THE UNIVERSITY of EDINBURGH

Integrated Supertagging and Parsing

Michael Auli

Doctor of Philosophy

Institute for Language, Cognition and Computation

School of Informatics

University of Edinburgh

2012

Abstract

Parsing is the task of assigning syntactic or semantic structure to a natural language sentence. This thesis focuses on syntactic parsing with Combinatory Categorial Grammar (CCG; Steedman 2000). CCG allows incremental processing, which is essential for speech recognition and some machine translation models, and it can build semantic structure in tandem with syntactic parsing. Supertagging solves a subset of the parsing task by assigning lexical types to words in a sentence using a sequence model. It has emerged as a way to improve the efficiency of full CCG parsing (Clark and Curran, 2007) by reducing the parser's search space. This has been very successful and it is the central theme of this thesis.

We begin by an analysis of how efficiency is being traded for accuracy in supertagging. Pruning the search space by supertagging is inherently approximate and to contrast this we include A* in our analysis, a classic exact search technique. Interestingly, we find that combining the two methods improves efficiency but we also demonstrate that excessive pruning by a supertagger significantly lowers the upper bound on accuracy of a CCG parser.

Inspired by this analysis, we design a single integrated model with both supertagging and parsing features, rather than separating them into distinct models chained together in a pipeline. To overcome the resulting complexity, we experiment with both loopy belief propagation and dual decomposition approaches to inference, the first empirical comparison of these algorithms that we are aware of on a structured natural language processing problem.

Finally, we address training the integrated model. We adopt the idea of optimising directly for a task-specific metric such as is common in other areas like statistical machine translation. We demonstrate how a novel dynamic programming algorithm enables us to optimise for F-measure, our task-specific evaluation metric, and experiment with approximations, which prove to be excellent substitutions.

Each of the presented methods improves over the state-of-the-art in CCG parsing. Moreover, the improvements are additive, achieving a labelled/unlabelled dependency F-measure on CCGbank of 89.3%/94.0% with gold part-of-speech tags, and 87.2%/92.8% with automatic part-of-speech tags, the best reported results for this task to date. Our techniques are general and we expect them to apply to other parsing problems, including lexicalised tree adjoining grammar and context-free grammar parsing.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Michael Auli*)

Preface

Most of the material presented in this thesis has been published. This applies to
(i) Chapter 4, which has been published in Auli and Lopez (2011b),
(ii) Chapter 5, which has been published in Auli and Lopez (2011a), and
(iii) Chapter 6, which has been published in Auli and Lopez (2011c).

Acknowledgements

I would like to thank my supervisors Philipp Koehn and Adam Lopez who guided me through the PhD process. Philipp Koehn inspired me to do research and allowed me to freely pursue my academic interests. Adam Lopez has given me invaluable guidance and support throughout my PhD.

The work in my PhD has been made possible by numerous past and present members of the CCG group at the University of Edinburgh as well as other people within the School of Informatics.

I would also like to acknowledge the financial assistance from an EPSRC scholarship (grant EP/P504171/1) and the EuroMatrixPlus project funded by the European Commission, 7th Framework Programme as well as a grant from Johns Hopkins University, which allowed me to visit this institution to broaden my knowledge. I have also made extensive use of the resources provided by the Edinburgh Compute and Data Facility (`http://www.ecdf.ed.ac.uk/`). The ECDF is partially supported by the eDIKT initiative (`http://www.edikt.org.uk/`).

Table of Contents

Chapter 1

Introduction

Parsing is the task of assigning syntactic structure to a sentence.[1] The inherent ambiguity in language makes this very challenging since there is a vast number of possible analyses for even the most simple sentences (Figure 1.1). Parsing is much studied in natural language processing and there is a high demand for both accurate and efficient parsers whose predictions can serve as input to downstream applications such as machine translation (Galley et al., 2004, 2006) or speech recognition (Chelba and Jelinek, 1998).

In this thesis we focus on Combinatory Categorial Grammar (CCG; Steedman 2000). CCG has many properties that make it attractive for downstream applications, including a fine-grained non-terminal set that has been successfully used within statistical machine translation models (Birch et al., 2007; Hassan et al., 2007) as well

[1] Parsing may also assign semantic structure but this thesis focuses on the syntactic component.

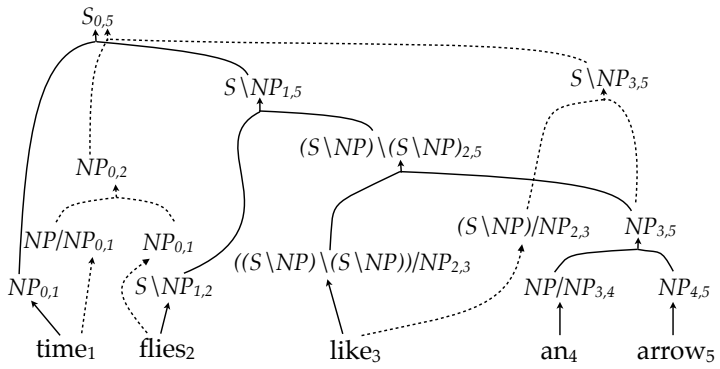


Figure 1.1: Representation of two possible syntactic structures, amongst many, for the sentence *time flies like an arrow*, the first is in solid and the second in a dashed line.

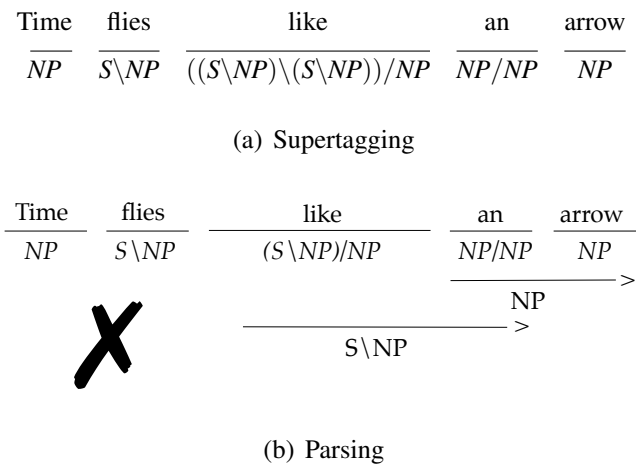


Figure 1.2: Illustration of supertagging (a) and a parse failure with wrong supertags (b).

as a natural representation of non-constituents which frequently occur in translation, e.g., CCG can assign syntactic or semantic structure to a phrase like *Peter is*. The latter property motivated the proposal of an actual parsing-based CCG model for machine translation (Auli, 2009). CCG is also suitable for incremental processing of language from left to right and has been used in this way for dialogue systems (Kruijff et al., 2007). Further applications include building semantic interpretations (Bos et al., 2004), language generation (White and Baldridge, 2003) and semantic parsing (Zettlemoyer and Collins, 2007; Kwiatkowski et al., 2010, 2011). Despite our focus on CCG and syntactic parsing, we would like to stress that the algorithms and models presented in this thesis are equally applicable to other grammar formalisms as well as semantic parsing.

The central theme of this thesis is the common use of a tagger with a parser. Tagging is the task of assigning lexical types, such as Parts-of-Speech, to each word in a sentence using a simple sequence model. Lexicalised formalisms usually rely on a much larger set of lexical types, or *supertags*, than standard context-free Penn Treebank grammars. Tagging with supertags is known as *supertagging* (Bangalore and Joshi, 1999), illustrated in Figure 1.2(a). It differs from full parsing in that overall grammaticality cannot be guaranteed (Figure 1.2(b)[2]) which in turn allows the use of much simpler and more efficient models. The dominant use of supertagging is to prune the set of lexical types considered by a parser. This approach chains supertagging and parsing in a pipeline in which the decisions of the supertagger directly influence the

[2]The parser we use in this thesis would actually be able to find an analysis for this sentence due to its use of type-changing rules (§3.3.3) which do not correspond to formal CCG combinatory rules (§2.1.2).

parser. The technique has been widely successful and drives the broad-coverage C&C parser (Clark and Curran, 2007). Although very effective, this technique is inherently approximate since it removes part of the parser's search space.

The first question we address in this thesis is how supertagging affects end-to-end parsing performance both in terms of efficiency and accuracy. Since supertagging is approximate, we compare and contrast it against exact search with A*, a classic heuristic search algorithm.

Although efficient, the pipeline approach does not allow the parser to recover from errors made by the tagger, nor does the parser use the probabilities estimated by the supertagger. Disregarding these probabilities is wasteful because the parser usually has a poorer model for lexical types. We address these problems by introducing a single integrated model of parsing and supertagging which does not suffer from these shortcomings. This integrated model is substantially more complex than the original pipeline approach and we need to be able to efficiently compute expectations for training and maximum probability solutions for parsing. We solve this by dual decomposition and loopy belief propagation approaches to inference during parsing and piecewise estimation during training. For our model, the inference approaches are based on extensions and generalisations of classical algorithms from the natural language processing literature.

The next problem we address is training the integrated model in order to further improve accuracy. We adopt the idea of optimising directly for a task-specific metric such as is common in statistical machine translation (Och, 2003). In our setting this reduces to augmenting the model expectations by a loss function representing the task-specific metric. For this purpose we introduce a novel dynamic programming algorithm that allows us to directly optimise sentence-level F-measure. Using these methods we achieve the best published performance to date on CCGbank, a CCG version of the Penn Treebank.

1.1 Outline of the Dissertation

Chapter 2 introduces relevant background to this thesis. We give an introduction to the CCG formalism and an overview of supertagging, focusing on the baseline against which we compare the integrated model. Next, we discuss probabilistic models and algorithms for training and decoding parsing and supertagging models. These algorithms are special cases of a general algorithm known as belief propagation. Finally,

we explain previously proposed models for CCG parsing and supertagging in terms of the algorithms discussed earlier.

Chapter 3 describes the experimental setup we use throughout the thesis. We discuss the parsing task, the evaluation methods, the baseline parsing and supertagging models as well as their configuration and estimation. This includes the previous state-of-the-art CCG parsing model of Clark and Curran (2007) which is the baseline for our integrated model.

Chapter 4 is about efficient search for CCG parsing. We present a thorough analysis of our baseline approach, the combination of supertagging and parsing in a pipeline, an approximate search technique. We compare and contrast this with A*, an efficient heuristic exact search algorithm. Finally, we combine A* and supertagging to effectively improve inexact search with an exact search technique.

In Chapter 5 we introduce an integrated model of both parsing and supertagging. We motivate the use of this model via an oracle experiment, showing that the upper bound on accuracy of a CCG parser is significantly lowered when its search space is pruned by a supertagger. The resulting increase in complexity is tackled with loopy belief propagation and dual decomposition approaches to inference.

In Chapter 6 we improve the integrated model by optimising each sub-model for a separate task-specific metric using the softmax-margin objective. We present novel dynamic programs to optimise towards F-measure in an exact fashion. We compare this to more efficient approximate methods to measure the reliability of these methods.

Finally, in Chapter 7 we conclude and present avenues for future work.

1.2 Research Contributions

This dissertation makes the following important research contributions.

- We describe a set of innovations which result in the most accurate CCG parser in the literature to date, raising parsing accuracy on CCGbank from 87.7% to 89.3% labelled F-measure with gold part-of-speech tags. Moreover, we introduce a new and principled approach to supertagging and parsing.

- We demonstrate that the interaction between a parser and a supertagger can be better exploited in an integrated model rather than simply using the supertagger to prune the parser's search space.

- We present the first empirical comparison of dual decomposition and loopy belief propagation on a structured prediction task. We find that the accuracy of both inference methods is nearly indistinguishable. Furthermore, we show that for our model the vast majority of initial solutions recovered by belief propagation is exact, despite a lack of formal guarantees. Recovering the same number of exact solutions takes many iterations for dual decomposition.

- We show that supertagging, while increasing efficiency, significantly lowers the upper bound on parsing accuracy.

- We present a novel dynamic programming algorithm to compute exact loss functions corresponding to labelled F-measure. We also experiment with approximations of sentence-level F-measure and find that they are excellent substitutes for exact loss functions, despite being computationally much less demanding. In fact, they are as easy to use as standard conditional log-likelihood while substantially improving accuracy.

- We demonstrate the viability of A* as a search algorithm for parsing with expressive grammar formalisms such as CCG, and what we believe to be the first evaluation of A* parsing on the stringent metric of CPU time.

Chapter 2

Background

This chapter covers the relevant background on which we will build in the rest of this thesis.

We begin by introducing Combinatory Categorial Grammar (CCG), the formalism we use throughout the thesis. We will discuss the two main building blocks of a CCG grammar, namely the lexicon and combinatory rules with which derivations can be formed, that is syntactic analyses for a sentence. CCG recovers semantically informed dependencies between predicates and arguments. For example, in *I like to run*, there is a dependency between *I* and *run*, whereas a standard dependency grammar would not assign such a dependency (§2.1).

Next, we discuss supertagging which is typically cast as a sequence modelling problem over lexical types for words, also referred to as supertags. A supertagged sentence vastly reduces the ambiguity a parser has to resolve, but it does not create the dependency links that parsing does. The predominant use of supertaggers is to prune the search space of a parser to improve efficiency. This is the basis for the state-of-the-art CCG parsing model (Clark and Curran, 2007), the baseline for our integrated supertagging and parsing model (§2.2).

Parsing and supertagging are usually modelled probabilistically by either generative or discriminative models. We focus on the discriminative approach and describe how the model parameters can be learned from data samples using expectations, that is the expected number of times a model believes an event *should* occur, compared to how often it actually *does* occur in the training sample. Once we have found the parameters, we can use the model for decoding, the task of predicting the best output given an input. Models for parsing and tagging differ in their structure and therefore different, specialised algorithms have been introduced. These algorithms can be used

to compute maximum probability solutions for decoding or expectations for training. For parsing we use the inside and outside algorithms and for tagging the forward and backward algorithms. We will show that they are all special cases of the general belief propagation algorithm which can be used to compute expectations and maximum solutions on general model structures (§2.3).

2.1 Combinatory Categorial Grammar

In the introduction (Chapter 1) we have briefly motivated the use of Combinatory Categorial Grammar (CCG; Steedman 2000) for machine translation and speech recognition. However, CCG has a range of other advantages such as the ability to recover semantically motivated dependencies, so called predicate-argument relations. CCG is particularly well suited to recover long-distance relationships between words. These dependencies are usually not recovered by standard treebank parsers such as those presented by Charniak (2000) and Collins (2003) which require dedicated post-processing steps for this purpose.

Furthermore, CCG is particularly interesting for semantic parsing (Bos et al., 2004; Bos, 2005; Zettlemoyer and Collins, 2007; Kwiatkowski et al., 2010, 2011) since the formalism provides a transparent interface between syntax and semantics, which allows semantic structure to be built in tandem with syntactic structure, providing a so-called *compositional semantics*. Steedman (2000) covers this topic in depth and we will not deal with it in this thesis since our focus is on syntactic parsing. However, all the algorithms we present still apply in this setting.

2.1.1 Categories and the Lexicon

The lexicon specifies for each word lexical categories which are either atomic or complex. Atomic categories are for example S (sentence), N (noun), NP (noun phrase), or PP (prepositional phrase); they can be further refined by features such as number, case or inflection. Complex categories are functors specifying directionality and valency by the slashes of their arguments; they are recursively built from basic categories. For example, the transitive verb *proves* carries the complex category $(S\backslash NP)/NP$, specifying that it takes first a noun phrase from the right and then a second noun phrase from the left to form a sentence. Lexical categories are also known as *supertags* and can be assigned with a supertagger as discussed in §2.2. The arguments of categories may be

numbered from left to right using subscripts to identify them uniquely such as in the following example lexicon:

$$\begin{aligned} \mathit{Mark} &\vdash NP \\ \mathit{completeness} &\vdash NP \\ \mathit{proves} &\vdash (S\backslash NP_1)/NP_2 \\ \mathit{walks} &\vdash S\backslash NP_1 \end{aligned}$$

where *Mark* and *completeness* are noun phrases, *proves* is a transitive verb and *walks* is an intransitive verb. The entries may be augmented by semantic interpretations in order to build semantic structure in parallel with syntactic structure. Steedman (2000) provides a detailed account of this topic.

2.1.2 Derivations and Predicate-Argument Relations

In parsing, adjacent spans are combined using a small number of binary combinatory rules on the spanning categories. The resulting structure is referred to as a **derivation**; we detail efficient algorithms for this task in §2.3.4. The categories are combined using a small set of combinatory rules, such as forward application ($>$) and backward application ($<$) which were inherited from Categorial Grammar (Bar-Hillel, 1953; Wood, 1993):

$$X/Y \quad Y \quad \Rightarrow \quad X \quad (>) \tag{2.1}$$

$$Y \quad X\backslash Y \quad \Rightarrow \quad X \quad (<) \tag{2.2}$$

where X and Y denote either basic or complex categories. Forward application combines a constituent of the form X/Y, followed by a constituent Y, into X, similarly for backward application.

Derivations are written as in Figure 2.1; underlines indicate combinatory reduction and arrows indicate the direction of the application:

$$\dfrac{\dfrac{\text{Mark}}{NP} \quad \dfrac{\dfrac{\text{proved}}{(S\backslash NP)/NP} \quad \dfrac{\text{completeness}}{NP}}{S\backslash NP}>}{S}<$$

Figure 2.1: Example CCG derivation using forward and backward application.

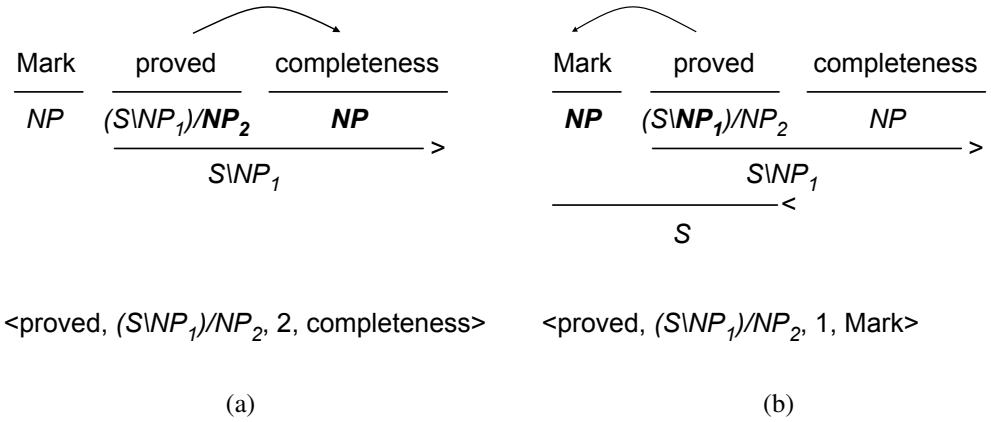


Figure 2.2: Illustration of creating predicate-argument relations during two derivation steps. In (a) the forward application ($>$) fills the object noun-phrase argument of *proved*, in bold, with *completeness* and creates a dependency between the two words, illustrated by the arrow. Similarly, in (b) the backward application ($<$) creates a dependency between *proved* and *Mark*. The relations built in each step are shown below the derivations; arrows point from predicates to arguments.

In this example, a predicate-argument relation is created in each derivation step, as illustrated in Figure 2.2. Formally, a predicate-argument relation is defined as a four-tuple (Clark and Hockenmaier, 2002; Hockenmaier, 2003a): $\langle h, c, s, d \rangle$ where h is a head word, c is the lexical category associated with the head word, s is the argument number of the lexical category filled by the dependency, and d is head word of the argument; a head word is the single most characterising word of a syntactic constituent.

A **predicate-argument structure** $y = \{\tau_1, \tau_2, \ldots\}$ is a set of predicate-argument relations τ_i; sometimes predicate-argument structures are also referred to as **dependency structures** and the relations are simply referred to as dependencies. Predicate-argument structures are the *main syntactic output* of a CCG parser and form the basis for evaluation (§3.2). It is worthwhile to point out that there can be multiple derivations for a single predicate-argument structure (§2.1.3) and that the actual syntactic output is not the derivation but the predicate-argument structure.

Difference to Standard Dependency-Grammars. It is worthwhile to point out that predicate-argument relations in CCG are different to the dependencies used by dependency parsers such as McDonald (2006) and Nivre et al. (2006). Standard dependency grammars allow only relations in which each word has exactly one head. This results

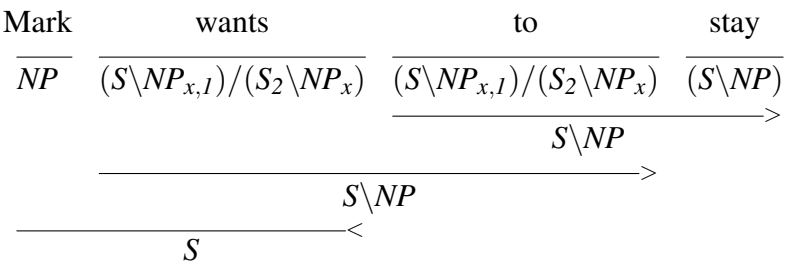


Figure 2.3: CCG derivation for the sentence used in Figure 2.4.

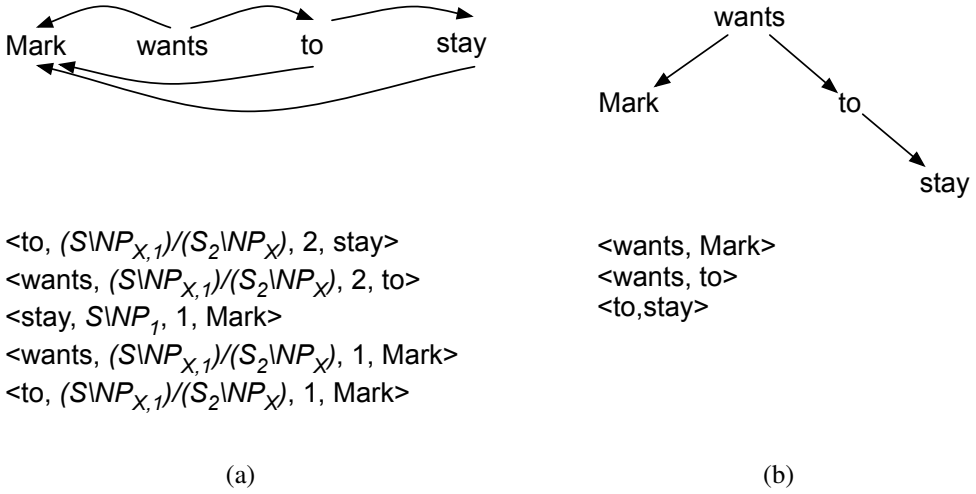


Figure 2.4: Illustration of the dependencies created in the CCG derivation of Figure 2.3 in (a) and a standard dependency analysis (b).

in a tree-shaped dependency graph with $n-1$ edges for an n-word sentence where each edge corresponds to a head-dependent relationship between two words. For CCG, the number of dependents is determined by the number of arguments each word takes according to its lexical category. Figure 2.3 shows a CCG derivation for a simple sentence and Figure 2.4 compares the predicate-argument relations for this derivation to the dependencies of a standard dependency grammar. The dependency analysis has three dependencies, whereas the CCG analysis has five predicate-argument relations in which *wants* and *to* have multiple heads. The CCG derivation contains lexical categories with variables x, these variables allow the creation of long-range dependencies by passing head information between categories via unification. For example, *to* has the lexical category $(S\backslash NP_{x,1})/(S_2\backslash NP_x)$ which identifies the head of argument NP_x also as the head of $NP_{x,1}$. The head being passed by the category for *to* is *stay*, which results in an extra dependency between *stay* and *Mark*. The distinction between CCG

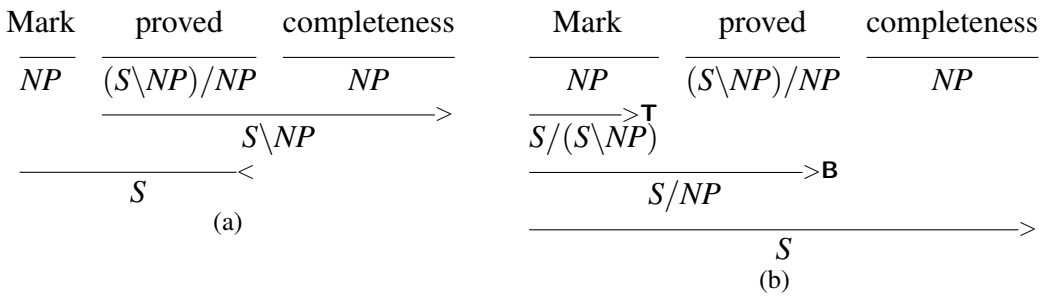


Figure 2.5: Illustration of two derivations for the same sentence resulting in the same predicate-argument relations. The example in (a) first fills the object noun phrase of *proved* and then the subject noun phrase, whereas the second example (b) does it in reverse order.

predicate-argument relations and standard dependency grammars is sometimes also referred to as "deep syntax" versus "shallow syntax", respectively.

2.1.3 Spurious Ambiguity and Normal-form Derivations

In the previous section we have discussed predicate-argument relations and how they are created during the derivation process. In this section we will describe the rules causing spurious ambiguity and how to deal with this ambiguity, in order to make practical parsing possible.

CCG extends classical Categorial Grammar (Ajdukiewicz, 1935; Bar-Hillel, 1953) by a range of combinatory rules such as composition and type-raising. Forward ($>\mathbf{B}$) and backward composition ($<\mathbf{B}$) combines two complex categories to form a new functor:

$$X/Y \quad Y/Z \quad \Rightarrow_{\mathbf{B}} \quad X/Z \quad (>\mathbf{B}) \tag{2.3}$$

$$Y\backslash Z \quad X\backslash Y \quad \Rightarrow_{\mathbf{B}} \quad X\backslash Z \quad (<\mathbf{B}) \tag{2.4}$$

Type-raising takes a category X and turns it into a complex functor, $T/(T\backslash X)$ or $T\backslash(T/X)$, which can compose with other categories, where T can be instantiated with any category that takes X as an argument.

These rules allow a different analysis of our earlier example (Figure 2.1), shown in Figure 2.5 where we type-raise *Mark* and then compose it with *proved*. This creates the same predicate-argument relations as in the original derivation but in a different order. If a grammar permits multiple derivations to infer a single set of predicate-argument

relations, then this is referred to as **spurious ambiguity**.

Normal-Form Derivations. One way to deal with spurious ambiguity is to consider only *normal-form* derivations which use composition and type-raising only when forward and backward application are not sufficient (Eisner, 1996). For a grammar which does not use type-raising, the **Eisner constraints** (Eisner, 1996) can remove spurious ambiguity via restricting analyses such that there is only a single derivation for each predicate-argument structure[1]. This can be accomplished by the following restriction: Any constituent that is the result of forward composition cannot serve as the primary (or left) functor for another instance of forward composition or forward application. Similarly, any constituent based on backward composition cannot serve as the primary (right) functor for another backward composition or backward application.

Here we have focused on the most common combinatory rules. A range of other combinatory rules such as backward crossing composition and substitution are detailed in Steedman (2000).

