Loop over midpoints
  for $A_i \rightarrow A_j A_k \in P$ **do** ▷ Loop over grammar rules
    $x \leftarrow \mathrm{P}(A_i \rightarrow A_j A_k)\, \alpha_j(b, m-1)\, \alpha_k(m, e)$
    if $x > \alpha_i(b, e)$ **then**
      $\alpha_i(b, e) \leftarrow x$

Algorithm 3.2 Grammar intersection via left child grammar loop

$\alpha(b, e) \leftarrow 0$
for $m = b + 1$ to $e - 1$ **do** ▷ Loop over midpoints
  for $j \in \alpha(b, m-1)$ **do** ▷ Observed left children
    for $A_i \rightarrow A_j A_k \in P$ **do** ▷ Grammar rules matching j
      $x \leftarrow \mathrm{P}(A_i \rightarrow A_j A_k)\, \alpha_j(b, m-1)\, \alpha_k(m, e)$
      if $x > \alpha_i(b, e)$ **then**
        $\alpha_i(b, e) \leftarrow x$

productions (Alg. 3.2).

Conventional wisdom holds that parsing time depends primarily on the number of constituents populated in the chart. Song et al. [169] began with this assumption, leading them to investigate grammar transformations which reduce total chart population. They explored the efficiency effects of several grammar intersection methods and found Algorithm 3.2 to be superior for right-factored grammars. They then proposed optimized binarizations to reduce chart population. Unfortunately, the efficiency gains from their binarization approaches are rather ambiguous. As demonstrated in Section 3.3, cache effects can be very large, and we suspect that they may have obscured the expected gains. In this section, we revisit the methods they described using the cache-efficient grammar encoding, and we explore their effects on a wider range of grammars. We find that a number of factors play a role in parsing efficiency, some of which may dwarf the effects of chart population. We will consider the following grammar-intersection methods (and implementation variants of some):

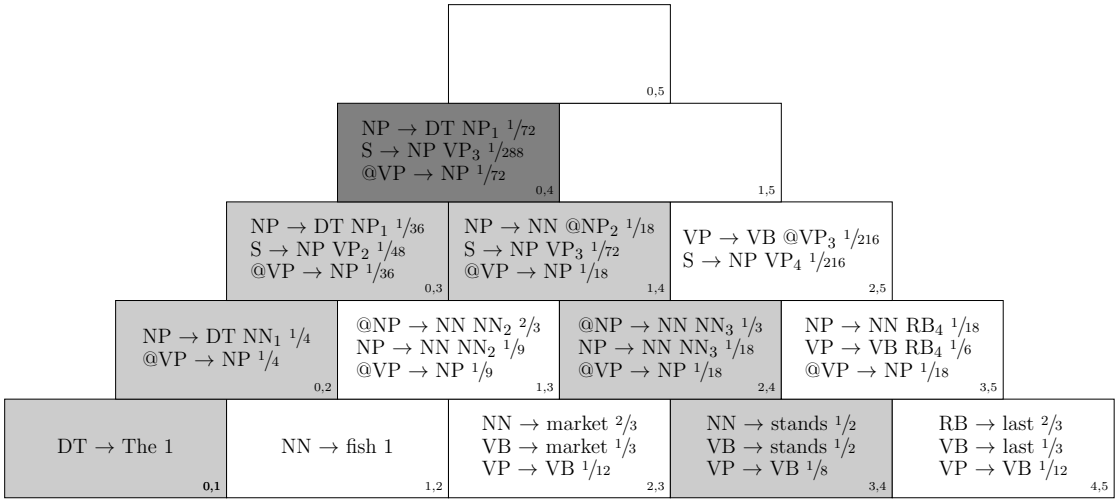


Figure 3.2: A partially-populated CYK chart. The cell spanning the first 4 words is marked in dark grey, and the potential subconstituents of that cell in lighter grey.

Grammar Loop. This approach, described in the previous chapter, and detailed in Algorithm 3.1, is the most intuitive approach to analyze. In short, we iterate through the entire grammar, looking for rules A → B C for which we find B in the left child cell and C in the right child cell. For example, when populating the highlighted cell (0,4) in Figure 3.2, we begin with the first pair of child cells—(0,1) (1,4)—and iterate through all binary productions looking for possible parents of the child pairs DT,NP; DT,S and DT,@VP. We repeat the process with each pair of child cells—(0,2), (2,4) and (0,3),(3,4) in this case. We must iterate through the grammar at each midpoint, and for each grammar rule we must examine both child cells.

Early discussions of grammar intersection methods, such as that in Goodman [75], found the grammar loop to be inefficient, and Song et al. [169] refer to it as 'generally not preferred in practice'. We, however, are not so quick to discard the grammar loop, for two reasons: 1) Iterating over P_b directly, instead of probing it repeatedly, presents a more linear memory access pattern to the processor, resulting in fewer cache misses. Previous discussions experimented on systems with flatter memory hierarchies (fewer layers of cache and smaller cache miss penalties), so a reexamination on modern hardware is warranted; and 2) Looping over the entire grammar parallelizes across a large number of processor

cores (most simply, by assigning to one thread all rules for a given parent). We might be willing to accept an increase in the total work required if we are able to exploit under-utilized processing units (c.f. the discussion of Canny et al. [27] in Section 3.8.1). We believe that attribute alone justifies consideration of the method.

Left-child loop (Algorithm 3.2) For each non-terminal B in the left child cell, iterate through all rules A → B C looking for C in the right child cell and populate A in the chart. The iteration is similar to that in Algorithm 3.1, but in this case, when processing the first pair of child cells, we need only consider productions with DT as the left child, and those with NP or @VP when processing the second midpoint. Presuming the productions are represented in memory such that rules with the same left child are adjacent to one another, this approach should be quite cache-efficient. The iteration over those rules accesses memory in a linear fashion, so a CPU's cache prediction logic is able to pre-fetch successive rules and avoid cache misses. We note that the left-child loop is more difficult to parallelize efficiently, since we must prevent simultaneous updates of the same parent by different threads.

Right-child loop. Analogous to the left-child-loop approach, differing only in direction. For each C in the right child cell, iterate through all rules A → B C and look for B in the left child cell. All of the same comments apply to this method.

Cartesian-product loop. For all B,C combinations in left and right cells, look for grammar rules A → B C. We can further subdivide this method by the possible techniques used to look up grammar rules. A linear search ($O(P_b)$) is generally not practical; hashing or binary search are possible choices, but their memory-access patterns can result in frequent cache stalls [107].

Section 3.6 presents experimental trials comparing each of these methods and the matrix-vector method presented in the next section.

3.5 Matrix-Vector Grammar Intersection

We now present an intersection method, based on the grammar encoding from Section 3.2, which decouples midpoint iteration from grammar intersection and can reduce the cell population cost considerably. This approach has two beneficial properties: 1) the number of expensive grammar intersection operations is reduced from $O(n^3)$ to $O(n^2)$; and 2) since grammar intersection is reduced to a set of matrix operations, the resulting algorithm is amenable to fine-grained parallelization. We begin with an informal description, with midpoints omitted for clarity. In Section 3.5.2, we will formalize the method as an application of a lexicographic semiring.

3.5.1 SpMV Intersection

We represent the population of each chart cell $\boldsymbol{\alpha}$ as a vector in $\mathbb{R}^{|V|}$. Each dimension of this vector represents the (log) probability of a non-terminal in that cell. To perform the *argmax*, we populate a temporary vector $\mathbf{c}$ of $|V|^2$ dimensions with the cartesian product of all observed non-terminals from the left and right child cells *over all midpoints*. That is, each dimension of this vector represents an ordered pair of non-terminals from the grammar, and its length (score) is the product of the inside probabilities of the respective children. For any child pairs which occur at multiple midpoints, we need record only the most probable — i.e., the child pair which might participate in the Viterbi 1-best solution.

Following the example grammar intersection from Section 3.4, consider populating the highlighted cell (0,4) in Figure 3.2. The first midpoint (m=1) adds (DT,NP), (DT,S), and (DT,@VP) to $\mathbf{c}$; the second midpoint (m=2) will add (NP,@NP), (NP,@VP), (@VP,@NP), and so on. If we observe the same pair at multiple midpoints, we retain only the maximum score.

Given a matrix-encoded grammar, G, and the child-cell vector, $\mathbf{c}$, we simply multiply G by $\mathbf{c}$ to produce $\boldsymbol{\alpha}$, the population of the target cell. In Viterbi search, we perform this operation in the $\langle T, T \rangle$ lexicographic semiring, thus computing the maximum probability instead of the sum (described more fully in Section 3.5.2). Figure 3.3 demonstrates this Sparse-Matrix $\times$ Vector multiplication (SpMV). The SpMV is the only portion of our

Algorithm 3.3 Grammar intersection via Sparse Matrix × Vector Multiplication. $h(l, r)$ maps $l, r \in V$ to an index of $\mathbf{c}$ (Backpointer ζ omitted for clarity).

$\mathbf{c} \leftarrow 0$
for $m = b + 1$ to $e - 1$ **do**
  for $j = 1$ to $|V|$ **do**
    for $k = 1$ to $|V|$ **do**
      $i \leftarrow h(\alpha_j(b, m-1),\ \alpha_k(m, e))$
      if $\alpha_j(b, m-1)\, \alpha_k(m, e) > \mathbf{c}_i$ **then**
        $\mathbf{c}_i \leftarrow \alpha_j(b, m-1)\, \alpha_k(m, e)$
$\alpha(b, e) \leftarrow G \cdot \mathbf{c}$

algorithm which must access the grammar. We perform that operation once per cell, rather than once per midpoint, reducing the number of expensive grammar operations from $O(n^3)$ to $O(n^2)$ (the cost of constructing c is still $O(n^3)$, so the asymptotic complexity remains cubic, but the practical runtime is reduced).

In Section 3.1 we discussed the formalism of Valiant [177], which transforms parsing into boolean matrix multiplication, and Lee [110], which inverts that transformation. We note the similarity to those approaches, but that similarity is only superficial; Valient's algorithm populates an upper-triangular matrix, the elements of which are equivalent to CYK chart cells. Each matrix element is a subset of V, the observed population of the analogous chart cell. The matrix is populated by a transitive closure operation, which takes the place of the CYK algorithm. Our matrix operation, on the other hand, is concerned with the population of individual chart cells, the operation accomplished by Valient's $*$ operator.

Decoupling the midpoint iteration from grammar intersection is not contingent on our matrix-vector encoding. The optimization in Graham et al. [77] also refactors the CYK algorithm to result in $O(n^2)$ grammar intersection operations by changing the dynamic programming to iterate through right (or left) child cells and build new (parent) categories in multiple chart cells at once. Similarly, the grammar-loop intersection of Algorithm 3.1

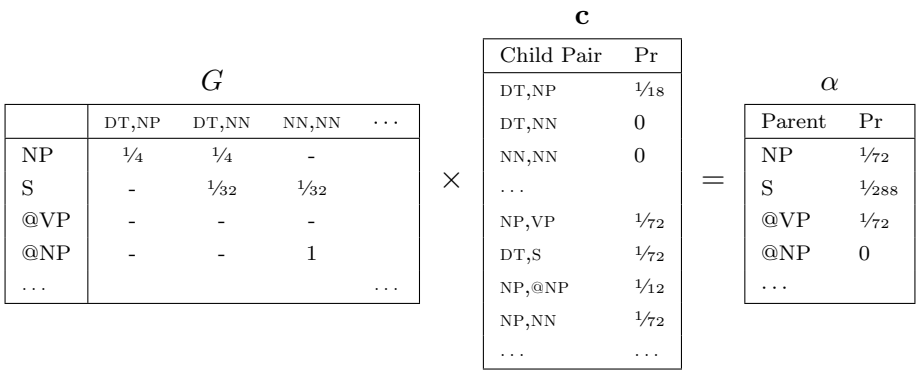


Figure 3.3: Example matrix-vector multiplication for cell 0,4 in Figure 3.2. The grammar G encodes binary rules as a $|V|\times|V|^2$ matrix, with rows representing parents and columns representing child pairs. The vector $\mathbf{c}$ contains non-terminal child pairs observed across all possible midpoints. The matrix-vector product of $G \times \mathbf{c}$ produces the target cell population, α. Factored categories are prefixed with '@', and backpointers are omitted for clarity.

could be modified to first maximize over all midpoints, then iterate over grammar productions as is done in Algorithm 3.3. However, neither variation lends itself to straightforward parallelization, and the required synchronization would severely impact parallel efficiency.

In contrast, the cartesian product and matrix-vector operations of our SpMV method parallelize easily across many cores. We partition the vector V into subvectors, one for each thread. Each thread iterates over its own subvector in the left child cell and combines with all entries in the right child cell, populating an entry in $\mathbf{c}$ for each observed child pair. $\mathbf{c}$ is represented as independent segments safe for lock-free mutation by independent threads (using the grammar's segmented hash system, as described in Section 3.2).

To perform the matrix-vector operation in parallel, we retain the same subvectors and G. Each thread t multiplies its subvector $G_t \cdot \mathbf{c_t}$, producing a vector $\alpha_\mathbf{t}$. We then merge the $\alpha_\mathbf{t}$ vectors into the final α. Since $|V| << |\mathbf{c}|$, this final merge is relatively inexpensive.

3.5.2 Lexicographic Semiring

We now present Algorithm 3.3 more formally as an application of a lexicographic semiring [73]. We frequently apply the tropical and log semirings to NLP tasks, but alternate semirings can provide a convenient and efficient representation of complex algorithms.

For instance, Roark et al. [158] recently applied lexicographic semirings to language-model encoding. We will summarize the formalism of the lexicographic semiring, following their notational conventions, and refer the interested reader to their detailed discussion.

A semiring is a ring, possibly lacking negation, defining two operations $\oplus$ and $\otimes$ and their respective identity elements $\bar{0}$ and $\bar{1}$ [103]. One common example in speech and language applications is the *tropical semiring* $(\mathbb{R} \cup \{\infty\}, min, +, \infty, 0)$. *min* is the $\oplus$ operation, with identity ∞, and $+$ is the $\otimes$, with identity 0. This definition is often used for Viterbi search, using negative log probabilities as costs.

A lexicographic semiring is defined over tuples of weights $\langle w_1, w_2 \dots w_n \rangle$, with the condition that the tuples can be ordered first by w_1, then by w_2, and so on (so named because of the similarity to lexicographic string comparison). We use the $\langle T, T \rangle$ semiring, defined as a pair of tropical weights:

$$\langle w_1, w_2 \rangle \oplus \langle w_3, w_4 \rangle = \begin{cases} \langle w_1, w_2 \rangle & \text{if } w_1 < w_3 \text{ or } (w_1 = w_3 \text{ and } w_2 < w_4) \\ \\ \langle w_3, w_4 \rangle & \text{otherwise} \end{cases}$$

$$\langle w_1, w_2 \rangle \otimes \langle w_3, w_4 \rangle = \langle w_1 + w_3, w_2 + w_4 \rangle$$

In our application, w_1 encodes the negative log probability of a production in G or of an observed non-terminal in the chart. w_2 encodes the midpoint of the maximum-probability analysis.[5] To perform grammar intersection using this semiring, we encode the grammar matrix as described in Section 3.2, and include 0 as w_2 for each grammar entry (since this weight is constant, it need not be encoded in the grammar representation).

We populate a vector of tuples $\mathbf{c}_i$ for each possible midpoint of the cell, and $\mathbf{c} = \mathbf{c_1} \oplus \mathbf{c_2} \oplus \dots \mathbf{c_{span}}$. $\oplus$ compares with *min*, so the entries in $\mathbf{c}$ will be from the maximum probability midpoints and the first midpoint will 'win' in the case of a tie.

When we multiply $G \cdot \mathbf{c}$ in the $\langle T, T \rangle$ semiring, we use the $\otimes$ multiplication operator on each individual element, and the $\oplus$ addition operator for the sum. Since w_2 is 0 for all entries in G, the midpoints are simply carried over from $\mathbf{c}$, and $G \cdot \mathbf{c}$ is the minimum-cost

[5] Since w_2 represents a midpoint, we could alter the definition to specify that $w_2 \in \mathbb{N}$, but the standard tropical semiring is adequate and slightly simpler.

path to each observed non-terminal.

SpMV implementations often represent the vector densely in memory (even if it is sparsely populated). We can then iterate through the populated matrix entries, accessing the (dense) vector as appropriate. The converse approach (dense matrix and sparse vector) is also possible, and a useful option if the input vector is much sparser than the matrix. In either case, the memory access pattern is highly irregular, and frequent cache stalls are likely [76]. The hash mechanism described in Section 3.2 is one approach to improving memory locality, but many other SpMV optimizations have been explored in the high-performance computing literature. In some cases, extracting dense blocks from a sparse matrix can improve cache efficiency [87, 1]. For some matrices, other transformations improve performance considerably [12]. Although some of these approaches are implemented in standard libraries, most operate in the real semiring; to our knowledge, none support the $\langle T, T \rangle$ semiring. Our implementation is intended to be grammar-agnostic, and we present experimental trials with a variety of grammars. The performance benefits of the various matrix transformations vary greatly with the structure of the specific matrix. Thus, we leave a full exploration of SpMV optimizations to future work, focused on optimizing a more limited range of grammars. Our expectation is that the benefit of standard SpMV optimizations would be greatest on grammars more densely populated than those we explore in this thesis.

3.6 Grammar Intersection Evaluation

We compare exhaustive and pruned parsing efficiency with several other competitive parsing implementations. The BUBS parser framework is grammar agnostic, permitting experiments on grammars of various sizes, and it implements both exhaustive inference and various pruning approches, as described in Section 2.12 and in Appendix A. For exhaustive parsing, we use BUBS implementation of Algorithms 3.1 and 3.2 and Mark Johnson's highly optimized C implementation, `lncky` [91] as baselines; for pruned inference, we compare with the Charniak parser [32], the Berkeley parser [142], and complete-closure pruning as implemented in BUBS. The Charniak parser is written in C and parses with a lexicalized grammar. The BUBS and Berkeley parsers are implemented in Java and parse with a latent-variable grammar.

We performed all trials on a 12-core Linux machine (2 $\times$ Intel® Xeon X5650 CPUs). Each core can execute 2 simultaneous threads, for a total of 24 concurrent threads. For

	M-0	M-2	Parent	LV
Parsing accuracy (F_1)	61.1	72.3	78.1	89.1
Non-terminal vocabulary (V)	99	2916	6712	1133
Binary productions (P_b)	4240	13,210	24,542	1.7m
Unary productions (P_u)	236	236	736	115k
Lexical productions (P_{lex})	52k	52k	52k	2.4m
	Speed (w/s)			
Johnson (2006)	245.0	**106.3**	**68.3**	.7
Grammar loop (Alg. 3.1)	278.2	51.0	24.8	0.9
Left-child loop (Alg. 3.2)	418.9	95.3	49.4	5.7
Cartesian Product				
Binary search	140.7	2.7	0.6	1.8
Binary search right children	159.0	38.2	4.7	2.0
Hash lookup	240.3	2.5	0.7	2.5
Hash lookup in right children	262.9	3.6	0.8	2.3
SpMV (Alg. 3.3)	**530.0**	33.6	7.6	**10.2**

Table 3.2: Grammar attributes and exhaustive Viterbi parse speeds over WSJ Section 22 for grammars using a variety of state-splitting methods; specifically, Markov-order-0, Markov-order-2, Markov-order-2 with parent annotations, and the Berkeley 6-cycle latent-variable grammar. All parsers produce the same maximum-likelihood parse trees (modulo minor differences in tie-breaking strategies).

the parsers implemented in Java, we used the Oracle 1.7.0_17 Virtual Machine.

3.6.1 Exhaustive Serial Search

Table 3.2 presents exhaustive search results with four grammars, each induced from the Penn Treebank Sections 2-21 [121]. The Markov-order-0 and Markov-order-2 grammars were markovized as described in Manning and Schuetze [119]. The parent-annotated grammar further splits the states of the Markov-order-2 grammar by annotating each

non-terminal with its parent category, as described in Johnson [89]. This expands the non-terminal vocabulary greatly, but the ruleset somewhat less so. The Latent-variable grammar [142] used here is one trained and selected by Slav Petrov, using the method described in Section 2.9. Its vocabulary is relatively small (particularly in comparison with the parent-annotated grammar), but the ruleset is quite large. We previously presented similar trials on both right- and left-binarized grammars, and found that the efficiency differences between the two depended primarily on implementation biases in grammar-intersection [58]. For example, left-child loop intersection (Algorithm 3.2) benefits from the smaller number of left-side children in a right-binarized grammar, and the analogous right-child loop method is appropriate for a left-binarized grammar. Since the efficiency impact of binarization direction is quite small, we limited these trials to right-binarized grammars, and biased all grammar-intersection implementations appropriately.

The comparison in Table 3.2 between SpMV and other approaches highlights the differences in grammar attributes. With the Markov-order-2 and Parent-annotated grammars, which have large vocabularies and relatively small rulesets, Johnson's C implementation and several other grammar-intersection methods outperform SpMV, but for the grammars with larger rulesets relative to their nonterminal space, SpMV provides a considerable speedup — a gain of 79% over Algorithm 3.2. Note that this improvement is in addition to the large gain from the matrix-encoded grammar, as reported in Table 3.1. For the remainder of this thesis, we are primarily interested in high-accuracy grammars, particularly for non-exact inference, so we focus the remaining empirical trials in this chapter on the Berkeley latent-variable grammar.

3.6.2 Pruned Serial Search

We now proceed to examine pruned inference with our SpMV method. We take as our primary baseline, the grammar-intersection methods and complete-closure pruning implemented in BUBS.[6] Our grammar intersection method works well with complete-closure and with adaptive beam search, but the serial initialization for adaptive beam search is considerably more expensive, so we chose complete-closure for these trials. Table 3.3 shows

[6] All experimental procedures follow the approach described in Section 2.13. Note that the trials in this chapter are limited to a fixed set of preexisting grammars (those listed in Table 3.2); we will explore a wider range of grammars in subsequent chapters.

	F_1	Words/sec
Charniak (2000)	90.3	40.1
Berkeley (2006)	90.4	107.9
Complete Closure		
Left-child loop (Alg. 3.2)	89.3	73.8
Cartesian-product hash	89.3	36.3
SpMV (Alg. 3.3)	89.3	97.4
Complete Closure and Beam Search		
Left-child loop (Alg. 3.2)	89.2	926.5
Cartesian-product hash	89.2	1313.2
SpMV (Alg. 3.3)	89.2	895.9

Table 3.3: Pruned parsing speeds, comparing several grammar intersection approaches and pruning methods. Evaluated using the Berkeley latent-variable grammar on WSJ Section 22. The Charniak implementation uses an agenda search, and the Berkeley parser coarse-to-fine pruning. Beam search trials used a lexical prioritization model [19] and a beam width of 20.

a speedup of over 2× vs. the baseline implementation, and an even greater advantage vs. other competitive parsers.

The Charniak and Adaptive Beam pruning systems both have tunable parameters, controlling their accuracy vs. efficiency operating point. Figures 3.4 and 3.5 shows empirical results over a range of those tuning parameters for those two implementations and for our approach. For a given parameterization, the search space explored by our approach is identical to that explored by Bodenstab et al., (modulo minor differences in unary processing), so the efficiencies achieved are directly comparable. We find consistently improved speed across all pruning thresholds.

3.7 Parallelism

Since smooth parallelization is one of the benefits of the algorithm presented in Section 3.5, we begin with background on some of the barriers to efficient parallelism. The overhead of parallelism takes many forms.[7] The operating system consumes processor cycles in thread

[7] We are concerned primarily with parallelism within a single machine. Cluster-level parallelism incurs network latency, shared filesystem, and other forms of overhead that do not concern us here.

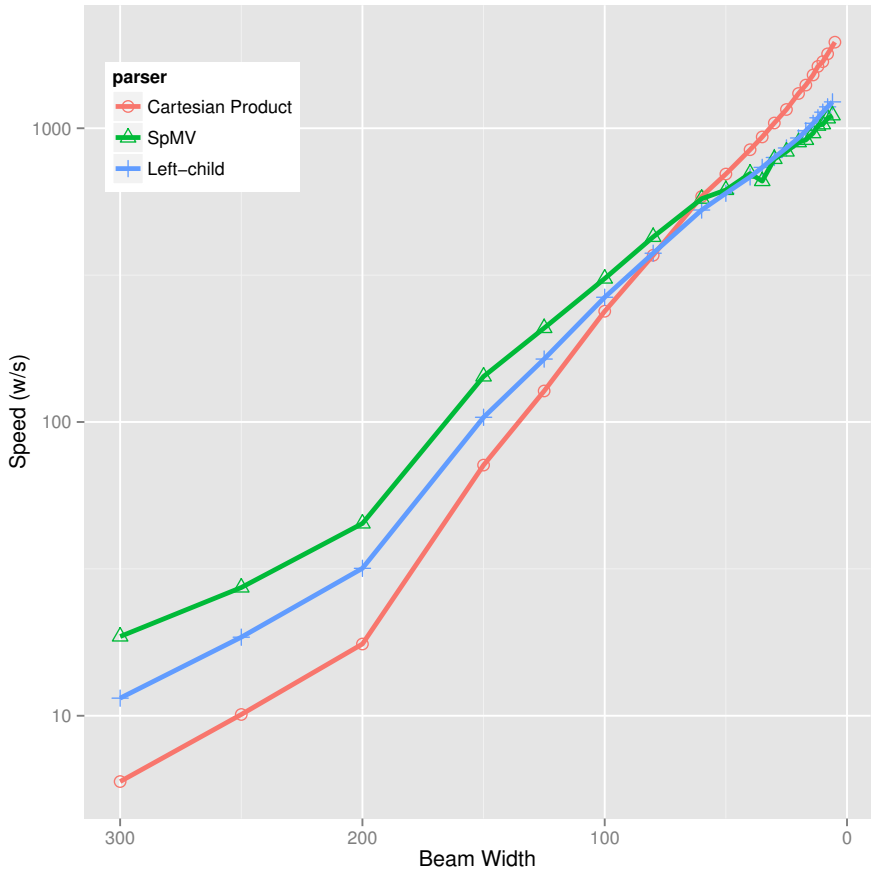


Figure 3.4: Speed vs. beam width, evaluated using complete-closure and a lexical prioritization model [19]. At a beam width of 300, SpMV is 62% faster than left-child-loop and 3.1× faster than cartesian-product intersection. As beam width is reduced, however, the benefit of SpMV lessens, and at very small beam widths, the cartesian-product intersection is considerably superior.

scheduling; coordination and synchronization of concurrent tasks can leave processors idle; and (more importantly to memory-bound applications such as parsing) context switching between threads often requires flushing the CPU cache, resulting in more memory contention and stalls. Further, some multi-core architectures share L2 or L3 caches between CPU cores, and nearly all share bandwidth to memory (the 'front-side bus', or FSB). Parallel execution threads may stall while competing for those resources. Thus, parallelism can introduce considerable hardware overhead, even if OS- and task-level overhead are minimal, but shared data structures can reduce this impact.

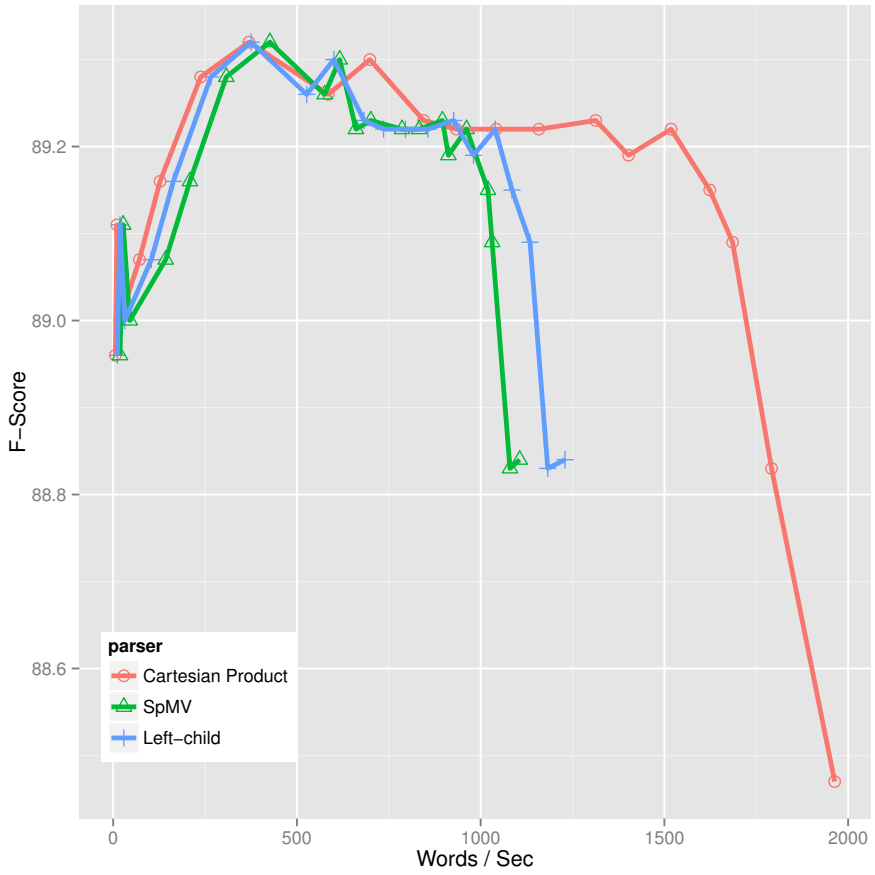


Figure 3.5: F_1 vs. speed, evaluated using complete-closure and a lexical prioritization model [19]. Accuracy of all three methods begins to drop off with beam widths below 10. Note: if beam width is reduced further, speed actually *decreases*, as parse failures force more reprocessing.

3.7.1 Parallel Parsing of Deterministic Languages

Parsing methods have been used for decades on deterministic languages, including programming languages and machine-readable data formats. Recent interest in high-volume data processing and the development of multicore architectures have prompted research in parallel parsing, particularly of large XML documents. Sequential parsing is sufficient for most small XML documents, but parallel methods are of interest for documents of hundreds of kilobytes or more. Parallelization methods that depend on unambiguous grammars do not apply straightforwardly to statistical parsing of ambiguous languages. However, we do find analogous mechanisms useful for linguistic parsing. In this section,

we will briefly summarize some methods of parallelizing deterministic parsing algorithms, and consider analogous applications in stochastic parsing.

Most parallel XML-parsing methods divide the document into subsegments using a simple serial mechanism, and process those segments in parallel [115, 134, 111]. The initial division (called *preparsing* by some authors) uses a simplified grammar, ignoring most well-formedness constraints.

One notable exception to that paradigm is the work of Cameron et al. [24]; they use hardware Single Instruction Multiple Data (SIMD) processing units (described in detail in Section 3.8) to process XML input 16 bytes at a time. They use hardware counters to analyze the penalty for out-of-order byte access (similar to our analysis in Section 3.3). El Hassan and Ionescu [64] demonstrates XML processing using custom hardware (Field Programmable Gate Arrays, or FPGAs). They achieve remarkable throughput, although limited to relatively small XML documents by the constraints of the FPGA architecture. Although our analysis does not extend to custom hardware, we can certainly envision a related approach applied to ambiguous languages. In the case of statistical parsing, the length of the input sequence is reasonable; we anticipate that the key limitation would be the size of the grammar (a problem we consider in chapters 4 and 5).

Although the techniques from the XML-processing literature do not apply directly, we find the success in parallel deterministic parsing promising. The constraint-based pruning methods implemented by the BUBS parser (and described in Appendix A) are in some ways similar to the 'preparsing' processes, and memory-access and cache analysis is clearly applicable in both cases.

3.7.2 Parallel PCFG Parsing

A sizable literature has explored parsing in parallel, demonstrating — theoretically, at least — that context-free and mildly context-sensitive parsing can be parallelized effectively. For example, Palis and Shende [133] presented an algorithm that processes Tree Adjoining Grammars in $O(log^2(n))$ time; however, their approach requires a machine with $O(n^6)$ processors, and effective communication between them; similarly, the approach of Rytter [160] requires polynomially many processing units. Subsequent decades have seen great advances in hardware technology, but that level of parallelism remains impractical (even using modern GPU hardware, which we will discuss in Section 3.8).

Unfortunately, the overhead of parallelizing can be considerable. For example, Chytil et al. [41] presented a parallel recognition algorithm for unambiguous context-free languages,

but the $O(n^3)$ complexity of their approach is considerably worse than the known $O(n^2)$ sequential solutions. Ninomiya et al. [130] explored cell-level parallelization on a 256-processor machine. Their method incurred an overhead of 6–10× vs. their baseline serial algorithm (depending on sentence length). That is, their parallel algorithm ran 6–10 times slower on a single core than a simpler serial implementation. So even if their approach scaled ideally, many cores would be required to match their serial baseline performance. In practice, their algorithm did not scale linearly and required approximately 64 CPUs to equal their baseline single-CPU performance, and the total speedup observed on 256 CPUs was only 2–4×.[8]

Further, the relationship between throughput and latency is often not linear. Given ideally efficient algorithms and hardware, there would be no tradeoff between the two — that is, we would be able to parallelize each sentence across an arbitrary number of processor cores, reducing latency and increasing throughput linearly with core-count. Unfortunately, hardware constraints and Amdahl's law ensure that we will never achieve that ideal speedup;[9] in practice, we are likely to see some tradeoff. So application developers might choose to optimize for latency, to improve user response time, even if that choice required a larger hardware investment to achieve a particular throughput. We will demonstrate interesting patterns of the tradeoff between throughput and latency with various parallelization methods, allowing consumers to tailor parsing strategies to particular application requirements.

We observe that CYK parsing can be parallelized in (at least) three distinct ways, each likely to have different advantages and disadvantages vis-à-vis the bottlenecks discussed in Section 3.7:

Sentence-level: The simplest way to parallelize parsing is to parse sentences or documents independently on separate cores. This approach is well-understood, simple to implement, and quite effective. Total throughput should scale roughly linearly with the number of cores available, at least until we reach the limits of memory bandwidth, but latency is not improved — and may actually increase. Further, the cost of scaling a cluster (or datacenter) increases at least linearly with the number of cores, and power consumption is increasingly a priority. We would prefer to improve not only parse throughput, but

[8] The inter-thread synchronization required by their algorithm probably constrains their scalability on a shared-memory CPU architecture, and renders an efficient GPU implementation unlikely.

[9] 'Amdahl's Law' is the observation that the theoretical maximum speedup of a parallel operation is limited by whatever portion of the operation which must run serially.

also latency and power consumption (sentences parsed per kWh).

Cell-level: In most forms of CYK iteration, we populate each cell separately, leading to a straightforward form of cell-level parallelism. For example, in bottom-up cell iteration order, we populate one chart row fully before proceeding to the next. The cells on each row are independent of one another, so we can process all cells of a row in parallel [129]. Unfortunately, as we move higher in the chart, there are fewer cells per row, and we must leave CPU cores idle. The highest cells in the chart are often the most densely populated (and require the most processing), an inherent limitation of this form of parallelism.[10] Additionally, filtering algorithms such as Earley [61, 182], with more dependencies between cells and alternative iteration orders, are not easily amenable to this form of parallelism.

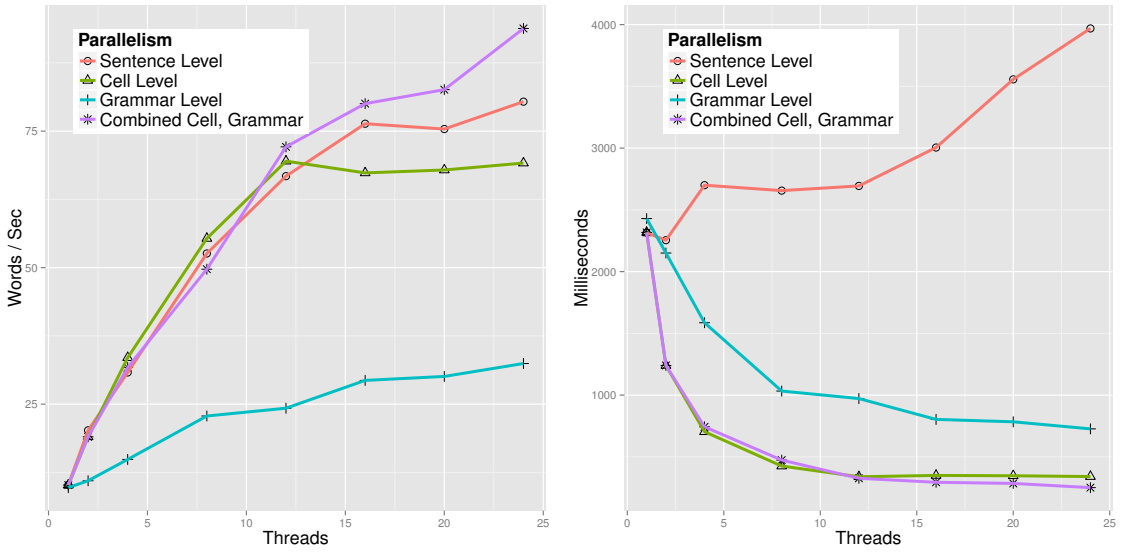
Grammar-level: Parallelization within a chart cell is more difficult to implement, but may avoid some of the weaknesses of the first two methods described. If we can fully parallelize cell population, we can make use of all available cores regardless of the cell iteration order or the current position in the chart [187].[11] Because each thread is operating on the same cell, their working sets may align more closely than in other forms of parallelism, reducing context-switch overhead. However, this method implies very fine-grained task divisions and close coordination between threads — when we split a single grammar intersection operation across many threads, each task is quite small. At this fine granularity, locking of shared data structures is impractical, so we must divide tasks such that they share immutable data (the grammar and current cell population) but do not simultaneously mutate the same target data structures (e.g., individual threads may populate separate ranges of non-terminals in the target cell, but must not attempt to populate the same range). Even with careful task division, the task management may overwhelm the potential gains.

3.7.3 Exhaustive Parallel Search

We now move to evaluating parallelization methods. Our baseline parsers could be parallelized at a sentence-level, and possibly at a cell-level, but having already established dramatic gains vs. those approaches for serial parsing, we will focus all these trials on our own SpMV algorithm — thus, the sentence-level results reported serve as a 'baseline' of sorts, albeit one already demonstrated to be a considerable improvement on standard

[10] If optimizing for throughput, those idle threads could be reassigned to subsequent sentences, but cache- and FSB-contention is likely to further increase latency.

[11] Of course, we can utilize sentence-level and cell-level parallelism as well.



(a) Throughput, in words per second.

(b) Latency, in milliseconds per sentence.

Figure 3.6: Exhaustive SpMV throughput and latency vs. thread-count.

baselines. We parallelize the SpMV implementation using the three parallelization strategies discussed in Section 3.7., and compare throughput and latency. To explore potential additive effects, we include a system combining cell-level and grammar-level parallelism, using several grammar-level threads for each cell-level thread, over the same range of total thread count. Note that some serial processing is required for each sentence (primarily initialization of the chart and extraction of the final parse tree), but these operations consume only 1.5% of the total time.

Executed with a single thread, the cell-level and grammar-level parallel implementations incur an overhead of less than 5%, which compares very favorably with the 500–900% overhead in Ninomiya et al. [130]. Figure 3.6 shows throughput and latency of each parallelization approach as thread count increases. All approaches show improved throughput with increased thread-count. Cell-level and grammar-level approaches begin to level off around 12 threads, when all physical cores are occupied; combining the two appears to benefit further from Hyper-threading, achieving throughput superior to sentence-level threading.

In Figure 3.6b, we see two interesting latency effects: 1) We expected the sentence-parallel approach to produce fairly constant latency, but instead found that latency jumped

considerably after only 2 threads, and continued to increase after 12 threads, as Hyper-threading came into use. Our (unproven) theory is that one parsing thread saturates the shared cache and/or front-side-bus on a physical die, and that additional threads on the die must compete for those shared resources. 2) Cell-level and grammar-level approaches show large decreases in latency as thread count increases, and the combination shows additive gains — an overall reduction in latency of more than 9× and an improvement of nearly 26.3% vs. cell-level threading alone. [12]

While that improvement is quite impressive, we anticipate that further gains might be possible. We found that both cell-level and grammar-level methods often leave numerous threads idle, and CPU monitoring rarely shows all cores being occupied. Our observations lead us to believe that much of the 'lost' processor time is going to the task handling and inter-thread communication; hardware threading (as described in Section 3.8) may extend the gains for those methods.

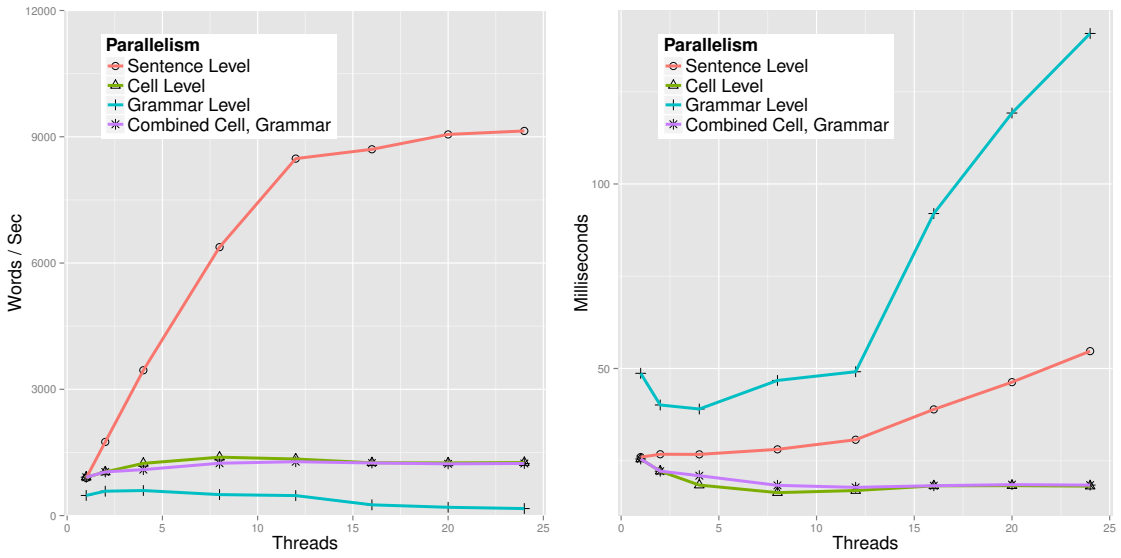
3.7.4 Pruned Parallel Search

Figure 3.7 presents similar trials for pruned parallel search (once again, all trials use SpMV grammar intersection). In this case, we find that the cell-level and combined approaches perform quite strongly — nearly optimally, in fact. In contrast to the relatively small serial portions of exhaustive search, pruned search requires some fairly expensive serial operations. The initial version of this work used an adaptive-beam model, which was much more costly to initialize. The complete-closure model used in these trials initializes considerably faster, but the total serial steps (chart and pruning initialization, and the beam-search pruning itself) account for 23% of the time, so the observed 27.7% increase in throughput and reduction in latency, although much smaller than that of sentence-level threading, is not unreasonable.[13]

For grammar-level parallelism in Figure 3.7b, however, we find a somewhat counterintuitive result: increasing the thread-count *increases* latency. This is due to the characteristics of the grammar and the severity of pruning during inference. The Berkeley grammar has a small non-terminal set ($|V| = 1134$), but a large ruleset ($|P_b| \approx 1.7$ million). When

[12] We found that varying the ratio of cell-level to grammar-level threads produced similar results, up to 4–6 grammar-level threads; we omit those combinations from the plots for clarity.

[13] We will re-incorporate an improved adaptive-beam model in the final trials presented in Chapter 7.



(a) Throughput, in words per second.

(b) Latency, in milliseconds per sentence.

Figure 3.7: Pruned search throughput and latency vs. thread-count.

Figure 3.8: Pruned search is constrained by the serial pruning initialization, so we see little benefit from parallelism beyond 2–4 threads.

performing exhaustive search, many cells are densely populated (the average cell population is 450 of 1134). When performing pruned search, the cell populations are naturally much sparser (at most 30 entries, and often fewer). Thus, the grammar-level parallel tasks are much smaller, and task management overhead overwhelms the potential gains of additional execution threads, a further motivation for hardware thread management.

3.8 SIMD

Single Instruction Multiple Data (SIMD) computation units execute the same instruction on multiple data streams in parallel. The architecture was formally classified in Flynn's taxonomy in the early 1970's [68], and the first practical implementations were the vector supercomputers of the same era. Most historical SIMD implementations use a large number of very simple processing units to accelerate matrix transformations and other inherently parallel tasks. Many multimedia processing operations are similarly parallelizable, so many recent microprocessor architectures have incorporated SIMD units specifically for that purpose.

	Speed (words/second)			
	Markov-0	**Markov-2**	**Parent Annotated**	**Latent Variable**
CPU	530.0	33.6	7.6	10.2
GPU	9.2	1.8	.42	1.2

Table 3.4: GPU SpMV parsing speeds (in w/s), using the same grammars as Table 3.2. Evaluated using an Nvidia Tesla M2090 GPU. CPU results repeated from Table 3.2.

A SIMD processor loads a large array of data from memory and executes the same operation on each element of that array. Example operations might increment each element, multiply by a fixed multiplicand, or take the natural logarithm of each.[14] SIMD processors are often applied to matrix operations, where the parallelism is explicit. Purpose-built graphics processing units (GPUs), such as those produced by Nvidia and AMD also incorporate SIMD processing, often with slight changes targeted on image manipulation.The large memory bandwidth and raw processing power of these units has motivated other applications as well (so-called 'General-purpose GPU computing', or GPGPU).

The SpMV parsing algorithm from Section 3.5 centers on a matrix-vector multiplication, and is thus amenable to SIMD processing. We implemented this algorithm in the OpenCL environment [126], and tested it on a Nvidia Tesla M2090 graphics-processing unit with 512 cores, 6 GB of RAM and 256 KB of global-memory cache. The results in Table 3.4 are disappointing, in that even massively parallel hardware underperforms the baseline CPU implementation. Effectively coding for GPU hardware is a very specialized skill-set (perhaps even an art form). We are certain that our simplistic implementation does not make full use of the available memory and computational bandwidth. We remain optimistic that engineers more skilled in GPU coding patterns could improve greatly on our proof-of-concept implementation.

3.8.1 Related Work

In a separate effort, Mark Johnson reimplemented the SpMV algorithm on CPU and GPU hardware [92]. His experimental trials focused on densely-populated grammar matrices,

[14] Note that this processing model require a sizable increase in memory bandwidth vs. a conventional processor.

as might be used for unsupervised grammar induction. He also found considerable gains on a CPU, and disappointing results on GPU hardware. Like us, he attributed that deficit to the highly specialized coding requirements for efficient GPU utilization. Yi et al. [186] and Canny et al. [27] also examined CYK inference on a GPU, and each of their implementations were more efficient. Yi et al. obtained a 25.8× speedup, albeit with a smaller latent-variable grammar and comparing to a somewhat slower baseline CPU implementation. Canny et al. parallelized at the cell level, but kept multiple sentences in flight simultaneously, allowing them to fully utilize the available memory bandwidth. They came near to the theoretical throughput bounds of their GPU hardware, obtaining over 2800 words per second under exhaustive inference.[15]

3.9 Discussion

In this chapter, we presented a matrix grammar encoding which has several beneficial properties. Access to grammar rules encoded in this manner is very cache-efficient, and it enables cell population using a Sparse Matrix × Vector grammar intersection. This reduction allows very fine-grained parallelism, and can reduce parse latency and improve throughput considerably. We found dramatic speedups on exhaustive serial parsing and a sizable improvement in pruned inference, vs. BUBS baseline encoding and other state-of-the-art parser implementations. Although results on SIMD / GPU hardware were disappointing, the combination of grammar encoding and intersection methods reduce latency as low as 17.8 ms, and produce throughput in excess of 9000 words per second on commodity SMP hardware. These results improve greatly on other common parsing implementations, and will be of great interest to end-user applications with response-time constraints and to semi-supervised model training constrained by parsing throughput.

Further, these approaches provide a very solid baseline for the approaches for training efficient grammar models we present in chapters 4 and 5. Although we do not claim that the implementation described in this chapter is fully optimal (i.e., making full use of available computational and memory resources), the large improvement in cache efficiency removes one large source of noise which otherwise would obscure any potential gains from training efficient grammar models. And an efficient and parallelizable implementation makes it practical to incorporate inference during grammar training in Chapter 5.

[15] Canny at al. predict further gains from pruning, although that claim is suspect; their approach depends on a full grammar-loop (Alg. 3.1), so it is unlikely to benefit greatly from a sparser chart population.

Chapter 4

Lexicon Simplification and Corpus Transformations

We now move from grammar-agnostic inference methods to approaches of training grammars which will enable further efficiency gains. In this chapter, we present corpus transformation methods, applied prior to training, which can reduce the size of the final grammar and increase inference speed.

4.1 Spurious State Splits

Annotated treebanks have revolutionized syntactic processing, providing the means to train accurate and generalizable statistical models. However, these models have inherent weaknesses that limit their effectiveness. Treebanks are constructed in hopes they encode general patterns, but grammars trained from raw treebanks often overfit to specific examples, encoding spurious productions and reducing performance on unseen sentences. Second, partially due to these productions, grammars learned from raw treebanks are quite large, dramatically increasing inference time.

For example, consider the non-terminal label CD. In the Penn Treebank annotation system, this preterminal (part-of-speech) is used for all cardinal numbers, including numeric values and textual representations thereof (e.g., '9.88', '91-23', 'eight', 'billion-plus', and 'mid-1970s'). Consider the rules in Table 4.1, drawn directly from a split-merge latent-variable grammar trained with the Berkeley Parser [142] on the Penn Treebank [121]. This particular grammar encodes 129,189 lexical rules for various splits of the CD (cardinal number) label, most of them for specific numbers — in fact, over half of these rules (66,096) are for tokens which occur only once in the training corpus. Clearly the majority of these rules are unlikely to help disambiguate between larger contexts in unseen

data. Further, these spurious productions increase the size of the grammar considerably, adding not only the required lexical productions, but increasing the binary ruleset as well. CD labels are usually children of NP or QP nodes, and the model is reserving probability mass in NP and QP productions specifically for the (unnecessarily) split CD labels. Thus, eliminating the spurious lexical productions may greatly reduce the size of the final grammar, yielding a more efficient model without loss in generalizability.

Parent		Terminal	Log Probability
CD_0	→	675,400,000	-13.3299269994
CD_1	→	675,400,000	-13.5714331797
CD_9	→	0.6287	-12.566730769
CD_10	→	0.6287	-12.1136975239
CD_11	→	0.6287	-12.6230114872

Table 4.1: A sample of grammar rules headed by various splits of the CD (cardinal number) non-terminal, taken from a 6-cycle latent-variable grammar.

We have focused this discussion on CD as an extreme example, but the same arguments apply to other open-class preterminals as well — for example, the proper noun label, NNP, behaves similarly, as the parent of many singleton tokens; many substates are learned, most of them unlikely to generalize beyond the training data.

In this chapter, we present two novel methods of reducing the number of spurious lexical productions in latent-variable grammars, and demonstrate an improvement of 27.2% in inference speed. In Section 4.2, we present a novel corpus transformation, based on the Berkeley Parser's class-based unknown-word handling system. We train grammars from transformed corpora, finding large reductions in grammar size with minimal degradation of parse accuracy. In Section 4.3, we derive a similar corpus transformation from automated clustering, and incorporate a discriminative cluster-assignment tagger into the inference process. This approach provides a comparable size reduction, but at a substantial cost in both speed and accuracy.

-INITC	Capitalized initial word of sentence
-KNOWNLC	Lower-case version of the word observed in grammar
-CAPS	Capitalized (but not sentence-initial)
-NUM	Contains one or more numerals
-DASH	Contains one or more hyphens
-s, -ed, -ing, -ion, -er, -est, -ly, -ity, -y, -al	Suffix features

Table 4.2: Class-based decision-tree features, in the order in which they are applied. The class of an unknown word is built up by appending each appropriate feature to the base UNK class. For example, the token '90th-story' would be assigned the unknown word class UNK-NUM-DASH-Y.

4.2 Class-based Rare Word Handling and Normalization

The Berkeley parser includes a simple but robust unknown-word system, which replaces each OOV word with a class signature [140].[1] OOV classes are assigned by a decision tree based on characteristics of the observed token. E.g., UNK-NUM for a word containing a number, UNK-NUM-DASH for one containing both numeral(s) and dashes, UNK-CAPS for a word in capital letters, and so on (the full list is in Table 4.2). Production probabilities for these classes are estimated using observation counts of rare words. In the Penn Treebank [121] training section, we observe 66 unique unknown-word classes. During inference, we replace any unknown words with the appropriate UNK class (using the same decision-tree hierarchy). In the rare case that the UNK class was also unobserved during training, we back off through the decision tree (removing suffixes from the predicted UNK class) until we find grammar productions matching the class. For example, the token '90's-era-fashion' would be assigned the class UNK-NUM-DASH-ion. If no productions are available for that class, we would back off to UNK-NUM-DASH, then to UNK-NUM, and finally to the base UNK class.[2]

[1] This signature-based system is similar to those used by Arun and Keller [5] and Crabbé and Candito [26] for French, and by Attia et al. [6] for Arabic.

[2] Although the backoff strategy of the resulting decision tree is not always linguistically ideal, we chose to retain the method unchanged, for consistency with earlier published results and the existing grammars we use as baselines.

POS	λ	F_1	$\Delta\rho$(%)	Δw/s (%)
baseline	-	88.6	-	-
CD	5	89.0	-1.9	+2.8
JJ	2	88.6	-6.6	+6.4
NN	1	88.6	-5.2	+0.6
NNP	4	88.7	-11.2	+2.3
NNPS	8	88.9	+0.3	+1.2
NNS	1	88.8	-2.1	+2.5
RB	1	88.5	-0.3	-1.0
VB	1	88.6	-0.1	+4.8
VBD	2	88.6	-0.4	+0.6
VBG	1	88.9	+0.1	-0.5
VBN	1	88.7	-0.8	+6.6
VBP	1	88.5	-0.2	+5.2
VBZ	1	88.9	+0.3	+3.3

Table 4.3: Thresholds which maximize F_1 for each preterminal, and the resulting change in grammar size ($\Delta\rho$) and speed (Δw/s). Evaluated on WSJ section 22.

4.2.1 Corpus Transforms

To eliminate some of the spurious productions described in Section 4.1, we use this class-based model to produce transformed training corpora. We replace rare open-class words — those observed in the training corpus less frequently than a threshold λ — with their class signature. A grammar trained on this transformed corpus learns production probabilities for the classes, without the spurious productions observed in the raw treebank.

We consider any preterminals which parents more than 500 unique lexical tokens to be open-class (namely, CD, JJ, NN, NNP, NNPS, NNS, RB, VB, VBD, VBG, VBN, VBP, and VBZ).[3] At $\lambda = 1$, we replace all singletons; as we tune λ upward, we replace more frequent words as well.[4]

[3] We chose an arbitrary threshold of 500, but the distribution of counts of unique lexical children is very much bimodal — closed-class preterminals have very few parents and open-class preterminals have many — so the open-class set is relatively stable across a wide range of possible thresholds.

[4] With the standard WSJ training corpus of 950k words, $\lambda = 5$ is equivalent to pruning lexical productions where $P(A \to w) < e^{-12.15}$.

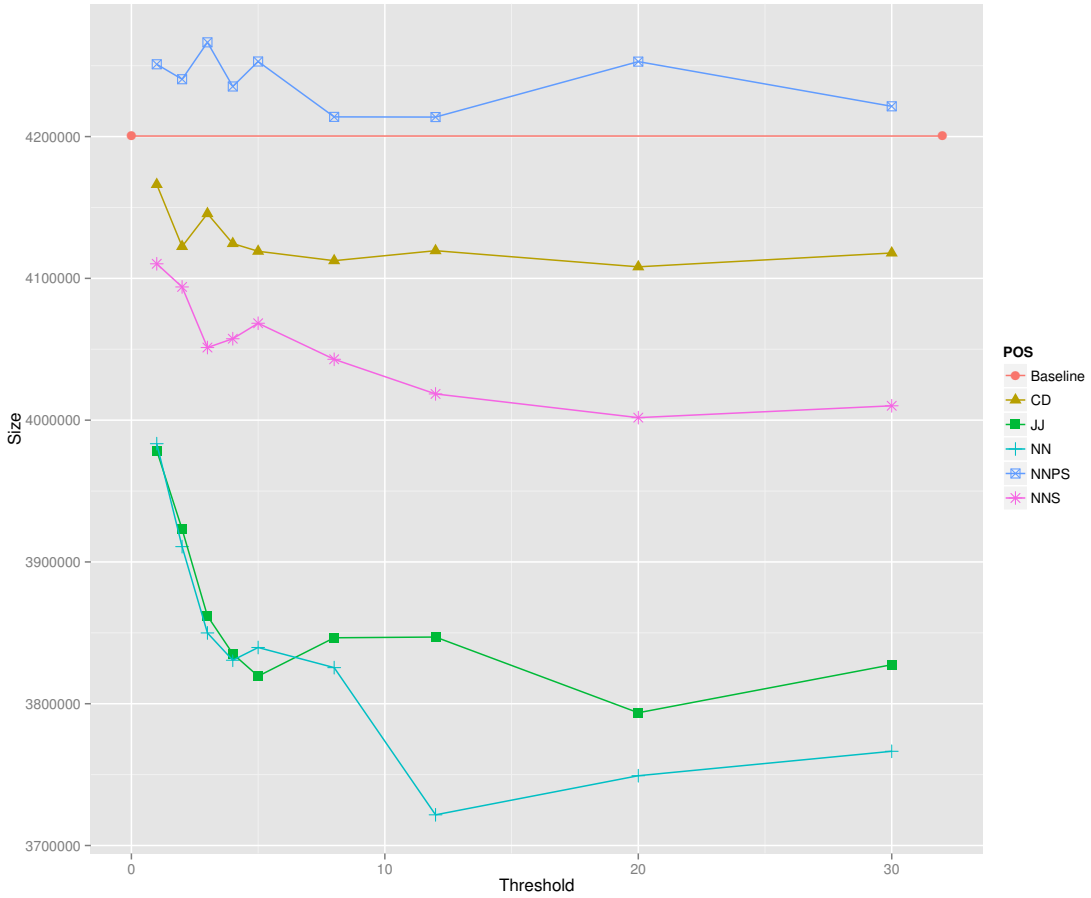


Figure 4.1: Grammar size at various normalization thresholds for selected preterminal labels. Averaged over 5 grammar-training trials and evaluated on WSJ section 22.

We performed a grid search over a variety of λ values, training latent-variable grammars on the transformed corpora with a variant of the Berkeley Parser's grammar-training system [142], as described in Section 2.9. The EM-based training mechanism is somewhat sensitive to initialization state [141], so we trained 5 separate grammars (using different random seeds) on each transformed corpus. All reported accuracy values are averages over these random grammars. We trained all grammars for 6 cycles, yielding grammars with approximately 1100 split non-terminals.

Figures 4.1 and 4.2 demonstrate the effects of these normalization on selected preterminals. As demonstrated in Figure 4.1, in most cases, normalizing most preterminals

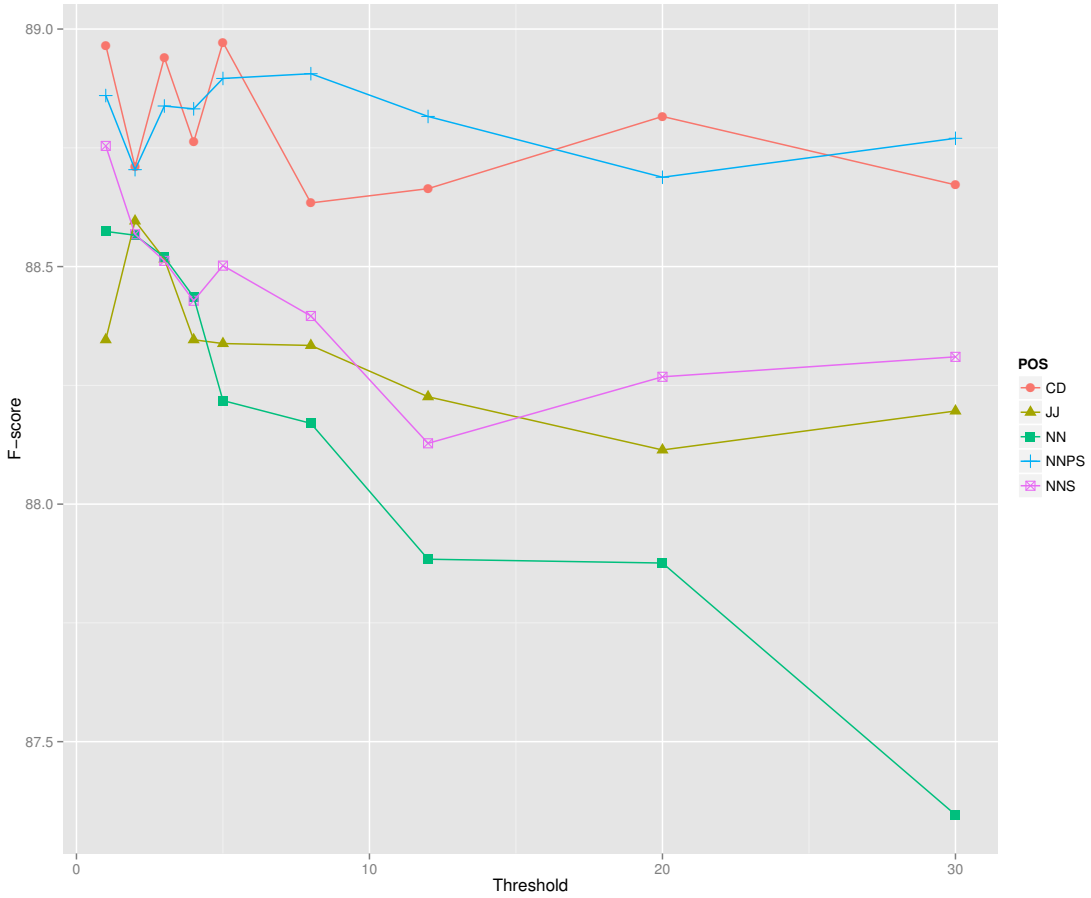


Figure 4.2: F_1 at various normalization thresholds for selected preterminal labels, evaluated on WSJ section 22, as in Figure 4.1.

produces a considerably more compact grammar (although not universally so; the grammars normalizing NNPS are somewhat larger than the baseline). Figure 4.2 demonstrates the effect on dev-set accuracy of normalizing each of these labels separately. Note that some preterminals are very sensitive to this normalization process, and accuracy drops off sharply with increasing λ, while others are more tolerant of aggressive normalization. In particular, CD and NNPS consistently show *increased* accuracy, even up to $\lambda = 30$. In the case of those two preterminals, this effect is somewhat unsurprising, as the class-based features in Table 4.2 (particularly -NUM, -CAPS, and -s) serve to uniquely identify cardinal numbers and plural proper nouns, and the normalization improves generalization without greatly sacrificing specificity in the learned productions.