2.2 Supertagging

Supertagging (Bangalore and Joshi, 1999; Clark, 2002) is the task of assigning lexical types or **supertags** to each word in a sentence; for CCG these are the lexical categories discussed in §2.1.1. Supertagging has been successfully applied to both Lexicalised Tree Adjoining Grammar (LTAG; Schabes 1991) by Bangalore and Joshi (1999) and to CCG by Clark (2002) and Curran et al. (2006).

In this section we will introduce supertagging as a tagging problem which relies on a simple sequence model over words (§2.2.1). We will also discuss how a supertagged sentence has much ambiguity removed, which is why it is sometimes referred to as almost parsing. In §2.2.2 we will discuss the state-of-the-art approach to parsing with lexicalised formalisms using a supertagger. This approach presents the baseline upon which this thesis improves.

[1]This assumes a grammar without type-raising. Practical parsers such as Clark and Curran (2007) implement type-raising but define a very constrained set of type-raising rules which limit the number of spurious derivations.

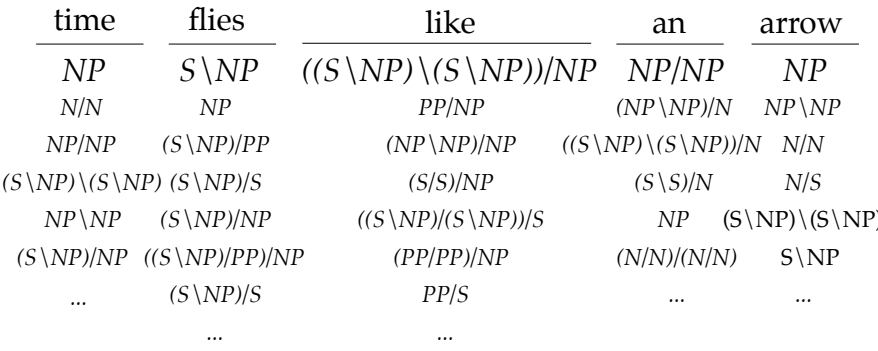


Figure 2.6: Illustration of the possible supertags for each word in a simple sentence.

2.2.1 Almost Parsing with Sequence Models

Supertags contain rich structural information, such as the direction and valency of arguments, which specify the possible syntactic environments of a word. However, there are typically many possible syntactic environments for each word, and therefore many possible supertags. Figure 2.6 illustrates the challenge in supertagging using a simple sentence. Every word has a large number of possible supertags e.g. "like" may be an adverbial preposition $((S\backslash NP)\backslash(S\backslash NP))/NP$ modifying "flies", or simply a preposition PP/NP amongst many others. Tagging is much harder for lexicalised grammars than for standard Penn Treebank-style grammars because they use a much larger non-terminal set. CCGbank (Hockenmaier and Steedman, 2007), a corpus of CCG normal-form derivations, contains 1287 different lexical types, whereas the Penn Treebank has less than 50 different Part-of-Speech tags. This leads to over 22 assignable supertags per word on average for the CCG supertagger of Clark et al. (2002).

Sequence Modelling. The most successful approach to supertagging is based on probabilistic sequence models that treat the assignment of supertags as a word labelling problem. Such models are usually referred to as n-gram taggers, where $n-1$ specifies the number of past labelling decisions taken into account when making future label assignments. For LTAG grammars trigram Hidden Markov Models, have been very successful (Bangalore and Joshi, 1999). Notably, the authors find that past supertag assignments have a larger impact on absolute accuracy for supertagging than for Part Of Speech (POS) tagging: Accuracy increases from a unigram model to a trigram model from 77% to 92%, whereas in POS-tagging the gap is narrowed from 91% to 97% (Bangalore and Joshi, 1999).[2] For CCG, discriminative sequence models (Clark,

[2]Relative error reduction is similar for both tasks but the absolute performance increase is signifi-

2002; Curran et al., 2006) following Ratnaparkhi (1996) have been very successful.

Almost Parsing. A supertagged sentence vastly reduces ambiguity due to the rich structural information encoded in the tags. This is why supertagging is sometimes referred to as almost parsing (Bangalore and Joshi, 1999). Parsing and supertagging are similar in the sense that they both assign a unique sequence of supertags to a sentence. What distinguishes them is that a parser also assigns a globally consistent derivation and dependency analysis. In fact, a supertagger may assign supertags which do not lead at all to a sentence spanning derivation and dependency analysis. However, there is previous LTAG work attempting to create dependency links during supertagging, based on the dependency requirements encoded by the assigned supertag (Joshi and Bangalore, 1994). This is only a partial solution since the assigned supertag sequence may still not result in a globally consistent analysis.

2.2.2 State-of-the-art CCG Parsing with Adaptive Supertagging

In the previous section we have discussed the effectiveness of supertagging to reduce the number of lexical categories considered for each word and how this is similar to parsing. We will now describe how supertagging is used in state-of-the-art CCG parsing (Clark and Curran, 2007), the baseline for our integrated model described in Chapter 5.

It is well understood that the number of possible supertags per word is a reliable indicator of parsing speed; even more so than sentence-length (Sarkar, 2000, 2010). Intuitively, a higher number of supertags licenses more derivations which decreases parsing efficiency. We will refer to this as **syntactic lexical ambiguity**. Supertagging directly addresses this issue by reducing the number of possible supertags considered by a parser.

Unfortunately, practical supertaggers do not provide accurate enough single-best supertag sequence predictions which can serve as input to a parser. Clark (2002) has shown that the 1-best output of a supertagger significantly lowers the coverage of a CCG parser. It is therefore common to use a multi-tagger (Curran et al., 2006) to predict more than one supertag per word. This increases the chances that the best supertag is not pruned, while removing a significant number of supertags, thus resulting in more efficient parsing (Clark and Curran, 2004b, 2007; Sarkar, 2010). Notably, Nasr and Rambow (2004) demonstrate for LTAG that parsing with the correct supertags is

cantly larger for supertagging.

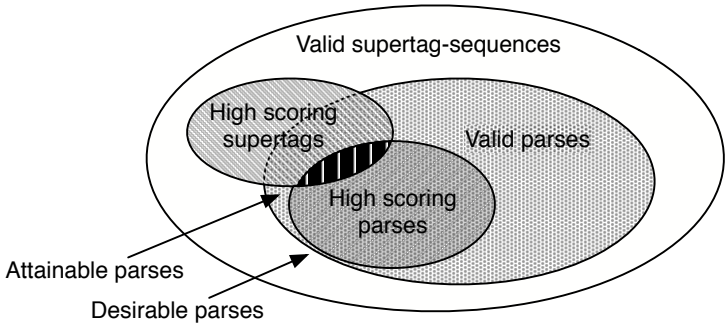


Figure 2.7: The relationship between supertagger and parser search spaces based on the intersection of their corresponding tag sequences.

extremely fast and accurate; a similar effect is very likely with a very accurate supertagger for CCG.

Adaptive supertagging. A problem that may arise with the use of a supertagger is that no parse can be found with the supertags returned (Figure 2.7), resulting in decreased parsing coverage; the number of sentences for which parses can be found. Clark and Curran (2004b) introduce adaptive supertagging, a technique that maximises parsing efficiency while maintaining good coverage. It is based on a step function over supertagger beam widths which determines the number of supertags returned. The function relaxes the pruning threshold for lexical categories only if the parser fails to find an analysis. Intuitively, adaptive supertagging can be described as follows: Initially, the supertagger returns very few supertags and the parser attempts to find an analysis; if it succeeds, the analysis is returned, otherwise the parser iteratively requests more supertags. This has the advantage of achieving high parsing speed while maintaining good coverage.

Adaptive supertagging somewhat integrates the supertagger into parsing, but fundamentally, they are still chained together into a **pipeline**. The two models only interact when the parser fails to find an analysis. Moreover, the probabilities estimated during supertagging are not used during parsing. This is wasteful since sequence models typically have much richer local contextual sensitivity than the tree-models for parsing. Clark and Curran (2004a,b) find for their parser that adaptive supertagging maintains, and in some cases increases, accuracy, while improving speed by an order of magnitude. In Chapter 4 we show that this efficiency increase results in lower accuracy for the parser of Hockenmaier (2003b) and in Chapter 5 that it decreases the upper bound

on accuracy for the parsing model of Clark and Curran (2007).

2.3 Probabilistic Models

The use of statistical or empirical methods is the prominent choice for modelling many natural language processing tasks such as parsing and tagging. The empirical view states that language is a natural phenomenon whose effects are observable in the world as text or speech. If we want a computer to reason about this data, then the best thing is to learn from it (Smith, 2006).

This section begins by discussing Conditional Random Fields (CRF), a probabilistic modelling approach we use for the main contributions in this thesis (§2.3.1). The parameters of CRFs can be estimated from training data with the use of expectations. Estimation adjusts parameters based on the expected number of times our model believes an event *should* occur, compared to how often it actually *does* occur in the training data (§2.3.2). Once estimated, we want to use the model to make predictions, or *decode* with it. We will describe Viterbi decoding to find the highest probability solution, and minimum risk decoding to find a solution that is most similar to all other high-probability solutions (§2.3.3).

We will then introduce the parsing task and how to efficiently tackle it with CKY, an efficient dynamic program. CKY has two sister algorithms, the inside algorithm and the outside algorithm, which can compute marginal probabilities over parses, the basis for estimating our model (see Chapter 3). A marginal probability is the probability of a piece of the syntactic structure, i.e., a single category or dependency, being part of any valid syntactic structure. We will succinctly describe these algorithms with semirings and weighted deduction to make their similarities and differences explicit (§2.3.4). There is a similar pair of algorithms for sequence models, the forward algorithm and the backward algorithm, which compute expectations in a similar fashion to their parsing counterparts (§2.3.5).

Finally, we discuss that all of these algorithms are special cases of the belief propagation algorithm, which can compute expectations and maximum solutions on general structures. We motivate this relationship by examining the semirings used by each algorithm and the order in which quantities in the different model structures are computed (§2.3.6).

2.3.1 Conditional Random Fields

In natural language processing we want to predict an output, such as a label sequence or a parse tree, for a given input sentence. There are two prevalent ways to design probabilistic models for this purpose. First, the generative approach models both inputs x and outputs y using a joint probability distribution $p(x,y)$. In order to make predictions we conditionalise the joint distribution using Bayes' rule $p(y|x) = \frac{p(x|y)p(y)}{p(x)}$. Second, the discriminative approach directly models the conditional distribution required for prediction $p(y|x)$. In this section we focus on Conditional Random Fields, a discriminative approach that is the basis for the best-performing CCG parser (Clark and Curran, 2007) which we improve further in this thesis.[3] The ensuing description follows the excellent introduction to CRFs by Cohn (2007).

Conditional Random Fields (Lafferty et al., 2001) are an instance of Markov Random Fields (MRF) which represent a probability distribution over a set of input and output variables $V = \{V_i\}_{i=1}^{L}$ with values $\boldsymbol{v} = \{v_i\}_{i=1}^{L}$ as a product of local potential functions

$$p(\boldsymbol{v}) = \frac{1}{Z}\prod_{c\in C}\psi_{V_c}(\boldsymbol{v}_c) \quad s.t.\ \psi_{V_c}(\boldsymbol{v}_c) > 0 \tag{2.5}$$

where C is the set of cliques. A clique is a subset of variables $V_c \subset V$ used by a potential function $\psi_{V_c}(\boldsymbol{v}_c)$, and Z is the partition function which ensures that p is a valid probability distribution, i.e., $\sum_{\boldsymbol{v}} p(\boldsymbol{v}) = 1$. The partition function is computed by summing out the numerator of Equation 2.5 for all possible value combinations:

$$Z = \sum_{v_1 \ldots v_L}\prod_{c\in C}\psi_{V_c}(\boldsymbol{v}_c) \tag{2.6}$$

This summation is usually very expensive but in sparse models, such as typical in NLP, dynamic programming can make the computation more efficient. We will present the belief propagation algorithm for efficient inference in §2.3.6.

CRFs redefine the potential functions as an exponential to avoid the positivity constraints in Equation 2.5:

$$\psi_{V_c}(\boldsymbol{v}_c) = \exp\ \phi_{V_c}(\boldsymbol{v}_c) \tag{2.7}$$

Equation 2.5 can be then be rewritten using the unconstrained functions in Equa-

[3] We do use a generative parser in Chapter 4 but our experimentation deals with efficient search methods rather than changing the probabilistic model of the parser.

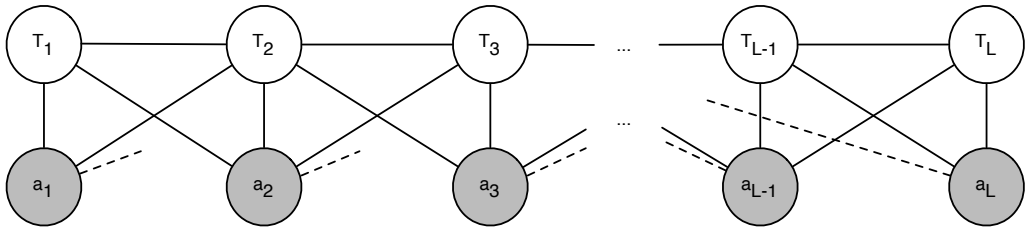


Figure 2.8: Illustration of a linear-chain Conditional Random Field for predicting tag labels for a sentence. White circles represent variables for individual tags $T_1 \dots T_K$ whose assignment is unknown and grey shaded circles are observed variables, representing words $a_1 \dots a_K$ in a sentence. The dotted lines summarise many outgoing edges which connect the output to all other variable. Edges between observed variables have been omitted since their correlation with each other has no bearing on the distribution of unobserved variables.

tion 2.7, also known as *log-potentials*:

$$p(\boldsymbol{v}) = \frac{1}{Z} \prod_{c \in C} \exp\ \phi_{V_c}(\boldsymbol{v}_c) \tag{2.8}$$

$$= \frac{1}{Z} \exp \prod_{c \in C} \phi_{V_c}(\boldsymbol{v}_c) \tag{2.9}$$

The use of log-potentials gives considerable freedom in designing potential functions, a compelling reason for using MRFs. CRFs are *log-linear models*, since they define the log-potentials as linear functions over sets of feature functions. Each feature detects aspects in the input and output variables, e.g., we may have a feature indicating the number of capitalised words. Features can be real-valued but we will consider only integer-valued functions.

Formally, the log-potentials used in CRFs are defined using K feature functions $h_1(\boldsymbol{v}_c) \dots h_K(\boldsymbol{v}_c)$, each weighted by its relative importance using parameters $\lambda_1 \dots \lambda_K \in \theta$:

$$\psi_{V_c}(\boldsymbol{v}_c) = \exp \sum_{k=1}^{K} \lambda_k h_k(\boldsymbol{v}_c) \tag{2.10}$$

Figure 2.8 shows a Conditional Random Field for a simple tagging problem. Given words $a_1 \dots a_L$, the observed variables, we want to predict a value assignment $t_1 \dots t_L$ to tag variables $T_1 \dots T_L$. The graph shows is a first-order Markov model, meaning that only the most recent tag assignment t_{i-1} to T_{i-1} is taken into account when assigning a new tag t_i to variable T_i. The graph has cliques for all individual tag variables as well

as all pairs of tag variables, i.e., $C = \{T_i\}_{i=1}^{L} \bigcup \{(T_{i-1}, T_i)\}_{i=2}^{L}$. Individual potential functions do not need to sum to one since the global distribution is normalised by the partition function Z. We denote functions over individual tags as "emission" potentials $e_i(t_i)$ and functions over pairs of tags as "transition" potentials $s_i(t_{i-1}, t_i)$. Note that the the observed word sequence $a_1 \dots a_L$ is an argument to every potential function but we omitted it for brevity. We can now define a CRF as a conditional MRF which is factorised as follows

$$p(t_1 \dots t_L | a_1 \dots a_L) = \frac{1}{Z} \exp \sum_{i=1}^{L} s_i(t_{i-1}, t_i) e_i(t_i) \tag{2.11}$$

where the partition function is

$$Z = \sum_{t_1 \dots t_L} \exp \sum_{i=1}^{L} s_i(t_{i-1}, t_i) e_i(t_i) \tag{2.12}$$

Computing the partition function Z is usually very expensive. For example, in tagging we need to sum over an exponential number of possible tag-sequences for a given sentence. We solve this problem with efficient dynamic programming algorithms described in §2.3.4 and §2.3.5.

2.3.2 Training of Probabilistic Models

So far we have described Conditional Random Fields, a discriminative approach to probabilistic modelling. This section discusses how to estiamte CRFs with training methods requiring the calculation of expectations.

We assume a supervised setting in which the training data $\mathcal{D}$ contains pairs of input sentences x and outputs y, e.g. tag sequences or parses, and that pairs are independently and identically distributed. The model parameters, $\hat{\theta}$, are chosen to maximise

$$\begin{aligned} \hat{\theta} &= \arg\max_{\theta} p(\theta|\mathcal{D}) && (2.13) \\ &= \arg\max_{\theta} \frac{p(\mathcal{D}, \theta)}{p(\mathcal{D})} && (2.14) \\ &= \arg\max_{\theta} p(\mathcal{D}|\theta) p(\theta) && (2.15) \end{aligned}$$

where we omit the denominator $p(\mathcal{D})$ since it is fixed for the maximising variable, θ. There are two scenarios for maximisation depending on the form of the prior distribution $p(\theta)$ over the parameters θ: First, *maximum likelihood estimation* assumes no prior, which requires to maximise only $p(\mathcal{D}|\theta)$. Second, *maximum a posteriori estimation* assumes a non-uniform prior distribution. We will illustrate training of log-linear

models under maximum likelihood estimation and then discuss changes for maximum a posteriori estimation.

Maximum Likelihood Estimation. The maximum likelihood estimate seeks to optimise the conditional likelihood $p(\mathcal{D}|\theta)$ to find the best parameters θ for modelling $\mathcal{D}$. One commonly optimises the logarithm of the conditional likelihood because the derivatives are easier to deal with in the logarithmic domain. Formally, we are given m training pairs $\mathcal{D} = \{\langle x^{(1)}, y^{(1)}\rangle, \ldots \langle x^{(m)}, y^{(m)}\rangle\}$, where each $x^{(i)} \in \mathcal{X}$ is drawn from a set of possible input sentences, and each $y^{(i)} \in \mathcal{Y}(x^{(i)})$ is drawn from a set of possible instance-specific outputs e.g. tag sequences. We want to learn the K parameters θ of a log-linear model, where each $\lambda_k \in \theta$ is the real-valued weight of an associated feature function $h_k(x,y)$ which itself is integer-valued. A function $f(x,y)$ maps input/output pairs to the vector $h_1(x,y),\ldots,h_K(x,y)$. The conditional log-likelihood of the data is then as follows:

$$\ell(\mathcal{D};\theta) = \sum_{i=1}^{m} \left[\theta^\mathsf{T} f(x^{(i)}, y^{(i)}) - \log \sum_{y \in \mathcal{Y}(x^{(i)})} \exp\{\theta^\mathsf{T} f(x^{(i)}, y)\} \right] \tag{2.16}$$

The first term is the weight of the correct prediction, whereas the second term is the *partition function*, the sum of the weights of all possible predictions. Intuitively, this objective is maximised when the weight of the correct prediction is large and the partition function is small. Therefore, one usually aims to choose large weights for features applying to the correct prediction and small, or even negative, weights for features firing on alternative predictions.

The parameters which maximise the conditional log-likelihood in Equation 2.16 cannot usually be found in closed form, however, numerical optimisation methods can be used to find the parameters. The log-likelihood is a convex function and therefore hill-climbing techniques can find the global optimum. There is a variety of such methods including Improved Iterative Scaling (Della Pietra et al., 1997) and Generalised Iterative Scaling (Darroch and Ratcliff, 1972) but we mostly focus on the Limited-memory Broyden–Fletcher-Goldfarb-Shanno method (L-BFGS) and to a lesser extent Stochastic Gradient Descent (SGD); both are detailed in §3.4. These optimisers usually require partial derivatives with respect to the individual features, which need to be set to zero:

$$\frac{\partial}{\partial \lambda_k} = \sum_{i=1}^{m} \left[-h_k(x^{(i)}, y^{(i)}) + \sum_{y \in \mathcal{Y}(x^{(i)})} \frac{\exp\{\theta^\mathsf{T} f(x^{(i)}, y)\}}{\sum_{y' \in \mathcal{Y}(x^{(i)})} \exp\{\theta^\mathsf{T} f(x^{(i)}, y')\}} h_k(x^{(i)}, y) \right] \tag{2.17}$$

The two terms in the brackets are both expectations over feature counts, the first, $h_k(x^{(i)}, y^{(i)})$, is referred to as the *empirical expectation* and the second as the *model expectation*. The empirical expectation of a feature simply indicates the number of times it has been observed in the training data. For example, a feature indicating if the word *likes* has been observed as a transitive verb will have in most sentences an empirical count of at most one. The model expectation of this feature depends on how often the model believes that *likes* should be a transitive verb. The training process tries to match the two expectations in order to find a model which can accurately predict the correct outputs given in the training data.

Maximum A Posteriori Estimation. One interpretation of the maximum likelihood estimate is that it assumes the prior distribution over parameters $p(\theta)$ to be uniform. The prior encodes some prior knowledge about the distribution of parameter values. Typically it is used for *regularisation* to minimise over-fitting to the training data in the hope that the parameters will generalise better to unseen data. A commonly used prior is based on a Gaussian distribution, which clusters parameter values around zero and ensures that few parameters have high magnitude. Usually, a simplified Gaussian prior is used, as given below:

$$G(\theta) = -\sum_{\lambda_k \in \theta} \frac{\lambda_k^2}{2\sigma^2} \tag{2.18}$$

This prior assumes zero mean and the standard deviations of all parameters to be equal, $\sigma_k = \sigma$; it leaves only a single hyper-parameter σ, hence simplifying tuning. The conditional log-likelihood in Equation 2.16 changes to

$$\ell(\mathcal{D};\theta)' = \ell(\mathcal{D};\theta) - G(\theta) \tag{2.19}$$

and the partial gradients of Equation 2.17 simply become

$$\left(\frac{\partial}{\partial \lambda_k}\right)' = \frac{\partial}{\partial \lambda_k} - \frac{\lambda_k}{\sigma_k^2} \tag{2.20}$$

2.3.3 Decoding: Viterbi and Minimum-Risk

In the previous section we discussed training of probabilistic models with maximum likelihood and maximum a posteriori estimates. This section deals with making predictions with trained models, also referred to as *decoding*. In natural language processing, there are usually many possible outputs for a given input, e.g., there is a large number

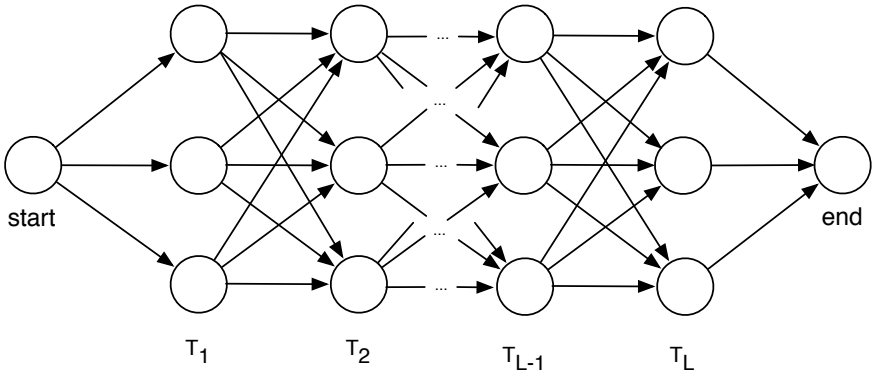


Figure 2.9: Illustration of a lattice representing all paths in a sequence model with tag variables $T_1 \ldots T_L$.

of possible supertag sequences for even the most simple sentences. For the sequence models we have seen in the previous sections, decoding can be imagined as choosing a single path through a lattice of tags, illustrated in Figure 2.9.

Another way to think about decoding is to select the optimal solution according to a *decision rule*. We introduce two such decision rules corresponding to Viterbi decoding, which selects the highest probability solution, and minimum-risk decoding, which chooses the solution that minimises the expected risk for a given function.

Viterbi Decoding. The decision rule for Viterbi decoding (Viterbi, 1967) selects the output with highest probability $\hat{y}$:

$$\hat{y} = \arg\max_{y \in \mathcal{Y}(x)} p(y|x) \tag{2.21}$$

The Viterbi algorithm computes the maximum probability solution by recursively choosing the best path leading to the current state. For example, in a bigram sequence model we move from the first to the last word choosing tags based on the best path so far. Paths to the current tag are computed recursively by re-using the probability of the best path to the *previous* tag and the probability of extending this path to the current tag (Figure 2.10). At the end of the sequence, we recursively follow backpointers to recover the sequence of tags with highest probability.

Minimum Risk Decoding. An alternative to Viterbi decoding is to select a solution

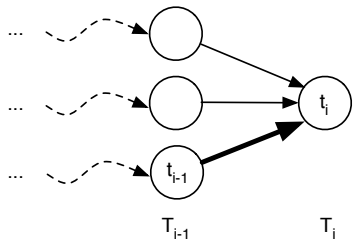


Figure 2.10: Illustration of the Viterbi algorithm. A path to tag t_i is computed by extending all best previous paths for t_{i-1} (dashed lines) to t_i (solid lines).

which minimises the *expected risk* or the *expected loss* for a given loss function:

$$\hat{y} = \arg\min_{y} \mathop{\mathbb{E}}_{y_i \sim P(y_i|x)} l(y_i, y) \tag{2.22}$$

$$= \arg\min_{y} \sum_{y_i} P(y_i|x) l(y_i, y) \tag{2.23}$$

where $l(y_i, y)$ is a function dependent on the expected best output y and another possible output y_i. In practice, minimum risk decoding is often re-cast as maximising the expected *gain* for a given function, often corresponding to a metric we would like to optimise (Goodman, 1996). If the function is simply the probability of a solution, then minimum risk decoding can be imagined as selecting a high-probability candidate which is at the same time most similar to other high-probability solutions. Clark and Curran (2007) have devised an algorithm based on minimum risk decoding for their CCG parser and we will detail it when describing their parser in §3.3.3.

The Viterbi parse can be found by using the Kronecker delta $\delta(y_i, y)$ as a gain function:

$$\hat{y} = \arg\min_{y} \sum_{y_i} P(y_i|x) \delta(y_i, y) \tag{2.24}$$

where $\delta(y_i, y)$ is defined as

$$\delta(y_i, y) = \begin{cases} 1 & \text{if } y_i = y \\ 0 & \text{otherwise} \end{cases} \tag{2.25}$$

This choice of loss function simply chooses the highest probability output, disregarding all other solutions.

2.3.4 Parsing: Computing Inside and Outside Probabilities

So far we have discussed modelling approaches, estimation and inference in general terms. This section introduces algorithms specific to parsing for computing marginal

probabilities, the basis for the expectations required in training, and to compute maximum solutions for decoding. We first introduce the parsing task and weighted deduction, a convenient notation to describe parsing algorithms. This is followed by a description of weighted derivations, the basis for recovering maximum solutions, and the inside and outside values which are essential for computing expectations during training. We then introduce efficient algorithms such as CKY to compute these quantities and show how semirings can be used to succinctly capture the similarities between some of these algorithms which are based on the same deductive logic. The use of weighted deduction and semirings is mostly intended to make the similarities between algorithms explicit but also to describe further algorithms in subsequent chapters.