4.2.2 Combined Normalization

We then combined normalizations of multiple open-class preterminals and trained grammars the resulting corpora. We began with the thresholds from Table 4.3, which maximize development-set F_1 for each preterminal. For these trials, we omitted RB and VBP, for which even normalizing singletons costs accuracy without a significant grammar-size reduction.

We produced a training corpus N_0, using the thresholds from Table 4.3, which maximize F_1. That is, $\lambda_{\text{CD}} = 5, \lambda_{\text{JJ}} = 2$, and so on. This normalization reduced the lexicon size by 58.6% (from 44.4k to 18.3k words) and the memory footprint of the lexicon itself even more so (from 377KB to 151KB). We call this corpus N_0. We then varied each λ equally to produce a series of corpora N_x, where $-1 \leq x \leq 20$. Thus, for N_3, $\lambda_{\text{CD}} = 8$ (recall that $\lambda_{\text{CD}} = 5$ for N_0), $\lambda_{\text{JJ}} = 5$, and so on. As we discussed in Section 4.1, eliminating lexical productions may allow a reduction in the number of phrase-level productions as well. If so, we may be able to re-merge more non-terminal splits than the default 50% at each cycle, reducing both $|V|$ and $|P|$. For each x, we trained two sets of grammars, one using a 50% merge, and one at 55%.[5] Figure 4.3 plots the accuracy and size of each resulting grammar and Table 4.4 presents a subset of the trials numerically. We found moderate efficiency gains for exhaustive parsing

Grammar N_0 is 32.4% smaller than the baseline, while sacrificing only .1 F_1. A 55% merge reduces the size by an additional 13.3% at a cost of an additional tenth of a point. But the speed gains on exhaustive inference are modest, and on pruned inference, we found no significant speed improvement. The most interesting result is that re-merging 55% of the candidate splits (without any corpus transformation) actually improves accuracy. We will explore this result further in Section 5.2.

The class assignments learned from this decision-tree incorporate a number of lexical features (described in Table 4.2), and generally provide robust probability estimates. But generalization is in many cases limited by the strict decision-tree structure. For example, the usage and context of the tokens 'fourteenth-story' and '90th-story' is likely to be quite similar, but their class assignments (UNK-DASH-y and UNK-NUM-DASH-y) do not incorporate that similarity. In fact, the two tokens would only be tied probabilistically

[5] The Berkeley parser defaults to merging 50% of candidate splits. Text-normalization may permit merging a larger percentage without loss; we will explore merge percentages more fully in Section 5.2.

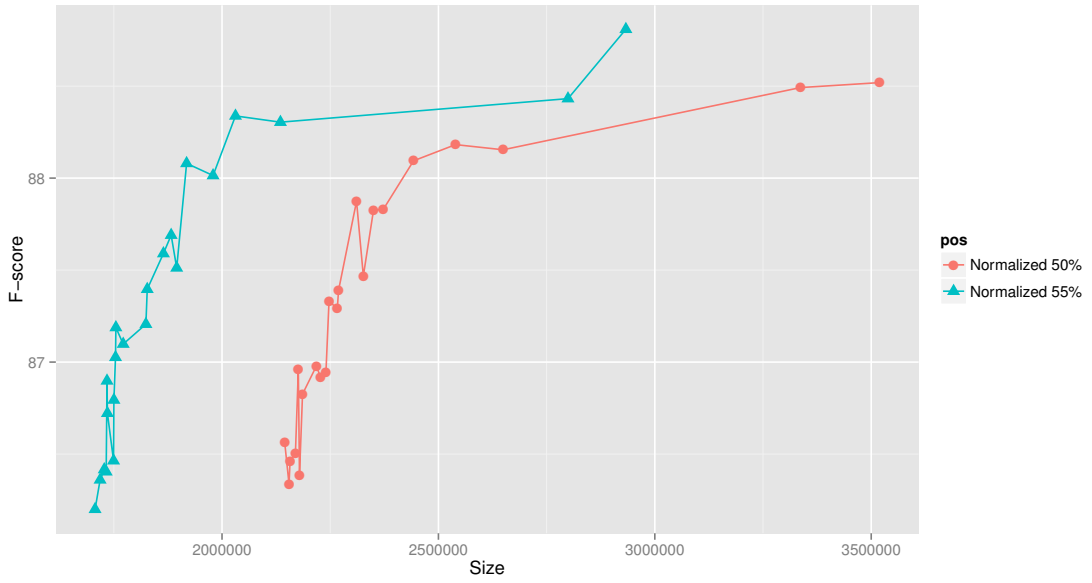


Figure 4.3: Accuracy vs. size for combined simple-normaliation grammars as the normalization threshold is varied from 0–20.

Grammar (λ, **merge**)	$\Delta\|P\|$	**Exhaustive**		**Pruned**	
		F_1 (σ)	**w/s**	F_1 (σ)	**w/s**
Baseline, 50%	-	88.6 (.10)	10.51	88.6 (.20)	1955
Baseline, 55%	-4.2%	89.1 (.05)	12.72	89.0 (.15)	2038
0, 50%	-32.4%	88.5 (.19)	10.66	88.6 (.17)	1952
0, 55%	-45.7%	88.4 (.17)	13.37	88.6 (.16)	2010
5, 50%	-45.0%	87.9 (.24)	10.72	88.2 (.22)	2035
5, 55%	-55.2%	87.7 (.26)	13.17	88.1 (.24)	2031

Table 4.4: Accuracy, size, and efficiency of grammars trained with normalized corpora, at selected operating points. Training corpora transformed by normalizing all open-class preterminals. Evaluated on WSJ section 22.

if (at inference time) the model was forced to back off fully to probabilities for the base UNK class. In the following section, we present a more sophisticated method of class assignment that can robustly link lexically-similar items.

4.3 Word Clustering

The class-based model described in the preceding section can be viewed as a deterministic hard-clustering mechanism, with a simple (again, deterministic) means for assigning unobserved tokens to existing clusters. In this section, we describe an alternate clustering method, which performs cluster assignment using a sequence tagger. The overall approach is similar to that of Seddah et al. [165], but the specific clustering and cluster-assignment methods are quite different, as our primary objective criteria remains efficient inference, whereas they targeted cross-domain generalization.

We clustered all rare open-class words — varying the threshold defining 'rare' as in Section 4.2. We extracted from the training corpus all rare words and the features thereof, as listed in Table 4.5. Note that in addition to lexical features similar to those of the class-based model, we are able to include information from the syntactic context as well. In some cases, this syntactic context (or an estimate thereof) will be available during inference to guide the cluster assignments of unknown words. Other features (e.g. span features of grandparent labels) are only available during training, but are still of utility to guide the clusters from which the inference-time tagging models are trained. We assigned those rare words to clusters using K-Means++ [4] as implemented in the Weka toolkit [79].

This clustering produces a set of word classes analogous to the class-based signatures described in Section 4.2, and the rare-word probabilities learned in Huang and Harper [85]. We again transformed the training corpus, replacing rare words with tokens denoting their cluster assignments. Training a PCFG on this transformed corpus learns production probabilities for the word clusters. The following section will describe a means of assigning unseen tokens to the appropriate clusters, incorporating the surrounding lexical and syntactic context.

4.3.1 Tagging

We trained a discriminative multiclass tagger to replicate the K-Means++ cluster assignments. Such taggers have proven effective in a variety of domains [48, 156]. We train an averaged-perceptron model [48], using the same features used during clustering (with the notable exception of those features only available from labeled training data, denoted with an '*' in Table 4.5).

We begin by POS-tagging the input sentence, using an averaged-perceptron tagger.

t_{i-1}, t_i, t_{i+1}	Parent POS tag of tokens i-1, i, i+1 (as assigned by a 1-best discriminative POS tagger)
l_{p2}, l_{p3}	Grandparent and great-grandparent non-terminals *
s_{p2}, s_{p3}	Span of the grandparent and great-grandparent labels (1, 2, 3, 4–5, 6+) *
s_1	Unigram suffix (final character of the token)
s_2	Bigram suffix
n_i	Contains-numeral
nf_i	Fraction of the entire token consisting of numerals ($\geq$ 20%, 40%, 60%, 80%, and 100%)
$@_i$	Contains '@'
$p_{\#}$	Starts-with '#'
p_{http}	Starts-with 'http'
$punct_i$	Contains-punctuation
$punctf_i$	Fraction of the token consisting of punctuation (binned as for the analogous numeral features).

Table 4.5: Word-clustering features. Features from labeled training data are marked with an '*'; these features are used during clustering, but are not available for cluster-assignment during inference. Some of features are more applicable to non-canonical genre, such as email, Twitter, and blog posts, and are not regularly observed in WSJ text, but we retain the same feature-set for the cross-domain trials in Section 4.4.1.

Although we did not engineer features sufficiently to match the state-of-the-art for POS-tagging, our tagger achieves accuracies > 96.3% on WSJ text, which is quite sufficient for our needs. Note that we do not strictly need to perform this step for an input sentence containing no unknown words, but the POS tags are shared with other pruning steps later in the parsing pipeline [19], so we tag each input sentence. The tagger processes input at upwards of 200k words per second, so it is a relatively inexpensive preprocessing step (in comparison to the overall cost of context-free inference).

We use the predicted POS tags, along with the lexical features from Table 4.5 as input to the cluster-assignment classifier. This tagging method — unlike the decision-tree approach and most other methods in widespread use — incorporates the surrounding context (including previous and subsequent parts-of-speech) into OOV-class assignment, so the cluster-assignment process is similar to that of a sequence tagger.

Although many of the features in Table 4.5 are English-specific, the syntactic features

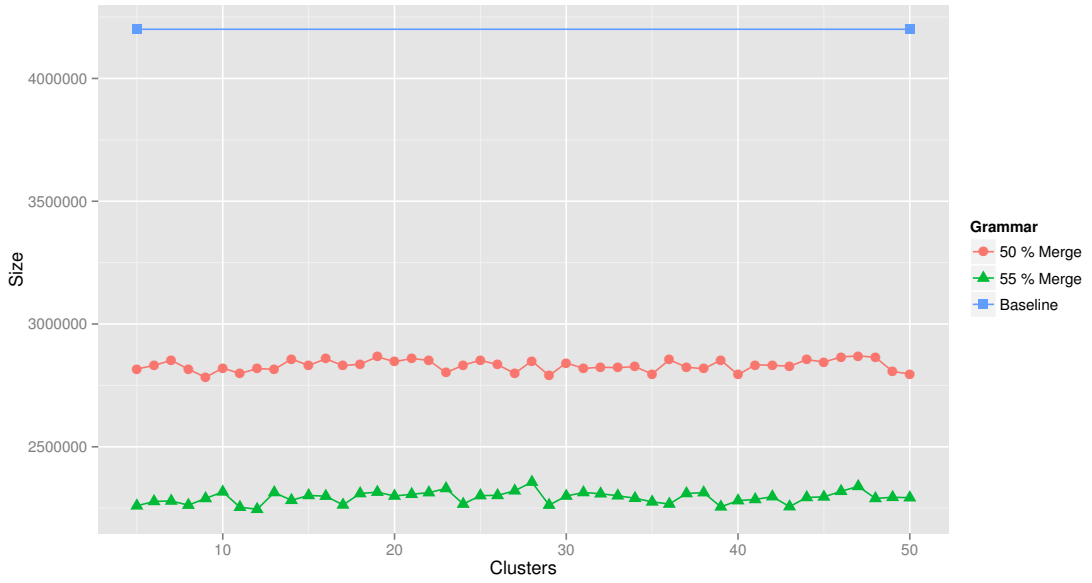


Figure 4.4: Size of clustering grammars, plotted against the number of clusters. The grammar size appears to be relatively independent of the cluster count, but we do see a considerable size improvement when merging an additional 5% of the non-terminal splits at each cycle.

apply across languages and genres. And in some cases, even the lexical features are applicable beyond English. For example, the newswire text of the Penn Chinese Treebank [185] includes numerous instances of foreign words and abbreviations (often English) incorporated into the Chinese text. Chinese words are generally quite short (1-4 characters), and Huang and Harper [85] found that a character n-gram model predicts word classes quite well; thus, although we did not extend the feature-set specifically for Chinese, the suffix features may capture a considerable portion of that predictive power.

4.3.2 Clustering Results

Table 4.7 demonstrates that the clustering and tagging approach achieves similar reductions in grammar-size to those we found in Section 4.2.2. However, at comparable grammar sizes, clustered-grammar accuracy is degraded considerably from that of the combined normalization method. Further, as indicated in Figure 4.4 and Figure 4.5, the tradeoff between size and accuracy is not smooth or consistent. Instead, we find that grammar

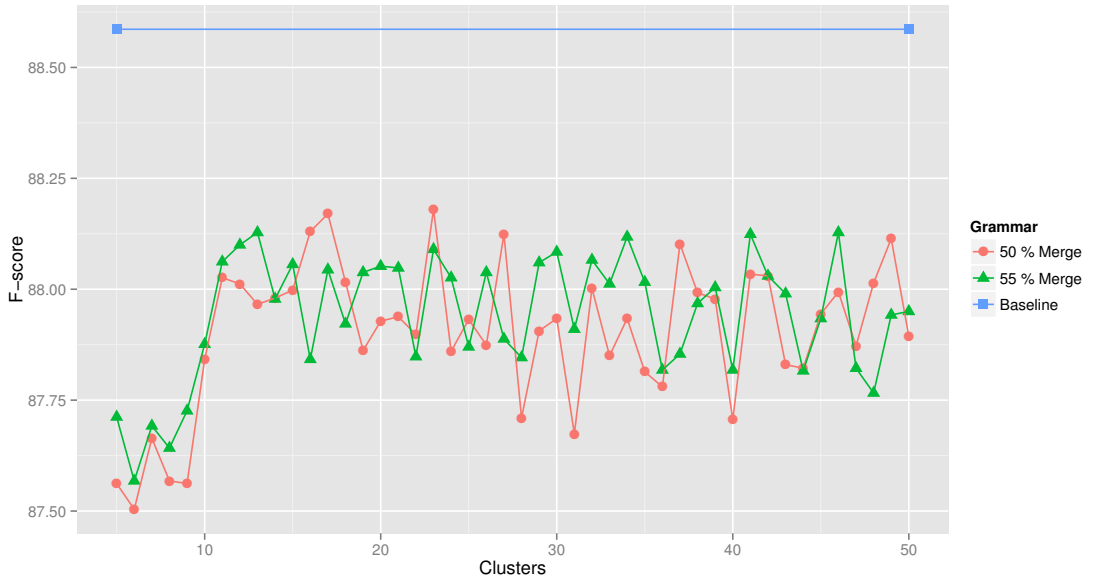


Figure 4.5: Accuracy of clustering grammars, plotted against the number of clusters. Unlike the grammar size plotted in Figure 4.4, accuracy is clearly impacted by the cluster count; with less than 10–15 clusters, accuracy degrades considerably. However, with sufficient clusters, the accuracy recovers somewhat, and it appears that the 55% merge fraction does not greatly degrade accuracy.

size is relatively invariant to the number of clusters, and that accuracy is also fairly flat (after a minimum cluster-count of approximately 10–12, below which accuracy drops considerably). In general, it appears that the manually tuned decision-tree 'clusters' better represent groupings of tokens which serve similar syntactic roles than do the automatic clusters we trained. Alternate clustering techniques might improve on these results, but we will leave that exploration for future work, and focus the following section and the trials in Chapter 7 on the more successful combined normalization approach from Section 4.2.2.

4.4 Test Set Results

We evaluated the 'combined' normalization method across several languages and genres. Table 4.7 presents the resulting accuracies and speeds on WSJ newswire text [121], Switchboard telephone conversations [71], and Chinese newswire [185]. We trained 5 separate grammars and present accuracies averaged across that set.

Clusters	$\|P\|$	$\mathbf{F}_1$ (σ)	w/s
10	2316k	87.9 (.19)	3.3
17	2263k	88.0 (.16)	3.3
50	2293k	88.0 (.09)	3.7

Table 4.6: Selected operating points from Figure 4.5. Accuracy, size, and efficiency of grammars trained with lexical clustering, evaluated using exhaustive inference on WSJ section 22.

Grammar	$\|P\|$ (σ)	Exhaustive		Pruned	
		$\mathbf{F}_1$ (σ)	w/s	$\mathbf{F}_1$ (σ)	w/s
WSJ					
Baseline	3543k (45.7k)	88.7 (.16)	13.1	88.7 (.19)	2043
Normalized	2281k (62.7k)	88.2 (.36)	14.0	88.4 (.32)	2074
Switchboard					
Baseline	1920k (15.7k)	86.9 (.15)	2.9	85.1 (2.3)	1697
Normalized	1652k (43.1k)	86.6 (.09)	3.0	86.8 (.21)	1776
Chinese					
Baseline	2702k (77.1k)	81.0 (.41)	4.1	79.4 (.70)	745
Normalized	2223k (27.8k)	79.8 (.44)	4.5	78.8 (.56)	753

Table 4.7: Test-set trials of grammars trained with the combined normalization method from Section 4.2.2 at $\lambda = 0$. All grammars trained merging 55% of non-terminal splits at each cycle. Standard deviations in parenthesis)

As in the previous development-set trials, we found that normalization imposed a modest accuracy cost, although the WSJ test-set degradation was slightly less than that on the development-set.[6] We found improvements of up to 10% in exhaustive parsing speed, but the gains for pruned inference were more modest. One note of interest regarding the accuracy of Chinese parsing: Bodenstab [19] reported that pruning improved accuracy vs. exhaustive search, whereas we found a loss of 1-1.5 points. As we discussed in Section 2.13,

[6] In Chapter 7, we will present similar trials over a larger set of grammars as we combine grammar training methods; in that case corpus transformation does not degrade accuracy measurably. Although the results in Table 4.7 were consistent across languages, they may still be indicative of the statistical noise inherent in a small set of grammars.

the PCTB test set is very small, so we are inclined to attribute this discrepancy to sampling noise over a limited number of grammars (Bodenstab tested a single grammar, and we examined 5).

4.4.1 Cross-domain Generalization

Although efficient inference is our primary objective, we also consider how lexicon simplification affects the cross-domain generalizability of the learned model. The distribution of token occurrences is likely to differ greatly across genres, particularly between well-formed newswire text and less well-formed genres. We applied models trained on WSJ text to the 5 genres of the English Web Treebank [14], and evaluated how the simplified lexicon — removing rare tokens specific to the WSJ text — generalizes to other domains. We performed no domain adaptation for the target genres, so the accuracies are unsurprisingly lower than those obtained by most submissions to the SANCL 2012 Shared Task [147]. But for our investigation, the relative effect of normalization is more important than the absolute accuracies obtained. Table 4.8 presents the results of these trials; we report the average accuracy over the 5 WSJ grammars from Table 4.7 and the maximum speed obtained in 5 individual trials with each. We found no consistent effect of the text-normalization on either accuracy or speed. The largest difference between the baseline mode and the $\lambda = 0$ model was a gain of .5 in F_1. That gain (on the 'Email' genre) might be meaningful, but the other differences (gain and loss) were negligible in magnitude.[7] The speed differences were similarly small; in general, the text-normalized grammars were somewhat faster (excepting a very small regression on the 'Newsgroups' genre), but none of the gains were material improvements.

4.5 Discussion

In this chapter, we presented two methods of corpus transformation that simplify the lexicon of a PCFG. We found that we can reduce the lexicon — and the memory footprint thereof — dramatically without great loss in parsing accuracy. This reduction may provide

[7] In the case of a search failure — relatively more common in non-canonical genres — our inference system automatically relaxes pruning parameters and re-parses. This mechanism increases reliability of inference, but occasionally causes increased parsing time with a smaller or less flexible model; c.f., for instance, the email genre at $\lambda = 5$.

Genre	**Baseline**		$\lambda = 0$		$\lambda = 5$	
	F_1 (σ)	w/s	F_1 (σ)	w/s	F_1 (σ)	w/s
Email	73.7 (.21)	1264	**74.2** (.39)	**1392**	73.3 (.15)	1053
Weblogs	**75.8** (.27)	434	75.7 (.44)	**456**	74.7 (.38)	437
Reviews	**74.4** (.20)	349	74.3 (.60)	**356**	73.2 (.38)	352
Newsgroups	72.2 (.27)	**308**	**72.4** (.30)	298	71.5 (.35)	306
Answers	**73.3** (.36)	1659	73.1 (.35)	1601	72.0 (.25)	**1742**

Table 4.8: Accuracy and speeds of a WSJ-trained grammar on the 5 non-canonical genres of the English Web Treebank. Each test set includes approximately 2000 sentences of annotated text drawn from web crawlers or other sources as appropriate for the domain. All trials performed with the BUBS parser using a complete-closure model [21] and beam-search guided by a lexical prioritization model [19].

real advantages in constrained-memory environments, or on some processor architectures (depending primarily on the specific cache capacity and layout). However, the lexical productions removed by these normalization techniques would often be unused — and not competing for cache storage. Their cost is mainly incurred while reading the grammar at startup, after which they lie idle in main memory. On the Intel Nehalem architecture we examined, we found that simplifying the lexicon provided minimal efficiency gains. The results from this chapter serve to underscore the nonlinear relationship between grammar size and speed. In the following chapter, we will further explore that relationship, and train models which can more effectively trade off speed and accuracy. In Chapter 7, we will combine the methods from this chapter with those from chapters 5 and 6; the combination yields interesting operating points, and in some cases, considerable additive efficiency gains.

Chapter 5

Regularization and Merge Objective Functions

In the previous chapter, we presented methods of altering training data to improve the efficiency of a trained model. In this chapter, we move on to the training process itself, developing methods of incorporating efficiency objectives into the split-merge grammar training approach described in Section 2.9. We present several variants of the approach and compare the effects thereof.

We begin by examining the parameter pruning threshold of the grammar training process. Optimizing this threshold yields a reduction in model size of approximately 60% and an improvement in parsing speed of 41% for exhaustive inference and 45% for pruned. We then proceed to describe methods of altering the model structure within the split-merge process, specifically by varying the objective function used to select candidate state-splits during the merge phase of the algorithm. As in the majority of this thesis, we are interested specifically in efficient inference, so most of the objective functions we consider will target compact and efficient grammars, but the learning methods we present are applicable to other goals as well — e.g., targeting specific parser error categories, or semi-automatic domain adaptation. These merge objectives all reduce the model size greatly, and the most successful thereof improves efficiency of exhaustive and pruned inference by 60% and 15% respectively.

5.1 Sparse Priors and Regularization

As we have demonstrated in earlier chapters, efficiency of inference usually benefits from a compact model. In some cases, we can encode this preference for compact models into the learning process, a bias known as a 'sparse prior'. This training bias prefers — in the

space of all possible models — to learn sparser solutions. Since a compact model is often also an efficient one, all the training methods we present in this chapter are biased in one way or another toward producing sparse models (although they differ considerably in the manner in which that bias is implemented). Thus, all these methods can all be considered sparse priors, in a very broad sense.

However, the term 'sparse prior' is often used to refer more specifically to specific regularization methods. Regularization simply means selecting a level of model complexity appropriate for the task, particularly to avoid overfitting (common) or underfitting (less common) the model to the training data. L_1 and L_2 regularizations are common in signal processing [35], statistics [135], and NLP applications [145, 69]. These regularization methods introduce a secondary objective — secondary, that is, to the primary model-fit objective — which minimizes some measure of model complexity. That objective is represented by a penalty term R, controlled by a hyperparameter λ, to balance between overfitting and oversimplification of the model. In the context of learning a least-squares linear regression model w, we perform the following minimization:

$$\hat{w} = \underset{w}{\operatorname{argmin}} \, (Aw - y)^2 + \lambda R \tag{5.1}$$

L_1 regularization [175] penalizes the sum of the absolute values of the model parameters — i.e., the L_1 norm of the model vector ($R = ||w||$). This penalty forces many model parameters to 0, producing sparse models [128]. This normalization can be effective in training compact conditional probability models, and is often applied to discriminative NLP modeling tasks. However, penalizing L_1 does not yield a useful objective function for training a generative model. In the case of a PCFG, the sum of the probabilities for each parent non-terminal is 1, so the L_1 norm of the overall model is simply the size of the non-terminal set V. While $|V|$ is a meaningful component of overall model complexity, and a significant predictor of inference cost (see Section 5.6), it is not the most important one.[1]. We will consider another method of controlling $|V|$ in Section 5.2, but we will focus our exploration of sparse priors on other methods.

L_2 regularization [176] penalizes the sum of the *squared* model weights ($R = ||w||^2$).[2]

[1] For example, recall the comparison of various grammars in Table 3.2. Although the non-terminal set of the parent-annotated grammar is approximately 6× that of the latent-variable grammar, the dramatically smaller ruleset yields faster inference under most grammar-intersection approaches.

[2] L_2 regularization and related methods go by a variety of names, including Tikhonov regularization, the TikhonovMiller method, the PhillipsTwomey method, and Ridge Regression.

L_2 regularization rewards small parameter values, and does not force weights to 0, so it generally does not yield as sparse a model as L_1. At the extreme, L_2 regularized PCFG training would yield a fully-populated grammar matrix with a perfectly flat weight distribution (i.e., every possible grammar production, each with an infinitesimal weight). Although L_2 regularization might be an effective smoothing method, it is unlikely to yield sparse and efficient PCFG models.

L_0 regularization [2, 164, 112] penalizes all non-zero parameters equally ($R = \sum_i I_{w_i \neq 0}$). This objective disregards the magnitude of the weights, instead counting and weighting equally all non-0 parameters. When applied to PCFG training, the L_0 norm is simply the size of the ruleset, and L_0 regularization reduces $|P|$. Thus, L_0 regularization may be a profitable approach for training efficient PCFGs. L_0 optimization consists of selecting the minimum-loss feature subset. Unfortunately, this feature-selection problem is NP-Hard [127], so L_0 is generally impractical when the feature-set is large; however, we will consider a greedy analog thereof in Section 5.4.

5.1.1 Uniform Parameter Pruning

After each split, merge, and smoothing phase of the split-merge process (as described in Section 2.9), we learn production probabilities using expectation maximization. As EM converges, some rules will be deemphasized, such that eventually the observed fractional counts are nearly 0. This is in some ways similar to the weight pruning of L_1 regularization — in both cases, an implementation will usually specify a small ϵ below which weights will be set to 0.[3] In the Berkeley Parser grammar training system, we prune rules between each EM iteration, at a threshold which defaults to $\epsilon = 10^{-30}$. Figure 5.1 demonstrates the effect of varying ϵ across a wide range. We find that ϵ can be increased considerably (reducing the resulting model size) before accuracy begins to degrade. And as demonstrated in Table 5.1, the more compact model yields an improvement of 45% in inference speed, along with a small gain in accuracy.[4] Note that the rule-pruning threshold ϵ is used only during grammar training, and is separate from any pruning done during inference.

[3] The limitations of floating-point arithmetic imply such a threshold on any practical machine; specifying ϵ allows a larger — and configurable — threshold.

[4] Note: in this chapter, we will examine the training methods separately. In Chapter 7, we will present trials combining the approaches from this chapter with those from chapters 4 and 6.

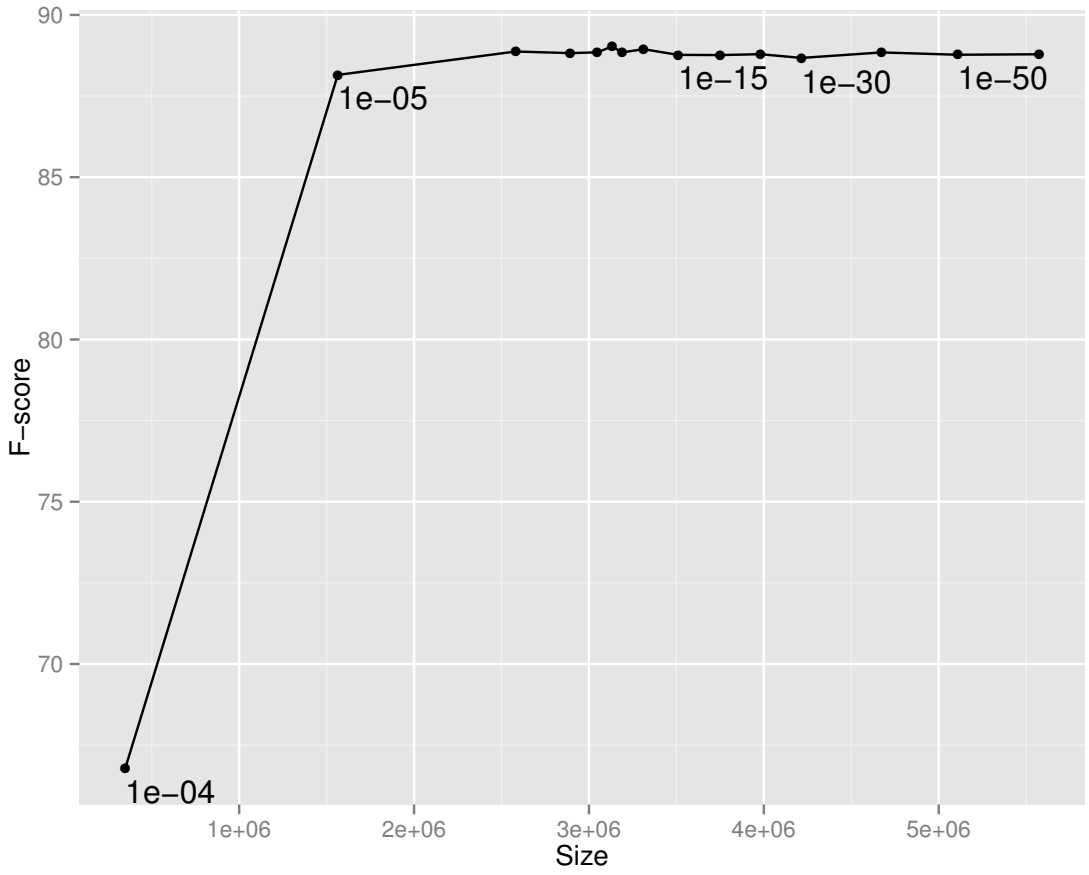


Figure 5.1: **Parameter Pruning** Accuracy vs. grammar size as the rule-pruning threshold is varied from 10^{-4} to 10^{-60} (with selected thresholds marked). Averaged over 8 grammars, trained on WSJ sections 02-21, and evaluated using exhaustive inference on section 22. The Berkeley Parser's trainer defaults to a threshold of 10^{-30}. We find that reducing that threshold does not improve performance, and that we can in fact increase the threshold to 10^{-6} and reduce the grammar size by 39% with no loss of accuracy.

5.2 Merge Fraction

In the merge phase of split-merge training, we re-merge a portion of the current set of non-terminal splits. Standard practice since Petrov et al. [142] has been to merge 50% of the candidate splits. In Section 4.2.2, we found that merging 55% produced a slightly smaller grammar and a minor efficiency gain, along with a considerable gain in accuracy. Before examining merge objective functions (which rank the merge candidates), we will

ϵ	Size ($\lvert P \rvert$)	Exhaustive		Pruned	
		$\mathbf{F}_1$	w/s	$\mathbf{F}_1$	w/s
10^{-30}	3215k (42.9k)	88.7 (.23)	10.3	88.6 (.17)	1952
10^{-15}	3509k (40.9k)	88.8 (.24)	12.3	88.7 (.27)	2163
10^{-6}	2581k (37.9k)	88.9 (.20)	14.6	88.8 (.16)	2837

Table 5.1: Pruned inference at selected operating points from Figure 5.1. Performed with a complete-closure model [21] and beam-search guided by a lexical prioritization model [19]. At $\epsilon = 10^{-6}$, we find a 45.3% improvement in speed, along with a slight gain in accuracy and reduction in variance.

explore operating points for this *merge fraction* hyperparameter.

A considerable body of work supports the benefits of the merge phase. Thus, we can exclude values near 0 (i.e., those that retain nearly all splits). While a merge fraction near 1 — and a commensurately larger number of training cycles — might be interesting, the dramatic increase in training time is impractical. Thus, we evaluated 4-, 5-, and 6-cycle training at merge fractions between .25 and .7. Figure 5.2 plots the average accuracies obtained at various operating points (for clarity, this plot includes merge fractions between 40% and 65%). Unsurprisingly, in each case a 6-cycle grammar produces the highest accuracy and lowest speed, and a 4-cycle grammar the inverse. Merging 55% at each cycle produces grammars with the most favorable performance — a combination of maximum accuracy and superior speed (vs. the 40%-merge grammars, which provide comparable accuracy, but at a speed penalty of approximately 10%). Thus, we will use a 55% merge hyperparameter for all grammars trained in the remainder of this chapter.

5.3 Merge Objective Functions

As discussed in Section 2.9, the merge phase of latent-variable grammar training improves generalization and constrains the size of the grammar, allowing more training cycles (typically 5 or 6, whereas splitting without merging is usually limited to 4) and thus more splits of certain non-terminals, when justified by the training data. All corpus-based PCFG training methods learn model *weights* from training data. We can think of the merge phase as automatically adapting the *structure* of the model as well (specifically the set of split non-terminals).

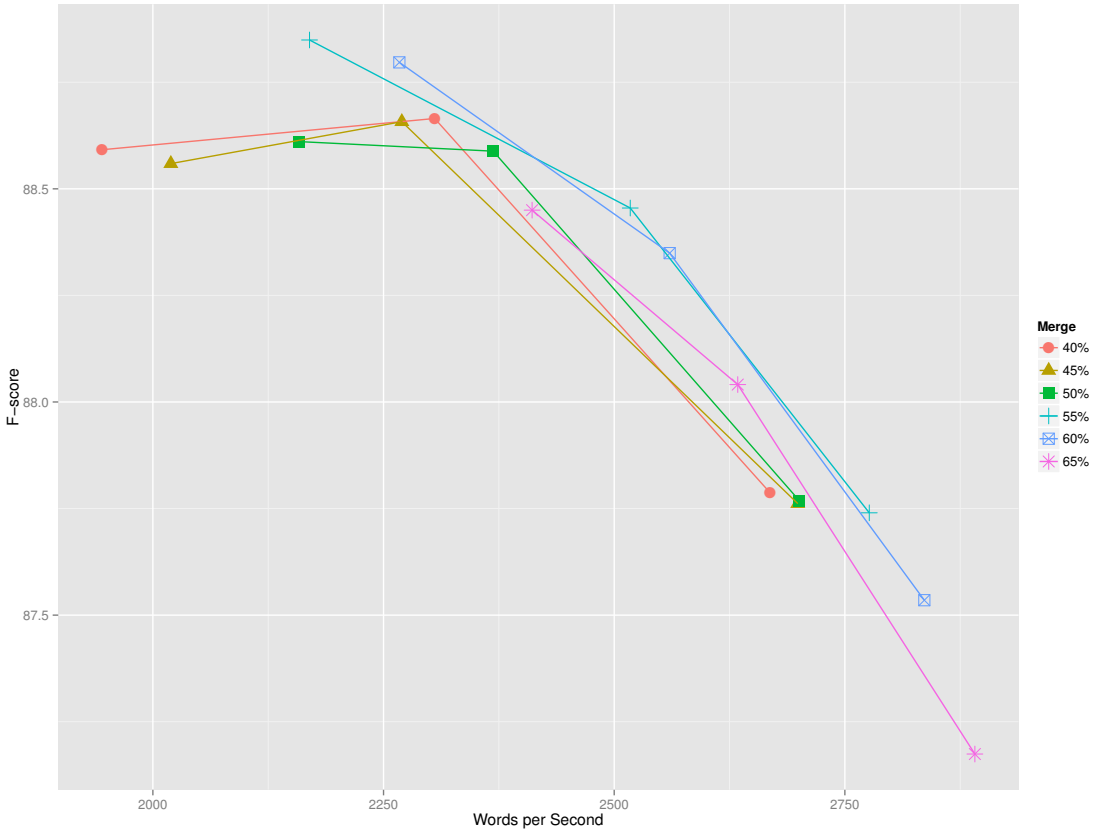


Figure 5.2: Accuracy vs. speed of grammars trained at different merge fractions. Averaged over 12 grammars trained at each operating point, and evaluated on WSJ section 22.

The regularization methods discussed in Section 5.1 adapt model weight training, biasing weights in a direction favorable to efficient inference — in particular, L_1 forces some of those weights to 0, thus reducing the memory footprint of the model. We will now consider analogous methods which alter merge candidate selection to introduce a similar bias.

During each cycle of LV grammar training, we split each non-terminal into 2 substates, and learn appropriate weights via EM. To constrain the model size, we must then select a portion of those splits to retain, and a portion to discard (re-merge). We rank candidate state splits by an estimate of the cost of re-merging the substates. In most cases, the cost function is chosen to approximate the potential loss in accuracy if the split is re-merged.

Petrov et al. [142] presented a likelihood-based approach. To estimate the cost of re-merging a split non-terminal, we compute two probability estimates for each training

tree — one probability retaining the split and one with the split re-merged (P and P^n respectively). The product of all estimates over all sentences w_i in the training corpus yields an approximation of the global likelihood loss if the split is re-merged.

$$\Delta_L(A_1, A_2) = \prod_i \prod_{n \in T_i} \frac{P^n(w_i, T_i)}{P(w_i, T_i)} \tag{5.2}$$

This likelihood-based merge criteria can be computed efficiently during training, and is in accordance with the overall likelihood objective of EM training. It has proven effective in the Berkeley Parser, and has been integrated into other L-PCFG learning frameworks as well (e.g. Le Roux et al. [109]). However, likelihood loss is not the only potential objective function for merge candidate selection; a wide array of other objectives are possible, and to our knowledge, few have been explored. In some cases, alternate merge objectives may better model the desired evaluation criteria or produce a grammar better-suited for a potential application domain. We adapted the grammar training system from the Berkeley Parser [142], adding the ability to rerank merge candidates by arbitrary functions. Each of the objective functions we will consider incorporates both an accuracy metric and an efficiency metric, allowing a smooth tradeoff between the two.

More formally, let $f(A_1, A_2)$ be a function that prioritizes a state split A_1, A_2 according to some desired criterion, and $r(f(A_1, A_2))$ a ranking function which assigns ordinal ranks to candidate state splits, as prioritized by f). Let $\hat{a}(A_1, A_2)$ be a function estimating the accuracy impact of a state-split and $\hat{e}(A_1, A_2)$ an analogous function estimating the efficiency effects.[5] We can then vary the operating point from entirely accuracy-driven to 100% efficiency-driven, using the following function:

$$rank(A_1, A_2) = \lambda \cdot r(\hat{a}(A_1, A_2)) + (1 - \lambda) \cdot r(\hat{e}(A_1, A_2)) \tag{5.3}$$

In the following sections, we will examine and compare several candidates for $\hat{a}(A_1, A_2)$ and $\hat{e}(A_1, A_2)$. This approach is in many ways similar to the speed-accuracy tradeoff in Eisner and Daumé [63] and Jiang et al. [88], although they operated in a different domain, using reinforcement learning to train efficient agenda parsers.

[5] Equation 5.2 is a natural candidate for a baseline $\hat{a}(A_1, A_2)$.

λ	Size ($\lvert P\rvert$)	Exhaustive		Pruned	
		$\mathbf{F}_1$ (σ)	w/s	$\mathbf{F}_1$ (σ)	w/s
0	4215k (40.6k)	88.7 (.23)	10.7	88.6 (.17)	1968
.25	3903k (36.5k)	89.0 (.16)	10.4	88.9 (.14)	1959
.40	2450k (33.7k)	88.9 (.30)	10.0	89.1 (.21)	2027
.45	1304k (53.6k)	88.7 (.15)	12.8	88.9 (.15)	2086
.5	390k (64.5k)	86.7 (.37)	16.5	87.2 (.33)	2228

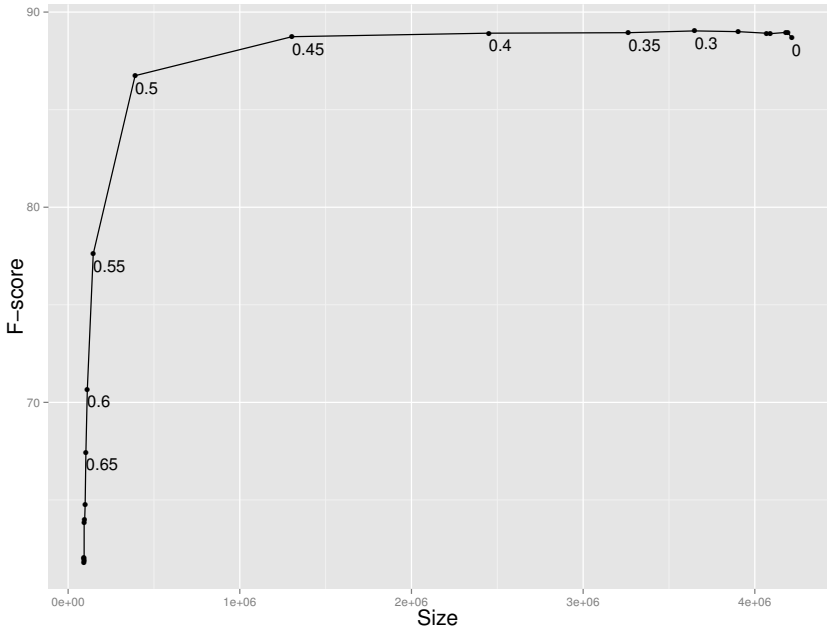
Table 5.2: Greedy L_0 Merge Objective — selected operating points from Figure 5.3. At $\lambda = .45$ — before accuracy begins to decline — the model size is reduced greatly, but the gains in parsing speed are modest — 19% for exhaustive inference and 6% for pruned. Pruned inference trials performed with a complete-closure model [21] and beam-search guided by a lexical prioritization model [19].

5.4 Greedy L_0 Merge Objective

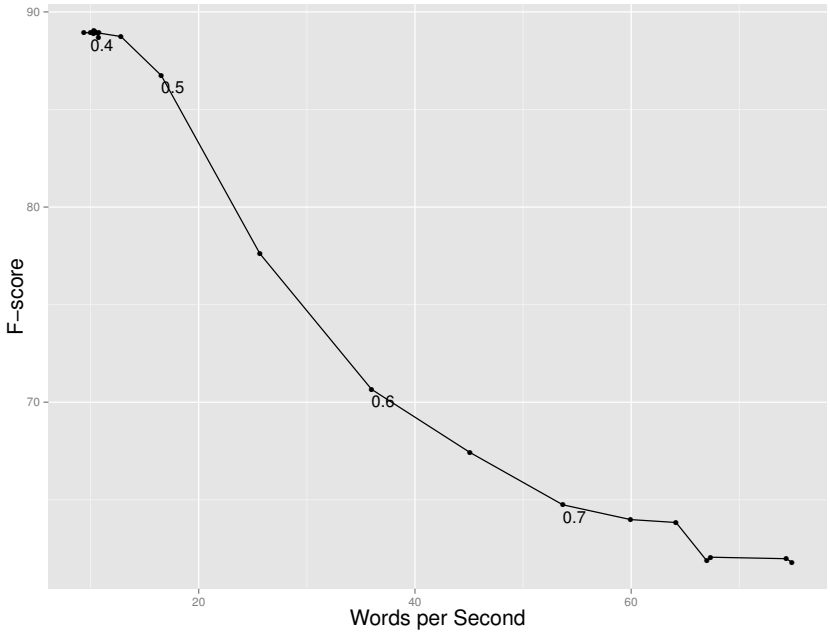
Although true L_0 optimization is NP-Hard, greedy approximations thereof are tractable, and can be quite effective [127]. In this section, we consider one such method, wherein the greedy feature-selection is done at the time of merge candidate selection. In this merge objective, we explicitly target the rule count of the post-merge grammar. We retain Equation 5.2 as $\hat{a}(A_1, A_2)$, and $\Delta_{|P|}$ as $\hat{e}(A_1, A_2)$ yielding the following ranking function:

$$r(A_1, A_2) = \lambda \cdot r(\Delta_L(A_1, A_2)) + (1 - \lambda) \cdot r(\Delta_{|P|}(A_1, A_2)) \tag{5.4}$$

We trained 8 grammars on WSJ sections 2-21 at each operating point $0 \leq \lambda \leq 1$, and evaluated on WSJ section 22. Figure 5.3 and Table 5.2 demonstrate that even this simple approach can be quite effective, reducing the size of the PCFG by 69% with no degradation of accuracy, and improving parsing speed by 19% and 6% (respectively) for exhaustive and pruned inference. Although the speed improvement is not dramatic, we are encouraged by the gain from even a very simple estimate of $\hat{e}(A_1, A_2)$, particularly as the gain was achieved without any loss of accuracy. In the following section, we will explore incorporating direct accuracy and efficiency measurement into the training process.



(a) Accuracy vs. grammar size (total productions)



(b) Accuracy vs. exhaustive parsing speed (w/s)

Figure 5.3: Greedy L_0 Objective: accuracy vs. grammar size and speed as λ is varied from 0 (likelihood) to 1 (pure rule count objective). Averaged over 8 grammars, trained on WSJ sections 02-21, and evaluated using exhaustive inference on section 22. At $\lambda = 0.45$, the grammar size is reduced by 69% and speed increased by 20% with no loss in accuracy.

5.5 Inference-Informed Merge Objective

Discriminative models are very effective for many NLP tasks, and are frequently applied to constituency parsing [66, 42, 151], and more specifically, to latent-variable models [145, 146]. Discriminative PCFG training — in contrast to the generative training methods we described in Sections 2.7–2.9 — incorporates inference. Although the training process can be quite expensive, the resulting models are in some cases superior to their generative equivalents. In many cases, that advantage is attributable to the flexibility in feature selection permitted by log-linear discriminative modeling, but performing inference directly into training may also aid in selecting the optimal model structure.

In this section, we incorporate accuracy and efficiency of actual inference (on a development set) into the merge objective function. The cost of that evaluation might be prohibitive if it required exhaustive inference with each merge candidate, but the pruning approaches used throughout this thesis integrate well into the learning process.

The merge process (step 6 in LV grammar learning) thus subdivides into the following steps:

- For each candidate pair:
- Re-merge the candidate split and train a prioritization model on the resulting grammar.[6]
- Parse a sizable development-set, using beam-search guided by the prioritization model and a complete-closure model (trained separately and shared across all training stages).

  Using section 24 as a 32.8k-word development set, this inference requires approximately 40 seconds per merge candidate during cycle 3 (on an Intel Nehalem processor) and 2 minutes per candidate at cycle 6.[7] Since each inference run, and in fact each sentence, is independent, this ranking function would parallelize smoothly across multiple cores or over a large cluster, but accurate comparison of inference speeds requires that the cores or cluster nodes be homogenous and not subject to interference from other processes. To limit potential interference from unrelated

[6] Prioritization model training can be performed effectively on parses constrained by the gold-chart, so this step it is very efficient, usually training in only a few seconds.

[7] Training the pruning models consumes over 70% of that time, and is a logical target for optimization, if training time becomes a bottleneck.

tasks, we limited grammar training tasks to a single node. Overall, grammar training requires approximately 24-36 hours. We can reduce that training cost somewhat further if we use the baseline likelihood-loss estimate to limit the set of merge candidates examined. In general, the splits ranked highest by that criteria are quite important (and should be retained), while the splits ranked lowest will (nearly) always be re-merged. We are attempting to discriminate between the candidates near the 'cutoff', and choose the most appropriate to merge. Thus, we can exclude those candidates near the top and bottom of the likelihood ranking, and measure efficiency only on the 'middle' splits. The trials reported here examine all candidates, but limiting inference to 50% of the candidates produces similar results.

- Rank the potential merges by Equation 5.3 — in this case, $\hat{a}(A_1, A_2)$ and $\hat{e}(A_1, A_2)$ are determined empirically (rather than estimated) over a sizable development set.

Pruning during training will certainly bias the choices of retained splits (vs. those splits which might be chosen were we to perform exhaustive search). However, the pruning models used for inference in this reranking function are very similar to those we normally use when parsing with the final model, so any bias introduced should contribute positively to the pruned performance of the final grammar.[8]

5.5.1 Training With An Inference-Informed Objective

As evidenced in Figure 5.4 and Table 5.3, the inference-informed merge objective appears to be dominated by noise, and did not provide a smooth tradeoff between accuracy and efficiency. We found instead a very small accuracy gain[9] and inference speed approximately flat from $\lambda = 0$ to $\lambda = 1$.

Limited subsequent experiments indicate that this counterintuitive result is at least partially attributable to the relatively small size of the development-set (approximately

[8] We used section 24 as the development set during training, the same section used to tune the complete-closure model. Reusing the same section may produce slightly more accurate cell-closure during the merge-objective inference; this may bias the set of retained splits somewhat but the bias should be toward those splits which perform well under an accurate complete-closure model, which we will generally have during final inference.

[9] A stratified shuffle test finds the accuracy gain significant at the .05 level, but a gain of this magnitude is unlikely to benefit downstream applications.

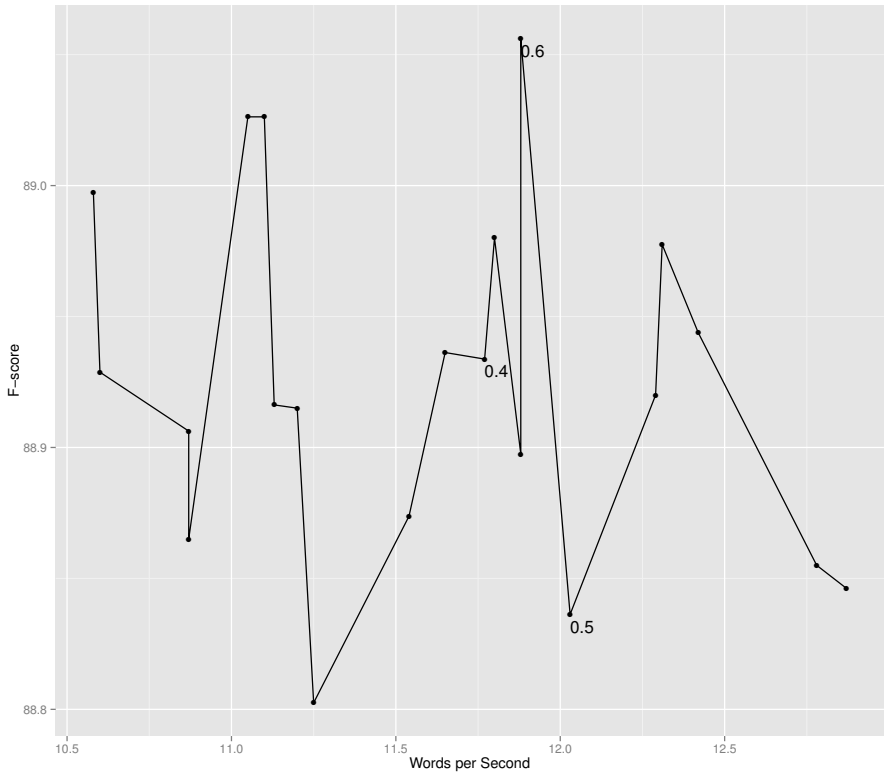


Figure 5.4: Accuracy vs. speed for grammars trained using an inference-informed merge objective. Trained on WSJ sections 02-21 and evaluated on section 22.

λ	**Size** ($\lvert P\rvert$)	**Exhaustive**		**Pruned**	
		$\mathbf{F}_1$ (σ)	**w/s**	$\mathbf{F}_1$ (σ)	**w/s**
0	3131k (57.6k)	89.0 (.19)	10.6	88.8 (.28)	2087
.25	3216k (50.4k)	88.9 (.13)	11.2	88.8 (.20)	2165
.50	3119k (73.1k)	88.8 (.24)	12.0	88.9 (.21)	2206
.75	3116k (95.6k)	88.9 (.32)	12.4	88.9 (.24)	2059
1	3140k (80.9k)	88.8 (.30)	11.3	88.8 (.37)	2069
	60k-sentence dev-set				
0	3585k (78.5k)	89.0 (.22)	10.4	89.0 (.12)	1982
.50	3538k (72.9k)	88.9 (.27)	10.9	88.9 (.25)	2026

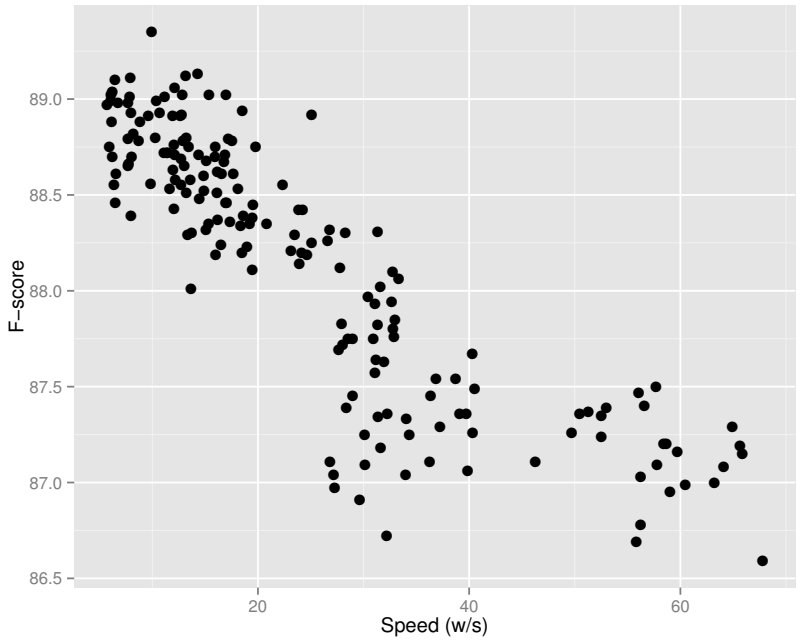
Table 5.3: Selected operating points from Figure 5.4, and trials with a larger development set at two selected operating points.

32k words). Inference speeds are high enough during the early cycles that real efficiency differences between splits are obscured by timing noise, so the splits chosen are not representative of the intended objective. Accuracy measurements during training are similarly obscured by noise, again — we believe — due to the limited dev-set. In fact, although we might anticipate this objective to reduce the variance in accuracy between grammars, we find that variance increases slightly vs. that of the other objectives.

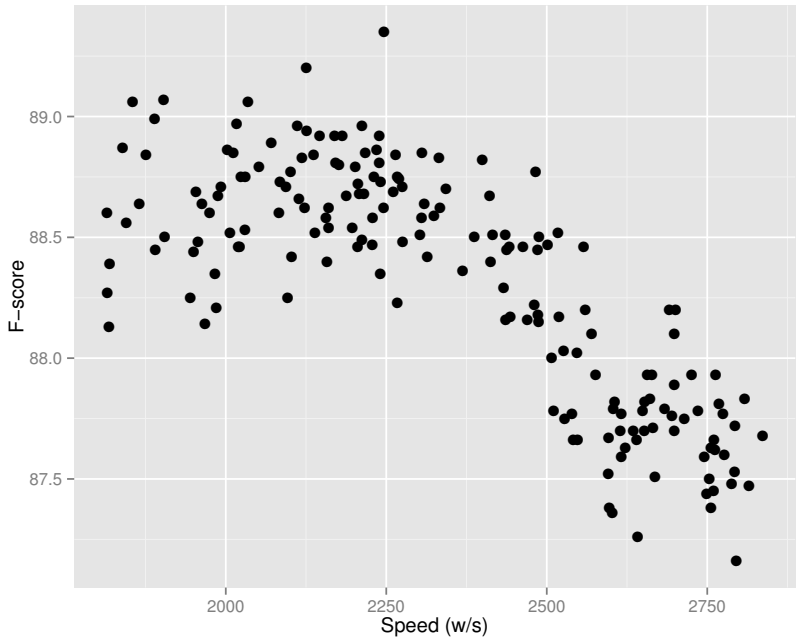
To (partially) address this concern, we combined the full training set with several repetitions of sections 00 and 24, creating an expanded dev-set of approximately 60k sentences and 1.4m words. Evaluating accuracy (during training) on the training set is perhaps not ideal, but should not overfit any more than the standard likelihood metric (also evaluated on the training set), and including the combination of sections 00, 2-21, and 24 produces a more diverse objective than the small dev-sets would alone.

Naturally, performing inference on this larger corpus is considerably more expensive, so we limited these trials to two operating points — $\lambda = 0$ and .5. Table 5.3 reports these trials, as well as selected operating points from the trials which performed inference only on section 24. The larger dev-set makes little difference in either accuracy or efficiency — certainly not enough to warrant the training cost.

These trials certainly do not exhaust the possibile methods of incorporating inference into the split-merge process. Thus far, we have examined each merge candidate in isolation, which may obscure interactions between merged non-terminals. To observe (and perhaps benefit from) those interactions, we could sample random sets of merge candidates, and examine the resulting grammars in their entirety. That is, if we are merging half of all merge candidates from the 'middle' 50% (as ranked by likelihood), we select 50% of that 50%, merge the selected candidates, and perform inference as before. We can then evaluate an entire grammar using same accuracy and efficiency metrics applied previously to an individual merge candidate. Preliminary trials using this method do not demonstrate any improvement over the other inference-informed methods, and the training method is even more costly than direct inference on a large dev-set (well over a month of training on our evaluation cluster). Thus, we will instead move on to considering an alternate method of estimating $\hat{e}(A_1, A_2)$ for merge candidates, a method which can achieve most of the benefits of inference-informed training at a very reasonable cost.



(a) Exhaustive inference



(b) Beam search, guided by a lexical prioritization model and constrained with a complete-closure model

Figure 5.5: Accuracy and efficiency of a large array of latent-variable grammars, trained on WSJ sections 02-21, and evaluated on section 22.

5.6 Modeled Merge Objective

We found in Section 5.4 that penalizing $|P|$ reduces the grammar size greatly, but provides a more modest improvement in parsing speed. This differential is unsurprising, since most grammar productions are lexical rules (nearly 60% in the baseline grammars). Lexical productions do not greatly impact inference time, as they are only accessed when populating span-1 chart cells. But an objective function based on $\Delta_{|P|}(A_1, A_2)$ penalizes all productions equally, overemphasizing the effectiveness of removing a lexical production. In this section, we consider attributes of a grammar other than $|P|$ which impact efficiency of inference. Additional binary rules are are likely to add considerably more cost than lexical rules, since they must be accessed once per midpoint, in each cell ($O(n^3)$).[10] Thus, a more sophisticated estimate of $\hat{e}(A_1, A_2)$ may better predict the efficiency impact of a merge candidate, and lead to greater efficiency gains at a comparable operating point.

To obtain inputs for our model of $\hat{e}(A_1, A_2)$, we began with a large number of latent-variable grammars, trained on WSJ text. We varied the random initialization seeds, the number of training cycles, and the percentage of non-terminal splits merged during each cycle. Figure 5.5 plots the accuracy and efficiency of each of these grammars. In most other accuracy trials reported in this thesis, we report averages over random seeds; in this case, we want to incorporate the observed variance, so Figure 5.5 displays the maximum speed obtained with each grammar, over 5 separate trials. We then created a regression model to predict the observed speeds, using the following grammar attributes:

- V_{Phrase} : The number of phrase-level non-terminals[11]
- V_{POS} : The number of preterminal (part-of-speech) non-terminals
- P_b : Binary rule count
- P_u : Unary rule count
- P_l : Lexical rule count
- $\mu(P_{POS})$: Mean preterminal rule-count (i.e., the number of lexical children of each preterminal)

[10] Unary rules are accessed once per cell, and are thus between the two extremes.

[11] The vocabulary of a PCFG is formally a single set, but in the linguistic grammars of interest here, the phrase-level labels and part-of-speech tags are disjoint sets.

Feature	Pruned		Exhaustive	
	β	p-value	β	p-value
V_{Phrase}	-.997	1.46×10^{-10}	-.135	$< 2 \times 10^{-16}$
V_{POS}	-1.67	1.73×10^{-13}	-	-
P_b	8.12×10^{-4}	$< 2 \times 10^{-16}$	8.813×10^{-5}	$< 2 \times 10^{-16}$
P_u	-	-	2.053×10^{-4}	.0089
P_l	-	-	-4.402×10^{-5}	2.02×10^{-9}
$\mu(p_{POS})$	-	-	.0110	.00028
$\widetilde{D_r}$	-.207	1.26×10^{-8}	-.0102	.00911
$\widetilde{D_c}$	-21.2	3.90×10^{-13}	-1.585	.00191

Table 5.4: Regression coefficients and P-values of the final linear model predicting pruned inference speed. The pruned model accounts for the vast majority of the variance in speed (Adjusted $R^2 = .973$).

- $\widetilde{D_r}$: Median row density of the binary grammar matrix (i.e., the number of child pairs for each phrase-level parent)
- $\widetilde{D_c}$: Median column density of the binary grammar matrix (i.e., the number of parents for each child pair)

With those grammars and appropriate pruning models, we recorded parsing speeds on WSJ section 22. We fit a least-squares linear model predicting those inference speeds using the grammar features listed and the `lm` function in the open-source R statistics package [149]. We found in our initial model that several of the features did not contribute significantly to the model predictions; we omitted those features with P-values $> .05$ (P_u, P_l, and $\mu(P_{POS})$, for pruned parsing). Table 5.4 includes P-values and model coefficients of the final regression models. The pruned-parsing model explains over 97% of the observed variance in parsing speed — $R^2 = .973$ when adjusted for the number of input variables; a similar model for exhaustive parse speeds is not as tight, but still explains 89.7% of the observed variance.