Parsing and Weighted Deduction. Parsing is the task of finding the syntactic or semantic structure of a sentence using a grammar, although as previously mentioned, we focus on the syntactic component in this thesis. Typically, grammars allow many different structures for a single sentence, as we have seen in §2.1. In fact, for a grammar whose rules are binary-branching, the number of possible parses is defined by the Catalan numbers, an exponential combinatoric function (Church and Patil, 1982). It is therefore undesirable to represent each parse individually. Instead, efficient dynamic programs such as the CKY algorithm (Kasami, 1965) are used to compute the possible parses in *polynomial* space and time. The CKY algorithm makes the assumption that the grammar is in a special form, the Chomsky Normal Form, in which productions are at most binary-branching. CCG is particularly well suited to the CKY algorithm since it is by definition already in this form.

A convenient way to describe parsing algorithms is *weighted deduction* (Pereira and Warren, 1983). This notation distinguishes between a **ruleset** and a **parameterisation**. The ruleset defines the space of possible parses and the parameterisation is usually given by a probabilistic model which allows us to recover the most likely analysis with a decoding algorithm.[4]

Before describing the actual algorithms for parsing, we will motivate their use by defining the quantities they compute and how these quantities are useful for calculating expectations and maximum solutions.

Inside and Outside Values to compute Expectations. For CCG the syntactic output

[4]In CCG the ruleset is theoretically infinite but practical parsers (Clark and Curran, 2007; Hockenmaier and Steedman, 2002) restrict the set of rules by default to the ones observed in the training data, which makes the grammar effectively a context-free approximation (Vijay-Shanker et al., 1987).

of a parser is a predicate-argument strucuture for which there can be many possible derivations. We defer the discussion of modelling predicate-argument structures to a later section (§3.3.3) and focus for now on single derivations. We define a **derivation** as a sequence of rule applications $r_1 \dots r_U$. The weight $w(E)$ of a derivation E is defined as the product of the weights of all involved rule applications $w(r_1) \dots w(r_U)$:

$$w(E) = \prod_{i=1}^{U} w(r_i) \tag{2.26}$$

Note that we will use the terms weight and probability interchangeably, higher weights imply higher probabilities. We define $A_{i,j}$, where $i < j$, as short-hand for non-terminal A spanning words $a_{i+1} \dots a_j$.[5] A derivation *rooted* at $A_{i,j}$ is a sequence of rule applications covering words $a_{i+1} \dots a_j$ leading to non-terminal $A_{i,j}$, where the rule applications are grammatical, i.e., valid according to the grammar.

The **inside value** of $A_{i,j}$ is the sum of the weights of all *inner-derivations* leading to this non-terminal. An inner-derivation P for $A_{i,j}$ is a derivation whose rules provide an analysis for $a_{i+1} \dots a_j$ that is rooted in A. The inside value is definedd as follows:

$$inside(A_{i,j}) = \sum_{P \in \text{inner-derivs}(A_{i,j})} w(P) \tag{2.27}$$

The **outside value** is the sum of the weights of all *outer-derivations* for $A_{i,j}$. An outer-derivation R is part of a full sentence-spanning derivation rooted at S which passes through $A_{i,j}$, but which does not include a sub-derivation for A. The outside value is defined as follows:

$$outside(A_{i,j}) = \sum_{R \in \text{outer-derivs}(A_{i,j})} w(R) \tag{2.28}$$

Intuitively, the outside value is the weight of all outer-derivations that surround a non-terminal, it is therefore in some sense the inverse of the inside value. The outside value of a category spanning the entire sentence is one if it is grammatical. Figure 2.11 illustrates the inside and outside values.

Finally, the normalised product of the inside and outside values is the **marginal probability**, or marginal, of a non-terminal:

$$p(A_{i,j}) = \frac{1}{Z} inside(A_{i,j}) \times outside(A_{i,j}) \tag{2.29}$$

where $Z = inside(S_{0,L})$, the total weight of the graph, this is essentially the partition function discussed in §2.3.1. For simplicity, we only consider sentence-spanning

[5] We use the indices to denote positions *between* words $a_1 \dots a_L$.

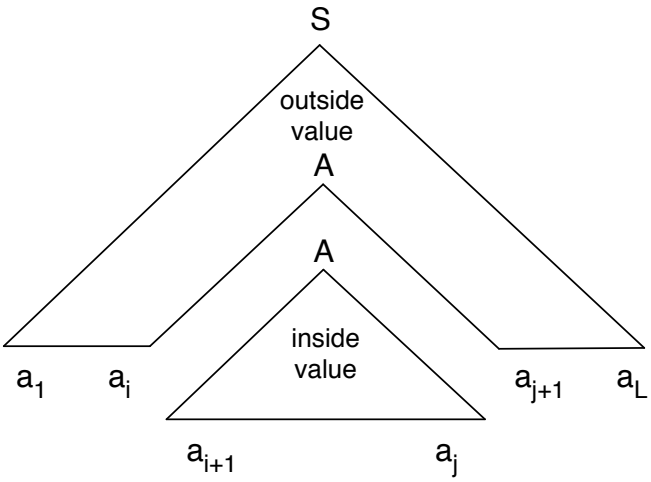


Figure 2.11: Illustration of inside and outside values. The inside value is the sum of all derivations rooted at $A_{i,j}$, the outside value is the sum of all derivations rooted at S excluding inner-derivations for A.

derivations rooted at S for the partition function, however, in practice other categories spanning the entire sentence are equally allowed (Clark and Curran, 2007). Intuitively, the marginal probability of a non-terminal is the probability of the sum of all rooted derivations that pass through the non-terminal.

Marginals form the basis for computing **expectations** required for training a log-linear model as described in §2.3.2. We distinguish between empirical expectations and model expectations. Empirical expectations are simply counts of how often a feature occurs in the training data. Model expectations are counts of how often the model predicts that a particular feature should occur in the training data; the aim of training is to make these counts as similar as possible (§2.3.2). The model expectation of a feature is the normalised sum of all marginals in which the feature occurs, see the second term in Equation 2.17.

The definitions of the inside and outside values involve summing over an exponential number of derivations, which is intractable. Tackling vast derivations spaces requires efficient algorithms. We first describe CKY, an algorithm to explore an exponential number of derivations in polynomial space and time. The CKY algorithm does not assign weights to derivations such as required for the inside and outside values. Therfore, we detail two sister algorithms , which use weights and whose purpose is the efficient computation of inside and outside values.

The CKY Algorithm. Our goal is to parse a sentence with words $a_1 ... a_L$. In order

chart($A_{i,j}$) = FALSE s.t. $0 \leq i \leq L, 1 \leq j \leq L, i < j$
for $i = 1$ to L **do**
    for all rules $a_i \Rightarrow A$ **do**
        chart($A_{i-1,i}$) = TRUE
for len $= 2$ to L **do** // length
    for $i = 0$ to $L -$ len **do** // start
        $j = i +$ len // end
        for all rules $B \Rightarrow A$ **do** // unary rules
            chart($A_{i,j}$) = chart($A_{i,j}$) $\vee$ (chart($B_{i,j}$)
        for $k = i + 1$ to $i +$ len $- 1$ // split-point
            for all rules $BC \Rightarrow A$ **do** // binary rules
                chart($A_{i,j}$) = chart($A_{i,j}$) $\vee$
                    (chart($B_{i,k}$) $\wedge$ chart($C_{k,j}$))

Figure 2.12: Pseudo-code for the CKY algorithm.

to abstract away from the particulars of CCG we will use the notation $a_i \Rightarrow A$ for lexical entries, representing supertags, and $BC \Rightarrow A$ to indicate that categories B and C combine to form category A via forward or backward composition or application.[6] Figure 2.12 presents pseudo-code for CKY.

From a weighted deduction point of view, the algorithm computes **items** of the form $A_{i,j}$ where A is a category, and i and j are indices ranging from $0 \ldots L$ where $i < j$. We refer to the quantity associated with an item as the *item value*. Inititally, this will only be its existance, i.e., a boolean value of TRUE if the item exists and FALSE otherwise. Items are usually arranged in a *chart* data structure, visualised in Figure 2.13. The basic idea is to compute items with longer spans based on items with shorter sub-spans. First items of the form $A_{i-1,i}$ are created for all lexical tokens $a_i \Rightarrow A$. Then longer items $A_{i,j}$ are constructed based on shorter adjacent items $B_{i,k}$, $C_{k,j}$ until items spanning the entire sentence $S_{0,L}$ are built.

We succinctly describe the CKY algorithm using a set of *inference rules* which deduce items. An inference rule requires a set of true items in the *antecedent*, the top of the rule, in order for the *consequent*, the bottom, to become true. Figure 2.14 shows

[6] These correspond to unary rules $A \rightarrow a_i$ and binary rules $A \rightarrow BC$ in a context-free grammar in Chomsky normal form.

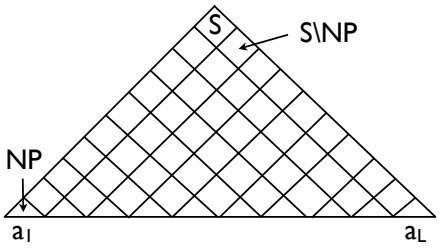


Figure 2.13: Illustration of a CKY chart which is essentially half of a two-dimensional array with L^2 entries.

$$\textbf{Item form}: A_{i,j} \qquad \textbf{Goal}: S_{0,L}$$

$$\textbf{CKY}-\textbf{Base}: \frac{a_i \Rightarrow A}{A_{i-1,i}}$$

$$\textbf{CKY}-\textbf{Unary}: \frac{B_{i,j} \quad B \Rightarrow A}{A_{i,j}}$$

$$\textbf{CKY}-\textbf{Binary}: \frac{B_{i,k} \quad C_{k,j} \quad BC \Rightarrow A}{A_{i,j}}$$

Figure 2.14: CKY algorithm in deductive form.

CKY in this notation which is borrowed from Shieber et al. (1995), Goodman (1999) and Lopez (2009). The base case (**CKY**−**Base**) corresponds to the first for-loop in the pseudo-code (Figure 2.12); its role is to assign lexical categories to every word. The **CKY**−**Unary** rule simply builds a new item $A_{i,j}$ based on an existing item $B_{i,j}$ with the same span using a rule $B \Rightarrow A$ such as type-raising. The **CKY**−**Binary** rule combines two adjacent items into a larger item; the deductive process finishes when all sentence-spanning items $S_{0,L}$ are built.

Figure 2.15 shows a hypergraph representation of a forest, a collection of multiple derivations based on structure-sharing. In hypergraph notation, items are referred to as *states* and instances of grammatical rules, such as forward application, as *transitions* between states, i.e., $BC \Rightarrow A$ is a transition from states B and C to state A. Figure 2.15 also illustrates the *GOAL* state: Sentence-spanning states $S_{0,L}$ can make a final tran-

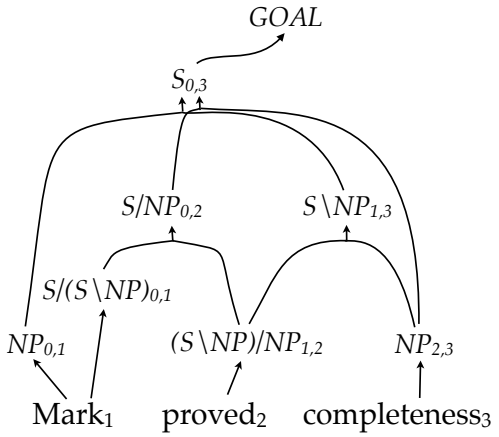


Figure 2.15: Hypergraph for a parse forest representing the two derivations for the sentence *Mark proved completeness* shown in Figure 2.5. The categories in the hypergraph have the start and ending position as subscripts; this is not to be confused with the argument indices of lexical categories. A hypergraph is a convenient way to visualise the structure sharing in dynamic programs for parsing.

sition to the *GOAL* state.[7] The *GOAL* state simply links to all grammatically valid sentence-spanning derivations.

The weighted deduction notation makes determining the **complexity** of an algorithm straightforward: Inspection of the inference rules and items is all that is required (McAllester, 1999; Lopez, 2009). For example, the item form reveals that there can be no more than $O(GL^2)$ items. An item specifies a single non-terminal from the set of grammar non-terminals G and indices range from $0 \ldots L$. Time complexity is $O(G^3L^3)$ since the most complex recursive rule **CKY−Binary** (Figure 2.14) contains at most three indices, i, k and j, which range over $0 \ldots L$, and the antecedent of the rule contains three non-terminals A, B and C, resulting in the cubic grammar constant.

The CKY algorithm described so far can be used to efficiently recognise valid sentences according to a given grammar, however, it cannot compute expectations and maximum probability parses. Next, we present a variant of the algorithm that can compute the inside value efficiently.

Inside Algorithm. The inside algorithm (Baker, 1979) computes inside values of items $I(A_{i,j})$. We give pseudo-code for this algorithm in Figure 2.16. Comparing the

[7] Practical parsers such as Clark and Curran (2007) also allow other non-terminals to transition to the *GOAL* state.

$\text{inside}(A_{i,j}) = 0 \text{ s.t. } 0 \leq i \leq L, 1 \leq j \leq L, i < j$
for $i = 1$ to L **do**
  for all rules $a_i \Rightarrow A$ **do**
    $\text{inside}(A_{i-1,i}) = w(a_i \Rightarrow A)$
for len $= 2$ to L **do** // length
  for $i = 0$ to $L - \text{len}$ **do** // start
    $j = i + \text{len}$ //end
    for all rules $B \Rightarrow A$ **do** // unary rules
      $\text{inside}(A_{i,j}) = \text{inside}(A_{i,j})$
        $+ \text{inside}(B_{i,j}) \times w(B \Rightarrow A)$
    for $k = i + 1$ to $i + len - 1$ // split-point
      for all rules $BC \Rightarrow A$ **do** // binary rules
        $\text{inside}(A_{i,j}) = \text{inside}(A_{i,j})$
          $+ \text{inside}(B_{i,k}) \times \text{inside}(C_{k,j})$
          $\times w(BC \Rightarrow A)\big)$

Figure 2.16: The inside algorithm in procedural form.

algorithm with CKY in Figure 2.12 reveals many similarities. The main difference is the operators used by the inside algorithm: the logical $\vee$ and $\wedge$ are substituted for $+$ and $\times$. Similarly, weights $w(a_i \Rightarrow A)$ replace true and false values.

We can succinctly capture these differences with **semirings**. A semiring is defined as a five tuple $\langle \mathbb{A}, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$ with elements $\mathbb{A}$, additive operator $\oplus$, multiplicative operator $\otimes$, additive identity $\mathbf{0}$ and multiplicative identity $\mathbf{1}$. The additive operator $\oplus$ needs to be associative $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ and commutative $a \oplus b = b \oplus a$, and the multiplicative operator $\otimes$ must be associative and needs to distribute over $\oplus$, so $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$. For the identities we require $a \oplus \mathbf{0} = a$ and $a \otimes \mathbf{1} = a$.

Semirings can be used to succinctly describe parsing algorithms (Goodman, 1999). For example, the difference between the CKY algorithm and the inside algorithm can be captured in the *boolean* and *inside* semirings, respectively, given in Table 2.1.

Restating the inside algorithm in terms of deductive logic and semirings (Figure 2.17) makes the similarities with CKY (see Figure 2.14) even more explicit. Note that Figure 2.17 introduces new notation to explicitly represent the weight of rule applications and item values: The weight $w(BC \Rightarrow A)$ of rule $BC \Rightarrow A$ is denoted as w_r in

$$\textbf{Item form}: \ I(A_{i,j}) \qquad \textbf{Goal}: \ I(S_{0,L})$$

$$\textbf{IN}-\textbf{Base}: \ \frac{a_i \Rightarrow A : w_r}{I(A_{i-1,i}) : w_r}$$

$$\textbf{IN}-\textbf{Unary}: \ \frac{I(B_{i,j}) : w_1 \quad B \Rightarrow A : w_r}{I(A_{i,j}) : w_1 \otimes w_r}$$

$$\textbf{IN}-\textbf{Binary}: \ \frac{I(B_{i,k}) : w_1 \quad I(C_{k,j}) : w_2 \quad BC \Rightarrow A : w_r}{I(A_{i,j}) : w_1 \otimes w_2 \otimes w_r}$$

Figure 2.17: Inside algorithm in deductive form.

notation $BC \Rightarrow A : w_r$.[8] The deductive logic description does not make explicit the summing, or the combination of items, when multiple derivations lead to the same item. Identical items are combined using the additive operator, which computes the sum of the weights for the inside algorithm. The values computed by the inside algorithm can be summarised by the following recursions:

$$\begin{aligned} I(A_{i-1,i}) &= w(a_i \Rightarrow A) \\ I(A_{i,j}) &= \bigoplus_{k,B,C} I(B_{i,k}) \otimes I(C_{k,j}) \otimes w(BC \Rightarrow A) \\ I(GOAL) &= \bigoplus_{S} I(S_{0,L}) \end{aligned} \tag{2.30}$$

The value of items based on lexical entries $I(A_{i-1,i})$ is the weight of the corresponding lexical rule $a_i \Rightarrow A$. Subsequent items are the sum of the weights of all items that can combine into A, this includes the weight of the rule licensing the combination $w(BC \Rightarrow A)$. The value of the *GOAL* state is the sum of the weights of all sentence-spanning items $I(S_{0,L})$.

We can use the **Viterbi semiring** in Table 2.1 to compute the weight of the best derivation (§2.3.3). The inside semiring sums where Viterbi takes the maximum. The highest probability derivation can be recovered by keeping back-pointers $b(A_{i,j})$ which index for each item $A_{i,j}$ the best previous items $B_{i,k}$, $C_{k,j}$ according to Equation 2.30.

[8] The deductive logic description of CKY in Figure 2.14 does not require explicit enumeration of the item values because the simple existence of an item suffices to operate the CKY deductive rules.

boolean	$\langle \{\text{TRUE}, \text{FALSE}\}, \vee, \wedge, \text{FALSE}, \text{TRUE} \rangle$
inside	$\langle \mathbb{R}_0^\infty, +, \times, 0, 1 \rangle$
Viterbi	$\langle \mathbb{R}_0^\infty, max, \times, 0, 1 \rangle$

Table 2.1: Semirings used within the CKY deduction logic.

The back-pointers are defined as:

$$b(A_{i,j}) = \langle k, B, C \rangle = \arg\max_{k,B,C} \; B_{i,k} \times C_{k,j} \times w(BC \Rightarrow A)$$

The best parse can be recovered by recursively following the back-pointers from the root $b(S_{0,L})$ to the leaves $b(A_{i-1,i})$ by which we find the full maximising parse.

Outside Algorithm. Outside values can be efficiently computed with the outside algorithm (Baker, 1979) given in Figure 2.19. The algorithm requires all inside values $I(A_{i,j})$ and proceeds in the reverse direction of the inside algorithm: Starting from sentence-spanning inside items $I(S_{0,L})$, outside items are computed based on other outside and inside items:

$$O(GOAL) = \mathbf{1} \tag{2.31}$$

$$O(B_{i,j}) = \bigoplus_{k,A,C} \left[O(A_{i,k}) \otimes I(C_{j,k}) \otimes w(BC \Rightarrow A) \right] \oplus \bigoplus_{k,A,C} \left[O(A_{k,j}) \otimes I(C_{k,i}) \otimes w(CB \Rightarrow A) \right] \tag{2.32}$$

The outside value of the GOAL state $O(GOAL)$ in Equation 2.31 is assigned the multiplicative identity of the inside-semiring, i.e., 1. The second recursion in Equation 2.32 has two parts, each dealing with B being either on the left or the right of its sibling A (Figure 2.18). The recursion for unary rules is omitted, it simply computes new outside values based on the parent-outside value and the weight of the unary rule.

Figure 2.20 shows the corresponding deductive inference rules. Inside items have been previously computed by the inside algorithm. The algorithm deduces the first outside item from an inside item covering the entire sentence using **OUT−Base**. It then builds smaller and smaller outside items by using adjacent left or right inside items as well as outside items for the parent category (**OUT−Left**, **OUT−Right**). Similar to the inside algorithm (Figure 2.17), the deductive rules for the outside algorithm do

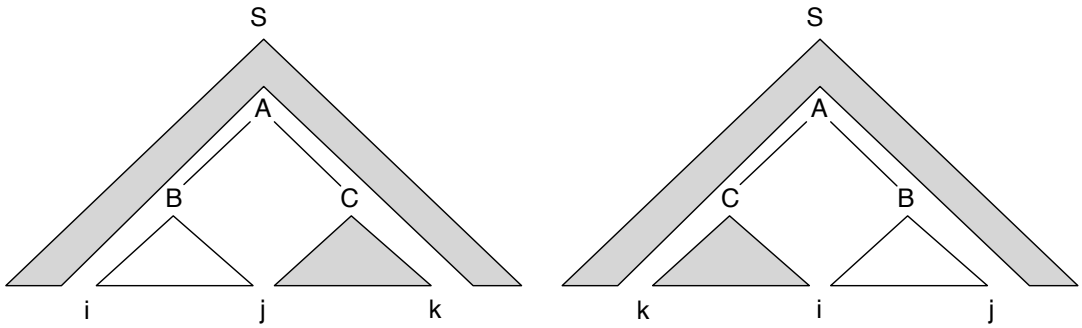


Figure 2.18: Illustration of the two sums computed in Equation 2.32 to compute $O(B_{i,j})$. The left subgraph corresponds to the first sum and the second subgraph to the second sum. Shaded parts of the parse trees correspond to the item values involved in the sum.

for len $= L$ to 2 **do** // span length
  for $i = 1$ to $L -$ len **do** // start
    $j = i +$ len // end
    for $k = i + 1$ to $i +$ len $- 1$ **do** // split-point
      for all rules $BC \Rightarrow A$ **do**

$$\text{outside}(B_{i,k}) = \text{outside}(B_{i,k}) + \text{outside}(A_{i,j}) \times \text{inside}(C_{k,j}) \times w(BC \Rightarrow A)$$

$$\text{outside}(C_{k,j}) = \text{outside}(C_{k,j}) + \text{outside}(A_{i,j}) \times \text{inside}(B_{i,k}) \times w(BC \Rightarrow A)$$

Figure 2.19: Pseudo-code for the outside algorithm.

$$\textbf{Item form}: \; O(A_{i,j}) \qquad \textbf{Goal}: \; O(A_{i-1,i})$$

$$\textbf{OUT}-\textbf{Base}: \; \frac{I(S_{0,L}) : w_1}{O(S_{0,L}) : \mathbf{1}}$$

$$\textbf{OUT}-\textbf{Left}: \; \frac{I(C_{j,k}) : w_1 \quad O(A_{i,k}) : w_2 \quad BC \Rightarrow A : w_r}{O(B_{i,j}) : w_1 \otimes w_2 \otimes w_r}$$

$$\textbf{OUT}-\textbf{Right}: \; \frac{I(C_{k,i}) : w_1 \quad O(A_{k,j}) : w_2 \quad CB \Rightarrow A : w_r}{O(B_{i,j}) : w_1 \otimes w_2 \otimes w_r}$$

Figure 2.20: Outside algorithm in deductive form.

not make explicit the summing, or the combination of items. But similar to before, identical items are combined using the additive operator, which computes the sum of the outside values leading to the same items.

2.3.5 Tagging: Forward and Backward Probabilities

So far we have discussed algorithms for efficiently computing expectations and maximum solutions for parsing. This section will discuss similar algorithms for tagging, treating the assignment of tags as a sequence labelling problem.

Sequence models take the form of chain graphs such as illustrated by the Conditional Random Field in Figure 2.21. We assume a first-order Markov model (§2.3.1) to predict a sequence of tags $t_1 \ldots t_L$. This is modelled by "emission" potential functions over individual tags $e_i(t_i)$, and "transition" potentials over pairs of tags $s_i(t_{i-1}, t_i)$; their dependence on words $a_1 \ldots a_L$ is omitted for brevity. Similarly to parsing models, we require marginal probabilities for computing feature expectations. The marginal probabilities are given by the forward and backward values which can be computed by the forward and backward algorithms, sister algorithms to the inside and outside algorithms in parsing.

Forward Algorithm. The forward value $F_i(t_i)$, of a tag t_i, is the sum of the weights

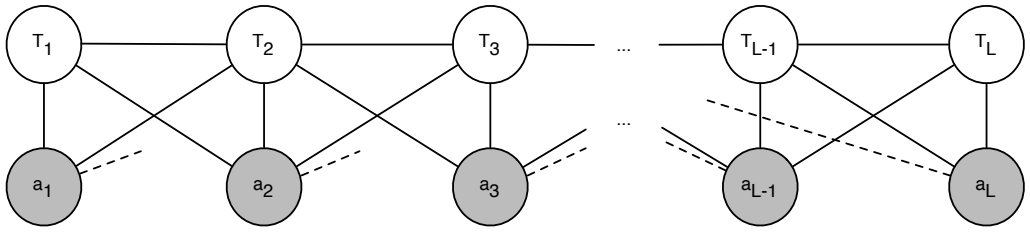


Figure 2.21: Illustration of a Conditional Random Field for tagging as given earlier in Figure 2.8.

of all partial paths starting at T_1 and ending at $T_i = t_i$ covering $a_1 \dots a_i$. The forward value for tag t_i is given as:

$$F_i(t_i) = \bigoplus_{t_1 \dots t_{i-1}} \left[\bigotimes_{j=1}^{i} s_j(t_{j-1}, t_j)\ e_j(t_j) \right] \tag{2.33}$$

where each $F_i(T_i)$ is a vector of values for all $t_i \in T_i$. Computing the forward value in this way involves summing over an exponential number of tag sequences which is intractable. The forward algorithm can compute the forward values efficiently. We will use the general semiring notation from §2.3.4 and assume the *inside semiring* from Table 2.1. The forward algorithm is given by the following recursions:

$$F_1(t_1) = \mathbf{1} \tag{2.34}$$

$$F_i(t_i) = \bigoplus_{t_{i-1}} [F_{i-1}(t_{i-1}) \otimes s_i(t_{i-1}, t_i) \otimes e_i(t_i)] \tag{2.35}$$

where Equation 2.34 forms the base case from which Equation 2.35 can be computed for $i = 2 \dots L$. The partition function is given by the sum of the final forward values $Z = \bigoplus_{t_L} F_L(t_L)$.

Backward Algorithm. Similarly, the backward value $B_i(t_i)$ for tag t_i is the sum of the weights of all partial paths starting at t_i and ending at T_L covering $a_i \dots a_L$. The backward values for tag t_i can be efficiently computed with the backward algorithm:

$$B_L(t_L) = \mathbf{1} \tag{2.36}$$

$$B_i(t_i) = \bigoplus_{t_{i+1}} [B_{i+1}(t_{i+1}) \otimes s_{i+1}(t_i, t_{i+1}) \otimes e_{i+1}(t_{i+1})] \tag{2.37}$$

Similarly to parsing, we can compute the **marginal probability** of a tag as the normalised product of the forward and and backward values which can be used for

$$\textbf{Item form}: F_i(D_i) \qquad \textbf{Goal}: F_L(D_L)$$

$$\textbf{Forw}-\textbf{Base}: \ : \frac{R(start, D) : w_r}{F_1(D_1) : w_r}$$

$$\textbf{Forw}-\textbf{Recursive}: \frac{F_i(D_i) : w_1 \quad R(D, E) : w_r}{F_{i+1}(E_{i+1}) : w_1 \otimes w_r}$$

Figure 2.22: Forward algorithm in deductive form.

$$\textbf{Item form}: B_i(A_i) \qquad \textbf{Goal}: B_1(D_1)$$

$$\textbf{Back}-\textbf{Base}: \frac{R(A, end) : w_r}{B_L(D_L) : w_r}$$

$$\textbf{Back}-\textbf{Recursive}: \frac{B_i(D_i) : w_1 \quad R(E, D) : w_r}{B_{i-1}(E_{i-1}) : w_1 \otimes w_r}$$

Figure 2.23: Backward algorithm in deductive form.

computing model expectations:

$$p(t_i) = \frac{1}{Z} F(t_i) B(t_i) \tag{2.38}$$

where the partition function is

$$Z = \sum_{t_L} F_L(t_L) = \sum_{t_1} B_1(t_1) \tag{2.39}$$

which is the probability of all tag sequences.

The highest probability tag sequence can be computed with the Viterbi algorithm described in §2.3.3, this is essentially the forward algorithm in Equation 2.34 and Equation 2.35 using the *Viterbi semiring* from Table 2.1. The Viterbi tag sequence can be recovered by keeping back-pointers, as described in §2.3.3.

Deductive Logic Description. We can also describe the algorithms using deductive logic. Figure 2.22 gives the forward algorithm in deductive form. The algorithm computes items $F_i(D_i)$ representing the forward value for tag D at position i. The algorithm begins at the start of the sentence and deduces items for position 1 using **Forw−Base**; this uses the rule $R(start, D) : w_r$ which licenses the combination of the *start* symbol and category D. Rules are augmented by weights w_r that combine the weights of the emission and transition factors. The recursive rule **Forw−Recursive** builds items from left to right until the end of the sentence is reached. The backward algorithm in Figure 2.23 computes backward items $B_i(D_i)$ following a similar logic but proceeds from right to left.

2.3.6 General Inference with Belief Propagation

In the previous section we have seen *specific* algorithms for computing marginal probabilities and maximum probability solutions for parsing and supertagging. This section describes belief propagation, an algorithm for computing these quantities in *general* model topologies.

Before introducing the algorithm, we will describe factor graphs, a convenient way to visualise joint probability distributions with potential functions over many variables. We then introduce the sum-product algorithm, a variant of belief propagation to compute marginal probabilities. Next, we discuss max-product belief propagation to compute maximum probability solutions. Finally, we discuss the relationship between the specific algorithms for parsing and tagging introduced earlier and belief propagation.

Factor graphs. These graphs (Kschischang et al., 1998) make the factorisation of global functions, or probability distributions, into local functions explicit. A factor graph is a bipartite graph with nodes for variables and squares for **factors**, which represent potential functions over subsets of variables. Borrowing notation from §2.3.1, we can describe a factor graph as the factorisation of a complex function $g(\boldsymbol{v})$, where $\boldsymbol{v}$ is a set of value assignments $\boldsymbol{v} = \{v_i\}_{i=1}^{L}$ to variables $V = \{V_i\}_{i=1}^{L}$. The complex function is a product of local potential functions $\psi_{V_f}(\boldsymbol{v}_f)$ over subsets of variables V_f:

$$g(\boldsymbol{v}) = \prod_{f \in F} \psi_{V_f}(\boldsymbol{v}_f) \tag{2.40}$$

where $f \in F$ is a factor from the set of all factors F for this graph. Figure 2.24 shows a factor graph for the CRF in Figure 2.21; the graph factors the words directly into the

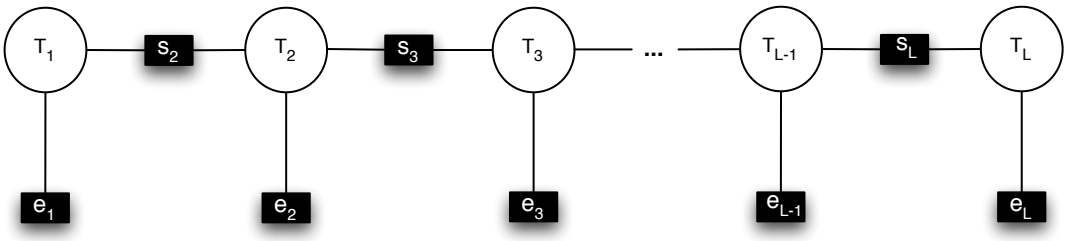


Figure 2.24: Factor graph representation of the Conditional Random Field in Figure 2.21. Squares represent factors and circles variables. There are emission factors $e_i(t_i)$ for each tag variable T_i and transition factors $s_i(t_{i-1}, t_i)$ computed over pairs of variables t_{i-1}, t_i.

emission factors e_i since their values are observed.[9]

Sum-Product Belief Propagation. The sum-product algorithm computes marginal distributions over variables in a factor graph. The marginal over variable V_i can be obtained by marginalising, or summing, over the remaining variables. For the purpose of explaining belief propagation we will use notation that indicates the variables over which we are *not* summing, which in this case is only V_i:[10]

$$p(v_i) = \sum_{\sim\{v_i\}} p(\boldsymbol{v}) \tag{2.41}$$

Computing this sum naively is intractable for most models since there is an exponential number of values in the number of variables. The sum-product algorithm (Pearl, 1988) computes the sums required by the forward value (Equation 2.33) or the inside (Equation 2.27) and outside values (Equation 2.28) recursively by passing messages in a factor graph.

The idea behind sum-product belief propagation (Pearl, 1988; Kschischang et al., 1998) is to pass **messages** between variables and factors in the graph after which marginals can be computed based on the message values. We can send a message between variable V_i and a factor f which we denote as $m_{V_i \to f}(v_i)$ where v_i is a value of V_i; messages are vectors with an element for each value of V_i, the vector quantities

[9] Equivalently, one could use a single emission factor connected to all variables since factors in CRFs have access to all observations.

[10] This notation is borrowed from Kschischang et al. (1998).

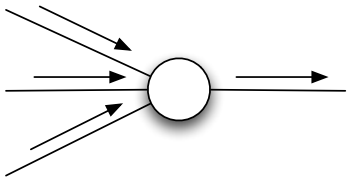


Figure 2.25: Illustration of message passing: Outgoing messages can only be sent over an edge if all other edges have received a message.

represent the result of summing out all other variables:

$$m_{V_i \to f}(v_i) = \prod_{h \in n(V_i) \setminus f} m_{h \to V_i}(v_i) \tag{2.42}$$

where $n(V_i)$ is the set of all neighbours of V_i. Because the graph is bipartite, all neighbours of a variable are factors and all neighbours of a factor are variables. It is common that messages are normalised to sum to one in order to avoid numerical overflow. A message $m_{f \to V_i}(v_i)$ from a factor to a variable aggregates all incoming messages and takes into account the factor-function:

$$m_{f \to V_i}(v_i) = \sum_{\sim\{v_i\}} \left[f(X) \prod_{V_u \in n(f) \setminus V_i} m_{V_u \to f}(v_u) \right] \tag{2.43}$$

where $X = n(f)$ is the set of arguments of the factor function f. Note that messages from variables to factors can be seen as messages from factors to variables with a factor-function that always returns 1.

Message Passing. The order in which messages are sent is referred to as the message passing schedule. Any schedule has to adhere to the **sum-product rule** (Kschischang et al., 1998) which states that nodes or factors with multiple neighbours must have received incoming messages from all neighbours, except for the neighbour to which a message will be sent, before they themselves can send messages. This requirement ensures that Equations (2.42) and (2.43) are valid and Figure 2.25 illustrates it.

Message passing typically starts from a leaf variable or factor with only a single neighbour and proceeds to a *root* variable. Once the root has received messages on all edges, its marginal can be computed as the product of all these messages; this corresponds to a single *outward pass*. However, typically we would like to compute the marginals of all variables, not only the root. This can be achieved by propagating

messages back from the root to the leaves, referred to as an *inward pass*. The message values converge in an *acyclic* graph after these two passes.

After message passing we are ready to compute the marginals of the variables. Marginals are also referred to as beliefs $\mathcal{B}_{V_i}(v_i)$; a belief is the normalised product of all incoming messages of a variable V_i.

$$\mathcal{B}_{V_i}(v_i) = \frac{1}{Z} \prod_{h \in n(V_i)} m_{h \to V_i}(v_i) \tag{2.44}$$

where the normalisation constant Z sums over all possible values v_i.

Max-Product Belief Propagation. For decoding we can use max-product belief propagation which computes the probability of the best variable assignment, i.e., $\hat{\boldsymbol{v}} = \arg\max_{\boldsymbol{v}} p(\boldsymbol{v})$. This requires only a small change to the sum-product algorithm:

$$m_{f \to V_i}(v_i) = \max_{\sim\{v_i\}} \left[f(V_i) \prod_{V_u \in n(f) \setminus V_i} m_{V_u \to f}(v_u) \right] \tag{2.45}$$

Equation 2.45 replaces the sum in Equation 2.43 with a max operator. The messages correspond to the maximum probability rather than the total probability. Messages are passed in a similar fashion to the sum-product algorithm but now we require only an outward pass since then we have found the value of the maximum variable assignment. The maximising configuration can be recovered by keeping back-pointers in the same way as with the Viterbi algorithm (§2.3.4).

Belief Propagation with Semirings. We have seen that the max-product algorithm is essentially the sum-product algorithm but with a different "summing" operator. More generally, belief propagation can be used with any pair of operators, if they form a commutative semiring (Cohn, 2007). To recap from §2.3.4, a **semiring** has a multiplicative operator $\otimes$ and an additive operator $\oplus$ which can be thought of as $\times$ and $+$ for sum-product and max and $+$ for max-product.

With this notation we generalise belief propagation as follows (Kulesza and Taskar, 2010): The message from a variable V_i to factor f (Equation 2.42) becomes

$$m_{V_i \to f}(v_i) = \bigotimes_{h \in n(V_i) \setminus f} m_{h \to V_i}(v_i) \tag{2.46}$$

and messages from factors to variables (Equation 2.43 and 2.45) become

$$m_{f \to V_i}(v_i) = \bigoplus_{\sim\{v_i\}} \left[f(X) \otimes \bigotimes_{V_u \in n(f) \setminus V_i} m_{V_u \to f}(v_u) \right] \tag{2.47}$$

sum-product	$\langle \mathbb{R}_0^\infty, +, \times, 0, 1 \rangle$
max-product	$\langle \mathbb{R}_0^\infty, max, \times, 0, 1 \rangle$

Table 2.2: Semirings for belief propagation (cf. semirings for parsing in Table 2.1).

which is the product of all but one incoming messages and the factor function; for sum-product we sum over all variable configurations $\sim \{v_i\}$, and for max-product we retain the maximum configurations.

The semirings for sum-product and max-product are given in Table 2.2. They are identical to the semirings we presented for the inside and outside algorithms in Table 2.1, which we also used in the forward and backward algorithms. This serves as the first evidence that these algorithms are *specific* cases of belief propagation and we will discuss this connection next.

Forward and Backward is Belief Propagation. We have already shown that the semirings for computing marginals and maxima in sequence models are identical to the semirings used by sum-product and max-product belief propagation. We now illustrate that the forward and backward values correspond to messages passed in a factor graph and that the forward and backward algorithms are special instances of the general belief propagation algorithm. Here we give the intuition following Sutton and McCallum (2011); formal proof can be found in Smyth et al. (1997). The main forward/backward equations are reproduced for reference:

$$F_i(t_i) = \bigoplus_{t_{i-1}} [F_{i-1}(t_{i-1}) \otimes s_i(t_{i-1}, t_i) \otimes e_i(t_i)] \tag{2.48}$$

$$B_i(t_i) = \bigoplus_{t_{i+1}} [B_{i+1}(t_{i+1}) \otimes s_{i+1}(t_i, t_{i+1}) \otimes e_{i+1}(t_{i+1})] \tag{2.49}$$

The forward algorithm chooses the final tag variable T_L as the root and sends messages from T_1 in its direction as illustrated by Figure 2.26. At T_2 we receive two messages whose product is the forward value $F_2(t_2)$ as given by the forward algorithm in Equation 2.48: The message $m_{s_2 \to T_2}(t_2)$ is the product of the sum of all previous messages $F_1(t_1)$ as well as the factor function $s_2(t_1, t_2)$; and $m_{e_2 \to T_2}(t_2)$ is the value of the emission factor function $e_2(t_2)$. The messages therefore correspond to the forward algorithm given in Equation 2.48. The backward algorithm chooses T_1 as the root and passes messages from T_L in the reverse direction. At T_2 we receive two messages

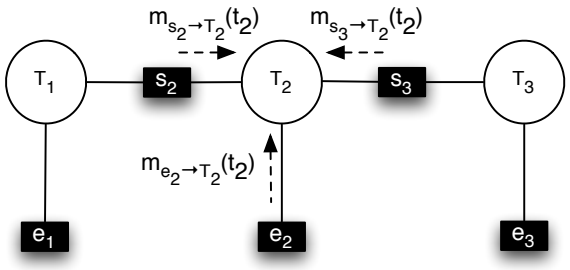
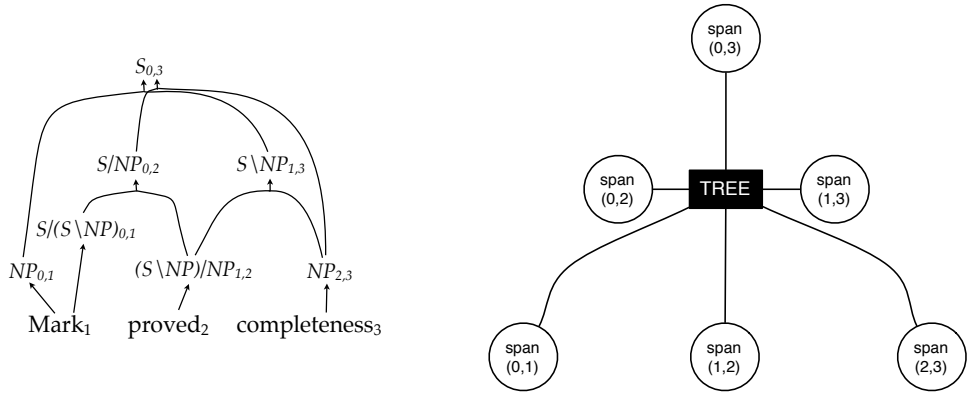


Figure 2.26: Illustration of message passing in a factor graph for a sequence model: The marginal probability of t_2 is the product of three messages: The product of $m_{s_1 \to T_2}(t_2)$ and $m_{e_2 \to T_2}(t_2)$ corresponds to the forward value $F_2(t_2)$, and $m_{s_2 \to T_2}(t_2)$ corresponds to the backward value $B_2(t_2)$.

whose product is the backward value $B_2(t_2)$ as given by the backward algorithm in Equation 2.49: Message $m_{s_3 \to T_2}(t_2)$ is the product of the sum of all previous messages $B_3(t_3)$ and the factor function of $s_3(t_2, t_3)$; and $m_{e_3 \to T_3}(t_3)$ is the value of the emission factor function $e_3(t_3)$. The messages therefore correspond to the backward algorithm.

Inside and Outside is Belief Propagation. The inside algorithm and the outside algorithm are special cases of belief propagation; formal proof can be found in Sato (2007). Parsing forests can be represented as hypergraphs as we have seen earlier in §2.3.4. Figure 2.27(a) shows the hypergraph for our running example and Figure 2.27(b) shows the corresponding simplified factor graph. The factor graphs arising in parsing take a special form: They have a quadratic number of variables $span(i, j)$ with values corresponding to the categories $C_{i,j}$ which can be built over words $a_{i+1} \ldots a_j$; we refer to them as *span variables*.

A `TREE` factor defines a distribution over the joint assignment to all span variables for a given sentence. The factor assigns only positive probability mass to value assignments which result in valid derivations according to the grammar. It also excludes overlapping child categories and invalid rule combinations. The factor is formalised by an extension to graphical models called case-factor diagrams (McAllester et al., 2008) which construct and/or graphs to represent the structure of possible derivations.



(a) Hypergraph (b) Factor graph

Figure 2.27: Hypergraph (a) and corresponding simplified factor graph (b).

2.4 Conclusions

In this chapter we have introduced the background on which the remainder of this thesis builds. We first discussed Combinatory Categorial Grammar (CCG), the formalism used throughout all experiments. CCG is based on a lexicon, containing a syntactic type for every word in the language as well as a small set of combinatory rules.

The central theme of this thesis is to combine supertagging and parsing. Supertagging is the task of assigning lexical types, or supertags, to words. Supertags contain a large amount of structural information, which is why supertagging massively reduces ambiguity. The baseline approach to the methods introduced in this thesis is adaptive supertagging, a shallow combination method for parsing and supertagging that maximises efficiency while maintaining good coverage. However, adaptive supertagging does not allow the parser to recover from errors made during tagging and it also discards the probabilities computed by the supertagger.

The basis for the methods presented in the following chapters are probabilistic models and inference methods. Probabilistic models are the predominant choice to tackle natural language processing tasks. We described Conditional Random Fields, a discriminative modelling approach which is the basis for the integrated supertagging and parsing model discussed in Chapter 5. Expectations are the basis for training discriminative models with gradient-based optimisation methods. Model parameters are adjusted such that empirical expectations, observed in the training data, are assimilated

to the model expectations. We experiment with two approaches to decoding: Viterbi decoding finds the highest-probability configuration while as minimum risk decoding finds a solution which minimises the expected risk given a loss function.

Parsing is the central theme of this thesis and we therefore reviewed CKY, an efficient algorithm to explore an exponential number of analyses in polynomial space and time. Its two sister algorithms, the inside algorithm and the outside algorithm, compute marginal probabilities, or just marginals, over categories. Marginals are the basis for model expectations of individual features. Substitution of appropriate semirings into the CKY logic allows the computation of inside values and Viterbi parses. Marginals in supertaggers can be computed with the forward algorithm and the backward algorithm.

Our combined supertagging and parsing model (Chapter 5) uses belief propagation for inference. Belief propagation is an algorithm for computing marginal and maximum probabilities in general model structures with the sum-product and max-product update rules, respectively. The inside and outside algorithms, as well as the forward and backward algorithms, are specialised variants of belief propagation.

The following chapters will build on these methods to describe more accurate as well as more efficient parsing models.

Chapter 3

Experimental Setup

This thesis proposes an integrated model for supertagging and parsing which we apply to parsing with Combinatory Categorial Grammar (CCG). We use efficient inference to make this model practical and to improve the accuracy of state-of-the-art CCG parsing.

This chapter presents the corpus we used to train and evaluate our models, the methods to measure performance and the configuration of the parsing and supertagging models. Finally, we detail the settings of the employed estimation methods. We aim to deviate as little as possible from previous work on CCG parsing in order to allow meaningful comparisons. We use the same data and evaluation methods, and, as much as possible, the same feature-sets, settings and training regimen of our baseline models.

The chapter is structured as follows: First, we describe CCGbank, the corpus we used for training and testing (§3.1). We then discuss how we measured performance both in terms of accuracy and efficiency (§3.2). Next we describe the baseline models of our work. Our description builds on the algorithms and probabilistic models we introduced in Chapter 2 but also details the exact settings we used with each system. First, we describe the supertagger of Clark (2002) and Curran et al. (2006) and its settings, which we used throughout this thesis (§3.3.1). Second, we detail the settings of the generative parser of Hockenmaier (2003a), which we use in Chapter 4 (§ 3.3.2). Third, we describe the discriminative parser of Clark and Curran (2007), which was the best performing model on CCGbank before this work and is therefore the baseline for our integrated model (§3.3.3). Finally, we detail the training of the parser of Clark and Curran (2007) since we used slightly different settings than described in their work (§3.4).

	Training	Development	Test
Sections	02-21	00	23
Words	929552	45422	55371
Sentences	39604	1913	2407

Table 3.1: Statistics for the CCGbank corpus split used for training, development and test.

3.1 CCGbank

CCGbank (Hockenmaier and Steedman, 2007) is the corpus we use to train and test our models throughout this thesis. It is a normal-form CCG version of the Penn Treebank (Marcus et al., 1993). Normal-form means that it uses composition and type-raising only when necessary. The corpus was created semi-automatically by converting the phrase-structure trees in the Penn Treebank. This conversion required some pre-processing such as correcting POS tagging errors and ensuring that the constituent structure conforms to the CCG analysis; Hockenmaier and Steedman (2007) describes the conversion procedure in detail.

Features. Some categories can carry features to distinguish different types of sentences and verb phrases. Sentence categories S can be either separated into declarative sentences $S[dcl]$, or wh-questions $S[wq]$ amongst others. Verb phrases $S\backslash NP$, can be distinguished between to-infinitives $S[to]\backslash NP$, or past participles in passive $S[pss]\backslash NP$. A complete list of features can be found in Hockenmaier and Steedman (2005).

Combinatory and Type-changing Rules. CCGbank uses the standard CCG combinatory rules described in §2.1 including application, composition, coordination and type-raising. There are also so-called type-changing rules which do not correspond to CCG combinatory rules. A very common such rule changes a noun into a noun phrase, $N \Rightarrow NP$ but it is also common to change verb phrases of any type into modifiers, following the schemes $Y \Rightarrow X/X$ and $Y \Rightarrow X\backslash X$. There are also binary type-changing rules which treat commas as coordinations following the schema , $X \Rightarrow X\backslash X$ (Clark and Curran, 2007), and rules dealing with punctuation: , $NP \Rightarrow NP$.

Corpus-split. The corpus is split into different sections. Following past work (Hockenmaier, 2003a; Clark and Curran, 2007), we used sections 02-21 for training, section 00 for development and section 23 for testing. Table 3.1 gives some statistics for this

split of CCGbank. For each sentence the corpus provides a normal-form derivation, a predicate-argument structure (see §2.1.2), a set of supertags and a set of POS-tags. We will refer to data from the corpus as *gold-standard* or as *ground truth*.

The training data contains 1,286 different lexical categories. The expected number of lexical categories per token is 19.2, however some words are observed with a very high number lexical categories, e.g., the word *as* was assigned 130 different lexical categories (Hockenmaier and Steedman, 2007).

3.2 Evaluation

The previous section described the experimental data we use to evaluate the accuracy and speed of various parsing models in the following chapters. In this section we describe the metrics for measuring accuracy and the setup for measuring parsing times.

3.2.1 Accuracy Measures

The accuracy of our models has been evaluated in terms of predicate-argument structures (§2.1.2) recovery. This evaluation method follows previous work in CCG parsing (Hockenmaier, 2003a; Clark and Curran, 2007) and allows for easy comparison with other results.

F-measure, Precision and Recall. Accuracy is measured in terms of precision, recall and F-measure, or F_1, over predicate-argument relations.[1] Given gold-standard dependencies y and dependencies output by a parser y', each dependency being a variable-sized set of predicate-argument relations, we can compute precision $P(y,y')$ and recall $R(y,y')$ as:

$$P(y,y') = \frac{|y \cap y'|}{|y'|} \tag{3.1}$$

$$R(y,y') = \frac{|y \cap y'|}{|y|} \tag{3.2}$$

where $|y \cap y'|$ is the number of correctly identified dependencies, $|y'|$ is the number of dependencies in the proposed output, and $|y|$ is the number of dependencies in the gold-standard.

Our aim is to achieve high precision, meaning that the returned dependencies are mostly correct, as well as high recall, meaning that we return most of the desired

[1] We use accuracy as a general term to refer to the performance on the test and development sets, which is measured using a range of metrics such as F_1, precision and recall.

dependencies. One can optimise for either precision or recall at the expense of the other measure. However, in most applications we would like to optimise both equally and this can be measured with F_1:

$$F_1(y,y') = \frac{2PR}{P+R} = \frac{2|y \cap y'|}{|y| + |y'|} \tag{3.3}$$

F-measure is the harmonic mean between precision and recall and is also referred to as F-score.

Sometimes parsers cannot produce an analysis for a given sentence which decreases its **coverage**, the ratio of sentences for which an analysis could be returned. Unless otherwise mentioned, the accuracy measures reported in this thesis are only over sentences for which a parse could be returned.

Labelled and Unlabelled Accuracy. We will evaluate accuracy using both labelled and unlabelled versions of precision, recall and F-measure. Labelled accuracy regards a predicate-argument relation only as correct if all fields in the relation match the gold-standard, that is, the lexical category, the argument slot and both head words (see §2.1.2). Unlabelled measures ignore the lexical category and the argument slot and therefore only evaluate head-argument dependencies.

Gold and Auto POS Tags. Previous work on CCG parsing and supertagging has performed most experiments using both gold-standard and automatically assigned POS-tags as input to their systems (Hockenmaier, 2003a; Clark and Curran, 2007). To make our results readily comparable, we also report results in both settings. For the latter we used the C&C POS tagger to assign POS-tags (Curran et al., 2006).

Exact-match and Supertagging Accuracy. Exact-match refers to the ratio of sentences for which the entire predicate-argument structure, that is all dependencies, match the gold-standard. We will use this measure in our final results presented in Chapter 6. Supertagging accuracy is the ratio of correct supertags assigned to the words in a sentence. It can be used to measure the accuracy of supertagging as well as parsing.

3.2.2 Timing

We measure parsing efficiency in terms of the time taken to parse the test and development portions of CCGbank. The times reported are the absolute wall clock times on a 2.5 GHz Intel Xeon machine with 32 GB memory. All results were averaged over ten runs to provide reliable results. The timings include only the core parsing times and exclude all preparatory steps such as loading of the model.

3.3 Baseline Models

We now turn to the baseline models used in the experiments. First, we detail the supertagger models, their settings and feature-sets (§3.3.1). Next, we describe the setup of the generative parser we experimented with in Chapter 4 (§3.3.2). Finally, in §3.3.3 we present the configuration of the discriminative parser that forms the basis of our integrated supertagging and parsing model (Chapter 5); we also used the parser for experimentation with a novel training technique (Chapter 6).

3.3.1 Supertagging

This section describes two supertaggers based on slightly different models: The C&C supertagger (Clark, 2002; Curran et al., 2006) and a reimplementation by Dennis Mehay, which is part of the OpenCCG project.[2] Experiments in this thesis are usually based on the idea of combining parsers and supertaggers. In the following chapters, we use two different parsers and in order to maintain coherent code-bases, we decided to combine each parser with a supertagger written in the same programming language: First, we use the C&C supertagger together with the C&C parser (Clark and Curran, 2007; §3.3.3), both written in C++; this setup is used in Chapters 5 and 6.[3] Second, the OpenCCG supertagger was coupled with the parser of Hockenmaier (2003b; §3.3.2), both implemented in Java; this setup is used in Chapter 4.