One point of particular interest regarding this linear model is the predicted effect of additional binary rules. Our intuition is that — holding all else equal — adding a binary

λ	Size ($\lvert P\rvert$)	Exhaustive		Pruned	
		$\mathbf{F}_1$ (σ)	w/s	$\mathbf{F}_1$ (σ)	w/s
0	3758k (53.3k)	88.8 (.25)	12.1	88.6 (.21)	2083
.25	3663k (24.5k)	88.8 (.19)	13.0	89.0 (.11)	2095
.45	2637k (202.5k)	88.6 (.31)	19.4	88.7 (.25)	2399
.50	1993k (130.6k)	87.5 (.47)	37.4	87.8 (.43)	2743

Table 5.5: Selected operating points from Figure 5.6.

rule to a PCFG will have greater negative impact on parsing speed than unary or lexical rules. Recall that we generally must intersect with the binary grammar $O(n^3)$ times, whereas access the lexical productions only $O(n)$ times — once for each span-1 cell — and the unary productions once in each cell ($O(n^2)$). Thus, we find it somewhat surprising that the coefficient for P_b is positive — i.e., the model indicates that adding binary rules to a grammar *increases* parsing speed.[12] Clearly, that cannot be true in the extreme, but — within the relatively narrow range of grammars we examined — the other factors have greater impact on efficiency. Note that $\widetilde{D_r}$ and $\widetilde{D_c}$ encode other attributes of the binary ruleset, and are thus positively correlated with $|P_b|$. However, even that correlation does not completely explain the unexpected result. The coefficient for $|P_b|$ remains positive for a model excluding $\widetilde{D_r}$ and $\widetilde{D_c}$. In fact, the same held true for any model except one fit on *only* $|P_b|$, which produces the expected negative coefficient (and R^2 of .851).

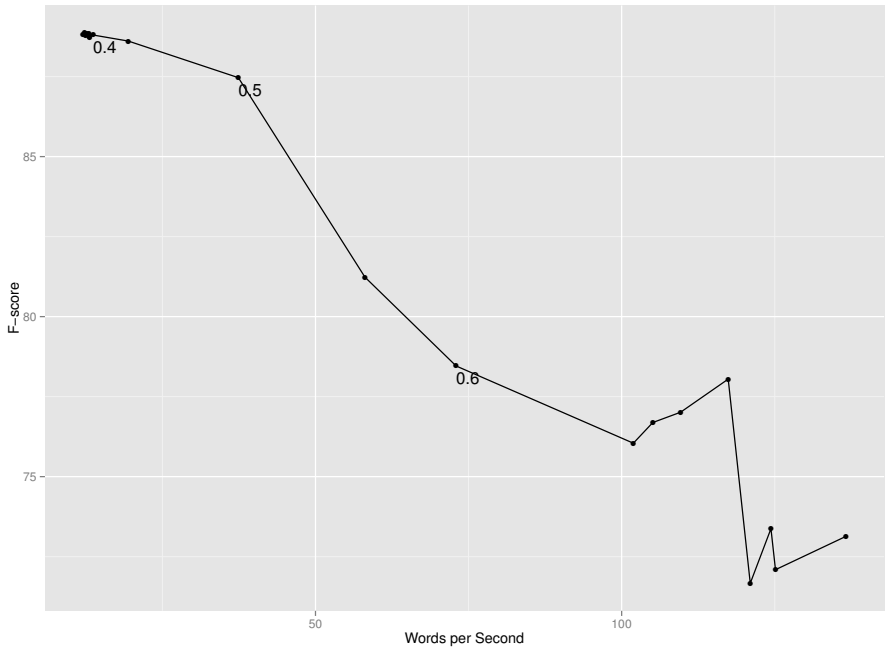
5.6.1 Training With A Modeled Merge Objective

We then used the linear model described in Table 5.4 as $\hat{e}(A_1, A_2)$, predicting the efficiency impact of each candidate merge. That is:

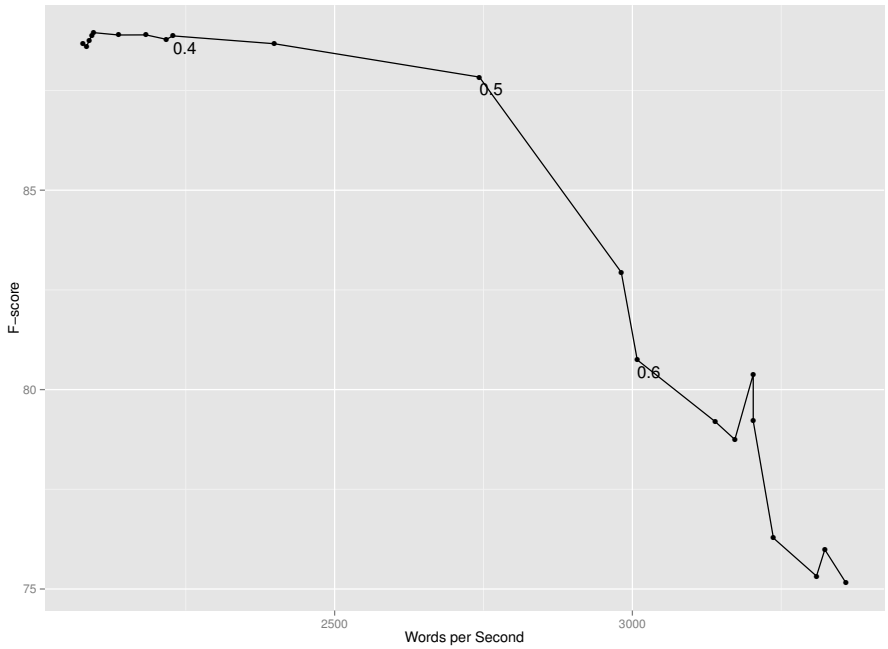
$$\hat{e}(A_1, A_2) = -.951 \cdot V_{Phrase} - 2.02 \cdot V_{POS} + .00082 \cdot P_b - .356 \cdot \widetilde{D_r} - .125 \cdot \widetilde{D_c} \qquad (5.5)$$

As in Section 5.4, we retained Equation 5.2 as $\hat{a}(A_1, A_2)$. Figure 5.6 plots size and accuracy of the resulting grammars, over a variety of operating points, and Table 5.5 presents size, accuracy, and inference speed for selected operating points along those curves. Grammar training guided by this linear model improves over the rule-count objective at nearly every

[12] The two models coefficients differ, but the directionality of this effect is consistent between the two.



(a) Exhaustive inference



(b) Pruned inference

Figure 5.6: Accuracy vs. speed for grammars trained using a modeled merge objective. Trained on WSJ sections 02-21 and evaluated on section 22.

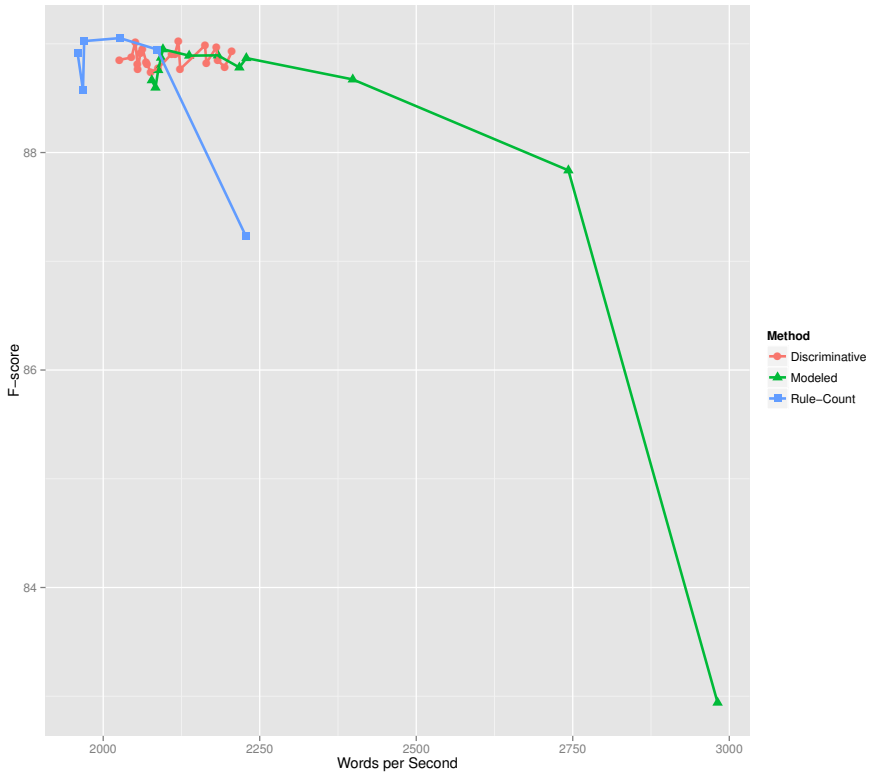


Figure 5.7: Accuracy vs. speed each merge objective function. The modeled objective provides the greatest speed gain before accuracy begins to fall off. Trained on WSJ sections 02-21 and evaluated on section 22.

operating point. We find a similar relationship between accuracy and the efficiency prioritization parameter λ as that in Table 5.2, but the efficiency gains are considerably greater for the modeling approach. Overall, we find this method yields a meaningful efficiency improvement with little loss in accuracy, providing an advantageous and controllable tradeoff between accuracy and efficiency.

5.7 Results and Discussion

Figure 5.7 compares the merge objectives of sections 5.4–5.5, again on WSJ section 22. Note that the inference-informed results are all clustered within a narrow range, and that the modeled objective provides a much faster operating point than the rule-count objective before accuracy begins to decline. In fact, the modeled objective can be pushed near 3000

Grammar	$\|P\|$	Exhaustive		Pruned	
		$\mathbf{F}_1$ (σ)	w/s	$\mathbf{F}_1$ (σ)	w/s
WSJ					
Baseline	4215k (40.6k)	88.7 (.17)	9.5	88.5 (.30)	1910
Uniform Pruning	2581k (37.9k)	88.7 (.26)	16.6	88.7 (.15)	2838
Rule-count Obj.	2450k (33.6k)	88.8 (.12)	9.7	88.9 (.15)	1950
Modeled Obj.	3160k (97.2k)	88.7 (.23)	14.0	88.8 (.21)	2238
Switchboard					
Baseline	2406k (20.0k)	86.6 (.17)	7.8	86.8 (.34)	1633
Uniform Pruning	1096k (10.3k)	86.8 (.11)	12.4	87.1 (.07)	2398
Rule-count Obj.	1521k (45.7k)	86.6 (.17)	7.6	87.3 (.13)	1721
Modeled Obj.	1899k (57.0k)	86.9 (.22)	10.0	87.2 (.19)	1712
Chinese					
Baseline	3073k (103.1k)	80.6 (.46)	3.5	79.3 (.60)	744
Uniform Pruning	1933k (39.3k)	80.8 (.23)	4.8	79.9 (.77)	1152
Rule-count Obj.	1909k (69.2k)	80.6 (.46)	3.5	80.4 (.43)	742
Modeled Obj.	2558k (73.7k)	81.0 (.55)	5.6	80.1 (.78)	811

Table 5.6: Test-set trials of grammars trained with selected grammar training methods from this chapter. All grammars trained merging 55% of non-terminal splits at each cycle.

words/second, although we could almost certainly obtain a comparable speed — and higher accuracy — with a 4- or 5-cycle grammar.

Table 5.6 presents test-set trials on WSJ, Switchboard, and Chinese. We trained uniform pruning grammars at $\epsilon = 10^{-6}$, and the rule-count and modeled objectives at a relatively conservative λ of .4 (we omitted the inference-informed objective from these trials, as the modeled objective outperformed it handily). For all genres, we find that uniform parameter pruning and the modeled objective each provide a considerable efficiency gain with little or no accuracy degradation. We will present trials combining these methods, along with those from Chapter 4, in Chapter 7.

In Table 5.7, we present similar trials, applying grammars trained on WSJ text with a modeled merge objective to the English Web Treebank (analogous to the experiments in Section 4.4.1). We observed a speed improvement comparable or greater than that

Genre	**Baseline**		**Modeled**	
	F_1 (σ)	w/s	F_1 (σ)	w/s
Email	74.0 (.30)	926	73.9 (.48)	1277
Weblogs	76.0 (.20)	339	76.1 (.46)	475
Reviews	74.8 (.28)	271	74.7 (.47)	386
Newsgroups	72.9 (.28)	250	72.9 (.41)	341
Answers	73.7 (.36)	1105	73.7 (.37)	1238

Table 5.7: Accuracy and speeds of modeled grammars on the English Web Treebank test sets. All grammars trained on WSJ text, without domain adaptation. Inference performed using a complete-closure model [21] and beam-search guided by a lexical prioritization model [19].

obtained for in-domain data, again with no loss in accuracy.

In summary, we presented several novel methods of optimizing LV grammars for efficiency, yielding speedups of up to 75% and 49% respectively for exhaustive and pruned inference. We found that we could obtain these efficiency gains with little or no loss in accuracy, and that the results are consistent across several genres. In Chapter 7, we will combine some of the methods from this chapter with those from chapters 4 and 6, yielding further gains in both efficiency and accuracy.

Chapter 6

Chart Decoding Methods

Using the methods presented in previous chapters, we can populate a chart structure very quickly, encoding a packed parse forest in the chart (see Figure 6.1, repeated here from Chapter 2 for convenience). We must then extract a target tree from that forest, and the decoding method chosen has considerable impact on both accuracy and efficiency. In this chapter, we will describe several decoding methods, compare them on both exhaustive and pruned inference, and present approximation methods that improve efficiency with minimal impact on accuracy.

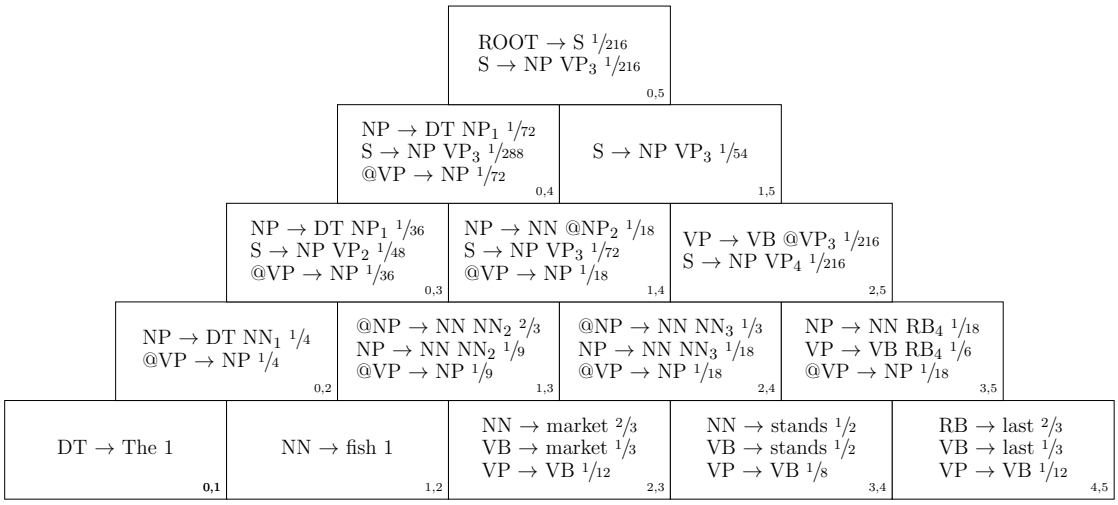


Figure 6.1: A CYK chart populated with a packed parse forest — a compact representation of many unique parse trees.

6.1 Viterbi Decoding

The simplest and most common decoding method finds the most probable complete tree populated in the forest $\mathcal{F}$. That is, we maximize:

$$\hat{y} = \operatorname*{argmax}_{y \in \mathcal{F}} P(y|G) \tag{6.1}$$

We often call this method *Viterbi* decoding, because of the obvious correlation to the Viterbi algorithm for finding the single best path through a Hidden Markov Model [178]. As described in Section 2.3, we often maintain as part of the chart structure, a set of backpointers ζ, recording the highest scoring path to each observed non-terminal. Given these backpointers, we simply start with the start symbol $S^\dagger$ in the topmost cell, and recurse downward through the start, following the backpointers and constructing the maximum-likelihood tree $\hat{y}$. If we choose to omit ζ during the initial pass, we can again recurse downward from $S^\dagger$, but we must recompute the highest-scoring path for each non-terminal by re-intersecting with the grammar.

6.2 Approximate Minimum-Bayes-Risk Decoding

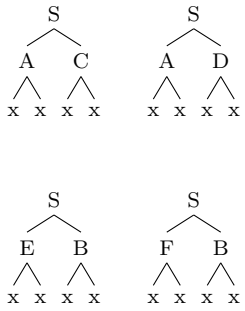
Minimum-Bayes-Risk (MBR) decoding attempts to find the candidate hypothesis which minimizes the expected loss according to the evaluation criteria. MBR methods are widely used in speech recognition [72] and machine translation [104], and have been shown to improve performance in both domains. The standard F_1 parse evaluation metric (described in Section 2.4) averages precision (LP) and recall (LR) of span labels, so minimizing risk under this evaluation must balance expected precision and recall.

Goodman [74] proposed a *max-recall* decoding method, which properly belongs to a family of methods we call *approximate minimum-Bayes-risk* (AMBR) methods. Figure 6.2, borrowed from Goodman, demonstrates that this *max-recall* method is able to produce parse trees which recover more correct nodes than the Viterbi parse, even if the resulting tree is not permitted by the grammar. Note: because standard PARSEVAL metrics do not score factored non-terminals or unary productions, true MBR is ill-defined — we can always improve recall by adding another unary production. The methods we describe are close approximations of true MBR, with bounds on unary chain length. The inside-outside algorithm described in Section 2.6 computes the posterior probability $\gamma(X)$ of each labeled span. We can use those posterior probabilities to optimize LR, by maximizing $\gamma(X)$.

Grammar

S	→	A	C	0.25
S	→	A	D	0.25
S	→	E	B	0.25
S	→	F	B	0.25
A,B,C D,E,F	→	x	x	1.0

Permitted Trees



Maximum-Recall Tree

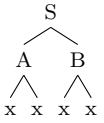


Figure 6.2: AMBR example from [74]. Trees are scored by the number of expected correct labels. Each permitted tree scores $1 + .5 + .25 = 1.75$. Although it is not permitted by the grammar, the maximum-recall tree scores $1 + .5 + .5 = 2$.

$$\max(LR) = \underset{T \in \tau}{\operatorname{argmax}} \sum_{X \in T} (\gamma(X)) \tag{6.2}$$

Because this method optimizes recall of scored non-terminals, it will tend to predict non-factored labels, even in contexts where a factored label (and an n-ary branching structure rather than a strictly binary one) is more appropriate, and will overpredict unary chains. We can introduce a term to penalize this spurious overgeneration, and optimize LP instead:

$$\max(LP) = \underset{T \in \tau}{\operatorname{argmax}} \sum_{X \in T} (\gamma(X) - 1) \tag{6.3}$$

Optimizing LP yields the opposite problem, overpredicting factored non-terminals (producing very flat trees) and by avoiding prediction of unary productions. However, by varying the overgeneration penalty, we can choose an operating point on the continuum between $max(LP)$ and $max(LR)$.[1]

$$\hat{T} = \underset{T \in \tau}{\operatorname{argmax}} \sum_{X \in T} (\gamma(X) - \lambda) \tag{6.4}$$

At $\lambda = 0$, this is equivalent to Equation 6.2, and at $\lambda = 1$ to Equation 6.3. $\lambda = 0.5$ balances expected recall and precision equally, minimizing the expected loss according to the F_1 evaluation metric.

[1] See Appendix A of Hollingshead and Roark [83] for the full derivation.

Inference with these methods is a 4-step process (in contrast to the 2-step process of Viterbi search):

1. An *inside* pass, proceeding upwards through the chart and computing the inside probabilities α of each labeled span.
2. A downward *outside* pass, computing the analogous outside probabilities β. Note that the combination of steps 1 and 2 is a single cycle of the inside-outside algorithm from Section 2.6, applied during inference rather than grammar induction.
3. An upward *maximization* pass, computing the argmax at each span and recording backpointers (similar to the computations in the upward pass of Viterbi decoding)[2]
4. A downward *decoding* pass, which follows those backpointers and extracts the tree

Most grammars of interest split their non-terminal spaces, but downstream processes (and thus our evaluation metrics) generally only consider the base (unsplit) non-terminals. We can choose to perform the AMBR argmax over the split nonterminals or while summing over splits of each base label. That is, in the maximization pass, we can accumulate scores for each non-terminal (e.g. NP_0, NP_1, ...), or we can instead accumulate a single score for NP. We refer to these approaches as AMBR-Max and AMBR-Sum; we find it unclear apriori which choice is superior, so we present results for each in Section 6.4.

6.3 Max-Rule Decoding

Petrov and Klein [144] presented a closely-related approach they call *Max-Rule* decoding. This method alters the maximization step, optimizing the number of expected correct *rules* (rather than the number of expected correct labels). Inference follows the same 4-pass process as the AMBR methods, performing inside and outside probability calculations, maximization, and finally decoding. However, instead of maximizing expectations over labels, we instead perform the argmax over grammar rules:[3]

$$r(A \rightarrow BC, i, k, j) = \sum_x \sum_y \sum_z \beta(A_x, i, j)\, \alpha(B_y, i, k)\, \alpha(C_z, k, j)\, P(A_x \rightarrow B_y C_z) \quad (6.5)$$

$$\hat{T} = \underset{T \in \tau}{\operatorname{argmax}} \prod_{X \in T} r(X) \quad (6.6)$$

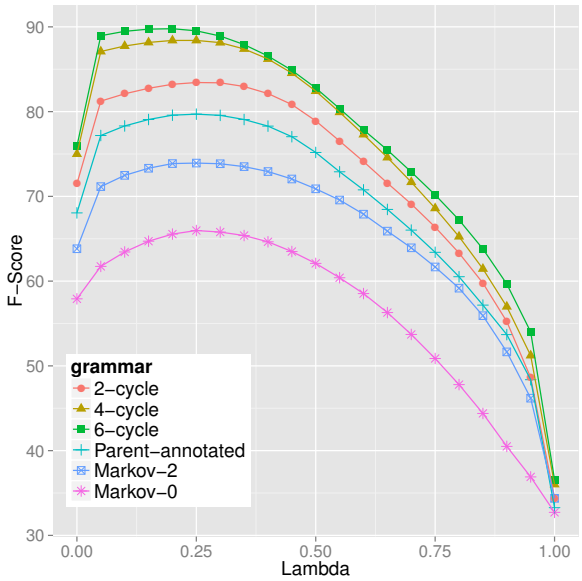
[2] See Goodman [74] for a full description of the maximization algorithm.

[3] This maximization function is taken directly from Petrov and Klein [144], with notation modified to match that in Section 2.6.

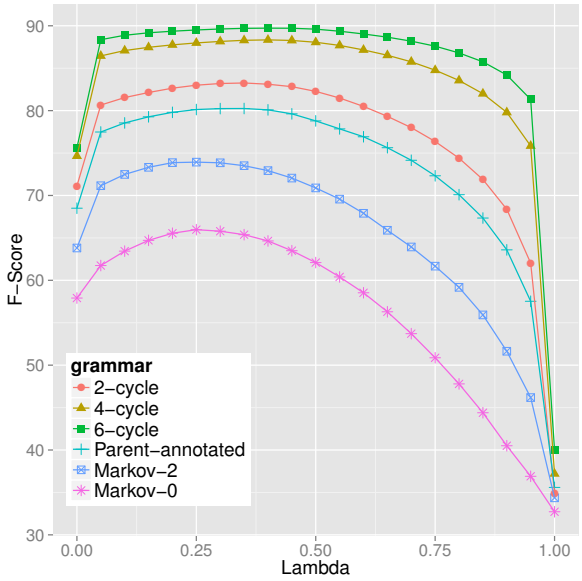
	V_{phrase}	V_{POS}	P_b	P_u	P_{lex}
WSJ					
M-0	53	46	4240	236	52k
M-2	2870	46	13.2k	236	52k
Parent	6600	46	24.5k	736	52k
2-cycle	112 (0.9)	106 (0.7)	21.9k (539)	3639 (119)	205k (39)
4-cycle	244 (4.0)	248 (1.0)	166k (3.3k)	19.9k (1k)	747k (6k)
6-cycle	540 (19.2)	579 (6.6)	1.25m (42k)	100k (7k)	2.4m (40k)
SWBD					
2-cycle	124 (0.9)	100 (0.7)	26.7k (681)	6.2k (107)	81k (2.0k)
4-cycle	280 (2.6)	227 (2.8)	174k (5.6k)	34.4k (851)	264k (2.0k)
6-cycle	664 (12.1)	480 (5.8)	1.1m (41k)	171k (6.3k)	742k (9.4k)
PCTB					
2-cycle	102 (0.8)	75 (0.8)	11.6k (245)	3.6k (157)	165k (1.5k)
4-cycle	229 (2.1)	166 (2.1)	110k (3.4k)	16k (640)	556k (11k)
6-cycle	558 (3.7)	327 (3.7)	966k (18k)	80k (2.5k)	1.8m (44k)

Table 6.1: Summary statistics of WSJ, Switchboard, and Chinese grammars, including vocabulary and rule-count distributions (binary, unary, and lexical). Statistics for the Markov-0, Markov-2, and Parent-annotated grammars are repeated from Table 3.2. Statistics for latent-variable grammars are averaged over 8 grammars trained with different random seeds, and include standard deviations in parenthesis.

Petrov [144] also explored a related optimization that maximizes the *sum* of rule scores in the tree, but found the product formulation superior, so we will constrain our experiments to the latter. Note that computing r requires revisiting the grammar rules during the maximization pass, a potentially expensive operation. In the next section, we compare and contrast the behaviors of these decoding methods, evaluated on a wide range of grammars.



(a) AMBR-Max



(b) AMBR-Sum

Figure 6.3: Development-set accuracy of the AMBR approaches over λ values from 0–1. Evaluated on WSJ section 22. Note that AMBR-Sum is much more robust to the choice of λ, while AMBR-Max performance is more peaked (with the exception of the Markov-0 grammar, which does not split nonterminals, and thus has no splits to sum over).

	M0	M2	Parent	2-cycle	4-cycle	6-cycle
Beam width	25	100	200	100	150	200
Viterbi	61.1	72.2	78.1	81.9 (.43)	87.4 (.21)	88.8 (.13)
AMBR-Max	66.0	73.9	79.7	83.4 (.27)	88.4 (.17)	89.8 (.20)
AMBR-Sum	66.0	73.9	80.3	83.3 (.30)	88.3 (.25)	89.7 (.19)
Max-Rule	64.4	73.5	79.6	83.5 (.23)	88.7 (.16)	90.3 (.15)

Table 6.2: WSJ development-set accuracies of each decoding method, using near-exhaustive inference (beam widths selected to minimize pruning error). Results for latent-variable grammars are averages over 8 grammars trained with different random seeds, and include standard deviations in parenthesis. AMBR approaches use a λ, selected to maximize accuracy on the same dev-set.

	2-cycle	4-cycle	6-cycle
Beam width	100	150	200
Viterbi	80.5 (.62)	86.4 (.21)	87.2 (.16)
AMBR-Max	82.5 (.50)	87.7 (.14)	88.4 (.14)
AMBR-Sum	81.7 (.70)	87.5 (.19)	88.4 (.22)
Max-Rule	81.7 (.49)	87.8 (.15)	88.7 (.20)

Table 6.3: Switchboard development-set accuracies of each decoding method, using near-exhaustive inference. AMBR approaches use the same λ values as Table 6.2.

6.4 Accuracy Evaluation

We evaluated each decoding method on a variety of genres and grammars.[4] Table 6.1 shows summary statistics of each grammar. To execute these trials in reasonable time, we executed a beam search, but with a beam determined empirically to be large enough to minimize search errors. These trials are labeled as near-exhaustive, since they produce results indistinguishable from exhaustive inference.

[4]Note: As discussed in Section 2.13, accuracy evaluations on the PCTB's small development and test sets may be quite noisy.

	2-cycle	4-cycle	6-cycle
Beam width	100	150	200
Viterbi	77.5 (.46)	82.7 (.65)	83.3 (.32)
AMBR-Max	79.7 (.39)	84.1 (.27)	84.1 (.29)
AMBR-Sum	79.7 (.39)	84.3 (.35)	84.8 (.29)
Max-Rule	79.4 (.56)	84.8 (.41)	85.4 (.34)

Table 6.4: Chinese development-set accuracies of each decoding method, using near-exhaustive inference. AMBR approaches use the same λ values as Table 6.2.

6.4.1 AMBR

We begin by examining the behavior of both AMBR variants at various operating points. As discussed in Section 6.2, we expect to balance precision and recall (and thus maximize F_1) at $\lambda = .5$. However, as demonstrated in Figure 6.3, the peak operating points for both variants and all grammars are found at $\lambda < .5$. Maximum-recall parsing ($\lambda = 0$) does not penalize additional label predictions, so the maximum score would be obtained by predicting infinite unary chains (and a fully binary-branching tree). Since an infinite chain is impractical, an implementation must bound the size of unary chain predictions. Our implementation is more flexible than some (e.g., the Berkeley Parser allows only a single level of unary parents at any span). We allow longer chains, but disallow repetitions within the chain (i.e., PRN $\rightarrow$ S $\rightarrow$ VP $\rightarrow$ VB is permitted, but NP $\rightarrow$ NP... and VP $\rightarrow$ S $\rightarrow$ VP... would be disallowed). Thus, the maximum length of a permitted unary chain is $|V_{Phrase}| + 1$ (one entry in the chain for each phrase-level non-terminal and a single preterminal). In practice, the sparsity of the unary grammar matrix constrains the length of these chains, and most observed chains are much shorter. This inherent bias toward shorter unary chains increases the observed performance near $\lambda = 0$ and shifts the peak operating point away from .5.

As demonstrated in Table 6.2, the two AMBR variants perform very similarly to each other, providing a consistent gain over Viterbi search for all grammars. Figure 6.3 demonstrates that AMBR-Sum is much less sensitive to the choice of λ, providing near-optimal results from approximately .25–.6 on the larger grammars. Since the peak operating points are similar, the greater stability of AMBR-Sum makes it a better choice for practical applications. In the remainder of our trials, we report only this variant.

6.4.2 Max-Rule

Our expectation was that the AMBR approaches, which directly maximize a criteria closely related to the evaluation metric, would outperform other decoding methods. However, max-rule shows further gains, improving over AMBR-Sum by .1 (for Switchboard) to .5 (for WSJ and Chinese). Max-rule decoding clearly makes better use of the same posterior probabilities used in the AMBR variants. One note of potential interest: In previous work using a single latent-variable grammar, we found that max-rule decoding under the Berkeley Parser's coarse-to-fine pruning system produced a similar gain [60]. This gain was in line with previously-reported results [144]. In this work, we tested numerous grammars trained with our variant of the Berkeley trainer (see Section 5.3). Those grammars are incompatible with the Berkeley inference system, so we did not attempt to replicate those earlier results on CTF pruning, but we expect the performance would be comparable with the Berkeley CTF pruning system.

6.4.3 Error Analysis

Although F_1 is a useful measure of overall parsing performance, many downstream applications benefit from a more granular evaluation, incorporating information about the linguistic errors present in parser output. To our knowledge, the first broad-based study of such errors (at least on modern parsers) was that of Kummerfeld et al. [105, 106]. We evaluated the output of each decoding method using the tools they developed in that work; Table 6.5 compares the syntactic error patterns of each objective.[5]

For many of the classic parsing errors — including Clause Attachment; Coordination; and Modifier-, NP- and PP- Attachment, AMBR decoding outperforms Max-Rule. However, it severely underpredicts unary productions. The gold trees in this development set contain 7489 unary productions (combining the two subclasses). Viterbi search and MaxRule decoding both recover a comparable number (7436.8 and 7353.0, respectively); AMBR decoding recovers only 6604.0. In the detailed analysis in Table 6.5, unaries are split across 2 categories — 'Single Word Phrase', which includes unary parents of preterminals, and 'Unary', encompassing all other unary productions. In combination, the two account for an average of 383 additional errors in AMBR decoding vs. Viterbi, and 474

[5]The 'UNSET *' categories contain bracketing errors which were not attributed to specific syntactic categories. The underlying causes of those errors are widely varied, and the differences between decoding methods are fairly small, so we will not attempt to analyze these unattributed errors.