The C&C supertagger implements a conditional Maximum Entropy model. The probability of a supertag t_i given a local context x_i containing t_i is as follows:

$$p(t_i|x_i) = \frac{\exp\left\{\theta^\top f(x_i,t_i)\right\}}{\sum_{t'} \exp\left\{\theta^\top f(x_i,t')\right\}} \tag{3.4}$$

Similar to before $f(x_i,t_i)$ is a function that returns a vector indicating which features apply to x_i and t_i; and θ is a vector of weights of the same length. Features include previous supertag assignments and word-based features within a local window of t_i; all encoded in the local context x_i, see below for a detailed list. During decoding, the tagger selects supertag-sequences based on the marginal probabilities of the supertags. The marginals are computed with the forward algorithm and the backward algorithm (§2.3.5, Curran et al. 2006).

[2] Dennis Mehay has not published his supertagger but it is part of the OpenCCG project (`http://sourceforge.net/projects/openccg`).

[3] The C&C tools can be obtained from `http://svn.ask.it.usyd.edu.au/trac/candc/wiki`

The OpenCCG supertagger also uses a conditional Maximum Entropy model but only to model features over local contexts. It relies on a separate sequence model for transition probabilities. The Maximum Entropy model is interpolated with a traditional multinomial, $p(t_i|t_{i-1})$, an n-gram model over supertags (McCallum et al., 2001):

$$p(t_i|x_i) = p(t_i|t_{i-1}) \frac{\exp\left\{\theta^{\mathsf{T}} f(x_i,t_i)\right\}}{\sum_{t'} \exp\left\{\theta^{\mathsf{T}} f(x_i,t')\right\}} \tag{3.5}$$

The two supertaggers share many similarities such as identical feature-sets and tag-set. Despite slightly different underlying models, the accuracy of both taggers is very similar when evaluated on 1-best output on the dev and test sets. While the two taggers have different models, it is important to note that we do not directly compare the two supertaggers in any experiment.

Features. The features are based both on words w_i and POS-tags t_i within a fixed window of five words; this is at most two positions before and after the current word. The following templates were used to extract the features:

- Unigrams of words: $w_{i-2},\ w_{i-1},\ w_i,\ w_{i+1},\ w_{i+2}$
- Unigrams of POS-tags: $t_{i-2},\ t_{i-1},\ t_i,\ t_{i+1},\ t_{i+2}$
- Bigrams of POS-tags: $\langle t_{i-2},t_{i-1}\rangle,\ \langle t_{i-1},t_i\rangle,\ \langle t_{i-1},t_{i+1}\rangle,\ \langle t_i,t_{i+1}\rangle,\ \langle t_{i+1},t_{i+2}\rangle$
- Unigrams and Bigrams over past supertagging decisions: $c_{i-2},\ c_{i-1},\ \langle c_{i-2},c_{i-1}\rangle$ (Only C&C supertagger)

Extracting features based on these templates from the training portion of CCGbank results in a total of $929,552$ instances for C&C. The model does not apply a frequency cutoff and therefore uses all of the features.

Category-set. We only model lexical categories which were observed at least 10 times in the training data. This simplifies the tagging problem by reducing the number of possible outcomes for the classifier from 1286 to 425. Subsequent work using the tagger with a parser (Clark and Curran, 2003, 2004a,b, 2007) demonstrates that this has negligible effect on coverage.

Beam Search. The beam size parameter β bounds the number of lexical categories considered for each word during training and testing. Supertags whose probability is lower than β times the probability of the best tag are removed. For the C&C supertagger we also use a forward-beam-ratio parameter ς set to 0.1. It completely removes supertags from the search graph during inference if the forward probability is $\varsigma \times \beta$ worse than the probability of the best tag.

Expansion probability	$p(exp\|\mathrm{P})$	$exp \in \{leaf, unary, left, right\}$
Head probability	$p(\mathrm{H}\|\mathrm{P}, exp)$	H is the head daughter
Non-head probability	$p(\mathrm{S}\|\mathrm{P}, exp, \mathrm{H})$	S is the non-head daughter
Lexical probability	$p(w\|\mathrm{P})$	

Table 3.2: Structural probabilities of the PCFG model. H, P, and S are categories, and w is a word.

Tagging Dictionary. The supertagger uses a tagging dictionary which contains, for each word, the set of categories the word has been seen with in the training data. If a word appears at least k times, the supertagger considers only categories from the dictionary for this word. Otherwise, a POS dictionary is consulted which lists all supertags for the POS-tag of the word.

Multi-tagging. The supertagger can predict multiple supertags per word. It returns for each word all categories whose marginal is within the beam β relative to the tag with the best marginal.

3.3.2 Generative Parsing

The parser of Hockenmaier (2001, 2003a,b) implements several different generative parsing models, we focus on two of them: First, PCFGModel treats the grammar as a simple binary context-free grammar without any internal structure. Second, HWDep models bilexical dependencies and is the most accurate model of their work. Each model incorporates a different interesting linguistic feature, and each has a generative story which completely describes the process of generating a parse tree and a sentence.

PCFG Model. The basic PCFG model has a very simple generative process: Given a parent category P, choose an expansion *exp* of P with **expansion probability** $p(exp|\mathrm{P})$, where *exp* can be either a *leaf*, for generating lexical categories, or *unary* for unary expansions such as type-raising, or *left* for binary trees with the head daughter on the left, or *right*, for binary trees with the head daughter on the right. If P is a leaf node, then generate its head word with **lexical probability** $p(w|\mathrm{P})$. Otherwise, generate its head daughter category H with **head probability** $p(\mathrm{H}|\mathrm{P}, exp)$ and its non-head daughter category with **non-head probability** $p(\mathrm{S}|\mathrm{P}, exp, \mathrm{H})$. Table 3.2 summarises these structural probabilities.

Expansion probability	$p(exp\|\mathrm{P}, c_\mathrm{P}\#w_\mathrm{P})$	$exp \in \{leaf, unary, left, right\}$
Head probability	$p(\mathrm{H}\|\mathrm{P}, exp, c_\mathrm{P}\#w_\mathrm{P})$	H is the head daughter
Non-head probability	$p(\mathrm{S}\|\mathrm{P}, exp, \mathrm{H}\#c_\mathrm{P}\#w_\mathrm{P})$	S is the non-head daughter
Lexcat probability	$p(c_\mathrm{S}\|\mathrm{S}\#\mathrm{P}, \mathrm{H}, \mathrm{S})$	$p(c_\mathrm{TOP}\|\mathrm{P}{=}\mathrm{TOP})$
Head word probability	$p(w_\mathrm{S}\|c_\mathrm{S}\#\mathrm{P}, \mathrm{H}, \mathrm{S}, w_\mathrm{P})$	$p(w_\mathrm{TOP}\|c_\mathrm{TOP})$

Table 3.3: Headword dependency model factorisation, backoff levels are denoted by '#' between conditioning variables: A#B#C indicates that $\hat{p}(\ldots|\mathrm{A},\mathrm{B},\mathrm{C})$ is interpolated with $\hat{p}(\ldots|\mathrm{A},\mathrm{B})$, which is an interpolation of $\hat{p}(\ldots|\mathrm{A},\mathrm{B})$ and $\hat{p}(\ldots|\mathrm{A})$. Variables c_P and w_P represent, respectively, the head lexical category and the head word of category P.

Extensions of this model use more features which are generated probabilistically. However, additional features mean more specific probability distributions for which some events may not have been observed in the training data. A standard technique to deal with this is to **smooth** the probability distributions such that they assign probabilities to unseen events. Hockenmaier uses a linear interpolation of probability distributions. The idea is to combine specific distributions with many conditioning variables with less specific distributions based on fewer variables (Hockenmaier, 2003a), e.g.,

$$p(y|x_1 \ldots x_i \ldots x_n) = \lambda p(y|x_1 \ldots x_i \ldots x_n) + (1-\lambda)p(y|x_1 \ldots x_i) \quad s.t.(0 \leq \lambda \leq 1) \quad (3.6)$$

where λ is a smoothing parameter, not a feature weight.

HWDep. Hockenmaier (2003b) describes a number of extensions to the `PCFG` model. The most noteworthy are the lexical category model `LexCat` and the head word dependency model `HWDep`. The `LexCat` model changes the generative story so that the parent node first generates a lexical category followed by the subtree rooted at P; in subsequent steps the lexical category is then used as additional conditioning context. The `HWDep` model follows a similar idea but captures word-word dependencies: It first generates a head word and then the lexical category, followed by the subtree. The generation of the subtree rooted at P is then based both on the head word as well as the lexical category. This is the best performing model presented in their work and its structural probabilities are given in Table 3.3.

Beam Search. In practice it is usually intractable to represent all parses for long sentences, even with efficient dynamic programming algorithms such as CKY (§2.3.4). It is therefore common to remove or *prune* low probability items during the search in

the hope that they will not lead to the best solution. This reduces memory requirements and increases speed but comes at the cost of exactness. Pruning is parameterised by a beam parameter $0 < b \leq 1$ which removes items whose probability is less than b-times the probability of the best comparable item; the best-item covers the same span of words and is in the same cell of the parsing chart. Such a threshold is also known as a global pruning threshold; we used the default setting of $b = 10^{-4}$.

Estimation. The models are estimated from training data where every sentence is annotated with a CCG derivation. The probability distributions of each model are maximum likelihood estimates (§2.3.2) based on frequency counts collected from the training data. For example, if we want to estimate the expansion probability in `PCFG`, for category NP being a leaf, we compute

$$\hat{p}(exp = leaf|\text{NP}) = \frac{f(\text{NP}, exp = leaf)}{\sum_{e'} f(\text{NP}, exp = e')} = \frac{f(\text{NP}, exp = leaf)}{f(\text{NP})} \tag{3.7}$$

where $f(\text{NP}, exp = leaf)$ is the frequency of observing NP as a leaf, while the denominator is simply the frequency of NP regardless of the expansion type.

Rare and Unknown Words. One of the problems with the parser of Hockenmaier is that the lexicon extracted from the training data does not provide good enough coverage for parsing the test-set. Parsers relying on a supertagger (Clark and Curran, 2007) transfer the burden of dealing with rare or unknown words to the tagger. The parser of Hockenmaier does not use a supertagger by default but it adopts a similar approach as the taggers described in §3.3.3: Every word falling below a frequency threshold is replaced by its POS tag. The intuition is that there are richer statistics for a POS tag than for a low-frequency word. The frequencies are computed on the training data and the threshold is tuned on held-out data. For all experiments reported, the frequency threshold was set to 30. During test time an unknown word is simply replaced by its POS tag and only supertags seen together with the POS tag are considered.

Grammar Implementation. The grammar is constructed based on the normal-form derivations in the training portion of CCGbank. The parser does not allow any categories to combine which have not been seen to combine in the training data. This restriction makes the grammar implemented by the parser effectively weakly context-free (Fowler and Penn, 2010).

3.3.3 Discriminative Parsing

This section describes the configuration of the Conditional Random Field (CRF) parser of Clark and Curran (2003, 2004a, 2007) which is part of the C&C tools. This CCG parser implements two different parsing models, the first is defined over dependency structures and the second over individual derivations. We will detail each of the models, their feature-sets, the way the grammar is implemented, a combined hybrid model, the various beam settings as well as the integration of the supertagger.

Log-Linear Form. Both models define the probability of a parse y and a sentence x in the usual log-linear form:

$$p(y|x) = \frac{1}{Z} \exp\left\{\theta^{\mathsf{T}} f(x,y)\right\} \tag{3.8}$$

However, the model is globally normalised, meaning that the partition function $Z = \sum_{y'} \exp\left\{\theta^{\mathsf{T}} f(x,y')\right\}$ is computed in terms of all possible parses y' for the sentence. The two models use different notions of what a parse constitutes and we will discuss each of them.

Dependency Model. This model defines a parse as a derivation-dependency structure pair $\langle d, y\rangle$. The probability of a dependency structure y (§2.1.2) is defined as the sum of the probability of all derivations d leading to y:

$$p(y|x) = \sum_{d \in \Delta(y)} p(\langle d, y\rangle|x) \tag{3.9}$$

where $\Delta(y)$ is the set of all derivations leading to dependency structure y. An alternative approach is to directly model dependencies as in Hockenmaier (2003b) and Clark et al. (2002). However, modelling both derivations and predicate-argument dependencies has the advantage that we can capture useful information within derivations to infer good dependency structures (Clark and Curran, 2007).

Both the dependency model and the normal-form model use a Gaussian prior term $G(\theta)$ to avoid overfitting. The training data $\mathcal{D} = \{\langle x^{(i)}, y^{(i)}\rangle\}_{i=1}^{m}$ for the dependency model is a set of sentences paired with a predicate-argument structure. The conditional

log-likelihood $\ell(\mathcal{D};\theta)'$ is as follows:

$$\ell'(\mathcal{D};\theta) = \ell(\mathcal{D};\theta) - G(\theta) \tag{3.10}$$

$$\ell'(\mathcal{D};\theta) = \log \prod_{i=1}^{m} p_\theta(y^{(i)}|x^{(i)}) - \sum_{\lambda_k \in \theta} \frac{\lambda_k^2}{2\sigma^2}$$

$$\ell'(\mathcal{D};\theta) = \sum_{i=1}^{m} \log \frac{\sum_{d \in \Delta(y^{(i)})} \exp\{\theta^\mathsf{T} f(x^{(i)}, \langle d, y^{(i)} \rangle)\}}{\sum_{\langle d', y' \rangle \in \Omega(x^{(i)})} \exp\{\theta^\mathsf{T} f(x^{(i)}, \langle d', y' \rangle)\}} - \sum_{\lambda_k \in \theta} \frac{\lambda_k^2}{2\sigma^2} \tag{3.11}$$

$$\ell'(\mathcal{D};\theta) = \sum_{i=1}^{m} \log \sum_{d \in \Delta(y^{(i)})} \exp\{\theta^\mathsf{T} f(x^{(i)}, \langle d, y^{(i)} \rangle)\}$$

$$- \sum_{i=1}^{m} \log \sum_{\langle d', y' \rangle \in \Omega(x^{(i)})} \exp\{\theta^\mathsf{T} f(x^{(i)}, \langle d', y' \rangle)\} - \sum_{\lambda_k \in \theta} \frac{\lambda_k^2}{2\sigma^2} \tag{3.12}$$

and the partial gradients are given by

$$\frac{\partial}{\partial \lambda_k} = \sum_{i=1}^{m} \sum_{d \in \Delta(y^{(i)})} \frac{\exp\{\theta^\mathsf{T} f(x^{(i)}, \langle d, y^{(i)} \rangle)\}}{\sum_{d' \in \Delta(y^{(i)})} \exp\{\theta^\mathsf{T} f(x^{(i)}, \langle d', y^{(i)} \rangle)\}} h_k(x^{(i)}, \langle d, y^{(i)} \rangle) \tag{3.13}$$

$$- \sum_{i=1}^{m} \sum_{\langle d, y \rangle \in \Omega(x^{(i)})} \frac{\exp\{\theta^\mathsf{T} f(x^{(i)}, \langle d, y \rangle)\}}{\sum_{\langle d', y' \rangle \in \Omega(x^{(i)})} \exp\{\theta^\mathsf{T} f(x^{(i)}, \langle d', y' \rangle)\}} h_k(x^{(i)}, \langle d, y \rangle) - \frac{\lambda_k}{\sigma_k^2}$$

where the Gaussian is parameterised by σ, which is the same for all feature-weights λ_k, and $h_k(x,y)$ is a feature-function returning the frequency of feature k for parse y of sentence x, and $\Omega(x)$ is the set of all dependency structures and derivations for x. The first term in the gradient (Equation 3.13) is the empirical expectation computed over the derivations that lead to a particular gold-standard dependency structure y; the second term computes the model expectations over all derivations for each sentence.

Normal-Form Model. This model defines a parse as a single derivation. Computationally, this is much less demanding than the dependency model which defines parses as sums over derivations. The training data $\mathcal{D} = \{\langle x^{(i)}, d^{(i)} \rangle\}_{i=1}^{m}$ consists of sentences x paired with normal-form derivations d. The objective function and the partial gradients are

$$\ell'(\mathcal{D};\theta) = \ell(\mathcal{D};\theta) - G(\theta) \tag{3.14}$$

$$= \log \prod_{i=1}^{m} p_\theta(d^{(i)}|x^{(i)}) - \sum_{\lambda_k \in \theta} \frac{\lambda_k^2}{2\sigma^2}$$

$$\frac{\partial}{\partial \lambda_k} = \sum_{i=1}^{m} h_k(x^{(i)}, d^{(i)}) \tag{3.15}$$

$$- \sum_{i=1}^{m} \sum_{d \in \rho(x^{(i)})} \frac{\exp\{\lambda_k h_k(x^{(i)}, d)\}}{\sum_{d' \in \rho(x^{(i)})} \exp\{\lambda_k h_k(x^{(i)}, d')\}} - \frac{\lambda_k}{\sigma_k^2}$$

where $d^{(i)}$ is the gold-standard normal-form derivation for sentence $x^{(i)}$ and $\rho(x)$ is the set of all derivations for sentence x. The empirical expectations are feature-counts over a single derivation-sentence pair and the model expectations are computed over all derivations for a sentence.

Viterbi and Minimum-Risk Decoding. Decoding with the normal-form model is straightforward since we can simply use the Viterbi algorithm (see §2.3.3) to recover the single-best derivation for a given sentence. However, for the dependency model we want to recover the highest scoring dependency structure, which is defined over sums of derivations, and computing these sums can be very expensive. Clark and Curran (2007) propose a minimum-risk decoding algorithm (§2.3.3) which recovers parses that maximise the *expected labelled recall rate*. Their algorithm is a dependency-variant of the algorithm developed by Goodman (1996). We outline the used algorithm briefly: A parse y is a set of dependencies τ; the parse $\hat{y}$ which maximises the expected recall rate is defined as

$$\hat{y} = \arg\max_{y} \mathop{\mathbb{E}}_{y_i \sim P(y_i|x)} [|y \cap y_i|] \tag{3.16}$$

The expected recall rate maximises the *unnormalised* expected recall which is why it is referred to as recall rate. Further, we can rewrite the expected recall rate for y as a sum over the individual dependencies in y:

$$\hat{y} = \arg\max_{y} \sum_{y_i} P(y_i|x)|y \cap y_i| \tag{3.17}$$

$$= \arg\max_{y} \sum_{y_i} P(y_i|x) \sum_{\tau \in y} 1 \text{ if } \tau \in y_i \tag{3.18}$$

$$= \arg\max_{y} \sum_{\tau \in y} \sum_{y' | \tau \in y'} P(y'|x) \tag{3.19}$$

The final expression is the sum of the expectations of all dependencies $\tau \in y$. We can use the inside probability of a dependency structure $I(\tau)$ and its outside probability $O(\tau)$ to efficiently compute these sums. The maximum expected recall rate parse is given as:

$$\hat{y} = \arg\max_{y} \sum_{\tau \in y} \frac{1}{Z} I(\tau) O(\tau) \tag{3.20}$$

Features. The features of the normal-form model are defined over local sub-trees of the parent and child categories in a derivation, whereas features of the dependency model are defined over predicate-argument dependencies. The two models have a number of features in common, such as features over both lexical categories and words,

Feature type	Example
LexCat + Word	NP + dog
LexCat + POS	NP + NN
RootCat	S[dcl]
RootCat + Word	S[dcl] + like
RootCat + POS	S[dcl] + VBD
Rule	$NP\ S[dcl]\backslash NP \Rightarrow S[dcl]$
Rule + Word	$NP\ S[dcl]\backslash NP \Rightarrow S[dcl]$ + liked
Rule + POS	$NP\ S[dcl]\backslash NP \Rightarrow S[dcl]$ + VBD

Table 3.4: Features common to both the normal-form model and the dependency model of Clark and Curran (2007).

or POS; features over the root category, extended by the head word of the root category and further generalised by the POS of the head word; there are also features over local rule sub-trees, extended by head words and the POS of head words; Table 3.4 summarises them and gives examples.

The dependency model uses predicate-argument relations, discussed in §2.1.2, as features. The relations used by the dependency model contain an extra field to indicate if the dependency is non-local. For example, the sentence *The equities that Vinken listed* has a long-range dependency between *equities* and *listed*:

$$\langle listed, (S\backslash NP_1)/NP_2, 2, equities, (NP\backslash NP)/(S[dcl]/NP)\rangle \tag{3.21}$$

This example dependency has the lexical category for *that* in the last field, which mediated the long-range dependency. The Word-Word dependency features are generalised by POS to yield features for Word-POS, POS-Word, and POS-POS dependencies; Table 3.5 shows some examples. Each variant is further extended by either distance information between the dependent words, which counts the number of intervening words, indicators for the presence of punctuation marks, and indicators on the presence of verbs; the possible distance values are 0, 1, 2, and more (> 2). The extended features do not include argument words, and similarly to before, there are variations which are generalised by POS-tags.

The normal-form model defines features in terms of the categories and head words of local subtrees, see Figure 3.6. These features are generalised to POS and variants

Feature type	Example
Word-Word	$\langle$ listed, $(S\backslash NP_1)/NP_2$, 2, equity, $(NP\backslash NP)/(S[dcl]/NP)\rangle$
Word-POS	$\langle$ listed, $(S\backslash NP_1)/NP_2$, 2, NN, $(NP\backslash NP)/(S[dcl]/NP)\rangle$
POS-Word	$\langle$ VBD, $(S\backslash NP_1)/NP_2$, 2, equity, $(NP\backslash NP)/(S[dcl]/NP)\rangle$
POS-POS	$\langle$ VBD, $(S\backslash NP_1)/NP_2$, 2, NN, $(NP\backslash NP)/(S[dcl]/NP)\rangle$
Word + Distance (words)	$\langle$ listed, $(S\backslash NP_1)/NP_2$, 2, $(NP\backslash NP)/(S[dcl]/NP)\rangle + > 2$
Word + Distance (punct)	$\langle$ listed, $(S\backslash NP_1)/NP_2$, 2, $(NP\backslash NP)/(S[dcl]/NP)\rangle + 0$
Word + Distance (verbs)	$\langle$ listed, $(S\backslash NP_1)/NP_2$, 2, $(NP\backslash NP)/(S[dcl]/NP)\rangle + 0$
POS + Distance (words)	$\langle$ VBD, $(S\backslash NP_1)/NP_2$, 2, $(NP\backslash NP)/(S[dcl]/NP)\rangle + > 2$
POS + Distance (punct)	$\langle$ VBD, $(S\backslash NP_1)/NP_2$, 2, $(NP\backslash NP)/(S[dcl]/NP)\rangle + 0$
POS + Distance (verbs)	$\langle$ VBD, $(S\backslash NP_1)/NP_2$, 2, $(NP\backslash NP)/(S[dcl]/NP)\rangle + 0$

Table 3.5: Features specific to the dependency model of Clark and Curran (2007).

with distance information similar to the dependency model.

Features are extracted from the training data and a frequency cutoff of two is applied, meaning that that only features observed at least twice are selected for the actual model. This results in 482,578 features for the normal-form model and 1,112,846 features for the dependency model.

Grammar Implementation. The parser models the same set of lexical categories as the supertaggers described before (§3.3.1). That is all supertags with a frequency of at least 10 in the training data, giving 425 lexical categories. The normal-form model, and the hybrid-dependency model described below, use two additional restrictions: First, the *Eisner constraints* (Eisner, 1996), discussed in §2.1.3, prevent constituents which are the outcome of forward composition to serve as the primary (or left) functor for another forward composition or application. Similarly, any constituent that is the result of backward composition cannot serve as the primary (right) functor for another backward composition or application. Second, only categories which have been seen to combine in the training data are allowed to combine during test time. Clark and Curran (2007) report that these restrictions have no detrimental effects on accuracy nor coverage. Neither of the constraints guarantees normal-form derivations but they eliminate enough derivations to vastly increase parsing speed. The constraints are used for both training and testing.

The grammar of the parser is carefully engineered to strike a good balance between

Feature type	Example
Word-Word	$\langle$ Mark, $NP\ S[dcl]\backslash NP \Rightarrow S[dcl]$, proved $\rangle$
Word-POS	$\langle$ Mark, $NP\ S[dcl]\backslash NP \Rightarrow S[dcl]$, VBD $\rangle$
POS-Word	$\langle$ NN, $NP\ S[dcl]\backslash NP \Rightarrow S[dcl]$, proved $\rangle$
POS-POS	$\langle$ NN, $NP\ S[dcl]\backslash NP \Rightarrow S[dcl]$, VBD $\rangle$
Word + Distance(words)	$\langle$ proved, $NP\ S[dcl]\backslash NP \Rightarrow S[dcl]\ \rangle + > 2$
Word + Distance(punct)	$\langle$ proved, $NP\ S[dcl]\backslash NP \Rightarrow S[dcl]\ \rangle + 0$
Word + Distance(verbs)	$\langle$ proved, $NP\ S[dcl]\backslash NP \Rightarrow S[dcl]\ \rangle + 0$
POS + Distance(words)	$\langle$ VBD, $NP\ S[dcl]\backslash NP \Rightarrow S[dcl]\ \rangle + > 2$
POS + Distance(punct)	$\langle$ VBD, $NP\ S[dcl]\backslash NP \Rightarrow S[dcl]\ \rangle + 0$
POS + Distance(verbs)	$\langle$ VBD, $NP\ S[dcl]\backslash NP \Rightarrow S[dcl]\ \rangle + 0$

Table 3.6: Features specific to the normal-form model of Clark and Curran (2007).

accuracy and coverage. Appendix A of Clark and Curran (2007) lists a range of design decisions. For example, type-raising is implemented as a set of only eight unary rules which are applied every time one of the categories NP, PP and $S[adj]\backslash NP$ are created. Similarly, only a subset of the type-changing rules occurring in CCGbank were implemented. This results in a very fast parser but it is at the same time language-specific.

While the restrictions make the parser very efficient, it has to be noted that their use makes the grammar effectively strongly context-free. This means it is possible to construct an equivalent context-free grammar, which can model the same analysis as the CCG grammar. Two restrictions are equally responsible for the implemented grammar being strongly context-free (Fowler and Penn, 2010): Allowing only a finite-set of categories, and suppressing the combination of categories which have not been seen to combine in the training data. The parser allows the removal of these restrictions but we did not do this in our experiments.

Used Model: Hybrid Dependency Model. Estimation of the dependency model requires vast amounts of memory since it does not use the grammar restrictions described above. To make estimation possible, Clark and Curran (2007) introduce a hybrid model which is the dependency model with the same restrictions imposed on the normal-form model i.e. the Eisner normal-form derivations and allowing only observed category combinations. This model consumes considerably less memory than the dependency model and achieves the best performance presented in Clark and Curran (2007). We use it in Chapters 5 and 6.

Beam Search and Maximum Chart Size. The parser implements a global beam-threshold which was recently extended to speed up the parser even further (Clark et al., 2009). However, we did not use it for our experiments, following previous work (Clark and Curran, 2007) which solely relied on the supertagger to make search efficient.