Error Category	**Viterbi**	**AMBR-Sum**	**Max-Rule**
Clause Attachment	802.3	578.1	725.8
Coordination	598.1	499.0	536.3
Different label	550.3	461.5	478.3
Modifier Attachment	574.1	502.9	522.3
NP Attachment	516.6	402.4	438.4
NP Internal Structure	424.9	401.0	395.8
PP Attachment	1506.8	1284.9	1302.8
Single Word Phrase	551.0	630.1	500.9
UNSET add	156.1	226.3	130.0
UNSET move	242.3	219.6	221.8
UNSET remove	173.1	195.3	133.0
Unary	326.0	630.6	285.9
VP Attachment	194.8	233.9	173.3
XoverX Unary	1.0	1.0	1.0
Total	6617.2	6266.5	5845.1

Table 6.5: Bracketing errors attributed to various syntactic error classes, as measured on WSJ section 22 by the Berkeley Parser Evaluator. Averaged over 8 6-cycle grammars. Measured using complete-closure and guided beam search; near-exhaustive results are similar. Error categories are described inKummerfeld et al. [105].

vs. Max-Rule. Note that the additional unary errors more than account for the total difference between AMBR and Max-Rule; if AMBR's unary performance were improved to match that of Max-Rule, it would outperform Max-Rule overall.

Although Max-Rule and AMBR utilize the same posterior probabilities (of non-terminal labels at each span), those posteriors combine probabilities of binary and unary productions. That is, if a non-terminal X has probability 1/3 of participating in a parse tree as a binary parent at a particular span, and 1/6 as a unary parent (at the same span), the posterior will combine those, recording only probability 1/2 for X at that span. Our Max-Rule implementation follows that of the Berkeley Parser in computing unary probability sums separately during decoding, and limiting each span to a single unary production. Thus, revisiting the grammar during decoding serves to separate those two sources of probability mass. This opens a possibility for further work in decoding, combining the

advantages of AMBR with those of Max-Rule — potentially improving accuracy and/or avoiding the expense of grammar intersection during decoding.

6.5 Relationship with Pruning

Thus far, we have focused on near-exhaustive inference, which produces a very dense chart, in which nearly every cell is populated with many non-terminals. Pruned search can produce a much sparser chart, and may in some cases interact with subsequent decoding in interesting and unpredictable ways.

For example, consider a simple beam search. Since Viterbi decoding retains only the maximum-scoring non-terminal in each cell, beam search does not impact the resulting tree, unless a non-terminal which would participate in that tree is mistakenly pruned from the chart. Inside-outside chart population, however, accumulates probability sums, so even non-terminals which do not participate in the final tree may contribute materially to the chart scores. Thus, the alternate decoding methods we are examining may behave much differently under beam search.

Cell-closure approaches, as described in Appendix A and implemented in the BUBS parser, classify certain cells as highly unlikely to participate in the target parse, and do not populate those *closed* cells. Unless a cell participating in the maximum-likelihood tree is closed, these cell-closure methods have no effect on Viterbi search, but the reduction in summed inside and outside scores may impact alternate decoding methods.

Conversely, the alternate decoding methods may in some cases be more robust to search errors than Viterbi search. For example, if the complete-closure model closes a cell which would otherwise participate in the maximum-likelihood tree, Viterbi search may fail to find an alternate solution — in which case the parse fails — whereas the summed probabilities of inside-outside methods can discover alternatives. The standard `evalb` bracket-evaluation tool ignores parse failures. A failed parse may have considerable detrimental impact on a downstream application, so for all evaluations in this chapter, we use a variant which penalizes recall in the event of a parse failure, thus demonstrating the overall impact of beam width on accuracy.

Table 6.6 compares the accuracy and speed of each decoding method under a variety of pruning conditions, including complete-closure [21], beam search, and the combination thereof. We chose beam search parameters that minimize parse failures, even under the

more susceptible Viterbi search. We found the effects of the beam search and complete-closure vary considerably between grammars and genres. In some cases (particularly with smaller grammars), the pruning models supply enough additional information to improve accuracy over unpruned search; when accuracy is degraded (as with the largest WSJ grammars, the loss is typically fairly small, and the improvement in efficiency is quite large).

Note also that — even with heavily pruned search — the accuracy gains of alternate decoding methods come at a considerable efficiency cost vs. Viterbi decoding (more than 10× in most cases). In the following sections, we will consider several methods of alleviating that cost.

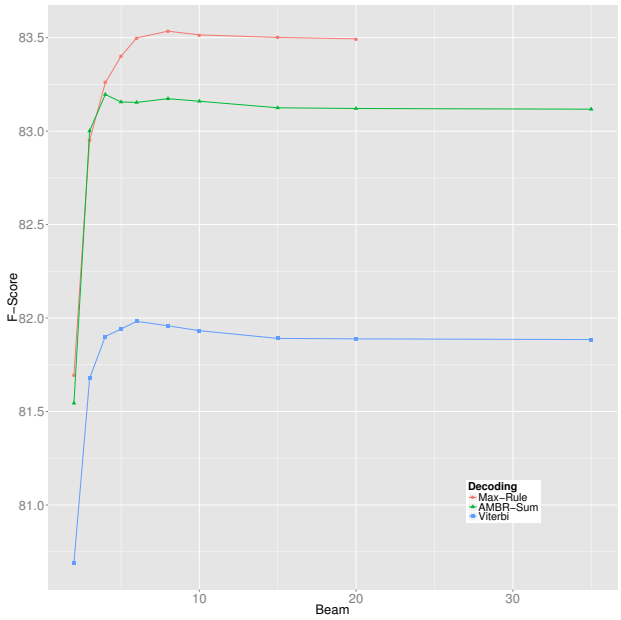
6.6 Efficient Approximations

The results in the previous sections indicated consistent accuracy gains from alternate decoding methods, but as shown in Tables 6.6–6.8, the accuracy improvements come at a steep computational cost. In this section we will consider several less-costly approximations, in hopes of finding an efficient decoding method that provides an accuracy boost over simple Viterbi decoding without incurring the computational cost of a full inside-outside computation.

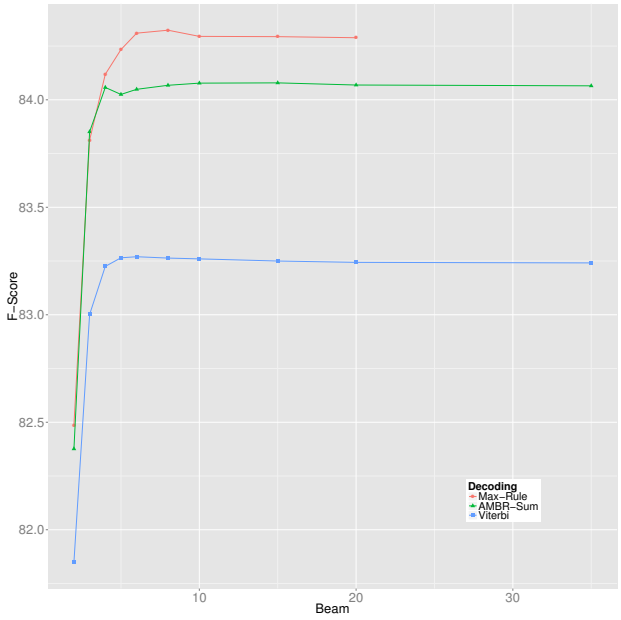
To avoid numeric underflow during inference, we generally represent observed non-terminal probabilities in the log domain ($log_e(P)$). Viterbi search requires only the maximum inside probability for each non-terminal, so we can use the Tropical semiring and accumulate logs very efficiently with simple arithmetic sums. For example, if computing the probability of label A given the children B and C ($P(A) = P(B) \times P(C) \times P(A \rightarrow BC)$), we can simply add the relevant log probabilities:

$$log(P(A)) = log(P(A \rightarrow BC)) + log(P(B)) + log(P(C))$$

This operation is handled very efficiently by modern hardware. However, the inside-outside algorithm (Equation 2.5) operates in the real semiring instead, summing probabilities over all paths. This requires that we accumulate log sums:

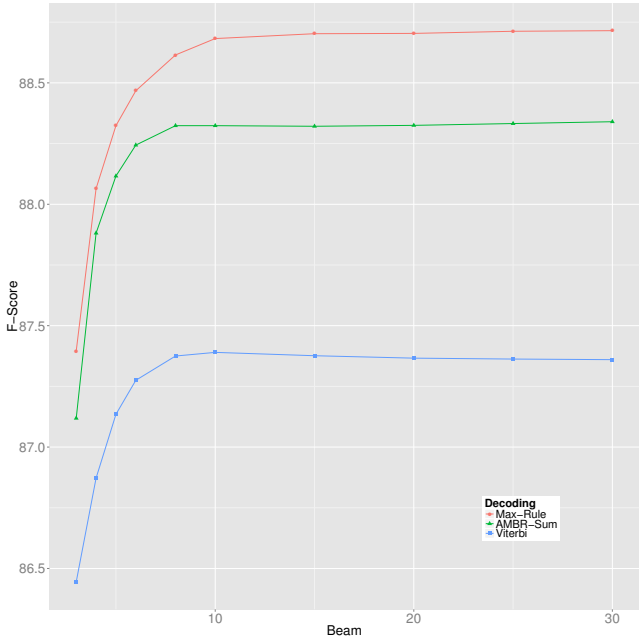


(a) Guided beam search

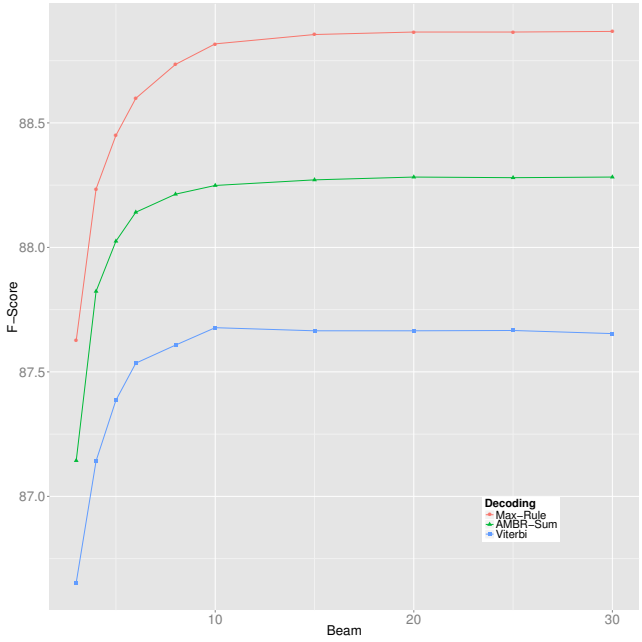


(b) Guided beam search and complete closure

Figure 6.4: Accuracies of 2-cycle LV grammars at various pruning thresholds. Executed on WSJ section 22, using a lexical prioritization model and λ selected to maximize accuracy on that section. Evaluated with a variant of the standard `PARSEVAL` that penalizes recall in the event of a parse failure.

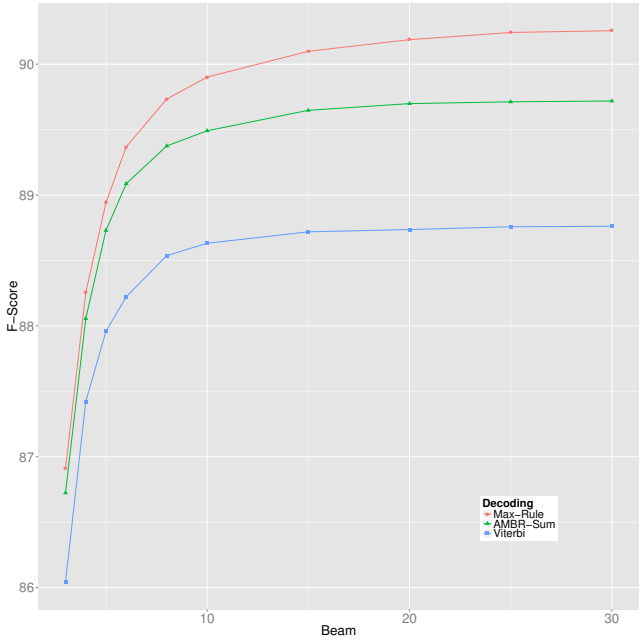


(a) Guided beam search

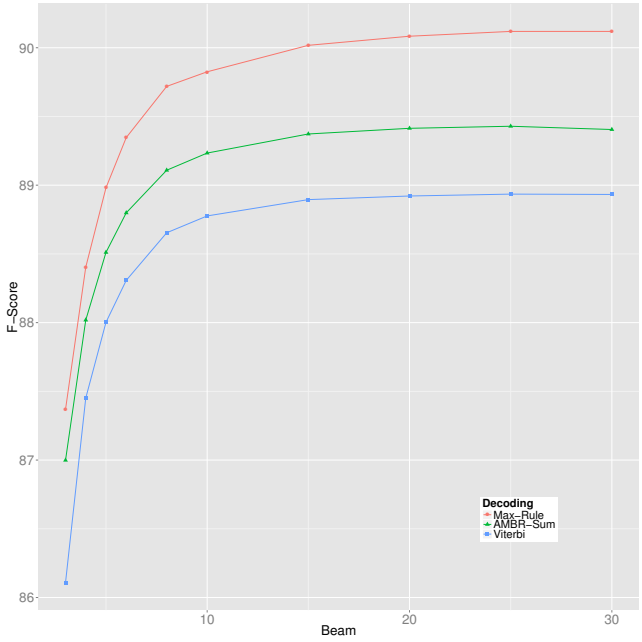


(b) Guided beam search and complete closure

Figure 6.5: Accuracies of 4-cycle LV grammars, using the same experimental settings as Figure 6.4.



(a) Guided beam search



(b) Guided beam search and complete closure

Figure 6.6: Accuracies of 6-cycle LV grammars, using the same experimental settings as Figure 6.4.

	2-cycle		4-cycle		6-cycle	
	F_1 (σ)	w/s	F_1 (σ)	w/s	F_1 (σ)	w/s
Viterbi						
Near-Exhaustive	81.9 (.43)	81	87.4 (.21)	24.4	88.8 (.13)	11.5
Complete-Closure	83.0 (.29)	979	87.8 (.16)	279	89.0 (.10)	114
Beam-Search	82.0 (.40)	2473	87.4 (.21)	494	88.8 (.13)	192
Beam-Search+CC	83.3 (.29)	6009	87.7 (.09)	2210	88.9 (.13)	859
AMBR-Sum						
Near-Exhaustive	83.3 (.30)	11.4	88.3 (.27)	3.4	89.7 (.19)	1.4
Complete-Closure	84.1 (.27)	74	88.2 (.22)	30.7	89.4 (.20)	10.9
Beam-Search	83.2 (.31)	267	88.3 (.24)	42.0	89.7 (.17)	15.1
Beam-Search+CC	84.0 (.31)	1029	88.3 (.23)	211	89.4 (.16)	65.4
Max-Rule						
Near-Exhaustive	83.5 (.23)	6.0	88.7 (.17)	1.6	90.3 (.15)	0.8
Complete-Closure	84.3 (.19)	67	88.9 (.17)	16.2	90.1 (.13)	6.6
Beam-Search	83.5 (.21)	203	88.7 (.17)	29.5	90.3 (.16)	11.7
Beam-Search+CC	84.3 (.22)	739	88.9 (.19)	146	90.1 (.14)	50.9

Table 6.6: WSJ development-set accuracies and speeds of each decoding method on latent-variable grammars, using various pruning approaches. Accuracies averaged over runs with 8 grammars trained using different random seeds. Beam search guided by a lexical prioritization model. Beam widths (6, 20, and 30) chosen to minimize accuracy impact, and AMBR approaches use $\lambda = .35$, selected to maximize accuracy on the same dev-set.

$$log(P(A)) = log(e^{log(\alpha_A)} + e^{log(P(B))+log(P(C))+log(P(A \rightarrow BC))})$$

The repeated exponentiations and logarithms of this so-called `logsumexp` operation are very expensive. Even on modern processors, these are expensive computations, and our profiling indicated that it consumes the vast majority of the inference time (and accounts for the considerable speed penalty of max-rule vs. AMBR, since max-rule recomputes probability sums during the maximization pass). Note that this CPU-bound processing is in contrast to Viterbi inference, which (as demonstrated in Chapter 3) is generally memory-bound. Thus, we focus most of our optimization methods on the `logsumexp`

	2-cycle		4-cycle		6-cycle	
	F_1 (σ)	w/s	F_1 (σ)	w/s	F_1 (σ)	w/s
Viterbi						
Near-Exhaustive	80.5 (.62)	67.7	86.4 (.21)	28.5	87.2 (.14)	14.9
Complete-Closure	82.2 (.31)	456	86.9 (.17)	181	87.6 (.13)	77.3
Beam-Search	80.6 (.65)	2115	86.4 (.22)	441	87.2 (.10)	184
Beam-Search+CC	82.3 (.33)	4236	86.9 (.18)	1403	87.6 (.12)	514
AMBR-Sum						
Near-Exhaustive	81.7 (.70)	7.8	87.5 (.19)	3.4	88.4 (.22)	1.8
Complete-Closure	83.1 (.45)	46.9	87.8 (.17)	18.0	88.6 (.18)	8.4
Beam-Search	81.8 (.73)	208	87.5 (.19)	35.1	88.4 (.18)	13.3
Beam-Search+CC	83.1 (.46)	624	87.8 (.18)	120	88.6 (.14)	41.4
Max-Rule						
Near-Exhaustive	81.7 (.49)	4.2	87.8 (.16)	1.8	88.7 (.20)	1.0
Complete-Closure	83.2 (.37)	27.5	88.0 (.12)	10.4	88.9 (.13)	5.3
Beam-Search	81.8 (.54)	171	87.8 (.15)	25.8	88.6 (.18)	10.4
Beam-Search+CC	83.2 (.37)	486	88.0 (.11)	87.6	88.8 (.12)	32.8

Table 6.7: Switchboard development-set accuracies and speeds of each decoding method on latent-variable grammars, using various pruning approaches, and the same configuration as reported in Table 6.6.

operation, examining the following approximations thereof:

Log-sum pruning at various heuristic operating points When adding two probabilities, each stored as $log_e(p)$ in 32-bit IEEE floating point number, the largest difference between the two which can be represented in the final sum is approximately e^{-16}. Thus, if $a-b>16$, $log_e(e^a+e^b)=a$. This allows us to prune the number of `logsumexp` operations considerably, by computing $|a-b|$ before the expensive exponentiation and logarithm operations. If we are willing to accept more error in our `logsumexp` estimate, we can prune further, by eliminating calculations where $|a-b|<\delta$. Figure 6.7 demonstrates that we can tune δ considerably below 16 with minimal impact on parse accuracy. Max-rule accuracy begins to degrade at $\delta<4$, and AMBR accuracy actually peaks at $\delta=3$ before dropping off. Tables 6.9 and 6.10 label this method Approximate Log Sum.

	2-cycle		4-cycle		6-cycle	
	F_1 (σ)	w/s	F_1 (σ)	w/s	F_1 (σ)	w/s
Viterbi						
Near-Exhaustive	77.5 (.46)	59.2	82.7 (.65)	17.1	83.3 (.32)	8.3
Complete-Closure	78.2 (.48)	374	83.0 (.63)	106	83.7 (.32)	43.4
Beam-Search	77.6 (.42)	1894	82.6 (.68)	350	83.4 (.31)	134
Beam-Search+CC	78.2 (.43)	3295	83.0 (.67)	1010	83.8 (.35)	355
AMBR-Sum						
Near-Exhaustive	79.7 (.39)	8.0	84.3 (.35)	1.9	84.8 (.29)	0.8
Complete-Closure	80.2 (.37)	42.3	84.6 (.44)	9.2	84.9 (.25)	3.3
Beam-Search	79.8 (.39)	200.0	84.3 (.35)	25.6	84.8 (.33)	9.6
Beam-Search+CC	80.1 (.35)	554.6	84.6 (.44)	81.3	85.0 (.28)	26.3
Max-Rule						
Near-Exhaustive	79.4 (.56)	3.9	84.8 (.41)	0.9	85.4 (.34)	0.4
Complete-Closure	79.9 (.57)	22.9	85.2 (.48)	4.8	85.7 (.33)	2.0
Beam-Search	79.4 (.58)	153	84.8 (.39)	18.0	85.4 (.36)	7.4
Beam-Search+CC	80.0 (.56)	431	85.2 (.50)	60.1	85.7 (.48)	21.0

Table 6.8: Chinese development-set accuracies and speeds of each decoding method on latent-variable grammars, including complete-closure pruning and beam search guided by a POS-Boundary prioritization model (the Brown clusters used to train the lexical prioritization model are only available for English).

IEEE approximation of `logsumexp` The standard hardware floating-point representations allow efficient and reasonably accurate approximations of the exponential and natural logarithm functions [163], and replacing the true logarithm and exponentiation functions in `logsumexp` may provide a considerable efficiency gain.

As demonstrated in Table 6.9, both approximation methods are accurate enough for effective parsing, and improve efficiency considerably. However, contrary to our expectation, the IEEE approximation did not combine smoothly with Approximate Log Sum; as shown in Table 6.9, accuracy declines greatly for all genres when the two approximations are used in conjunction.

We omit here a third potential approximation. The lexical prioritization model we use for beam-search parsing uses (as part of its non-terminal ranking function) an heuristic

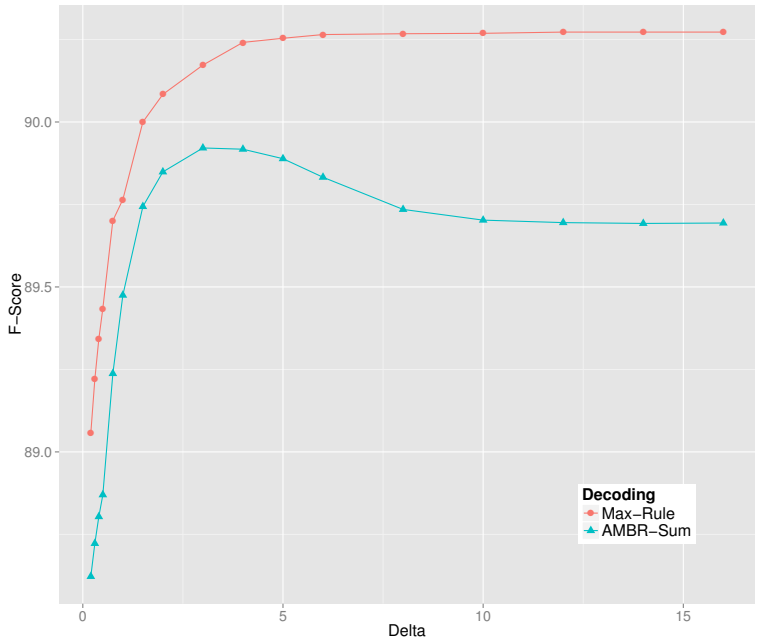


Figure 6.7: Approximating the `logsumexp` operation. WSJ development-set accuracies of 6-cycle latent-variable grammar, as δ is varied from .2–16. Evaluated with near-exhaustive beam search. Max-rule decoding appears to benefit from incorporating the very small probabilities pruned at lower δ values, while AMBR-Sum accuracy peaks earlier and declines as δ increases.

outside score for each non-terminal label. If this score were a sufficiently accurate estimate of the outside probability, we could use it in place of β, and eliminate the outside pass. However, preliminary trials with this method resulted in large accuracy loss. The outside score, although useful for local prioritization, is not a true outside-probability estimate, and using it as such results in severe numeric instability. We leave a more reliable outside heuristic as a possibility for future investigation.

6.7 Scaling in the Real Semiring

The approximations described in the previous section improve `logsumexp` efficiency considerably; in this section, we describe an alternate approach, borrowed largely from the Berkeley Parser [142] — and similar to that described by Rabiner [150] in the context of HMM estimation — that does away completely with the `logsumexp` operation. As noted previously, we generally maintain chart probabilities in the log domain (and thus, it is

	Baseline		Approx. Log-sum		IEEE Approx.		Log-sum and IEEE	
	F_1	w/s	F_1 (σ)	w/s	F_1 (σ)	w/s	F_1 (σ)	w/s
AMBR-Sum								
WSJ	89.4 (.16)	64.9	89.9 (.19)	113.7	89.5 (.15)	96.3	86.5 (.86)	54.9
SWBD	88.6 (.14)	30.3	88.6 (.17)	62.6	88.5 (.15)	54.9	86.7 (.28)	41.7
PCTB	84.8 (.27)	24.4	84.0 (.45)	60.3	83.8 (.79)	54.0	81.3 (1.04)	37.1
Max-Rule								
WSJ	90.0 (.13)	50.1	90.1 (.14)	73.8	90.1 (.15)	73.8	86.8 (.80)	37.0
SWBD	88.8 (.12)	30.3	88.8 (.12)	41.4	88.8 (.13)	43.6	87.0 (.28)	33.0
PCTB	85.4 (.35)	19.3	84.9 (.46)	38.5	84.9 (.48)	41.7	82.6 (.49)	26.7

Table 6.9: Development-set accuracies and speeds of various approximate methods on 6-cycle latent-variable grammars. All trials used the same grammars and beam-search settings as Table 6.6, with complete-closure and prioritization models trained for each genre (lexical boundary prioritization models for English genres, POS boundary models for Chinese). Log-sum deltas (8 for max-rule and 3 for AMBR-Sum) chosen to maximize near-exhaustive F_1 on WSJ section 22.

convenient to represent production probabilities as logarithms as well, although underflow is less of a problem in grammar rules).

The standard 64-bit IEEE floating-point representation [86] provides 53 binary digits of precision and 10 binary digits of exponent (i.e., it can represent numbers between approximately 10^{-307} and 10^{307}, or e^{-707}–e^{707}), with roughly 16 decimal digits of precision. Underflow in this range is relatively infrequent, but still problematic, particularly when processing longer sentences.

Note that the probability range represented within a single cell is likely to be much smaller than the range of the hardware representation, even if the probabilities are near the lower bound of the floating-point format. We can avoid underflow by re-scaling all probability representations by a fixed amount, keeping the maximum probability within a fixed range (e.g., $> e^{-200}$, well within the range of a 64-bit double). Thus, we record the grammar probabilities in the real domain (as 64-bit doubles), and all cell probabilities in (possibly) scaled real domain. We populate non-terminal probabilities in a cell using the inside-outside summations from Section 2.6. After processing each cell, we multiply all

	Viterbi	Approx Log Sum		Scaled Real Semiring	
		AMBR-Sum	Max-Rule	AMBR-Sum	Max-Rule
WSJ					
F_1 (σ)	89.0 (.12)	89.7 (.21)	89.9 (.13)	89.4 (.16)	90.0 (.13)
w/s	882.7	113.0	83.9	217.8	121.9
SWBD					
F_1 (σ)	87.1 (.09)	88.1 (.13)	88.3 (.07)	88.2 (.09)	88.3 (.07)
w/s	524.4	61.1	47.6	144.5	84.5
PCTB					
F_1 (σ)	80.9 (.64)	82.2 (.45)	83.1 (.62)	81.7 (.72)	82.9 (.68)
w/s	300.5	40.6	29.6	91.0	9.5

Table 6.10: Final accuracies and speeds of various decoding methods and representation choices. Evaluated on WSJ, SWBD, and PCTB test sets, using complete-closure and beam search, with parameterizations selected to minimize accuracy loss on development sets.

probabilities by a multiplier chosen to keep the maximum value within our chosen range.[6]

We can represent the grammar using the same matrix encoding (as described in Chapter 3), but the grammar and chart representations have approximately twice the memory footprint of our standard representation, since we must replace all 32-bit floating-point values with their 64-bit equivalents. However, as demonstrated in Table 6.10, the additional memory access is outweighed by eliminating the expensive `logsumexp` operations.

6.8 Test Set Results and Discussion

Table 6.10 presents accuracy and timing trials on the test sets of each corpus, executed with complete closure and a guided beam search (using lexical prioritization models for WSJ and SWBD and a POS-boundary model for PCTB). In general, the scaled real-value representation improves speed measurably (but not dramatically) over the log-domain representation. In some cases (e.g. WSJ AMBR-Sum), the accuracy results differ between the

[6] By retaining the multipliers for each cell, we can recover real-valued probabilities.

two representations, presumably due to numeric instability in the summed probabilities.[7]

In this chapter, we have examined several competing chart decoding methods, including two approximations of Minimum-Bayes-Risk decoding, which have previously been described but not empirically validated. However, several related areas of research remain. We presented summary results averaged over numerous latent-variable grammars; recent work has demonstrated that incorporating multiple grammars into a single decoding pass can be very effective, at least for max-rule decoding [141, 165]. Incorporating this approach into AMBR decoding methods might yield further gains there as well.

In this work, we report only F_1 of each decoding method (standard practice in parsing research). However, the performance of downstream consumers of constituency parse output is likely to be only loosely correlated with F_1, depending instead on the parser's performance vis-à-vis specific parsing errors (e.g. PP-attachment and coordination errors). Recent work has also begun to explore finer details of such errors across parser implementations [105], and a similar exploration of chart decoding methods would yield further insight on their relative merits.

Finally, a decoding algorithm that incorporated some of max-rule's 'reordering' effects directly into the outside pass might improve accuracy over straightforward AMBR methods without requiring the (expensive) second grammar intersection of max-rule.

[7] As demonstrated in Figure 6.7, AMBR-Sum may in some cases benefit from *less* numeric precision. It remains unclear which representation method suffers more from numeric instability, and thus, whether AMBR-Sum decoding suffers or benefits from those effects.

Chapter 7

Method Combination and Discussion

The goals of this research were to examine and improve the efficiency of context-free inference, particularly with high-accuracy latent-variable grammars. In support of these goals, we have explored numerous methods — in both training and inference — of improving parsing efficiency (summarized in Table 7.1). We described a compact and efficient grammar representation, and demonstrated large efficiency gains from that encoding; we presented several novel methods of grammar training that yield more compact models and more efficient inference; and we compared the accuracy and efficiency of various chart decoding methods. In so doing, we have improved considerably on the state-of-the-art for efficient context-free inference and broadened the array of operating-point choices available to an application designer, when trading off accuracy and efficiency.

In Chapter 3, we presented a grammar encoding that dramatically improves cache utilization and improves parsing throughput by 7–10×. Subsequent chapters presented model-training and inference methods, each of which made use of that grammar encoding. In this final chapter, we combine the methods from chapters 4–6 and examine the potential for additive gains. We compare those trials with previous state-of-the-art systems; we find an efficiency gain of approximately 2.5× over the best previously-reported results, which themselves benefited from a 7× gain from the approaches from Chapter 3. We also present practical guidance for training efficient parsing models, and discuss opportunities for future work in efficient constituency parsing.

7.1 Combining Methods

In this section we will combine the methods presented in chapters 4–5, and examine the potential for additive gains (Note that the trials presented in those chapters already incorporate the grammar encoding from Chapter 3).