The maximum chart size parameter limits the number of items in the parser chart. We follow Clark and Curran (2007) and set it to 1,000,000 for testing and 300,000 for estimation. If this limit is exceeded, then the best sentence-spanning analysis is returned, or if there is none, a parse failure is reported. However, when the parser is used in adaptive supertagging mode, then the next iteration will be initiated.

Supertagger-Integration. The parser is tightly integrated with a supertagger which prunes the set of possible lexical categories for each word (Clark and Curran, 2004b). We use the C&C supertagger described in §3.3.1 with this parser. The supertagger is used both during training and test time. The test-time settings are identical to the ones used by Clark and Curran (2004b) and we will detail them in Chapter 5.

3.4 Estimation

This section covers the hyper-parameters and settings used during estimation of the discriminative parsing model of Clark and Curran (2007); its test time configuration was discussed in §3.3.3. We do not describe the training process for the generative parsing models of Hockenmaier (2003a) since we used it exclusively for decoding; details about the training process can be found in their work.

Training Beams. The estimation methods we used require repeated calculation of expectations over the entire training corpus, potentially hundreds of times. Computing model expectations over parses entails the construction of all possible derivations for all sentences in the training data. Moreover, since expectations are repeatedly computed, it is usually necessary to keep the derivations in memory for efficiency.

However, the memory requirements of all possible derivations is usually prohibitively large, even for simple sentences. In order to make estimation feasible we use two pruning methods: We limit the maximum number of chart items per sentence to 300,000 and use the supertagger (Clark, 2002; Curran et al., 2006) to prune the set of lexical categories. The beam step function for the supertagger starts with a wide beam and narrows it at each iteration if the maximum chart size is exceeded.

Figure 3.7 shows the beam step function we used (`Training`), as well as an alternative presented in previous work (Clark and Curran, 2003, 2004a,b, 2007). Our

Condition	Parameter	Iteration 1	2	3	4	5	6	7
Training	β	0.001	0.001	0.0045	0.0055	0.01	0.05	0.1
	k	150	20	20	20	20	20	20
C&C '07	β	0.0045	0.0055	0.01	0.05	0.1		
	k	20	20	20	20	20		

Table 3.7: Beam step function used for training. Parameter β is a beam threshold while k bounds the use of a part-of-speech tag dictionary which is used for words seen less than k times (see §3.3.1).

Condition	items per sent	total items
Training	16,116	605,885,365
C&C '07	4,606	172,793,049

Table 3.8: Size of training forests measured in number of items. We quote the average per sentence and the total number of items, both for our beam settings as well as the settings used in previous work (Clark and Curran, 2007).

beams are less aggressive than previous work since our experiments are more recent and we therefore had larger computing resources available. However, we found that the larger beams had only a marginal effect on accuracy: The new beams increased labelled F-measure from 87.24% to only 87.38% on the development set and from 87.64% to 87.73% on the test set. Despite negligible effect on accuracy, we find that the beam settings have a significant effect on the size of the forests generated for the training corpus, as shown in Table 3.8. Our beam step function more than triples the size of the training forests, which results in very high memory requirements of over 100 gigabytes of main memory when loaded.

Training Data Coverage. Expectation-based training requires that each forest contains the gold-standard parse; for the normal-form model this is the treebank derivation, and for the dependency-model we require at least one derivation to produce the gold-standard predicate-argument structure. The utilisation of the training data is about 91.5% for both models since the gold-standard parse cannot be recovered for all sentences. There are several reasons why: First, the parser cannot produce all rule combi-

nations occurring in CCGbank, because some rules do not correspond to instances of CCG rule combinations, and others are rare punctuation handling rules which are not modelled for efficiency reasons (Clark and Curran, 2007). Second, the beam settings above may result in the loss of the gold-standard parse and therefore a decrease in the number of sentences available for training. However, we found that the use of our less aggressive beam settings made no difference which suggests that loosening the beam settings will not alleviate the problem.

One issue arising with the supertagger is that it may not always find the supertag required for the gold-standard parse. In these cases, we simply add the gold-standard supertag to the chart, following Clark and Curran (2007). Adding gold-supertags is required for 5.6% of training sentences.[4]

Parallel Estimation. As previously discussed, we need to keep all training forests in memory which is impractical on a single machine. To solve this problem we used the MPI architecture developed for the C&C parser by Clark and Curran (2007) to distribute the training forests across a cluster of machines. This allows the parallel computation of expectations. A master machine aggregates the computed expectations and takes the gradient-step, upon which the new parameters are distributed to compute the next set of expectations. We used the MPICH2 implementation (Gropp et al., 1999) to manage the communication between machines.

L-BFGS. We use the Limited-memory Broyden–Fletcher-Goldfarb-Shanno method (L-BFGS) for training the feature-weights, an iterative numerical optimisation algorithm. At each iteration it requires a function value, the conditional log-likelihood, and the first and second order partial derivatives of this function in order to choose the next gradient step direction. The second order partial derivatives are approximated using the previous function evaluations and the corresponding first order derivations. This approximation avoids full computation of the second order partial derivatives, full computation is usually intractable. L-BFGS retains only the last few function evaluations, which is why it limits memory.

We used the implementation that is part of the C&C tools (Clark and Curran, 2007). Training with L-BFGS requires repeated evaluation of the conditional log-likelihood

[4]The data utilisation figure of 91.5% does already include adding gold-supertags.

function, as defined in §3.3.3 and reproduced below:

$$\ell'(\mathcal{D};\theta) = \ell(\mathcal{D};\theta) - G(\theta) \tag{3.22}$$

$$\ell'(\mathcal{D};\theta) = \log \prod_{i=1}^{m} p_\theta(y^{(i)}|x^{(i)}) - \sum_{\lambda_k \in \theta} \frac{\lambda_k^2}{2\sigma^2} \tag{3.23}$$

The Gaussian prior term $G(\theta)$ has a single parameter which we set to $\sigma = 0.6$ following Clark and Curran (2007). L-BFGS converges if the percentage change in the objective function is less than 0.0001% in any iteration and we keep a history of the last ten evaluations.

SGD. L-BFGS examines all training examples before taking a step, whereas Stochastic Gradient Descent (SGD) uses only a sample of the training data to compute gradients. The underlying idea is that examining the entire training set for a single step may be wasteful, as it may contain much duplicate information. Clearly, the steps made by L-BFGS may be better but it can make far fewer than SGD in the same time (Sutton and McCallum, 2011). The hope is that making many low quality steps may be faster and as good as taking a few very accurate steps. The gradients used in SGD are based on samples of the training data $\mathcal{D}_b$ of size b such that

$$\mathbb{E}\left[\sum_{i=1}^{B} \ell(\mathcal{D}_b^{(i)};\theta)\right] = \ell(\mathcal{D};\theta) \tag{3.24}$$

where B is the number of samples drawn from the training data, possibly with replacement. Note that any priors must be scaled down by a factor of $b/|\mathcal{D}|$. The weight update used in SGD is

$$\theta_{q+1} = \theta_q - \mu_q \nabla \ell(\mathcal{D}_b^{(i)};\theta_q) \tag{3.25}$$

where μ_q is a temperature parameter that is adjusted according to the following schedule:

$$\mu_q = \mu_0 \frac{\zeta}{\zeta + q} \tag{3.26}$$

where μ_0 is the initial temperature and ζ determines how quickly μ changes over iterations $q = 1, \ldots, Q$. The main disadvantage of SGD is that it requires the tuning of various settings, whose optimal value often depends on the particular task. There are many settings which can substantially influence performance such as the initial temperature μ_0, the speed by which the temperature changes over time ζ, the batch size b, or the number of training iterations Q (Sutton and McCallum, 2011).

For our experiments we used the parameter settings proposed by Finkel et al. (2008) for their context-free grammar parser, which generalised well to our models. We ran

SGD for ten passes through the training data. Each sample was of size $b = 30$, drawn with replacement. The initial temperature for the update schedule was set to $\mu_0 = 0.1$ and we set ζ, the parameter controlling the decay of the temperature μ, so that μ would half after exactly five passes.

3.5 Conclusions

This chapter outlined our experimental setup to evaluate the accuracy and efficiency of CCG parsing. The following chapters present a novel integrated supertagging and parsing model and the evaluation methods discussed here are used to measure its accuracy and efficiency. Specifically, we have covered the following:

- CCGbank, the treebank we use to train and evaluate our models.
- Evaluation metrics to measure the accuracy of CCG parsers and the setup to measure efficiency in terms of CPU time.
- The models and the configuration of the parsers and supertaggers used in this thesis.
- The estimation of a discriminative CCG parsing model on which our integrated model is based.

The following chapters present an analysis of supertagging, the baseline for the integrated supertagging and parsing model, as well as a training method to find accurate parsing models.

Chapter 4

Efficient Search for CCG Parsing

This chapter deals with efficient parsing of Combinatorial Categorial Grammar (CCG; Steedman, 2000; §2.1). Parsing with CCG can be much harder than with Penn Treebank-style context-free grammars, since the number of non-terminal categories is generally much larger, leading to higher parsing complexity (§2.3.4). Where a typical Penn Treebank grammar may have fewer than 100 non-terminals, we found that a CCG grammar derived from CCGbank contained nearly 1600. The same grammar assigns an average of nearly 20 lexical categories per word, resulting in a very large space of possible derivations (Hockenmaier and Steedman, 2007).

The most successful strategy to date for efficient parsing of CCG is *adaptive supertagging* (§2.2.2) which is based on pruning lexical categories with a supertagger. However, pruning means approximate search: if a lexical category used by the highest probability derivation is pruned, the parser will not find that derivation. Since the supertagger enforces no grammaticality constraints it may even prefer a sequence of lexical categories that cannot be combined into *any* derivation. Empirically, we show that supertagging improves efficiency by an order of magnitude, but the tradeoff is a significant loss in accuracy (§4.2).

Can we improve on this tradeoff? The line of investigation we pursue in this chapter is to consider more efficient *exact* algorithms for Viterbi-parse selection (§2.3.3). In particular, we test different variants of the classical A* algorithm (Hart et al. 1968; §4.1), which has met with success in Penn Treebank parsing with context-free grammars (Klein and Manning, 2003; Pauls and Klein, 2009a,b). We can substitute A* for standard CKY (§2.3.4) on either the unpruned set of lexical categories, or the pruned set resulting from supertagging. Our empirical results show that on the unpruned set of lexical categories, heuristics employed for context-free grammars show substantial

speedups in hardware-independent metrics of parser effort (§4.3). To understand how A* compares to the CKY baseline, we conduct a carefully controlled set of timing experiments. Although our results show that improvements on hardware-independent metrics do not always translate into improvements in CPU time due to increased processing costs that are hidden by these metrics, they also show that when the lexical categories are pruned using the output of a supertagger, then we can still improve efficiency by 15% with A* techniques (§4.4).

4.1 A* Parsing

Irrespective of whether lexical categories are pruned in advance using the output of a supertagger, most CCG parsers we are aware of use some variant of the CKY algorithm.[1] Although CKY is easy to implement, it is *exhaustive*: it explores all possible analyses of all possible spans, irrespective of whether such analyses are likely to be part of the highest-probability derivation. Hence it seems natural to consider exact algorithms that are more efficient than CKY.

A* is an agenda-based best-first graph search algorithm which finds the lowest cost parse exactly without necessarily traversing the entire search space (Klein and Manning, 2003). Both CKY and A* search the same hypergraph defined by the deductive system described in §2.3.4. However, A* does not process items in topological order using a simple control loop such as CKY, or the inside algorithm. Instead, it processes promising items before less-likely ones. The ensuing description of A* parsing follows Klein and Manning (2003).

An A* parser maintains two data-structures: An agenda of newly-formed items waiting to be processed and, similar to CKY, a chart data structure to record items for which the highest-probability parse has already been found. The main control loop removes the item with highest probability from the agenda and combines it with items already in the chart to build new items. For example, if item $S\backslash NP_{1,3}$ were removed from the agenda and item $NP_{0,1}$ is already in the chart, then item $S_{0,3}$ would be created, and added to the agenda, if it is not in the chart already (Figure 4.1).

During parsing we are not only interested if certain items can be built at all, but also what the highest probability of an item is. We therefore record estimates of item probabilities and update them as better parses for the item are found. If we find a better

[1] The recently introduced shift-reduce parser of Zhang and Clark (2011) is a notable exception.

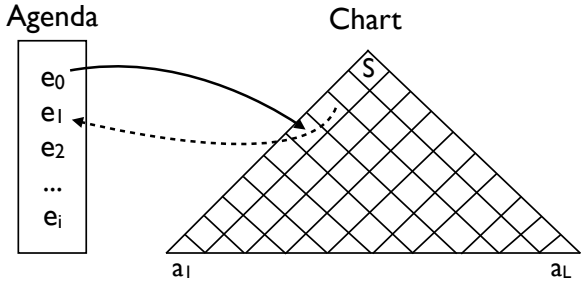


Figure 4.1: Illustration of agenda-based parsing. Items e_i compete on an agenda to be put into the chart and to be used in building new items, which are again pushed onto the agenda.

way to construct an item already on the agenda, then we simply update the estimate; new items with a worse estimate than previously recorded are simply discarded.

Items on the agenda are processed by their priority: Promising items are explored first, others can wait on the agenda indefinitely. We would like to assign priorities which speed up parsing but also guarantee exactness of the recovered parses, meaning that the first sentence-spanning parse returned is indeed the highest-probability parse. Exactness is ensured by requiring that for any item e, all items contained in the best derivation of e get removed from the agenda before e itself. If the item priorities ensure this ordering, then the items leaving the agenda will carry their correct probabilities. One way to guarantee a correct ordering is to assign shorter spans higher priority than longer ones; this priority gives essentially the CKY algorithm but has the disadvantage of building all items. We will later simulate CKY using this priority.

Formally, we define A* parsing using *Viterbi inside items* $\bar{I}(A_{i,j}) : w$ with Viterbi inside probability w, the weight of the highest-probability inner-derivation for $A_{i,j}$ discovered so far. As long as an item is on the agenda, it represents the best estimate of its true probability discovered so far, however, once removed, it does represent the true probability, given that the pre-conditions for exactness are met. Note that in this chapter we are only concerned with the recovery of Viterbi parses, i.e., the highest-probability parse for a sentence, and therefore $\bar{I}(A_{i,j})$ does not represent the inside probability of $A_{i,j}$ but rather the probability of a single derivation rooted at $A_{i,j}$. The A* parser maintains estimates for the Viterbi inside probabilities and improves them over time as better analyses for $A_{i,j}$ are discovered. Priorities $p = w \otimes h(A_{i,j})$ for Viterbi inside items are defined as the product[2] of the Viterbi inside probability w and

[2]Assuming the Viterbi semiring in Table 2.1.

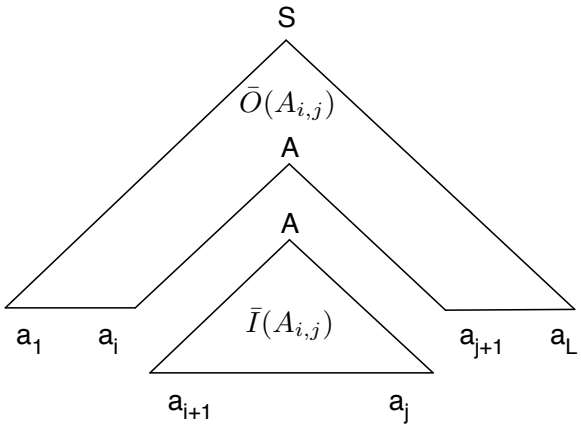


Figure 4.2: Illustration of the Viterbi inside and outside probabilities.

a heuristic estimate $h(A_{i,j})$ of the Viterbi outside probability. Note that we are not strictly required to use a heuristic and we could simply use $p = w$; no heuristic can also lead to improvements as we show in §4.3.1. Estimates are model-specific and we consider several variants in our experiments based on the CFG heuristics developed by Klein and Manning (2003; §4.1.1) and Pauls and Klein (2009; §4.1.2). We also define *Viterbi outside items* $\bar{O}(A_{i,j})$ to represent the probability of an outer-derivation of $A_{i,j}$. The deductive system to compute Viterbi inside probabilities is identical to the inside algorithm (Figure 2.17) with the Viterbi semiring (Table 2.1), similarly for Viterbi outside probabilities and the outside algorithm (Figure 2.20). Figure 4.2 illustrates Viterbi inside items and Viterbi outside items.

In order for A* search to be exact we require the heuristic estimates $h(A_{i,j})$ to be both *admissible* and *monotonic*. Admissibility means that the heuristic never overestimates the true Viterbi outside probability, i.e., the probability to complete the parse. Monotonicity means that the priority $p = w \otimes h(A_{i,j})$ never increases when building up a parse. If these conditions are met, then we can be sure that the probabilities of the items popped off the agenda are exact; proof can be found in Klein and Manning (2002). A* parsing terminates if we pop off the first sentence-spanning Viterbi inside item $\bar{I}(S_{0,L})$ since it represents the highest-probability parse for the sentence.

Next we detail two types of heuristic estimates: Pre-computed heuristics which offer constant-time lookup (§4.1.1) and heuristics computed at test-time wich are based on grammar-projections (§4.1.2).

4.1.1 Pre-computed Heuristics

Pre-computed heuristics provide an admissible estimate of the Viterbi outside probability for items encountered during parsing. Since the estimates are pre-computed, the cost can simply be looked up in constant time. The look-up is based on the item as well as a summary of the *context*, a summary of the parts of the sentence not covered by the item for which we require an estimate. Generally, more detailed context summaries provide better estimates and allow more efficient search, however, more detailed summaries make estimates also more expensive to compute and to store.

In this setting, a heuristic estimate is the lowest probability parse for *any* sentence fitting the context summary, which ensures that the heuristic is admissible. A number of pre-computed heuristics have been devised for context free grammars resulting in substantial reduction of parsing effort (Klein and Manning, 2003). We will apply the same idea to improve efficiency for CCG parsing (§4.3.1).

For our experiments we chose the `SX` heuristic devised by Klein and Manning (2003), which has been shown to offer a good trade-off between a reduction in parsing effort and computational effort.[3] The estimate $[A_{i,j},k,l]$ provides a bound on the Viterbi outside probability of $A_{i,j}$ when there are k words in the right context of the item and l words on the left (Figure 4.3). The heuristic estimate for $A_{i,j}$ is pre-computed by considering all parses fitting the context of $[A_{i,j},k,l]$ and choosing the parse with lowest-probability.[4]

Formally, we are given sentence $s = a_1 \dots a_L$ and context $c = (a_1 \dots a_i, A_{i,j}, a_{j+1} \dots a_L)$, abbreviated as $c(A_{i,j},s)$ and a context summary function $\sigma(c(A_{i,j},s))$ provides a summary of $c(A_{i,j},s)$. For example, given the sentence *time flies like an arrow* we have the following context and summary for $I(S\backslash NP_{1,2})$ with the `SX` heuristic:

$$c(S\backslash NP_{1,2},s) = (\text{time}, S\backslash NP_{1,2}, \text{like}, \text{an}, \text{arrow})$$
$$\sigma(c(S\backslash NP_{1,2},s)) = [S\backslash NP_{1,2}, 1, 3]$$

If summary contains no information, then the estimate is always 0; we call this the `NULL` estimate, it corresponds to simply using the Viterbi inside probability alone as priority for an item $I(A_{i,j})$.

For notational convenience we use A to denote non-terminal $A_{i,j}$, and $\bar{O}(A,s)$ to denote the Viterbi outside probability of $A_{i,j}$ in sentence $s = a_1 \dots a_L$. We can then

[3] The large non-terminal set for our CCG grammar made it impossible to experiment with more sophisticated pre-computed heuristics discussed in (Klein and Manning, 2003).

[4] Pseudo-code can be found in Figure 10 of Klein and Manning (2003).

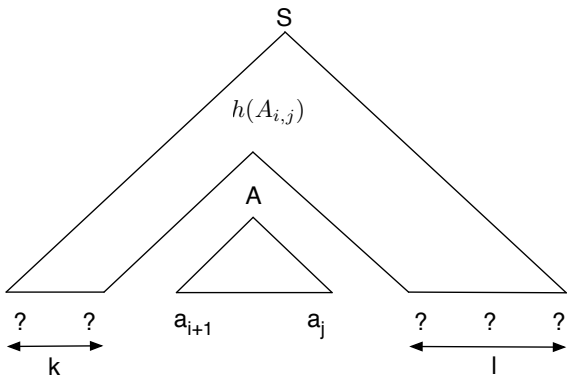


Figure 4.3: Illustration of the SX heuristic that provides an estimate of the Viterbi outside probability for an item when there are k words in the left context and l words on the right.

compute the estimate $h(A)$ as follows:[5]

$$h(A) = \min_{(A',s'):\sigma(c(A',s'))=\sigma(c(A,s))} \bar{O}(A,s)$$

which is the exact Viterbi outside probability $\bar{O}(A',s')$ for *some* context $c(A',s')$ whose context summary fits the summary of the current sentence $c(A,s)$. This requires constructing all Viterbi parses for every possible context summary which can be intractable for complex heuristics.

4.1.2 Hierarchical A* and Grammar Projection Estimates

Pre-computed heuristics provide fast look-up but they are based on context summaries rather than the actual context of the sentence that is currently being parsed. Estimates based on context summaries become increasingly inaccurate for large contexts because the parses, which back these estimates, have less and less to do with the actual contexts (Klein and Manning, 2003).

Grammar projection estimates use the true context, i.e., the actual sentence, rather than summaries (Klein and Manning, 2003; Felzenszwalb and McAllester, 2007; Pauls and Klein, 2009a). This allows the heuristics to be more specific than pre-computed estimates, mainly because the actual context is known and one is not constrained by the issues of computing and storing estimates for all possible context summaries. As a downside we have to calculate the heuristic during parsing, whereas pre-computed heuristics can simply be looked-up. Grammar projection estimates do not use the true

[5]Weights are not log-probabilities, i.e., minimisation returns the lowest probability parse.

grammar G with which we are trying to parse, but rather project G to some G' with which parsing is much simpler. The idea is to first parse with G' exhaustively and to use the result to guide the search in the original grammar G.

An extension to the idea of grammar projection estimates is *Hierarchical A** (HA*; Felzenszwalb and McAllister 2007, Pauls and Klein 2009a) which avoids having to to parse exhaustively with G' in order to find the best parse in G. Intuitively, we may not require all of the items in G' to provide estimates for parsing with G. HA* can parse with multiple grammar projections as well as the true grammar at the same time, making items of different grammars compete on a single agenda and prioritising them such that necessary estimates are computed before they are required.

We consider the case of a single grammar projection G' and the true grammar G, the more general case is discussed in Pauls and Klein (2009a): The parser first computes Viterbi inside probabilities in G', then uses the resulting items to compute Viterbi outside probabilities in G', and then uses the Viterbi outside items as estimates for parsing the true grammar.

The grammar projection is constructed as follows: Projection G' defines a mapping $\pi : \Sigma \to \Sigma'$ of the non-terminals Σ in G to a reduced set, the non-terminals Σ' in G'. The rule combinations licensed by G' should never have higher weights (or probabilities) than their counterparts in G. Otherwise the resulting heuristic may overestimate the true probabilities, which would make the heuristic inadmissible. More formally, for all rules r of the form $BC \Rightarrow A : w_r$ with weight w_r in G, we require the weight $w_{r'}$ of the projected counterpart r', that is $B'C' \Rightarrow A' : w_{r'}$, to be smaller or equal $w_{r'} \leq w_r$. Weights for rule projections can be found by minimising over the rules of the target grammar collapsed by π:

$$w_{r'} = \min_{r \in G : \pi(r) = r'} w_r \tag{4.1}$$

We extend the item notation to make explicit the priorities assigned to items: Viterbi inside items $\bar{I}(A_{i,j}) : \langle w_1, p \rangle$ show both the Viterbi inside probability w_1, and priority p. The deductive rules for A* parsing in the true grammar with a single grammar projection are given in Figure 4.4. Both deductive rules require an outside item of the gramamr projection G' to prioritise the new item. The figure only shows the deductive rules for building Viterbi inside items $\bar{I}(A_{i,j})$ in the true grammar using the previously computed grammar projection estimates; items for the projection are built with the standard inside algorithm (Figure 2.17) and the outside algorithm (Figure 2.20). The priority of grammar projection items on the agenda is simply the Viterbi inside

$$\textbf{IN}-\textbf{Base}: \frac{a_i \Rightarrow A : w_r \quad \bar{O}(A'_{i-1,i}) : w_o}{\bar{I}(A_{i-1,i}) : \langle w_r, w_o \rangle}$$

$$\textbf{IN}-\textbf{Rec}: \frac{BC \Rightarrow A : w_r \quad \bar{O}(A'_{i,j}) : w_o \quad \bar{I}(B_{i,k}) : w_1 \quad \bar{I}(C_{k,j}) : w_2}{\bar{I}(A_{i,j}) : \langle w_r \otimes w_1 \otimes w_2, w_o \rangle}$$

Figure 4.4: Deduction schema for A* parsing with a grammar projection. We explicitly show the calculation of priorities of new items using notation $I(A_{i,j}) : \langle w, p \rangle$ where A is a non-terminal of the true grammar G, and w is the Viterbi inside probability, and p is the agenda-priority. A' is a non-terminal of the grammar projection that is mapped to A in the true grammar G.

or outside probability, depending on the item type. We describe the actual grammar projections used for experimentation in §4.3.2.

4.2 Adaptive Supertagging Experiments

Supertagging has been shown to improve the speed of a generative parser, although little analysis has been reported beyond speedups (Clark, 2002). We ran experiments to understand the time/accuracy tradeoff of adaptive supertagging, and to serve as baselines to A* parsing (§4.3).

For our experiments we used the generative CCG parser of Hockenmaier and Steedman (2002). Generative parsers have the property that all item weights are non-negative, which is required for A* techniques.[6] Although not quite as accurate as the discriminative parser of Clark and Curran (2007) in our preliminary experiments, this parser is still quite competitive. It implements the CKY algorithm and we focus on the `PCFG` and `HWDep` parsing models described in §3.3.2. For supertagging we used Dennis Mehay's implementation of Clark (2002; §3.3.1). Due to differences in smoothing of the supertagging and parsing models, we occasionally drop supertags returned by the supertagger because they do not appear in the parsing model.[7] We evaluate on *all* sentences and thus penalise lower coverage.

Adaptive supertagging is parameterised by a beam size β and a dictionary cut-

[6] Indeed, all of the past work on A* parsing that we are aware of uses generative parsers (Pauls and Klein, 2009b, *inter alia)*.

[7] Less than 2% of supertags are affected by this.

Condition	Parameter	Iteration 1	2	3	4	5	6
AST	β (beam width)	0.075	0.03	0.01	0.005	0.001	
	k (dictionary cutoff)	20	20	20	20	150	
AST-covA	β	0.075	0.03	0.01	0.005	0.001	0.0001
	k	20	20	20	20	150	150
AST-covB	β	0.03	0.01	0.005	0.001	0.0001	0.0001
	k	20	20	20	20	20	150

Table 4.1: Beam step function used for standard (AST) and high-coverage (AST-covA and AST-covB) supertagging.

off *k* that bounds the number of lexical categories considered for each word (§3.3.1). Table 4.1 shows both the standard beam levels (AST) used by the C&C parser and looser beam levels: AST-covA, a simple extension of AST with increased coverage and AST-covB, also increasing coverage but with better performance for the HWDep model.