Efficient and Parallelizable CYK Chart Parsing. Chapter 3 presented a compact and efficient PCFG representation, and application thereof in a parallelizable grammar intersection approach.

Analysis of Efficiency Characteristics of PCFGs. The regression models of Chapter 5 demonstrate the efficiency impact of various PCFG characteristics, and provide guidance for subsequent exploration of parameterization approaches likely to yield further efficiency gains.

Methods of training efficient latent-variable grammars Chapter 5 presented corpus transformations aimed at compact and efficient grammars, and Chapter 5 demonstrated the effectiveness of incorporating efficiency measures directly into the split-merge training process.

Evaluation of alternate chart decoding methods In Chapter 6, we compared several decoding methods and approximations thereof.

Table 7.1: Principal contributions of this thesis

The pruned inference trials presented in chapters 3–6 all use a complete-closure cell pruning model [21] and a lexical prioritization model [19]. For these final trials, we re-implemented adaptive beam pruning [21] and unary constraint systems [156], using memory representations similar to those described in Chapter 3 to improve CPU cache efficiency and reduce memory stalls. This greatly reduces the sentence-level initialization time for these pruning methods, and allows us to combine them with a lexical prioritization model, improving the effectiveness of beam search. Bodenstab [19] presented inference results using a lexical prioritization model and others incorporating adaptive-beam pruning, but did not combine the two; our implementation allows us to include trials combining both.

Since the pruning methods we use are strongly related to those in Bodenstab [19], we also compare with the final timing trials presented there; we executed our trials on the same hardware, so those results provide an appropriate baseline and well-represent the current state-of-the-art. One other note on that baseline is important: Bodenstab's final results make use of the matrix grammar encoding from Chapter 3; On WSJ text, he reported an improvement from 188 to 1331 w/s when replacing a previous encoding with our matrix encoding, and a further increase to 1581 when unary constraints were added. To place the full contributions of this thesis in proper context, the lower figure might be a more appropriate baseline; as presented, Tables 7.2–7.4 emphasize the further gains from chapters 4–6. The results of those trials are presented in tables 7.2–7.4. We incorporated

	F_1	LP	LR	W/S
Baselines				
Coarse-to-Fine MaxRule [142]	90.2	90.3	90.0	110
Adaptive Beam + Unary [19]	88.8	89.0	88.6	1581
This Thesis				
Matrix Grammar + CC	88.7 (.18)	88.8 (.20)	88.5 (.18)	2285
+ Param. Pruning + Modeled Obj.	88.6 (.17)	88.8 (.18)	88.4 (.18)	3313
+ Unary	88.5 (.17)	88.8 (.18)	88.3 (.18)	3862
+ Adaptive Beam	87.7 (.22)	87.8 (.23)	87.5 (.21)	4188
Pruning + Mod. Obj. + Lex. Simp.	88.3 (.19)	88.5 (.19)	88.1 (.22)	3363
1-Best Grammar (Parameter Pruning + Modeled Obj)				
Viterbi	88.8	89.0	88.6	3214
+ Unary Constraints	88.7	88.9	88.4	3734
Max-Rule	89.8	90.1	89.5	448

Table 7.2: WSJ accuracy and efficiency, combining the efficient-inference techniques discussed in this thesis. The baselines are from Bodenstab [19], executed on the same Intel® Xeon X5650 "Nehalem" systems, and include adaptive-beam pruning, unary constraints, and begin- and end-constituent chart-cell closures [156].

the uniform parameter pruning from Section 5.1.1 into the modeled efficiency objective, training a total of 444 grammars (at $\epsilon = 10^{-}6$) and fitting a new linear regression to the inference speeds observed with those models. We trained all modeled grammars in this chapter at $\lambda = .35$, a relatively conservative ranking function.

	F_1	LP	LR	W/S
Matrix Grammar + CC	87.1 (.26)	87.5 (.28)	86.6 (.25)	1834
+ Param. Pruning + Modeled Obj.	87.4 (.22)	87.8 (.22)	86.9 (.23)	2753
+ Unary Constraints	87.2 (.22)	87.8 (.22)	86.7 (.22)	3214
+ Adaptive Beam	86.6 (.60)	87.1 (.44)	86.1 (.78)	3731
Pruning + Mod. Obj + Lex. Simp.	8721 (.11)	87.7 (.13)	86.7 (.11)	2822
1-Best Grammar (Parameter Pruning + Modeled Obj.)				
Viterbi	87.8	88.2	87.4	2670
+ Unary Constraints	87.6	88.1	87.1	3102
Max-Rule	88.5	89.2	87.8	339

Table 7.3: Switchboard accuracy and efficiency, using the same settings as in Table 7.2.

As per our standard practice throughout this thesis, we report accuracies and speeds over a sizable number of grammars (16 in this case). We believe these trials over a wide range of grammars reduce the variance introduced by a random factor in EM training, and present an accurate reflection of the effects of each grammar training method. We executed 5 trials for each configuration, and report the maximum speed observed. As the inference speed increases, variances induced by small environmental influences becomes more apparent; to dampen any such effects, we replicated the test corpora (3× for WSJ and Switchboard and 20× for Chinese) to produce files of 160–180k words.

The baseline trials from Bodenstab [19] were executed with a single grammar (the Berkeley Parser's default 6-cycle English and Chinese grammars). Those grammars were selected from set of similar grammars for their superior performance. Since end-user applications will generally operate similarly on a single model, presumably one selected similarly from a set of competing models. To represent this operating condition, we selected the most accurate grammar — as measured on development data, without regard to speed — and present accuracies and timings for that grammar as well. The WSJ corpus

	F_1	LP	LR	W/S
Baselines				
Coarse-to-Fine MaxRule [142]	83.9	84.5	83.3	56.8
Adaptive Beam + Unary [19]	81.1	82.3	80.0	1169
This Thesis				
Matrix Grammar + CC	79.7 (.83)	80.0 (.87)	79.4 (.83)	824
+ Param. Pruning + Modeled Obj.	80.2 (.60)	80.5 (.73)	80.0 (.57)	1379
+ Unary	80.4 (.57)	80.7 (.65)	80.0 (.57)	1442
+ Adaptive Beam	79.7 (.58)	79.9 (.72)	79.6 (.49)	1815
1-Best Grammar (Parameter Pruning + Modeled Obj.)				
Viterbi	80.5	81.1	79.8	1344
+ Unary Constraints	80.5	81.1	79.8	1403
Max-Rule	83.6	85.2	82.0	147

Table 7.4: Chinese test-set accuracy and efficiency, using the POS prioritization model; other settings are the same as in Table 7.2. Note: the lexical simplification approaches from Chapter 4 cost considerable accuracy without a commensurate efficiency gain, so we omit that approach in these combined trials.

includes multiple development sets, so we selected the model which maximizes the product of F_1 on sections 22 and 24 (i.e. $F_1\,(22) \cdot F_1\,(24)$). For each of the other genres, we chose the grammar which maximizes F_1 on the standard development set. We anticipate that accuracy on the 63.7k-word Switchboard development set will be a reasonable predictor of test-set accuracy. We hesitate to make the same assumption about the Chinese corpus, as accuracy on the small 7.2k-word development set — and on the 8k-word test set — is likely to be dominated by noise.

		Words/Sec	
	F_1	X5650	3520M
		Nehalem	Ivy Bridge
Viterbi			
WSJ	88.7	3734	5302
Switchboard	87.6	3101	4531
Chinese	80.5	1442	2211
Max-Rule			
WSJ	89.8	445	728
Switchboard	88.5	339	530
Chinese	83.6	147	228

Table 7.5: Trials with 1-best grammars on the Intel® 'Ivy Bridge' architecture. All trials use the same 1-best grammars from Tables 7.2–7.4; accuracies and Nehalem timings are repeated from those tables. Ivy Bridge trials performed on the 3520M model at 2.9 GHz, executed on Java 1.7.0_45 under Mac OS 10.8.

We found that the lexical simplification methods from Chapter 4 and the pruning and modeling methods from Chapter 5 combine smoothly, yielding additive efficiency gains with minimal accuracy degradation. The final throughput on WSJ text is approximately double that of the fastest result in Bodenstab [19], and we found a small accuracy gain as well — the average accuracy of the models we trained was nearly equal to the baseline, and the best model improved on that baseline by a small margin. Integrating adaptive-beam pruning appears to incur search errors that lower accuracy — likely an unjustified cost, even for the additional speed gained.

To provide results directly comparable to previously-published work, we limited the trials in Tables 7.2–7.4 to Intel Nehalem hardware. Table 7.5 repeats the 1-best grammar trials, this time on an Intel 3520M 'Ivy Bridge' CPU; we found that the newer processor architecture provides a consistent gain — of approximately 50–70% — across corpora and decoding methods. Our expectation is that these trials should be representative of expected throughput on relatively recent hardware.

7.2 Best Practices

Potential parsing consumers — whether utilizing parse data in research or developing end-user applications — need accurate and efficient models, but few users want to commit significant time to exploring grammar training. Several of the models we've examined in this thesis are available for download along with the BUBS parser distribution, and are documented on the BUBS wiki. However, these models will not cover the needs of every application, so we present in this section general guidelines for training and use of accurate and efficient latent-variable grammars and pruning models for them.

We made available the grammar training system used in chapters 4 and 5 as part of the BUBS project. This system is an adaptation of code from the Berkeley Parser, adding the ability to rerank merge candidates by arbitrary objective functions (such as the inference-informed model from Section 5.5 and the regression model from Section 5.6). In addition, the constrained inference during EM incorporates some of the cache-efficient representation methods from Chapter 3, reducing training time by approximately 50% in the default configuration. Step-by-step instructions for grammar training are also included in the BUBS project wiki.[1]

7.2.1 Model Training

Although we showed in Chapter 5 that the relationship between model size and efficiency is not strictly linear, it is still the case that the smallest grammar (that achieves the required accuracy) will generally yield the fastest performance. Many training choices affect the ultimate size of the model; the BUBS training system provides for configuration of the following parameters — listed approximately in order of the expected return on modeling-time investment:

- The number of split-merge cycles — parameter pruning and some of the other methods listed below can vary the rate of expansion considerably, but a rough rule-of-thumb is that each additional cycle will approximately double the number of productions. And in some cases, additional cycles will overfit to the training corpus, so a 4- or 5-cycle grammar may be superior to 6-cycle training for some applications.

- Uniform Parameter Pruning (Section 5.1.1). The Berkeley training system defaults to pruning any parameters of probability below 10^{-30}. The BUBS implementation

[1] https://code.google.com/p/bubs-parser/w/list

defaults to 10^{-12}, but in many genres, even that threshold is overly conservative, and can be increased considerably before accuracy declines. A grid-search over a range of values should yield an effective threshold.

- Explicit efficiency objectives (Chapter 5)
- Lexical simplification (Chapter 4). As we found in Chapter 4, normalizing out rare words is unlikely to improve inference speed dramatically, but it will reduce the model's memory footprint and initialization time, both of which may be important in some applications.

As always, evaluation on a sizable in-domain development set is the most reliable way to determine whether the model achieves accuracy goals. In many cases, optimizing for (relatively) minor improvements in F_1 will not be reflected in improved application performance. For applications particularly sensitive to specific grammatical errors (e.g. PP-attachment errors, erroneous modifier attachment, and coordination errors), the error analysis tools we used in Section 6.4.3 [105] may provide insight into the sources of such errors, and inform the choice of an appropriate speed/accuracy operating point.

7.2.2 Inference

At inference time, an efficient grammar representation and effective pruning are crucial. As we demonstrated in Chapter 3, a cache-efficient grammar encoding can improve speed by 10× or more. Accurate pruning of the search space is equally crucial. The BUBS and Berkeley parsers both implement effective (albeit very different) pruning approaches, but the BUBS grammar representation is somewhat more efficient (as demonstrated by the relative speeds in tables 7.2 and 7.4).

If maximum accuracy is required, Petrov [141] showed that the differing performance characteristics of grammars trained with different random initialization seeds can often be harnessed jointly, and the weaknesses of a particular grammar can in some cases be ameliorated by the strengths of the others. However, this approach requires a separate inference run with each of the target grammars. Even if those tasks are parallelized to reduce latency, the total system throughput drops by (at least) a factor of the number of

grammars.[2] While the approaches described in this thesis are applicable to products-of-grammars inference, the BUBS framework does not currently support this method, and we leave further investigation of optimization thereof to future work (see also the discussion of interpolating and merging latent-variable grammars below).

7.3 Conclusions and Future Work

In this thesis, we explored a wide range of methods of improving constituency parsing efficiency. We presented a grammar encoding which greatly improves throughput on CPU architectures, and allows parallelization at a finer level than any other practical approach to-date. We demonstrated that parsing throughput can be predicted quite accurately given a relatively few characteristics of the grammar, and incorporated that measure into grammar training, yielding more compact and efficient grammars with minimal accuracy loss. We demonstrated the effectiveness of our methods across multiple languages and genres; in combination, we found improvements of 10–20× over the previous state-of-the-art for efficient constituency parsing.

Nevertheless, many avenues remain for further progress in efficient parsing. As discussed in Section 1.1, dependency structure can in some cases be recovered by very efficient algorithms [131, 123]. Although the pruning methods used in this thesis yield an average-case complexity of approximately $O(N^{1.5})$ [19], the linear complexity and lower constant factors of transition-based dependency parsing methods result in dramatically faster inference (often tens of thousands of words per second). The timings in Tables 7.2–7.4 show that we have closed this gap somewhat; the larger constant factors make it unlikely that PCFG-based inference will ever reach the speeds of transition-based parsers, but there may be room to leverage the strengths of the two approaches.

Greedy transition-based inference can in some cases recover constituency structure as well as dependency [161, 95], although thus far such systems do not obtain the accuracy of state-of-the-art PCFG-based parsers, and our methods provide a superior accuracy/efficiency operating point for most applications. Even so, transition-based parsers provide another valuable operating-point choice for applications where maximum throughput and minimum latency are crucial.

[2] In some cases, pruning passes may be shared between separate grammars, reducing — but not eliminating — this effect. Petrov discusses reuse of multi-pass coarse-to-fine pruning [141], and some of the pruning methods described in Appendix A can be applied similarly.

Another promising avenue would leverage dependency parsing as a pre-processing stage to constrain traditional chart-parsing inference. The structure of a projective dependency tree is closely related to that of the analogous constituent tree — i.e., each subtree in the dependency structure corresponds to a constituent spanning the same set of words. Wang and Zong [180] noted that, given the dependency tree, we can 1) Close CYK cells which conflict with that subtree structure, and 2) Tag cells within that substring as incapable of participating in spans beyond the fixed subtree. Thus, we can use dependency inference analogously to the chart-pruning methods described in Appendix A, and prune a considerable fraction of the CYK search space.[3]

This approach depends crucially on confidently classifying dependency arcs for use in the chart cell closure — as with other chart-constraint methods we need very high precision (to limit search errors) and the effectiveness of the method depends on the recall obtainable at that high-precision operating point. Unfortunately, obtaining that classification accuracy may be a difficult task. Mejer and Crammer [124] attempted a similar classification on graph-based arcs, with limited success. Their reported classification accuracy on incorrect arcs was only around 50%; effective application of this method will require a considerable improvement in arc classification.

Finally, we see great potential in exploration of methods of combining latent-variable grammars. As we discussed in the previous section, scoring with multiple latent-variable models can leverage the strengths of each and improve accuracy. However, to our knowledge, existing implementations of this approach require separate inference passes with each model. If the final models are all derived from a single parent model, the Berkeley Parser implementation is able to leverage that shared structure and reuse early stages of its multi-level coarse-to-fine process (up until the models diverge). However, even with that shared structure, inference is slowed dramatically.

Since the latent annotations in two latent-variable models are unrelated, merging or interpolating between two such models is not straightforward. However, if we were able to cluster and merge those annotations, we might obtain some of the benefits of product-of-grammars inference, with a single model. Even if that PCFG is larger (and slower) than

[3] Like constituency parsing, transition-based dependency parsers produce only projective dependencies (i.e., they cannot recover structure with crossing dependencies). Note: transition-based dependency parsers cannot recover structure with crossing dependences. This constraint is often considered a weakness, but the languages which exhibit frequent non-projective dependencies are (by definition) those for which constituency structure is least useful. For languages which are well-described by constituency structure, the limitation to projective dependencies is not problematic.

any of the source models, inference with a single model would almost certainly be more efficient than populating and combining multiple separate charts.

7.4 Applications Outside Constituency

Although this thesis has focused exclusively on constituency parsing, many of the methods we explored are extensible into other NLP domains. Some such applications have already been explored, and a few are in common use; although it is not a primary contribution of this thesis, we have already utilized some of the cache-sensitive model representation methods from Chapter 3 in a POS tagger (integrated into the BUBS parser, and also usable as a standalone tagger). Other examples include model encodings in language modeling [152, 70, 136] and machine translation cube pruning [81], or model pruning methods [153]. To our knowledge, previous work has not explored the methods from Chapter 5 — specifically, predicting model efficiency and incorporating efficiency objectives into training; we see great potential in adapting those methods for machine translation, graph-based dependency parsing, named entity recognition, semantic processing, and other NLP applications. In summary, while the contributions of this thesis have greatly improved constituency-parsing throughput, we see great opportunities for further efficiency gains — in constituency parsing and in other areas of NLP as well.

Bibliography

[1] Agarwal, R. C., Gustavson, F. G., and Zubair, M. A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing* (Minneapolis, Minnesota, United States, 1992), IEEE Computer Society Press, pp. 32–41.

[2] Akaike, H. Information theory and an extension of the maximum likelihood principle. In *Second international symposium on information theory* (1973), p. 267–281.

[3] Amedro, B., Caromel, D., Huet, F., Bodnartchouk, V., Delbé, C., and Taboada, G. L. HPC in java: experiences in implementing the NAS parallel benchmarks. In *Proceedings of the 10th WSEAS international conference on applied informatics and communications and 3rd WSEAS international conference on Biomedical electronics and biomedical informatics* (2010), p. 221–230.

[4] Arthur, D., and Vassilvitskii, S. k-means++: the advantages of carefull seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms* (2007), pp. 1027–1035.

[5] Arun, A., and Keller, F. Lexicalization in crosslinguistic probabilistic parsing: The case of french. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics* (Stroudsburg, PA, USA, 2005), ACL '05, Association for Computational Linguistics, p. 306–313.

[6] Attia, M., Foster, J., Hogan, D., Le Roux, J., Tounsi, L., and van Genabith, J. Handling unknown words in statistical latent-variable parsing models for arabic, english and french. In *Proceedings of the NAACL HLT 2010 First Workshop on Statistical Parsing of Morphologically-Rich Languages* (Los Angeles, CA, USA, June 2010), Association for Computational Linguistics, p. 67–75.

[7] Baker, J. K. Trainable grammars for speech recognition. *The Journal of the Acoustical Society of America 65* (1979), S132.

[8] Bangalore, S., Boulllier, P., Nasr, A., Rambow, O., and Sagot, B. MICA: a probabilistic dependency parser based on tree insertion grammars application note. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Short Papers* (Stroudsburg, PA, USA, 2009), NAACL-Short '09, Association for Computational Linguistics, p. 185–188.

[9] Banko, M. *Open Information Extraction for the Web*. PhD dissertation, University of Washington, Seattle, Washington, 1999.

[10] Baum, L. E., and Petrie, T. Statistical inference for probabilistic functions of finite state markov chains. *The Annals of Mathematical Statistics 37*, 6 (Dec. 1966), 1554–1563.

[11] Baum, L. E., Petrie, T., Soules, G., and Weiss, N. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The Annals of Mathematical Statistics 41*, 1 (Feb. 1970), 164–171.

[12] Bell, N., and Garland, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland, Oregon, 2009), ACM, pp. 1–11.

[13] Bellman, R. E. The theory of dynamic programming, 1954.

[14] Bies, A., Mott, J., Warner, C., and Kulick, S. English web treebank, 2012.

[15] Bikel, D. M. Intricacies of Collins' parsing model. *Computational Linguistics 30*, 4 (2004), 479–511.

[16] Björne, J., Ginter, F., Pyysalo, S., Tsujii, J., and Salakoski, T. Complex event extraction at PubMed scale. *Bioinformatics 26*, 12 (June 2010), 382–390.

[17] Black, E., Abney, S., Flickenger, F., Grishman, R., Harrison, P., Hindle, D., Ingria, R., Jelinek, F., Klavans, J., Liberman, M., Marcus, M., Roukos, S., Santorini, B., and Strzalkowski, T. A procedure for quantitatively comparing the syntactic coverage of english grammars. In *Proceedings of the Fourth DARPA Speech and Natural Language Workshop* (1991).

[18] Bod, R. An efficient implementation of a new DOP model. In *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics - Volume 1* (Stroudsburg, PA, USA, 2003), EACL '03, Association for Computational Linguistics, p. 19–26.

[19] Bodenstab, N. *Prioritization and Pruning: Efficient Inference with Weighted Context-Free Grammars*. Ph.D. thesis, Oregon Health & Science University, 2012.

[20] Bodenstab, N., and Dunlop, A. BUBS parser, 2012.

[21] Bodenstab, N., Dunlop, A., Roark, B., and Hall, K. Beam-width prediction for efficient context-free parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics* (Portland, Oregon, June 2011), pp. 440–449.

[22] Boullier, P., and Sagot, B. Are very large context-free grammars tractable? In *Proceedings of the 10th International Conference on Parsing Technologies* (Prague, Czech REpublic, 2007), ACL, pp. 94–105.

[23] Brants, T. Inter-annotator agreement for a german newspaper corpus. In *In Proceedings of Second International Conference on Language Resources and Evaluation LREC-2000* (2000).

[24] Cameron, R. D., Herdy, K. S., and Lin, D. High performance XML parsing using parallel bit stream technology. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds* (New York, NY, USA, 2008), CASCON '08, ACM, p. 17:222–17:235.

[25] Campbell, R. Using linguistic principles to recover empty categories. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics* (Stroudsburg, PA, USA, 2004), ACL '04, Association for Computational Linguistics.

[26] Candito, M., and Crabbé, B. Improving generative statistical parsing with semi-supervised word clustering. In *Proceedings of the 11th International Conference on Parsing Technologies* (Stroudsburg, PA, USA, 2009), IWPT '09, Association for Computational Linguistics, p. 138–141.

[27] Canny, J., Hall, D., and Klein, D. A multi-teraflop constituency parser using GPUs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (Seattle, Washington, USA, Oct. 2013), Association for Computational Linguistics, p. 1898–1907.

[28] Caraballo, S. A., and Charniak, E. New figures of merit for best-first probabilistic chart parsing. *Computational Linguistics 24* (June 1998), 275–298.

[29] Carreras, X., Collins, M., and Koo, T. TAG, dynamic programming, and the perceptron for efficient, feature-rich parsing. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning* (Stroudsburg, PA, USA, 2008), CoNLL '08, Association for Computational Linguistics, p. 9–16.

[30] Cer, D., Marneffe, M.-C. d., Jurafsky, D., and Manning, C. D. Parsing to stanford dependencies: Trade-offs between speed and accuracy. In *7th International Conference on Language Resources and Evaluation (LREC 2010)* (2010).

[31] Charniak, E. *Statistical language learning*. The MIT Press, 1996.

[32] Charniak, E. A maximum-entropy-inspired parser. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference* (Seattle, Washington, 2000), Morgan Kaufmann Publishers Inc., pp. 132–139.

[33] Charniak, E., and Elsner, M. EM works for pronoun anaphora resolution. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics* (Stroudsburg, PA, USA, 2009), EACL '09, ACL, p. 148–156.

[34] Charniak, E., and Johnson, M. Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics* (Ann Arbor, Michigan, 2005), ACL, pp. 173–180.

[35] Chen, S. S., Donoho, D. L., and Saunders, M. A. Atomic decomposition by basis pursuit. *SIAM Review 43*, 1 (Jan. 2001), 129–159.

[36] Chi, Z., and Geman, S. Estimation of probabilistic context-free grammars. *Comput. Linguist. 24*, 2 (June 1998), 299–305.

[37] Chiang, D. A hierarchical phrase-based model for statistical machine translation. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics* (Ann Arbor, Michigan, 2005), ACL, pp. 263–270.

[38] Chomsky, N. *Syntactic Structures*. Mouton, Paris, 1957.

[39] Chomsky, N. A. Three models for the description of language. *IRE Transactions on Information Theory 2*, 3 (1956), 113–124.

[40] Church, K. W. A finite-state parser for use in speech recognition. In *Proceedings of the 21st Annual Meeting on Association for Computational Linguistics* (Stroudsburg, PA, USA, 1983), ACL '83, Association for Computational Linguistics, p. 91–97.

[41] Chytil, M., Crochemore, M., Monien, B., and Rytter, W. On the parallel recognition of unambiguous context-free languages. *Theoretical Computer Science 81*, 2 (Apr. 1991), 311–316.

[42] Clark, S., and Curran, J. R. Parsing the WSJ using CCG and log-linear models. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics* (Stroudsburg, PA, USA, 2004), ACL '04, Association for Computational Linguistics.

[43] Click, C., Tene, G., and Wolf, M. The pauseless GC algorithm. *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments* (2005), 46–56.

[44] Click, C. N., Vick, C. A., and Paleczny, M. H. System and method for range check elimination via iteration splitting in a dynamic compiler, May 2007. US Patent 7,222,337.

[45] Cocke, J., and Schwartz, J. T. Programming languages and their compilers. Technical report Preliminary notes, Courant Institute of Mathematical Sciences, NYU, 1970.

[46] Collins, M. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics* (Madrid, Spain, 1997), ACL, pp. 16–23.

[47] Collins, M. *Head-Driven Statistical Models for Natural Language Parsing*. PhD dissertation, University of Pennsylvania, 1999.

[48] Collins, M. Discriminative training methods for hidden markov models: theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 conference on Empirical Methods in Natural Language Processing* (Philadelphia, July 2002), vol. 10, ACL, pp. 1–8.

[49] Collins, M. J. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics* (Santa Cruz, California, 1996), ACL, pp. 184–191.

[50] Coppersmith, D., and Winograd, S. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing* (New York, New York, United States, 1987), ACM, pp. 1–6.

[51] Cuff, J. A., and Barton, G. J. Application of multiple sequence alignment profiles to improve protein secondary structure prediction. *Proteins: Structure, Function, and Genetics 40*, 3 (2000), 502–511.

[52] Culy, C. The complexity of the vocabulary of bambara. *Linguistics and Philosophy 8*, 3 (Aug. 1985), 345–351.

[53] Cuppu, V., Jacob, B., Davis, B., and Mudge, T. High-performance DRAMs in workstation environments. *IEEE Transactions on Computers 50*, 11 (2001), 1133–1153.

[54] Dempster, A. P., Laird, N. M., and Rubin, D. B. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological) 39*, 1 (Jan. 1977), 1–38. ArticleType: research-article / Full publication date: 1977 / Copyright © 1977 Royal Statistical Society.

[55] Dienes, P., and Dubey, A. Antecedent recovery: experiments with a trace tagger. In *Proceedings of the 2003 conference on Empirical methods in natural language processing* (Stroudsburg, PA, USA, 2003), EMNLP '03, Association for Computational Linguistics, p. 33–40.

[56] Drepper, U. What every programmer should know about memory. Technical report, Red Hat, Inc., Nov. 2007.

[57] Dreyer, M., and Eisner, J. Better informed training of latent syntactic features. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing* (Stroudsburg, PA, USA, 2006), EMNLP '06, Association for Computational Linguistics, p. 317–326.

[58] Dunlop, A., Bodenstab, N., and Roark, B. Reducing the grammar constant: an analysis of CYK parsing efficiency. Technical report CSLU-2010-02, OHSU, 2010.

[59] Dunlop, A., Bodenstab, N., and Roark, B. Efficient matrix-encoded grammars and low latency parallelization strategies for CYK. In *Proceedings of the 12th International Conference on Parsing Technologies* (Dublin, Ireland, Oct. 2011), ACL, pp. 163–174.

[60] Dunlop, A., and Roark, B. Contrasting objective functions for CYK chart decoding. In *NW-NLP 2012* (Redmond,WA, 2012).

[61] Earley, J. An efficient context-free parsing algorithm. *Commun. ACM 13*, 2 (1970), 94–102.

[62] Eisner, J. Bilexical grammars and their cubic-time parsing algorithms. In *Advances in Probabilistic and Other Parsing Technologies*, H. Bunt and A. Nijholt, Eds. Kluwer Academic Publishers, Oct. 2000, p. 29–62.

[63] Eisner, J., and III, H. D. Speed-accuracy tradeoffs in nondeterministic inference algorithms. In *Proceedings of COST: NIPS 2011 Workshop on Computational Tradeoffs in Statistical Learning* (Sierra Nevada, Spain, 2011).

[64] El-Hassan, F., and Ionescu, D. SCBXP: an efficient hardware-based XML parsing technique. In *5th Southern Conference on Programmable Logic, 2009. SPL* (Apr. 2009), pp. 45 –50.

[65] Ferrucci, D. A. IBM's Watson/DeepQA. *SIGARCH Comput. Archit. News 39*, 3 (June 2011), –.

[66] Finkel, J. R., Kleeman, A., and Manning, C. D. Efficient, feature-based, conditional random field parsing. *IN PROC. ACL/HLT* (2008).

[67] Fisher, S., and Roark, B. The utility of parse-derived features for automatic discourse segmentation. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics* (Prague, Czech Republic, June 2007), ACL, pp. 488–495.

[68] Flynn, M. Some computer organizations and their effectiveness. *IEEE Transactions on Computers C-21*, 9 (Sept. 1972), 948–960.

[69] Gao, J., Andrew, G., Johnson, M., and Toutanova, K. A comparative study of parameter estimation methods for statistical natural language processing. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics* (Prague, Czech Republic, June 2007), Association for Computational Linguistics, p. 824–831.

[70] Germann, U., Joanis, E., and Larkin, S. Tightly packed tries: How to fit large models into memory, and make them load fast, too. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing* (Stroudsburg, PA, USA, 2009), SETQA-NLP '09, Association for Computational Linguistics, p. 31–39.

[71] Godfrey, J. J., Holliman, E. C., and McDaniel, J. SWITCHBOARD: telephone speech corpus for research and development. In *Proceedings of the 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing* (Los Alamitos, CA, USA, 1992), vol. 1, IEEE Computer Society, pp. 517–520.

[72] Goel, V., and Byrne, W. J. Minimum bayes-risk automatic speech recognition. *Computer Speech & Language 14*, 2 (Apr. 2000), 115–135.

[73] Golan, J. S. *Semirings and their Applications*. Springer, July 1999.

[74] Goodman, J. Parsing algorithms and metrics. *Proceedings of the 34th annual meeting on Association for Computational Linguistics* (1996), 177–183.

[75] Goodman, J. Global thresholding and multiple-pass parsing. *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing (EMNLP)* (1997), 11–25.

[76] Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., and Koziris, N. Understanding the performance of sparse matrix-vector multiplication. In *PDP'08: Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (2008).

[77] Graham, S. L., Harrison, M., and Ruzzo, W. L. An improved context-free recognizer. *ACM Trans. Program. Lang. Syst. 2*, 3 (July 1980), 415–462.

[78] Green, Bert F., J., Wolf, A. K., Chomsky, C., and Laughery, K. Baseball: an automatic question-answerer. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference* (New York, NY, USA, 1961), IRE-AIEE-ACM '61 (Western), ACM, p. 219–224.

[79] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. The WEKA data mining software: an update. *SIGKDD Explor. Newsl. 11*, 1 (Nov. 2009), 10–18.

[80] Hart, P., Nilsson, N., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics 4*, 2 (July 1968), 100 –107.

[81] Heafield, K., Koehn, P., and Lavie, A. Language model rest costs and space-efficient storage. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning* (Stroudsburg, PA, USA, 2012), EMNLP-CoNLL '12, Association for Computational Linguistics, p. 1169–1178.

[82] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach, 4th Edition*, 4 ed. Morgan Kaufmann, Sept. 2006.

[83] Hollingshead, K., and Roark, B. Pipeline iteration. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics* (Prague, Czech Republic, June 2007), ACL, p. 952–959.

[84] Hopcroft, J. E., and Ullman, J. D. *Introduction to automata theory, languages, and computation*. Addison-wesley, Reading, Massachusetts, 1979.

[85] Huang, Z., and Harper, M. Self-training PCFG grammars with latent annotations across languages. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 2 - Volume 2* (Stroudsburg, PA, USA, 2009), EMNLP '09, Association for Computational Linguistics, p. 832–841.

[86] IEEE. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008* (Aug. 2008), 1–58.

[87] Im, E.-J., Yelick, K., and Vuduc, R. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications 18*, 1 (Feb. 2004), 135–158.

[88] Jiang, J., Teichert, A., III, H. D., and Eisner, J. Learned prioritization for trading off accuracy and speed. In *Advances in Neural Information Processing Systems 25* (Dec. 2012).

[89] Johnson, M. PCFG models of linguistic tree representations. *Comput. Linguist. 24*, 4 (1998), 613–632.

[90] Johnson, M. A simple pattern-matching algorithm for recovering empty nodes and their antecedents. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Stroudsburg, PA, USA, 2002), ACL '02, Association for Computational Linguistics, p. 136–143.

[91] Johnson, M. lncky, 2006.

[92] Johnson, M. Parsing in parallel on multiple cores and GPUs. In *Proceedings of the Australasian Language Technology Association Workshop 2011* (Canberra, Australia, Dec. 2011), p. 29–37.

[93] Joshi, A., Shanker, K. V., and Weir, D. The convergence of mildly context-sensitive grammar formalisms. *Technical Reports (CIS)* (Jan. 1990).

[94] Joshi, A. K., Levy, L. S., and Takahashi, M. Tree adjunct grammars. *J. Comput. Syst. Sci. 10*, 1 (Feb. 1975), 136–163.

[95] Kalt, T. Induction of greedy controllers for deterministic treebank parsers. In *Proceedings of EMNLP 2004* (Barcelona, Spain, July 2004), D. Lin and D. Wu, Eds., Association for Computational Linguistics, p. 17–24.

[96] Kasami, T. An efficient recognition and syntax-analysis algorithm for context-free languages. Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA, 1965.

[97] Klein, D., and Manning, C. D. Parsing and hypergraphs. In *In IWPT* (2001), p. 123–134.

[98] Klein, D., and Manning, C. D. Parsing with treebank grammars: Empirical bounds, theoretical models, and the structure of the penn treebank. In *Proceedings of 39th Annual Meeting of the Association for Computational Linguistics* (Toulouse, France, July 2001), pp. 338–345.

[99] Klein, D., and Manning, C. D. A* parsing: Fast exact viterbi parse selection. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology (NAACL '03)* (Edmonton, Canada, 2003), pp. 40–47.

[100] Klein, D., and Manning, C. D. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1* (Sapporo, Japan, 2003), ACL, pp. 423–430.

[101] Klein, D., and Manning, C. D. Fast exact inference with a factored model for natural language parsing. In *Advances in Neural Information Processing Systems 15*, S. T. S. Becker and K. Obermayer, Eds. MIT Press, Cambridge, MA, 2003, p. 3–10.

[102] Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., and Cox, D. Design of the java HotSpot™ client compiler for java 6. *ACM Transactions on Architecture and Code Optimization (TACO) 5* (May 2008), 7:1–7:32.

[103] Kuich, W., and Salomaa, A. *Semirings, Automata, Languages.* EATCS Monographs on Theoretical Computer Science, Number 5. Springer-Verlag, Berlin, Germany, 1985.

[104] Kumar, S., and Byrne, W. Minimum bayes-risk decoding for statistical machine translation. In *HLT-NAACL 2004: Main Proceedings* (Boston, Massachusetts, USA,

May 2004), D. M. Susan Dumais and S. Roukos, Eds., Association for Computational Linguistics, p. 169–176.

[105] Kummerfeld, J. K., Hall, D., Curran, J. R., and Klein, D. 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 2012* (2012).

[106] Kummerfeld, J. K., Tse, D., Curran, J. R., and Klein, D. An empirical examination of challenges in chinese parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* (Sofia, Bulgaria, Aug. 2013), Association for Computational Linguistics, p. 98–103.

[107] Ladner, R., Fortna, R., and Nguyen, B.-H. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Experimental Algorithmics*, vol. 2547 of *Lecture Notes in Computer Science*. Springer Berlin, 2002, pp. 78–92.

[108] Lari, K., and Young, S. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech & Language 4*, 1 (Jan. 1990), 35–56.

[109] Le Roux, J., Foster, J., Wagner, J., Kaljahi, R. S. Z., and Bryl, A. DCU-Paris13 systems for the SANCL 2012 shared task. In *Working Notes of SANCL 2012* (Montreal, Quebec, Canada, 2012).

[110] Lee, L. Fast context-free parsing requires fast boolean matrix multiplication. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics* (Madrid, Spain, July 1997), ACL, pp. 9–15.

[111] Li, X., Wang, H., Liu, T., and Li, W. Key elements tracing method for parallel XML parsing in multi-core system. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies* (Dec. 2009), pp. 439 –444.

[112] Lin, D., Foster, D. P., and Ungar, L. H. A risk ratio comparison of l0 and l1 penalized regressions. *University of Pennsylvania, Tech. Rep* (2010).

[113] Lloyd, S. Least squares quantization in PCM. *IEEE Transactions on Information Theory 28*, 2 (1982), 129–137.

[114] Lowerre, B. T. *The Harpy speech recognition system.* Ph.D. thesis, Carnegie Mellon University, Apr. 1976.

[115] Lu, W., Chiu, K., and Pan, Y. A parallel approach to XML parsing. In *7th IEEE/ACM International Conference on Grid Computing* (Sept. 2006), pp. 223 – 230.

[116] Maclachlan, A., and Rambow, O. Cross-serial dependencies in tagalog. In *In Proceedings of the Sixth International Workshop on Tree Adjoining Grammar and Related Frameworks (TAG+6), 100–104, Università di Venezia* (2003).

[117] Macqueen, J. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability* (1967), pp. 281–297.

[118] Magerman, D. M., and Marcus, M. P. *P*earl: a probabilistic chart parser. In *Proceedings of the fifth conference on European chapter of the Association for Computational Linguistics* (Stroudsburg, PA, USA, 1991), EACL '91, Association for Computational Linguistics, p. 15–20.

[119] Manning, C. D., and Schuetze, H. *Foundations of Statistical Natural Language Processing*. The MIT Press, June 1999.

[120] Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. Building a large annotated corpus of english: the penn treebank. *Computational Linguistics 19*, 2 (1993), 313–330.

[121] Marcus, M. P., Santorini, B., Marcinkiewicz, M. A., and Taylor, A. *Treebank-3*. Linguistic Data Consortium, Philadelphia, 1999.

[122] Matsuzaki, T., Miyao, Y., and Tsujii, J. Probabilistic CFG with latent annotations. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics - ACL '05* (Ann Arbor, Michigan, 2005), pp. 75–82.

[123] McDonald, R., Pereira, F., Ribarov, K., and Haji\vc, J. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing* (Stroudsburg, PA, USA, 2005), HLT '05, Association for Computational Linguistics, p. 523–530.

[124] Mejer, A., and Crammer, K. Are you sure? confidence in prediction of dependency tree edges. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (Montréal, Canada, June 2012), Association for Computational Linguistics, p. 573–576.

[125] Moore, R. C. Improved left-corner chart parsing for large context-free grammars. *New developments in parsing technology* (2004), 185–201.

[126] Munshi, A. OpenCL 1.0 specification, Oct. 2009.

[127] Natarajan, B. K. Sparse approximate solutions to linear systems. *SIAM J. Comput. 24*, 2 (Apr. 1995), 227–234.

[128] Ng, A. Y. Feature selection, l1 vs. l2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning* (New York, NY, USA, 2004), ICML '04, ACM, p. 78–.

[129] Nijholt, A. The CYK approach to serial and parallel parsing, June 1991.

[130] Ninomiya, T., Torisawa, K., Kinjiro, T., and Jun'ichi, T. A parallel CKY parsing algorithm on large-scale distributed-memory parallel machines. In *PACLING '97* (Tokyo, Japan, 1997), pp. 223–231.

[131] Nivre, J. Algorithms for deterministic incremental dependency parsing. *Comput. Linguist. 34*, 4 (Dec. 2008), 513–553.

[132] Paleczny, M., Vick, C., and Click, C. The java hotspot server compiler. *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium* (2001), 1–12.

[133] Palis, M. A., and Shende, S. Sublinear parallel time recognition of tree adjoining language. Technical report, ScholarlyCommons@Penn, 1988.

[134] Pan, Y., Lu, W., Zhang, Y., and Chili, K. A static load-balancing scheme for parallel XML parsing on multicore CPUs. In *Seventh IEEE International Symposium on Cluster Computing and the Grid, 2007. CCGRID 2007* (May 2007), pp. 351 –362.

[135] Park, M. Y., and Hastie, T. L1-regularization path algorithm for generalized linear models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 69*, 4 (2007), 659–677.

[136] Pauls, A., and Klein, D. Faster and smaller n-gram language models. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1* (Stroudsburg, PA, USA, 2011), HLT '11, Association for Computational Linguistics, p. 258–267.

[137] Pauls, A., Klein, D., and Quirk, C. Top-down k-best a* parsing. In *Proceedings of ACL 2010* (Morristown, NJ, USA, 2010), p. 200–204.

[138] Penn, G., and Munteanu, C. A tabulation-based parsing method that reduces copying. In *Proceedings of ACL '03* (Sapporo, Japan, 2003), pp. 200–207.

[139] Pereira, F., and Schabes, Y. Inside-outside reestimation from partially bracketed corpora. In *Proceedings of the 30th annual meeting on Association for Computational Linguistics* (Newark, Delaware, 1992), ACL, pp. 128–135.

[140] Petrov, S. *Coarse-to-Fine Natural Language Processing*. PhD thesis, University of California at Bekeley, Berkeley, CA, USA, 2009.

[141] Petrov, S. 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* (Los Angeles, June 2010), ACL, pp. 19–27.

[142] Petrov, S., Barrett, L., Thibaux, R., and Klein, D. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics* (Sydney, Australia, 2006), ACL, pp. 433–440.

[143] Petrov, S., Chang, P.-C., Ringgaard, M., and Alshawi, H. Uptraining for accurate deterministic question parsing. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing* (Stroudsburg, PA, USA, 2010), EMNLP '10, Association for Computational Linguistics, p. 705–713.

[144] Petrov, S., and Klein, D. Improved inference for unlexicalized parsing. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics* (Rochester, New York, Apr. 2007), ACL, pp. 404–411.

[145] Petrov, S., and Klein, D. Discriminative log-linear grammars with latent variables. In *Advances in Neural Information Processing Systems 20 (NIPS)* (Cambridge, MA, 2008), J. C. Platt, D. Koller, Y. Singer, and S. Roweis, Eds., MIT Press, p. 1153–1160.

[146] Petrov, S., and Klein, D. Sparse multi-scale grammars for discriminative latent variable parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing* (Stroudsburg, PA, USA, 2008), EMNLP '08, Association for Computational Linguistics, p. 867–876.

[147] Petrov, S., and McDonald, R. Overview of the 2012 shared task on parsing the web. In *Notes of the First Workshop on Syntactic Analysis of Non-Canonical Language (SANCL)* (2012).

[148] Punyakanok, V., Roth, D., and Yih, W.-t. The importance of syntactic parsing and inference in semantic role labeling. *Computational Linguistics 34*, 2 (2008), 257–287.

[149] R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2012. ISBN 3-900051-07-0.

[150] Rabiner, L. R. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE 77*, 2 (1989), 257–286.

[151] Ratnaparkhi, A. A linear observed time statistical parser based on maximum entropy models. *arXiv:cmp-lg/9706014* (June 1997).

[152] Riley, M., Allauzen, C., and Jansche, M. OpenFst: an open-source, weighted finite-state transducer library and its applications to speech and language. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts* (Boulder, Colorado, May 2009), Association for Computational Linguistics, p. 9–10.

[153] Roark, B., Allauzen, C., and Riley, M. Smoothed marginal distribution constraints for language modeling. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Sofia, Bulgaria, Aug. 2013), Association for Computational Linguistics, p. 43–52.

[154] Roark, B., and Hollingshead, K. Classifying chart cells for quadratic complexity context-free inference. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)* (Manchester, UK, Aug. 2008), D. Scott and H. Uszkoreit, Eds., ACL, pp. 745–752.

[155] Roark, B., and Hollingshead, K. Linear complexity context-free parsing pipelines via chart constraints. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics* (Boulder, Colorado, June 2009), ACL, pp. 647–655.

[156] Roark, B., Hollingshead, K., and Bodenstab, N. Finite-state chart constraints for reduced complexity context-free parsing pipelines. *Computational Linguistics 38*, 4 (Mar. 2012), 719–753.

[157] Roark, B., and Sproat, R. *Computational Approaches to Morphology and Syntax*. Oxford University Press, USA, Sept. 2007.

[158] Roark, B., Sproat, R., and Shafran, I. Lexicographic semirings for exact automata encoding of sequence models. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics* (Portland, Oregon, June 2011), ACL.

[159] Russell, S. J., and Norvig, P. *Artificial Intelligence: A Modern Approach*, 1st ed. Prentice Hall, Jan. 1995.

[160] Rytter, W. Parallel time o(log n) recognition of unambiguous context-free languages. *Information and Computation 73*, 1 (Apr. 1987), 75–86.

[161] Sagae, K., and Lavie, A. A classifier-based parser with linear run-time complexity. In *Proceedings of the Ninth International Workshop on Parsing Technology* (Stroudsburg, PA, USA, 2005), Parsing '05, Association for Computational Linguistics, p. 125–132.

[162] Sagae, K., and Lavie, A. A best-first probabilistic shift-reduce parser. In *Proceedings of the COLING/ACL on Main conference poster sessions* (Stroudsburg, PA, USA, 2006), COLING-ACL '06, Association for Computational Linguistics, p. 691–698.

[163] Schraudolph, N. A fast, compact approximation of the exponential function. *Neural Computation 11*, 4 (1999), 853–862.

[164] Schwarz, G. Estimating the dimension of a model. *The Annals of Statistics 6*, 2 (Mar. 1978), 461–464. Mathematical Reviews number (MathSciNet): MR468014; Zentralblatt MATH identifier: 0379.62005.

[165] Seddah, D., Candito, M., and Anguiano, E. H. A word clustering approach to domain adaptation: Robust parsing of source and target domains. *Journal of Logic and Computation* (Feb. 2013).

[166] Shieber, S. M. Evidence against the context-freeness of natural language. *Linguistics and Philosophy 8*, 3 (Aug. 1985), 333–343.

[167] Skut, W., Krenn, B., Brants, T., and Uszkoreit, H. An annotation scheme for free word order languages. In *Proceedings of the fifth conference on Applied natural language processing* (Stroudsburg, PA, USA, 1997), ANLC '97, Association for Computational Linguistics, p. 88–95.

[168] Smith, N. A., and Johnson, M. Weighted and probabilistic context-free grammars are equally expressive. *Computational Linguistics 33*, 4 (Dec. 2007), 477–491.

[169] Song, X., Ding, S., and Lin, C.-Y. Better binarization for the CKY parsing. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing* (Honolulu, Hawaii, Oct. 2008), ACL, pp. 167–176.

[170] Strassen, V. Gaussian elimination is not optimal. *Numerische Mathematik 13*, 4 (1969), 354–356.

[171] Tarjan, R. E., and Yao, A. C.-C. Storing a sparse table. *Communications of the ACM 22*, 11 (1979), 606–611.

[172] Tene, G., Choquette, J. H., Sellers, S., and Click, C. N. Array access, Aug. 2009. US Patent 7,577,801.

[173] Tewarson, R. P. *Sparse Matrices. Mathematics in Science and Engineering Volume 99*. Academic Press, Apr. 1973.

[174] Thue, A. *Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln, von Axel Thue...* J. Dybwad, 1914.

[175] Tibshirani, R. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological) 58*, 1 (1996), 267–288.

[176] Tikhonov, A. N. On the stability of inverse problems. In *Dokl. Akad. Nauk SSSR* (1943), vol. 39, p. 195–198.

[177] Valiant, L. G. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences 10*, 2 (Apr. 1975), 308–314.

[178] Viterbi, A. J. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory 13*, 2 (1967), 260–269.

[179] Vogel, S., Ney, H., and Tillmann, C. HMM-based word alignment in statistical translation. In *Proceedings of the 16th conference on Computational linguistics - Volume 2* (Stroudsburg, PA, USA, 1996), COLING '96, Association for Computational Linguistics, p. 836–841.

[180] Wang, Z., and Zong, C. Phrase structure parsing with dependency structure. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters* (Stroudsburg, PA, USA, 2010), COLING '10, Association for Computational Linguistics, p. 1292–1300.

[181] Winograd, T. *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*. Ph.D. thesis, Massachusetts Institute of Technology, Feb. 1971.

[182] Wirén, M. A comparison of rule-invocation strategies in context-free chart parsing. In *Proceedings of the third conference on European chapter of the Association for Computational Linguistics* (Stroudsburg, PA, USA, 1987), EACL '87, Association for Computational Linguistics, p. 226–233.

[183] Wu, C. On the convergence properties of the EM algorithm. *The Annals of Statistics 11*, 1 (1983), 95–103.

[184] Würthinger, T., Wimmer, C., and Mössenböck, H. Array bounds check elimination for the java HotSpot\™ client compiler. *Proceedings of the 5th international symposium on Principles and practice of programming in Java* (2007), 125–133.

[185] Xue, N., Xia, F., Chiou, F.-D., and Palmer, M. The penn chinese TreeBank: phrase structure annotation of a large corpus. *Natural Language Engineering 11*, 02 (2005), 207–238.

[186] Yi, Y., Lai, C.-Y., Petrov, S., and Keutzer, K. Efficient parallel CKY parsing on GPUs. In *Proceedings of the 12th International Conference on Parsing Technologies* (Dublin, Ireland, Oct. 2011), pp. 175–185.

[187] Yonezawa, A., and Ohsawa, I. Object-oriented parallel parsing for context-free grammars. In *Proceedings of the 12th conference on Computational linguistics - Volume 2* (Stroudsburg, PA, USA, 1988), COLING '88, ACL, p. 773–778.

[188] Younger, D. H. Recognition and parsing of context-free languages in time n^3. *Information and Control 2*, 10 (1967), 189–208.

Appendix A

Search Pruning Methods

Inference speed depends greatly on the size of the search space. As noted in Section 2.3, full CYK inference is $O(n^3)$, but heuristic pruning can reduce the space, often with minimal impact on accuracy. Common pruning methods include beam search, coarse-to-fine [142], A* [99, 137], best-first [28, 32], and beam search [47]. Pruning reduces the search space greatly, allowing effective search in reasonable time, although (of the methods listed), only A* guarantees finding the globally optimal solution.

Search pruning approaches are not a principal contribution of this thesis, but will make use of several recent pruning approaches in our learning and inference trials, and compare them in a variety of domains. We chose these pruning approaches for the following reasons: 1) They are rapid to train for an arbitrary grammar; 2) They are (relatively) easy to tune such that the maximum-likelihood tree remains in the beam, thus retaining the accuracy of exhaustive search; and 3) They provide state-of-the-art efficiency, yielding a very strong baseline.

This section introduces briefly some of the pruning methods implemented in the BUBS parser, focusing on the algorithms we utilize in chapters 3–6. The approaches are described more fully in Roark and Hollingshead [154], Roark and Hollingshead [155], Roark et al. [156], and Bodenstab et al. [21].

A.1 Agenda Parsing and A*

Context-free parsing can be formulated as search over a hypergraph for a target node, corresponding to the CFG start symbol $S^{\dagger}$ [97]. The A* graph-search algorithm [80], as extended for hypergraphs by Klein and Manning [101] reduces the graph search space while ensuring the optimal solution is returned. A* (and many other graph-search methods) depends on a *heuristic* — an estimate of the cost of the remaining distance from a source

node to the target node. The search is prioritized by the combination of the (known) cost from the start node $g(\cdot)$ and the estimated cost $h(\cdot)$ to the target node. To guarantee that the first solution found will be optimal, $h(\cdot)$ must be *admissible* — that is, it must never overestimate the true cost to the target [159]. The more closely $h(\cdot)$ approximates $\alpha(\cdot)$, the more quickly an agenda search will complete.

While agenda search can be quite effective for CFG parsing [32], managing the a global agenda and computing complex prioritization functions is quite expensive [19]. However, similar heuristic estimates can be applied effectively over a smaller context within a standard bottom-up search.

A.2 Beam Search and Prioritization Functions

Beam search [114] is a heuristic best-first graph search algorithm which limits the number of candidate solutions explored at each step. The number of solutions explored (the 'beam width') is often fixed, but can be varied during search to adapt to contexts of greater or lesser ambiguity (c.f. Bodenstab et al. [21]). In the context of CYK chart parsing, a beam search usually means limiting the cell population — the number of non-terminals populated in a cell. Beam search requires a heuristic prioritization of the candidate sub-solutions (also called a 'local prioritization', or 'figure-of-merit'). The inside probability of each non-terminal label is one simple and effective heuristic — we simply rank candidates by inside probability and retain the n most probable. In empirical trials with a 6-cycle latent-variable grammar, we found we can reduce the beam to approximately 100 before accuracy begins to decline (the total vocabulary of this grammar is over 1100, and exhaustive search averages over 400 entries per cell, so a beam of 100 is constraining the search greatly).

We can improve on inside probability by estimating the candidates' posterior probability as well. Combining with a heuristic estimate of the outside probability with the (known) inside probability gives us an estimate of the posterior [28]. Bodenstab [19] presented a lexicalized prioritization function which provide dramatically improved rankings, maintaining accuracy with a beam of 30 or fewer on the same 6-cycle grammar. A straightforward beam search improves efficiency by orders of magnitude [19], but combining beam search with coarse-to-fine pruning approaches can yield further gains.

A.3 Coarse-to-fine

Coarse-to-fine (CTF) search proceeds in 2 or more stages, first processing the sequence with a simpler 'coarse' model and using the output of that stage to guide and constrain subsequent inference with a more accurate model. The general CTF approach is common in many NLP tasks, and is described well in Petrov [140]. When applied to constituency parsing, CTF approaches generally involve annotating cells of the parse chart with constraints from the coarse stage which can then be applied during the fine stage(s). For example, the Charniak-Johnson parser [32, 34] performs an agenda parse with a finely-tuned bilexical grammar, requiring computation of probabilities and smoothing during inference. These on-the-fly calculations are quite expensive, so it first parses with a small, explicitly enumerable PCFG, and then guides its agenda parse using the posterior probabilities from the initial search. Petrov et al. [142] adopts a similar approach, extending it to a multi-level hierarchical grammar trained with the algorithm described in Section 2.9.

In this thesis, we make use of CTF approaches which perform $O(n)$ or $O(n^2)$ pre-processing stages, prior to the $O(n^3)$ chart-parsing operation. The following sections introduce several examples of such approaches.

A.4 Chart Cell Constraints

Roark and Hollingshead [154] presented a tagging approach that *closes* certain chart cells, disallowing entries in positions unlikely to contribute to a complete parse tree. For example, the word *the* is quite unlikely to end a complete constituent (subtree). Cells which require *the* to end a subtree can be closed, reducing the search space. To apply this approach, we train a pair of taggers to classify each word in the target sentence as potentially capable of beginning or ending a subtree. We want to run these two taggers at a very high-precision operating point, such that the vast majority of the 'cannot-begin' and 'cannot-end' classifications are correct.

We then extend the classifications upwards in the chart along a diagonal from each word, marking the cells which would contribute to disallowed subtree constructions, as demonstrated in Figure A.1. For simplicity, we omit here the details of accommodating incomplete constituents (factored categories) and the time-complexity proofs; both are detailed in the original publications [154, 155, 156].

Bodenstab et al. [21] introduced a modification of this approach they called *complete*

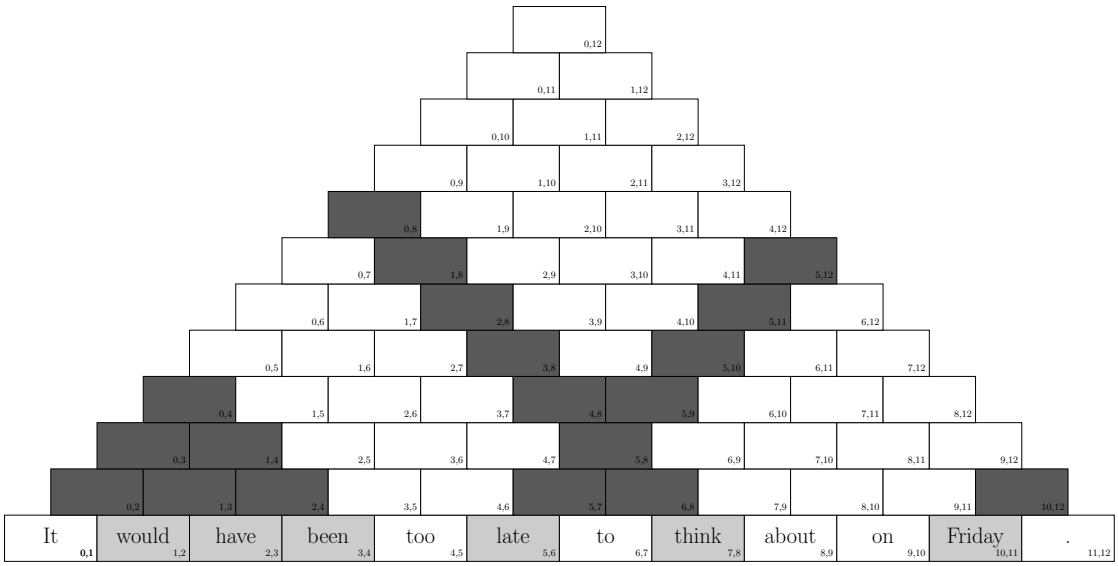


Figure A.1: CYK chart highlighting cells closed by Roark and Hollingshead [154] chart cell constraints. The words 'late' and 'Friday' are highlighted as unable to begin a multi-word constituent, and 'would', 'have', 'been', and 'think' as unable to end one. The cells closed by those constraints are marked in dark gray. (Note: this simple example ignores the nuances of factored categories and partially-open cells).

closure, which applied a similarly-trained classifier to individual chart cells instead of terminals. Figure A.2 displays a sample chart. Complete closure performs $O(n^2)$ open/closed classifications (rather than $O(n)$), allowing in some cases finer distinctions between cells within a diagonal. Chart-constraint methods are very effective, reducing average-case parse times by an order of magnitude or more, and both variants interoperate smoothly with beam search and local prioritization.

A.5 Adaptive Beam Models

Bodenstab et al. [21] further refined and generalized cell constraints. A local prioritization function — as described in Section A.2 — ranks non-terminal labels within each cell. An effective prioritization function will often rank very highly the labels which participate in the final parse tree. Thus, a fixed beam width often includes many unnecessary cell entries.

An *Adaptive Beam* model is a regression model, trained to predict the appropriate

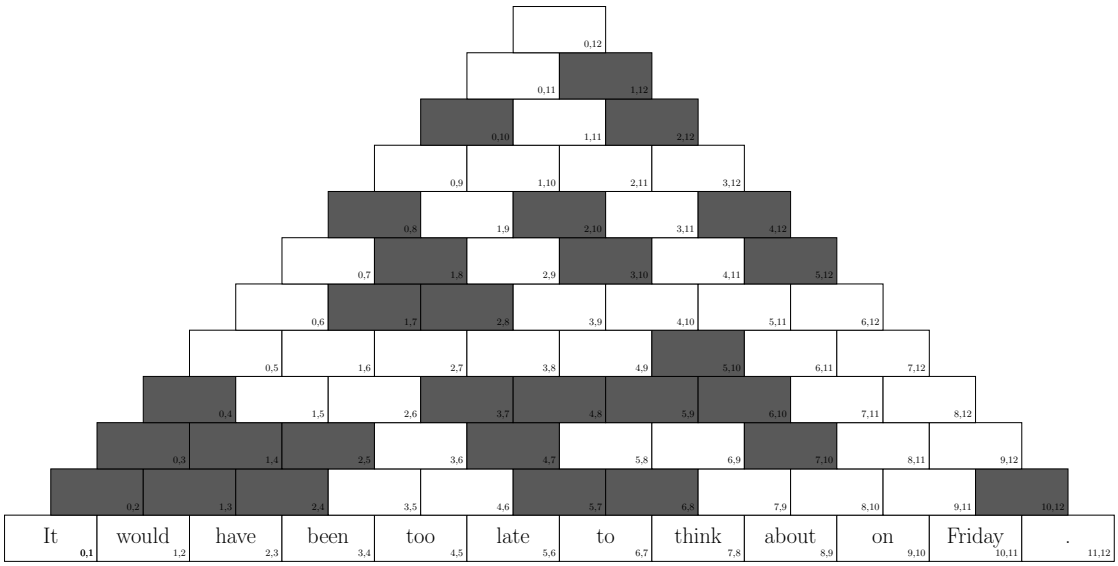


Figure A.2: CYK chart highlighting cells closed by 'complete closure' [21]. Note the similarities and differences between this chart and that in Figure A.1. More cells are closed, but not always in straight diagonals.

population of each individual chart cell. When the model predicts less ambiguity, the cell population is reduced below the maximum beam width. For some unambiguous cells, the modeled beam width may be 1 (i.e., the top-ranked constituent is the only one stored in the chart). As in complete closure, the model may also close cells (i.e., predict a beam width of 0). Adapting the beam width on a per-cell basis allows heavier pruning in less-ambiguous areas of the chart, without risking search errors in more other contexts.

While this is fundamentally a regression problem (i.e., for each cell, predict the appropriate beam width), Bodenstab et al. [21] found it more effective to train a small set of binary classifiers — e.g., beam width=0, beam-width=1, beam-width=2, ... — and combine them into a single model. All adaptive-beam pruning models used in this thesis follow the same approach.

Biographical Note

Aaron J. Dunlop was born in January 1975, in Portland Oregon. He received his Bachelor of Arts in Computer Science from Willamette University in 1996 and his Masters of Science from Oregon Health & Science University in 2008. He continued his studies at OHSU, doing his doctoral research while working as a Software Engineer at TransCore and later at Intel. His professional interests include syntactic parsing, language modeling, and information extraction. He has co-authored several papers in peer-reviewed conferences and other venues.