4.2.1 Results

Parsing results for the AST settings (Tables 4.2 and 4.3) confirm that it improves speed by an order of magnitude over a parser without AST. Perhaps surprisingly, the number of parse failures decreases with AST in some cases. The reason is that without AST, the parser needs to prune more aggressively in order to make search feasible, and the additional pruning removes more sentence-spanning derivations than AST would, which leads to more parse failures.[8]

AST also clearly narrows the speed gap between the two models: Without AST, PCFG is over five times faster than HWDep, but with AST the difference narrows to a factor of two. The next section analysis the speed an accuracy tradeoff with AST in more detail.

[8]Hockenmaier and Steedman (2002) saw a similar effect.

	Time(sec)	Sent/sec	Cats/word	Fail	UP	UR	UF	LP	LR	LF
PCFG	290	6.6	26.2	5	86.4	86.5	86.5	77.2	77.3	77.3
PCFG (AST)	**65**	**29.5**	1.5	14	**87.4**	85.9	86.6	**79.5**	78.0	78.8
PCFG (AST-covA)	67	28.6	1.5	6	87.3	**86.9**	**87.1**	79.1	**78.8**	**78.9**
PCFG (AST-covB)	69	27.7	1.7	5	87.3	86.2	86.7	79.1	78.1	78.6
HWDep	1512	1.3	26.2	5	**90.2**	**90.1**	**90.2**	**83.2**	**83.0**	**83.1**
HWDep (AST)	**133**	**14.4**	1.5	16	89.8	88.0	88.9	82.6	80.9	81.8
HWDep (AST-covA)	139	13.7	1.5	9	89.8	88.3	89.0	82.6	81.1	81.9
HWDep (AST-covB)	155	12.3	1.7	7	90.1	88.7	89.4	83.0	81.8	82.4

Table 4.2: Results on development set (CCGbank section 00) when applying adaptive supertagging (AST) to two models of a generative CCG parser. Performance is measured in terms of parse failures, labelled and unlabelled precision (LP/UP), recall (LR/UR) and F-score (LF/UF).

	Time(sec)	Sent/sec	Cats/word	Fail	UP	UR	UF	LP	LR	LF
PCFG	326	7.4	25.7	29	85.9	85.4	85.7	76.6	76.2	76.4
PCFG (AST)	**82**	**29.4**	1.5	34	**86.7**	84.8	85.7	**78.6**	76.9	77.7
PCFG (AST-covA)	85	28.3	1.6	15	86.6	85.5	86.0	78.5	**77.5**	**78.0**
PCFG (AST-covB)	86	27.9	1.7	14	86.6	**85.6**	**86.1**	78.1	77.3	77.7
HWDep	1754	1.4	25.7	30	**90.2**	**89.3**	**89.8**	**83.5**	**82.7**	**83.1**
HWDep (AST)	**167**	**14.4**	1.5	27	89.5	87.6	88.5	82.3	80.6	81.5
HWDep (AST-covA)	177	13.6	1.6	14	89.4	88.1	88.8	82.2	81.1	81.7
HWDep (AST-covB)	188	12.8	1.7	14	89.7	88.5	89.1	82.5	81.4	82.0

Table 4.3: Results on test set (CCGbank section 23) when applying adaptive supertagging (AST) to two models of a CCG parser (cf. Table 4.2).

4.2.2 Efficiency versus Accuracy

The most interesting result is the effect of the speedup on accuracy. As shown in Table 4.4, the vast majority of sentences are actually parsed with a very tight supertagger beam, raising the question of whether many higher-scoring parses are pruned.[9] Despite heavy pruning, labelled F-score improves by up to 1.6 for the PCFG model, although it harms accuracy for the HWDep model as expected.

β	Cats/word	Parses	%
0.075	1.33	1676	87.6
0.03	1.56	114	6.0
0.01	1.97	60	3.1
0.005	2.36	15	0.8
$0.001_{k=150}$	3.84	32	1.7
Fail		16	0.9

Table 4.4: Breakdown of the number of development set sentences parsed for the HWDep model (see Table 4.2) at each of the supertagger beam levels from the most to the least restrictive setting when using adaptive supertagging.

In order to understand this effect, we filtered section 00 to include only sentences of between 18 and 26 words (resulting in 610 sentences) for which we can perform exhaustive search without pruning[10], and for which we could parse without failure at all of the tested beam settings. We then measured the log-probability of the highest probability parse found under a variety of beam settings, relative to the log-probability of the unpruned exact parse, along with the labelled F-score of the Viterbi parse under each beam setting (Figure 4.5). The results show that PCFG actually finds worse results as it considers more of the search space. In other words, the supertagger can actually "fix" a bad parsing model by restricting it to a small portion of the search space. With the more accurate HWDep model, finding worse results in larger search spaces does not appear to be a problem and there is a clear improvement when considering the larger search space. The next question is whether we can exploit this larger search space without paying as high a cost in efficiency.

[9] Similar results are reported by Clark and Curran (2007).

[10] The fact that only a subset of short sentences could be exhaustively parsed demonstrates the need

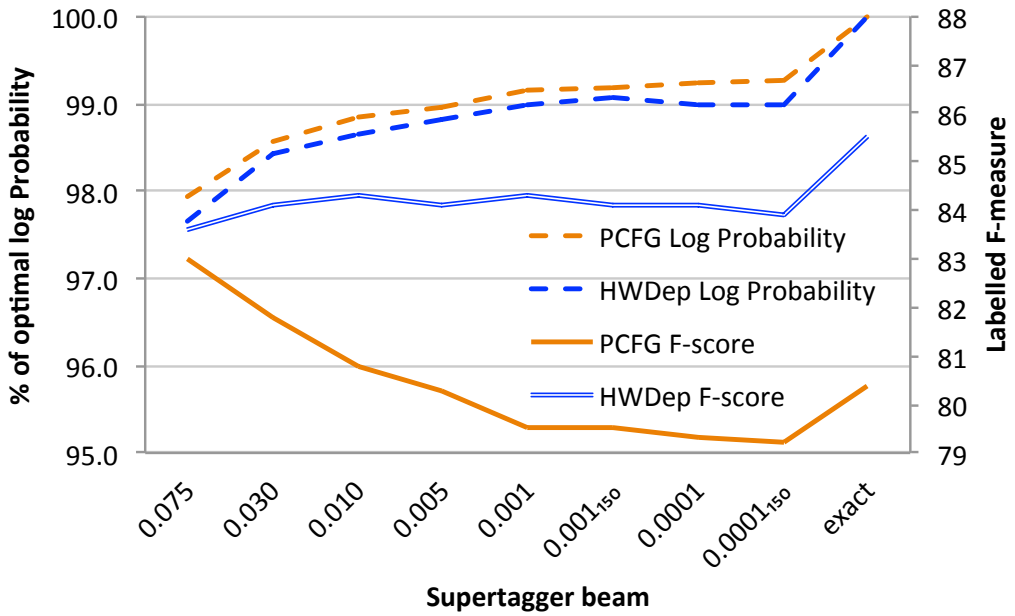


Figure 4.5: Log-probability of parses relative to exact solution vs. labelled F-score at each supertagging beam-level.

4.3 A* Parsing Experiments

To compare approaches, we extended our baseline parser to support A* search. Following Klein and Manning (2003) we restrict our experiments to sentences on which we can perform exact search via using the same subset of section 00 as in §4.2.2. Before considering CPU time, we first evaluate the amount of work done by the parser using three hardware-independent metrics. We measure the number of *items pushed* (Pauls and Klein, 2009a) and *items popped*, corresponding to the insert/decrease-key operations and the remove operation of a priority queue, respectively. Finally, we measure the number of *traversals*, which counts the number of item probabilities computed, regardless of whether the probability is discarded due to the prior existence of a better probability.

We experiment with both `PCFG` and `HWDep`. Since `HWDep` is lexicalised it was not feasible to experiment with pre-computed heuristics for this grammar. We therefore considered different A* variants for each model: for `PCFG` we used A* with a simple precomputed heuristic (§4.1.1), and for the more complex `HWDep` we used the hierarchical A* algorithm discussed earlier (§4.1.2) with two simple grammar projections.

for efficient search algorithms.

4.3.1 Hardware-Independent Results: PCFG

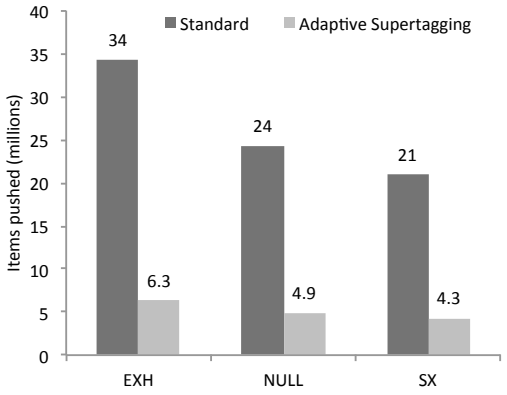
For the PCFG model, we compared three agenda-based parsers: EXH prioritises items by their span length, thereby simulating the exhaustive CKY algorithm; NULL prioritises items by their Viterbi inside probability only; and SX is an A* parser that prioritises items by their Viterbi inside probability times an admissible Viterbi outside probability estimate (§4.1.2).[11] The parsers are tested with and without adaptive supertagging where the former can be seen as performing exact search (via A*) over the pruned search space created by AST.

Figure 4.6 shows that A* with the SX heuristic decreases the number of items pushed by up to 39% on the unpruned search space. Although encouraging, this is not as impressive as the 95% speedup obtained by Klein and Manning (2003) with the same heuristic on their CFG. On the other hand, the NULL heuristic works better for CCG than for CFG, with a speedup of 29% for CCG and 11% for CFG (Klein and Manning, 2003). These results carry over to the AST setting which shows that A* can improve search even on the highly pruned search space. Note that A* only saves work in the final iteration of AST, since for earlier iterations it must process the entire agenda to determine that there is no spanning analysis.

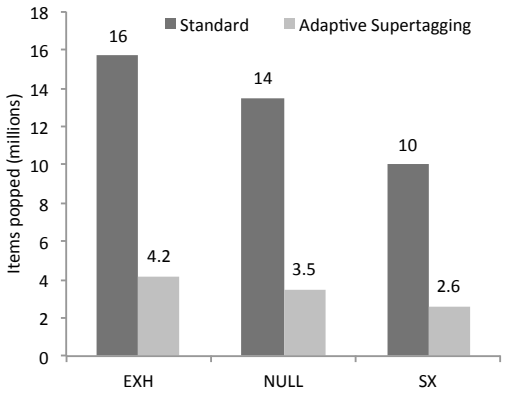
Since there are many more categories in the CCG grammar than for a standard Penn Treebank grammar we might have expected the SX heuristic to work better for CCG than for a CFG. Why doesn't it? We can measure how well a heuristic bounds the true Viterbi outside probability in terms of *slack*: the difference between the true and estimated outside probability. Lower slack means that the heuristic bounds the true cost better and guides us to the exact solution more quickly. Figure 4.7 plots the average slack for the SX heuristic against the number of words in the outside context. Comparing this with an analysis of the same heuristic when applied to a CFG by Klein and Manning (2003), we find that SX is less effective in our setting.[12] Note the near zero slack for contexts of size one: The main reason for the minimal slack is that a single word in the context is in many cases the full stop at the end of a sentence, a very predictable parsing step in a nearly finished sentence. However for larger context sizes the slack becomes very large: The flexibility of CCG to analyse sentences in many

[11] The NULL parser is a special case of A*, also called uniform cost search, which in the case of parsing corresponds to Knuth's algorithm (Knuth, 1977; Klein and Manning, 2001), the extension of Dijkstra's algorithm to hypergraphs.

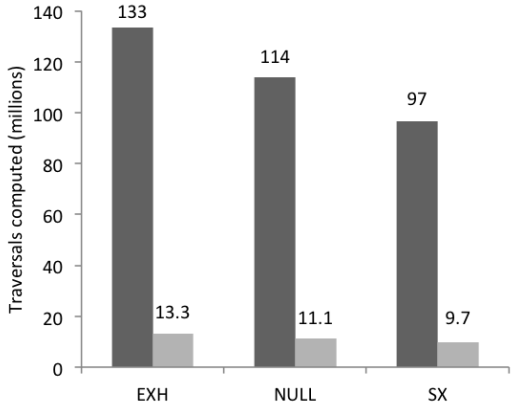
[12] Specifically, we refer to Figure 9 of their paper which uses a slightly different representation of estimate sharpness.



(a) Items pushed



(b) Items popped



(c) Traversals Computed

Figure 4.6: Exhaustive search (EXH) or simulated CKY, A* with no heuristic (NULL), and A* with the SX heuristic in terms of millions of items pushed, items popped and item traversals computed using the PCFG grammar with and without adaptive supertagging.

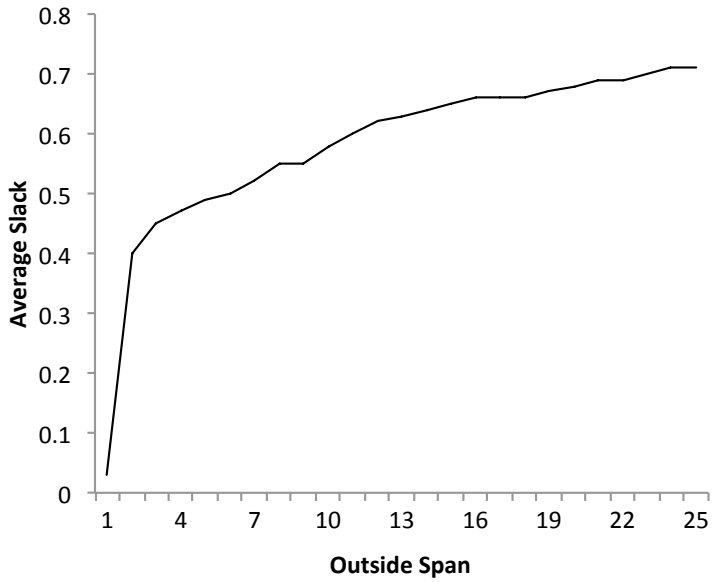


Figure 4.7: Average slack of the SX heuristic. The figure shows the relative difference between the estimated Viterbi outside probability and the true Viterbi outside probability across the number of words in the context.

different ways means that the outside estimate for a non-terminal can be based on a great number of different outside derivations, which does not bound the true probability well.

4.3.2 Hardware-Independent Results: HWDep

Lexicalisation in the HWDep model makes the precomputed SX estimate impractical, we therefore experiment with two grammar projections using the hierarchical A* (HA*) algorithm described earlier (§4.1.2). The grammar projections are constructed by defining simplified structural probabilities for HWDep (Table 3.3) using Equation 4.1. We consider two projections: First, PCFGProj projects the grammar to a PCFG, completely removing lexicalisation, i.e., head words and lexical categories (Table 4.5). The resulting structural probabilities $p'(\ldots)$ are nearly identical to PCFG. Second, LexcatProj removes only head words but retains lexical categories (Table 4.6). In the experiments we use HWDep as the true grammar and each projection individually, i.e., we do not chain the two projections together in a hierarchy.

Figure 4.8 compares exhaustive search (EXH), A* with no heuristic (NULL), and

Expansion probability	$p'(exp\|P) = \min_{c_\mathrm{P}, w_\mathrm{P}} p(exp\|\mathrm{P}, c_\mathrm{P}\#w_\mathrm{P})$
Head probability	$p'(\mathrm{H}\|\mathrm{P}, exp) = \min_{c_\mathrm{P}, w_\mathrm{P}} p(\mathrm{H}\|\mathrm{P}, exp, c_\mathrm{P}\#w_\mathrm{P})$
Non-head probability	$p'(\mathrm{S}\|P, exp, \mathrm{H}) = \min_{c_\mathrm{P}, w_\mathrm{P}} p(\mathrm{S}\|\mathrm{P}, exp, \mathrm{H}\#c_\mathrm{P}\#w_\mathrm{P})$
Lexcat probability	$p'(c_\mathrm{S} = \hat{c_\mathrm{S}}\|\mathrm{S}\#\mathrm{H}, exp, \mathrm{P}) = \min_{c_\mathrm{S}} p(c_\mathrm{S}\|\mathrm{S}\#\mathrm{H}, exp, \mathrm{P})$
Head word probability	$p'(w_\mathrm{S} = \hat{w}_\mathrm{S}\|c_\mathrm{S} = \hat{c}_\mathrm{S}\#\mathrm{P}, \mathrm{H}, \mathrm{S}, w_\mathrm{P} = \hat{w}_\mathrm{P}) = \min_{c_\mathrm{S}, w_\mathrm{S}, w_\mathrm{P}} p(w_\mathrm{S}\|c_\mathrm{S}\#\mathrm{P}, \mathrm{H}, \mathrm{S}, w_\mathrm{P})$

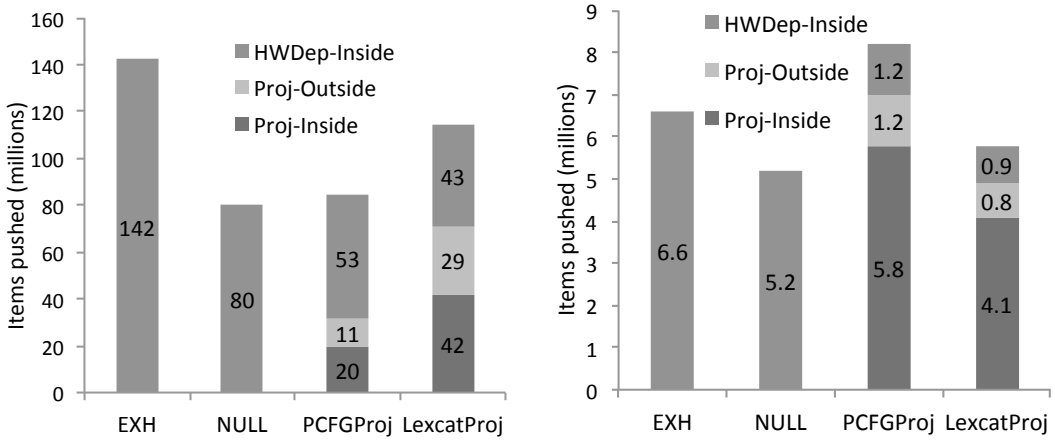
Table 4.5: Factorisation of the `PCFGProj` grammar projection which simplifies the structural probabilities of the `HWDep` grammar given in Table 3.3.

Expansion probability	$p'(exp\|\mathrm{P}, c_\mathrm{P}) = \min_{w_\mathrm{P}} p(exp\|\mathrm{P}, c_\mathrm{P}\#w_\mathrm{P})$
Head probability	$p'(\mathrm{H}\|\mathrm{P}, exp, c_\mathrm{P}) = \min_{w_\mathrm{P}} p(\mathrm{H}\|\mathrm{P}, exp, c_\mathrm{P}\#w_\mathrm{P})$
Non-head probability	$p'(\mathrm{S}\|\mathrm{P}, exp, \mathrm{H}\#c_\mathrm{P}) = \min_{w_\mathrm{P}} p(\mathrm{S}\|\mathrm{P}, exp, \mathrm{H}\#c_\mathrm{P}\#w_\mathrm{P})$
Lexcat probability	$p'(c_\mathrm{S}\|\mathrm{S}\#\mathrm{H}, exp, \mathrm{P}) = p(c_\mathrm{S}\|\mathrm{S}\#\mathrm{H}, exp, \mathrm{P})$
Head word probability	$p'(w_\mathrm{S} = \hat{w}_\mathrm{S}\|c_\mathrm{S}\#\mathrm{P}, \mathrm{H}, \mathrm{S}, w_\mathrm{P} = \hat{w}_\mathrm{P}) = \min_{w_\mathrm{S}, w_\mathrm{P}} p(w_\mathrm{S}\|c_\mathrm{S}\#\mathrm{P}, \mathrm{H}, \mathrm{S}, w_\mathrm{P})$

Table 4.6: Factorisation of the `LexcatProj` grammar projection which simplifies the structural probabilities of the `HWDep` grammar given in Table 3.3.

HA* in the two variants. For HA* (columns PCFGProj and LexcatProj), parsing effort is broken down into the different item types computed at each stage: We distinguish between the work carried out to compute the heuristic estimates, i.e., the inside and outside items of the projection (Proj-Inside, Proj-Outside), and the work to compute the inside items of the true grammar (HWDep-Inside). A* `NULL` applied to `HWDep` saves about 44% of items pushed, which makes it slightly more effective than for the `PCFG` model. However, the effort to compute the grammar projections outweighs their benefit. We suspect that this is due to the large difference between the target grammar and the projection: The PCFG projection is a simple grammar and so we improve the probability of an item less often than in the target grammar.

The Lexcat projection performance is worst, we believe this is due to two reasons. First, the projection requires about as much work to compute as the target grammar without a heuristic (`NULL`). Second, the projection itself does not save a large amount of work as can be seen in the statistics for the target grammar.



(a) Standard Search (b) Adaptive Supertagging

Figure 4.8: Comparison between a CKY simulation (EXH), A* with no heuristic (NULL), hierarchical A* (HA*) using two grammar projections for (a) standard search and (b) AST. The graphs show a breakdown of the work carried out in computing various items.

4.4 CPU Timing Experiments

Hardware-independent metrics are useful for understanding agenda-based algorithms, but what we actually care about is CPU time. We are not aware of any past work that measures A* parsers in terms of CPU time, but as actual running time is the real objective, experiments of this type are important. The savings in items processed by an agenda parser come at a cost: operations on the priority queue data structure can add significant runtime.

Timing experiments of this type are very implementation-dependent, so we took care to implement the algorithms as cleanly as possible and to reuse as much of the existing parser code, written in Java, as we could. An important implementation decision for agenda-based algorithms is the data structure used to implement the priority queue. Preliminary experiments showed that a Fibonacci heap implementation outperformed several alternatives: Brodal queues (Brodal, 1996), binary heaps, binomial heaps, and pairing heaps.[13]

We carried out timing experiments on the best A* parsers for each model (SX and NULL for PCFG and HWDep, respectively), comparing them with our CKY implementation (CKY) and the agenda-based CKY simulation (EXH); we tested on the same portion of section 00 as in §4.2.2. Details on the setup for the timing experiments are in §3.2.2.

[13]We used the Fibonacci heap implementation at http://www.jgrapht.org

	Standard		AST	
	PCFG	HWDep	PCFG	HWDep
CKY	536	24489	34	143
EXH	1251	26889	41	155
A* NULL	1032	21830	36	121
A* SX	889	-	34	-

Table 4.7: Parsing time in seconds of CKY and agenda-based parsers with and without adaptive supertagging. There are no timing results for HWDep with SX because the heuristic is too difficult to compute for lexicalised grammars, we also omitted the HA* parsers because they were outperformed by the simple NULL estimate (§4.3.2).

	Standard		AST	
	PCFG	HWDep	PCFG	HWDep
CKY	80.4	85.5	81.7	83.8
EXH	79.4	85.5	80.3	83.8
A* NULL	79.6	85.5	80.7	83.8
A* SX	79.4	-	80.4	-

Table 4.8: Labelled F-score of exact CKY and agenda-based parsers with and without adaptive supertagging. Within columns all parses have the same probability, thus variances in accuracy are due to implementation-dependent differences in tie-breaking.

Table 4.7 presents the cumulative running times with and without adaptive supertagging, while Table 4.8 reports F-scores.

The results (Table 4.7) are striking. Although the timing results of the agenda-based parsers track the hardware-independent metrics, they start at a significant disadvantage to exhaustive CKY with a simple control loop. This is most evident when looking at the timing results for EXH, which in the case of the full PCFG model requires more than twice the time than the CKY algorithm that it simulates. A* makes modest CPU-time improvements in parsing the full space of the HWDep model. Although this decreases the time required to obtain the highest accuracy, it is still a substantial tradeoff in speed compared with AST.

On the other hand for AST, the tradeoff between speed and accuracy improves

significantly: by combining AST with A* we observe a decrease in running time of 15% for the A* NULL parser of the HWDep model over CKY.[14] As the CKY baseline with AST is very strong, this result shows that A* holds real promise for CCG parsing.

4.5 Conclusions

This chapter has shown that adaptive supertagging is a strong technique for efficient CCG parsing. Our analysis confirms tremendous speedups, and shows that for weak models, it can even result in improved accuracy. However, for better models, the efficiency gains of adaptive supertagging come at the cost of accuracy. One way to look at this is that the supertagger has good precision with respect to the parser's search space, but low recall.

To our knowledge, we are the first to measure A* parsing speed both in terms of running time and commonly used hardware-independent metrics. It is clear from our results that the gains from A* do not come as easily for CCG as for CFG, and that agenda-based algorithms like A* must make very large reductions in the number of items processed to result in real-time savings, due to the added expense of keeping a priority queue. However, we have shown that A* can yield real improvements even over the highly optimised technique of adaptive supertagging: in this pruned search space, a 44% reduction in the number of items pushed results in a 15% speedup in CPU time.

In Chapter 5 we will exploit the observations in this chapter by combining parsing and supertagging models in a principled way, making the supertagger impose a soft constraint on the parser rather than a hard constraint.

[14] Our A* parsers always return the highest-probability solution despite our experiments showing large fluctuations in accuracy (Table 4.8). These differences are due to unfortunate tie-breaking since the solutions have the same probability.

Chapter 5

Integrated Supertagging and Parsing

In Chapter 4 we analysed the most successful approach to CCG parsing which is based on a pipeline strategy of first supertagging and then parsing (§5.1). Variations on this approach drive the widely-used, broad coverage C&C parser (Clark and Curran, 2004b, 2007; Kummerfeld et al., 2010). Our analysis explored the tradeoff between the efficiency and accuracy of this approach and concluded that, while substantially increasing efficiency, this method results in lower accuracy for the parser of Hockenmaier (2003b).

In this chapter we show experimentally that this pipeline approach also significantly lowers the upper bound on parsing accuracy (§5.2). The same experiment shows that the supertagger prunes many bad parses. So, while we want to avoid the error propagation inherent to a pipeline, ideally we still want to benefit from the key insight of supertagging: that a sequence model over lexical categories can be quite accurate. Our solution is to combine the features of both the supertagger and the parser into a single, less aggressively pruned model. The challenge with this model is its prohibitive complexity, which we address with both *exact* and *approximate* methods: dual decomposition and belief propagation (§5.3). We present the first side-by-side comparison of these algorithms on an NLP task of this complexity, measuring accuracy, convergence behavior, and runtime. In both cases our model significantly outperforms the pipeline approach, leading to the best reported results in CCG parsing (§5.4).

Our approach to avoid the pipeline problem (Felzenszwalb and McAllester, 2007) is very much in line with much recent work in NLP. Tasks such as machine translation (Dyer et al., 2008; Dyer and Resnik, 2010; Mi et al., 2008), part-of-speech tagging (Jiang et al., 2008), named entity recognition (Finkel and Manning, 2009), and semantic role labelling (Sutton and McCallum, 2005a; Finkel et al., 2006) have been im-

proved by combined models. We focus on dual decomposition and belief propagation, which have recently received much attention in the computational linguistics community. Other work has tackled complex models using integer liner programming (Riedel and Clarke, 2006) for inference, or Monte Carlo Markov Chain methods (Finkel et al., 2006) for estimation, both of which have been successfully applied to complex NLP models.

5.1 Supertagging and Parsing in a Pipeline

Supertagging (Bangalore and Joshi, 1999; Clark, 2002; Curran et al., 2006) is often used to prune the parser's search space: the parser only considers lexical categories with high posterior probability (or other figure of merit) under the supertagging model (Clark and Curran, 2004b). The posterior probabilities are then discarded and it is the extensive pruning of lexical categories that leads to substantially faster parsing times. Pruning the categories in advance this way has a specific failure mode: sometimes it is not possible to produce a sentence-spanning derivation from the tag sequences preferred by the supertagger, since the supertagger does not enforce grammaticality.

A workaround for this problem is *adaptive supertagging* (AST; §2.2.2) whose accuracy and efficiency tradeoff we explored in Chapter 4. Recall that AST is based on a step function over supertagger beam widths, relaxing the pruning threshold for lexical categories only if the parser fails to find an analysis. The process either succeeds and returns a parse after some iteration or gives up after a predefined number of iterations. While efficient, the technique is inherently approximate: it will return a lower probability parse under the parsing model if a higher probability parse can only be constructed from a supertag sequence returned by a subsequent iteration. In this way it prioritises speed over exactness, although the tradeoff can be modified by adjusting the beam step function. Regardless, the technique remains approximate.

We will also explore *reverse adaptive supertagging*, a much less aggressive pruning method that seeks only to make sentences parse-able when they otherwise would not be due to an impractically large search space. Reverse AST starts with a wide beam, narrowing it at each iteration only if a maximum chart size is exceeded. In this way it prioritises exactness over speed.

Condition	Parameter	Iteration 1	2	3	4	5
AST	β (beam width)	0.075	0.03	0.01	0.005	0.001
	k (dictionary cutoff)	20	20	20	20	150
Reverse	β	0.001	0.005	0.01	0.03	0.075
	k	150	20	20	20	20

Table 5.1: Beam step function used during test time for standard (AST) and less aggressive (Reverse) AST throughout our experiments. Parameter β is a beam threshold while k bounds the use of a part-of-speech tagging dictionary, which is used for words seen less than k times (see §3.3.1 for an explanation of tagging dictionaries).

5.2 Oracle Parsing

What is the effect of these approximations? To answer this question we extend our analysis of the tradeoff between practical efficiency and accuracy in §4.2.2. Specifically, we compute oracle best and worst values for labelled dependency F-score. The oracle is simply the best or worst possible value obtainable by a given model under certain settings and therefore presents an upper or lower bound on accuracy. In order to compute the oracle we use the algorithm of Huang (2008; §5.2.1) on the hybrid model of Clark and Curran (2007), the best model of their C&C parser (§3.3.3). We computed the oracle (§5.2.2) on our development data, using both AST and Reverse AST beams settings shown in Table 5.1.

5.2.1 Algorithm

Our CCG parser is evaluated on labelled, directed dependency recovery using F-measure (§3.2.1). Suppose we want to compute the oracle F-score (i.e. best or worst possible) on a combinatorial space of outcomes represented by a dynamic program. Recall that under this evaluation method we represent the output y' and ground truth y as variable-sized sets of predicate-argument relations or *dependencies*. Recall that we can compute *precision* P as $\frac{|y \cap y'|}{|y'|}$ and *recall* R as $\frac{|y \cap y'|}{|y'|}$. F-measure is then given by:

$$F_1(y, y') = \frac{2PR}{P+R} = \frac{2|y \cap y'|}{|y| + |y'|} \tag{5.1}$$

For a given test problem, y is fixed, and F_1 is a function of two integers, $|y \cap y'|$ and

$|y'|$, which we will call n and d, respectively.[1] Note that we will use the terms *item* and *state* from now on interchangeably.

The key idea is to use a dynamic program which treats F_1 as a *non-local feature* of the parse, dependent on values n and d. This requires the splitting of each item in an otherwise usual CKY computation (§2.3.4) by all pairs $\langle n, d \rangle$ incident at that item. We will use the term *state-splitting* to refer to splitting an item $A_{i,j}$ into many items $A_{i,j,n,d}$, one for each $\langle n, d \rangle$ pair. The functions $n_+(\cdot)$ and $d_+(\cdot)$ represent the number of correct and total dependencies introduced by a parsing action. The algorithm to compute these state-split states is then given below:

$$A_{i,i+1,n,d} = \text{TRUE} \quad \text{iff } n = n_+(a_{i+1} \Rightarrow A),\ d = d_+(a_{i+1} \Rightarrow A)$$

$$A_{i,j,n,d} = \bigoplus_{k,B,C} \bigoplus_{\substack{\{n',n'':n'+n''+n_+(BC\Rightarrow A)=n\},\\ \{d',d'':d'+d''+d_+(BC\Rightarrow A)=d\}}} B_{i,k,n',d'} \otimes C_{k,j,n'',d''} \otimes n_+(BC \Rightarrow A)$$

$$\hat{F}_1 = \max_{n,d} S_{0,L,n,d} \left(1 - \frac{2n}{d + |y|}\right)$$

where we use the boolean semiring (Table 2.1). The first two recursions build all possible items permitted by the grammar and record their n and d values. The final equation computes the maximum $\hat{F}_1$ based on sentence-spanning items $S_{0,L,n,d}$.

However, F_1 is strictly increasing in n, since we can never obtain a higher F_1 by choosing a lower n for some d. This observation allows us to compute the oracle $\hat{F}_1$ as a maximisation over defined values of d, i.e.

$$\hat{F}_1 = \max_d \left(\max_n \frac{2n}{d + |y|} \right) \tag{5.2}$$

It is therefore sufficient to keep only the maximum value of n encountered for each d. This allows us to use a simplified dynamic program, which computes items $N(A_{i,j,d})$ maintaining the maximum value of n encountered for each d. The final oracle algorithm is then given by the following recursions:

$$N(A_{i,i+1,d}) = n_+(a_{i+1} \Rightarrow A) \quad \text{iff } d = d_+(a_{i+1} \Rightarrow A)$$

$$N(A_{i,j,d}) = \bigoplus_{k,B,C} \bigoplus_{\{d',d'':d'+d''+d_+(BC\Rightarrow A)=d\}} N(B_{i,k,d'}) \otimes N(C_{k,j,d''}) \otimes n_+(BC \Rightarrow A)$$

$$\hat{F}_1 = \max_d \left(1 - \frac{2N(S_{0,L,d})}{d + |y|}\right)$$

[1] For *numerator* and *denominator*.

	Viterbi F-score			Oracle Max F-score			Oracle Min F-score			
	LF	LP	LR	LF	LP	LR	LF	LP	LR	cats
AST	**87.38**	**87.83**	86.93	94.35	95.24	93.49	54.31	54.81	53.83	1.3-3.6
Reverse	87.36	87.55	**87.17**	**97.65**	**98.21**	**97.09**	18.09	17.75	18.43	3.6-1.3

Table 5.2: Comparison of adaptive supertagging (AST) and a less restrictive setting (Reverse) with Viterbi and oracle F-scores on CCGbank Section 00; we optimise for the highest and lowest possible F-score (Oracle Max/Min) under each setting. The table shows the labelled F-score (LF), precision (LP) and recall (LR) as well as the number of lexical categories per word used (from first to last parsing attempt).

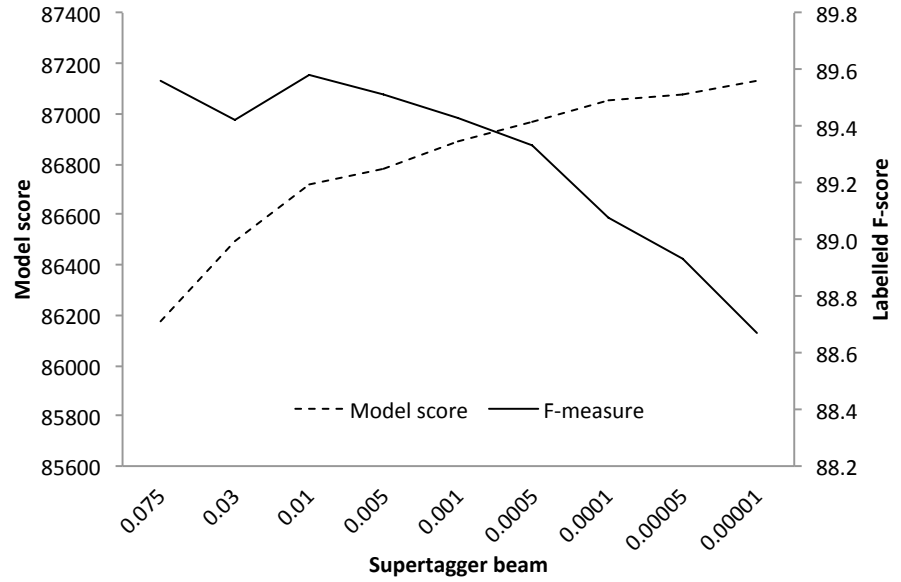
The recursions use the semiring $\langle \mathbb{N}_0^\infty, max, +, 0, 0 \rangle$ which operates over the positive integers. The additive operator $\oplus$ is max and the multiplicative operator $\otimes$ becomes $+$.

5.2.2 Results and Discussion

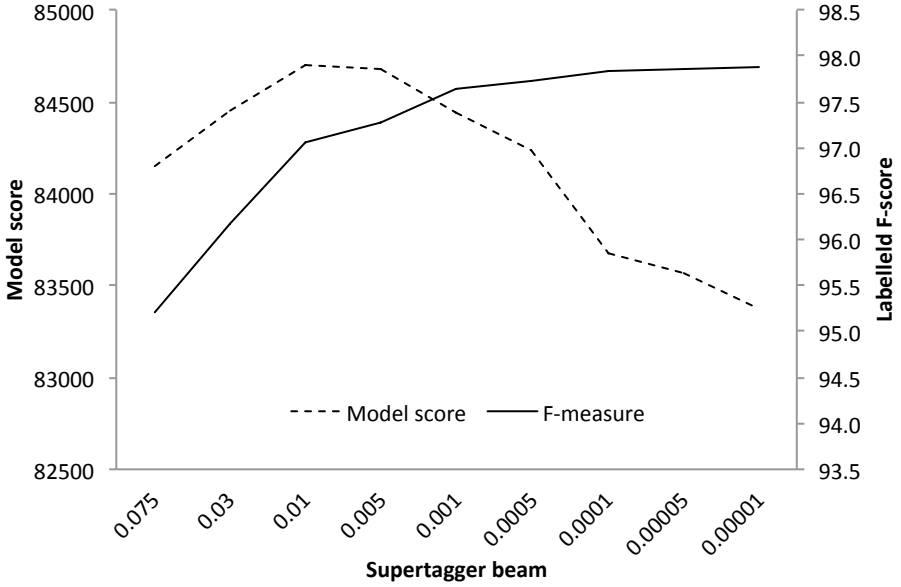
The results (Table 5.2) show that the oracle best accuracy for reverse AST is more than 3% higher than the aggressive AST pruning. In fact, it is almost as high as the upper bound oracle accuracy of 97.73% obtained using *perfect* supertags—in other words, the search space for reverse AST is theoretically near-optimal.[2] We also observe that the oracle worst accuracy is much lower in the reverse setting. It is clear that the supertagger pipeline has two effects: while it beneficially prunes many bad parses, it harmfully prunes some very good parses. We can also see from the scores of the Viterbi parses that while the reverse condition has access to much better parses, the model does not actually find them. This mirrors the result of Clark and Curran (2007) that they use to justify AST.

Digging deeper, we compared the model score of the Viterbi solutions returned by the parser against their F-score (Figure 5.1(a)) at a variety of fixed beam settings. We did a similar experiment for the parses returned by the oracle algorithm (Figure 5.1(b)) and considered for both experiments only the subset of our development set which could be parsed with *all* beam settings. Why is the parser unable to find better solu-

[2]This idealised oracle reproduces a result from Clark and Curran (2004a). The reason that using the gold-standard supertags does not result in 100% oracle parsing accuracy is that some of the development set parses cannot be constructed by the learned grammar.



(a) Viterbi



(b) Oracle

Figure 5.1: Comparison between model score and Viterbi F-score (a); and between model score and oracle F-score (b) for different supertagger beams on a subset of CCGbank Section 00. The model score in (b) decreases because the parsing model does not necessarily assign higher probability to solutions with higher F-score.

tions with decreasing beam settings despite having access to much better parses (Figure 5.1(a))? First, the parser was trained with supertags with beam setting $\beta = 0.001$ (see §3.4) explaining its poor performance beyond that. Second, it is common for parsing models to perform better when pruned by a high-precision constraint such as a supertagger (Roark and Hollingshead, 2009). The inverse relationship between model score and F-score shows that the supertagger restricts the parser to mostly good parses (under F-measure) that the model would otherwise disprefer. Exactly this effect is exploited in the pipeline model. However, when the supertagger makes a mistake, the parser cannot recover.

5.3 Combination Methods

The supertagger obviously has good but not perfect predictive features. An obvious way to exploit the supertagger features without being bound by its decisions is to incorporate these features directly into the parsing model. In our case both the parser and the supertagger are feature-based models, so from the perspective of a single parse tree, the change is simple: the tree is simply scored by the weights corresponding to all of its active features. However, since the features of the supertagger are all Markov features on adjacent supertags, the change has serious implications for search. If we think of the supertagger as defining a weighted regular language consisting of all supertag sequences, and the parser as defining a weighted mildly context-sensitive language consisting of only a *subset* of these sequences, then the search problem is equivalent to finding the optimal derivation in the weighted intersection of a regular and mildly context-sensitive language. Even allowing for the observation of Fowler and Penn (2010) that our practical CCG is context-free, this problem still reduces to the construction of Bar-Hillel et al. (1964), making search very expensive.

This intersection corresponds to a new grammar that introduces sensitivity to the Markov features of the supertagger. Roughly speaking, with a first-order tagging model, rules such as

$$NP \;\; S\backslash NP \Rightarrow S \tag{5.3}$$

are replaced with new rules

$$_{NP}NP_{NP} \;\; _{(S\backslash NP)/NP}\, S\backslash NP \,_{NP} \Rightarrow S \tag{5.4}$$

where each category is replaced with a category that tracks the first and last supertag of the sentence fragment spanning this category. For example, $_{(S\backslash NP)/NP}\, S\backslash NP \,_{NP}$ rep-

resents a verb phrase that begins with a transitive verb $(S\backslash NP)/NP$ and ends with an NP. The weights of these grammar rules include the weights of the original parser grammar as well as the weights from the tagger. We can then use a dynamic programming algorithm such as CKY (§2.3.4) to find the highest scoring structure according to both the parsing and tagging models. This is guaranteed to give an exact solution to the combined problem but it is often very inefficient, as can be seen by looking at the items the dynamic program computes in each case: Parsing in the original scenario uses items $A_{i,j}$ of which there can be at most $O(GL^2)$ where G is the number of nonterminals and L is the sentence length.[3] In the intersected case we have items $A_{i,j,B,C}$ of which there can be $O(G^3L^3)$ which is substantially more complex, particularly for CCG where G is very large. Therefore we need approximations.

Fortunately, recent literature has introduced two relevant approximations to the NLP community: loopy belief propagation (Pearl, 1988), applied to dependency parsing by Smith and Eisner (2008); and dual decomposition (Dantzig and Wolfe, 1960; Komodakis et al., 2007; Sontag et al., 2010, *inter alia*), applied to dependency parsing by Koo et al. (2010), lexicalised CFG parsing by Rush et al. (2010), event extraction (Riedel and McCallum, 2011), word-alignment for machine translation (DeNero and Macherey, 2011), and machine translation itself (Rush and Collins, 2011b; Chang and Collins, 2011). These algorithms are based on the exact dynamic programming algorithms we presented for parsing (§2.3.4) and tagging (§2.3.5) but avoid the expensive construction of Bar-Hillel. We apply both techniques to our integrated supertagging and parsing model.

5.3.1 Loopy Belief Propagation

Belief propagation (BP; §2.3.6) is an algorithm for computing marginals (i.e. expectations) on structured models. These marginals can be used for decoding (parsing) in a minimum-risk framework (Smith and Eisner, 2008; see §2.3.3); or for training using a variety of algorithms (Sutton and McCallum, 2011). We experiment with both uses in §5.4. Many researchers in NLP are familiar with two special cases of belief propagation: the forward-backward (§2.3.5) and inside-outside (§2.3.4) algorithms, used for computing expectations in sequence models and context-free grammars, respectively.[4]

[3] For simplicity we assume an unlexicalised grammar. In the lexicalised case, there would be at most $O(GL^3)$ items since every item can be lexicalised by one of the L words.

[4] Forward-backward and inside-outside are formally shown to be special cases of belief propagation by Smyth et al. (1997) and Sato (2007), respectively.

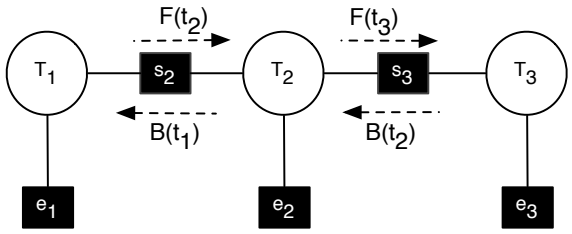


Figure 5.2: Supertagging factor graph with messages. Circles are variables and filled squares are factors. We omit the message indices for brevity i.e. $F(t_i)$ instead of $F_i(t_i)$.

Our use of belief propagation builds directly on these two familiar algorithms.

The supertagger defines a distribution over tags $T_1 ... T_L$ with values $t_1 ... t_L$, based on emission factors $e_1 ... e_L$ and transition factors $s_2 ... s_L$ (Figure 5.2). The message F_i a variable T_i receives from its neighbour to the left corresponds to the forward probability, while messages from the right correspond to backward probability B_i (§2.3.5).

$$F_i(t_i) = \sum_{t_{i-1}} F_{i-1}(t_{i-1}) s_i(t_{i-1}, t_i) e_{i-1}(t_{i-1}) \tag{5.5}$$

$$B_i(t_i) = \sum_{t_{i+1}} B_{i+1}(t_{i+1}) s_{i+1}(t_i, t_{i+1}) e_{i+1}(t_{i+1}) \tag{5.6}$$

Note how we reuse notation for forward and backward items introduced in §2.3.5 to represent messages. The quantities represented by forward and backward items are identical to the messages in a factor graph; we will also reuse parse item notation for messages in factor graphs for parse trees.

The current belief of a variable can be computed by taking the normalised product of all its incoming messages (Equation 2.44). In the supertagger model, this quantity is

$$p(t_i) = \frac{1}{Z} F_i(t_i) B_i(t_i) e_i(t_i) \tag{5.7}$$

where

$$Z = \sum_{t_i} F_i(t_i) B_i(t_i) e_i(t_i) \tag{5.8}$$

Note how the definition of the forward probability in Equation 5.5 does not include $e_i(t_i)$, this differs from the original definition in Equation 2.35. We made this change in order to make explicit that the emission probability is part of the belief of a tag.

Our parsing model is *also* a distribution over variables T_i, along with an additional quadratic number of $span(i, j)$ variables (see §2.3.6).[5] We add this complex distribu-

[5]The parsing model is also a distribution over the tag variables but it uses different features.

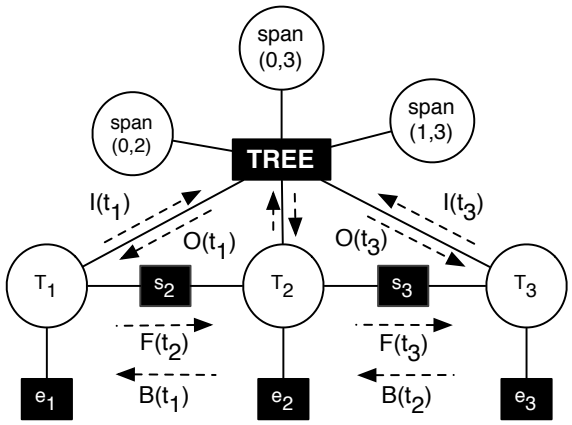


Figure 5.3: Factor graph for the combined parsing and supertagging model. Inside $I(A_{i,j})$ and outside messages $O(A_{i,j})$ for non-supertag span variables $span(i,j)$ are omitted for brevity.

tion to our model as a single factor (Figure 5.3). This is a natural extension to the use of complex factors described by Smith and Eisner (2008) and Dreyer and Eisner (2009).

When a factor graph is a tree as in Figure 5.2, BP converges in a single iteration to the exact marginals. However, when the model contains cycles, as in Figure 5.3, we can iterate message passing. In many cases *loopy* BP will converge to approximate marginals that are bounded under an interpretation from statistical physics (Yedidia et al., 2001; Sutton and McCallum, 2011). Furthermore, at convergence, the beliefs are not guaranteed to match the true marginals. However, in practice the beliefs are usually quite a close approximation to the real values (Ihler et al., 2005; Cohn, 2007).

The TREE factor receives *inside* messages $I(A_{i,j})$ and sends *outside* messages $O(A_{i,j})$ to the tag and span variables, taking into account beliefs from the sequence model. We will omit the unchanged outside recursion for brevity (given in §2.3.4), but inside messages $I(A_{i,j})$ for category $A_{i,j}$ in $span(i,j)$ are computed as follows:

$$I(A_{i,j}) = \begin{cases} w(a_{i+1} \Rightarrow A_{i,j}) \underbrace{F_i(A_{i,j}) B_i(A_{i,j}) e_i(A_{i,j})}_{\text{supertag messages}} & \text{if } j = i+1 \\ \sum_{k,B,C} I(B_{i,k}) I(C_{k,j}) w(BC \Rightarrow A) & i < k \leq j \end{cases} \tag{5.9}$$

Note that the only difference from the classic inside algorithm is that the recursive base case of a category spanning a single word has been modified by a message from the supertag that contains both forward and backward factors, along with a unary emis-

sion factor, which doubles as a unary rule factor. This difference is also mirrored in the forward and backward messages, which are identical to Equations 5.5 and 5.6, except that they also incorporate outside messages from the tree factor:

$$F_i(t_i) = \sum_{t_{i-1}} F_{i-1}(t_{i-1}) s_i(t_{i-1}, t_i) e_{i-1}(t_{i-1}) \underbrace{O(t_i)}_{\text{tree}} \tag{5.10}$$

$$B_i(t_i) = \sum_{t_{i+1}} B_{i+1}(t_{i+1}) s_{i+1}(t_i, t_{i+1}) e_{i+1}(t_{i+1}) \underbrace{O(t_{i+1})}_{\text{tree}} \tag{5.11}$$

Once all forward-backward and inside-outside probabilities have been calculated the belief of supertag t_i can be computed as the product of all incoming messages. The only difference from Equation 5.7 is the addition of the outside message.

$$p(t_i) = \frac{1}{Z} F_i(t_i) B_i(t_i) e_i(t_i) O(t_i) \tag{5.12}$$

The algorithm repeatedly runs forward-backward and inside-outside, passing their messages back and forth, until these quantities converge.

5.3.2 Dual Decomposition

Dual decomposition (Rush et al., 2010; Koo et al., 2010) is a decoding (i.e. search) algorithm for problems that can be decomposed into efficiently solvable subproblems together with linear constraints that enforce some notion of agreement between the subproblem solutions over a shared set of variables. A solution for the original complex problem can be extracted from the solutions of the subproblems. The subproblems are chosen so that they can be easily solved individually, for example, they may be tree-structured so that max-product belief propagation can efficiently find exact solutions (§2.3.6). The constraints are incorporated into the subproblems, and an iterative algorithm can recover solutions to the original problem. The ensuing description follows the excellent tutorial of Rush and Collins (2011a).

In our setting the subproblems are supertagging and parsing. Formally, assume we have L words in the input sentence, $\mathcal{Y}$ as the set of valid parses, $\mathcal{Z}$ as the set of valid supertag sequences, and $\mathcal{T}$ as the set of supertags. We define $y(i,t) = 1$ if parse tree y has tag t at position i such that $y \in \mathcal{Y}$, $t \in \mathcal{T}$, and $i \in \{1 \ldots L\}$; otherwise define $y(i,t) = 0$. Similarly, for any tag sequence z, we define $z(i,t) = 1$ if sequence z has tag t at position i, and $z(i,t) = 0$ otherwise. Figure 5.4 shows a parse and a supertagging sequence which both have $y(3,NP) = z(3,NP) = 1$.

$$\dfrac{\dfrac{\text{Mark}}{NP}\quad\dfrac{\dfrac{\text{proved}}{(S\backslash NP)/NP}\quad\dfrac{\text{completeness}}{NP}}{S\backslash NP}>}{S}<$$

(a)

$$\dfrac{\text{Mark}}{NP}\quad\dfrac{\text{proved}}{(S\backslash NP)/NP}\quad\dfrac{\text{completeness}}{NP}$$

(b)

Figure 5.4: Example CCG derivation and supertagging sequence.

Given this notation, we want to solve the following optimisation problem for parser $f(y)$ and supertagger $g(z)$:[6]

$$\begin{gathered}\arg\max_{y\in\mathcal{Y},z\in\mathcal{Z}} f(y)+g(z) \\ \text{such that } y(i,t)=z(i,t) \text{ for all } (i,t)\in I\end{gathered} \tag{5.13}$$

Here $I=\{(i,t): i\in 1\ldots L, t\in\mathcal{T}\}$ denotes the set of all possible supertags for each word. This corresponds to finding the best pair of y and z such that they share the same tags. Solving Equation 5.13 without the equality constraints $y(i,t)=z(i,t)$ is straightforward since we have efficient, exact algorithms for both tagging (§2.3.5) and parsing (§2.3.4); however, solving the combined problem with the constraints is as hard as the construction of Bar-Hillel et al. (1964) we illustrated in the example given in §5.3.

The first step to construct a dual decomposition algorithm is to formulate the *Lagrangian* for this problem. We introduce a Lagrangian multiplier $u(i,t)$ for each equality constraint $y(i,t)=z(i,t)$: The set of Lagrange multipliers is $u=\{u(i,t):(i,t)\in I\}$. The Lagrangian is defined as

$$L(u,y,z)=f(y)+g(z)+\sum_{i,t}u(i,t)\left(y(i,t)-z(i,t)\right) \tag{5.14}$$

which can be re-arranged as

$$L(u,y,z)=\left(f(y)+\sum_{i,t}u(i,t)y(i,t)\right)+\left(g(z)-\sum_{i,t}u(i,t)z(i,t)\right) \tag{5.15}$$

Using the Lagrangian we can define the *dual objective* as

$$\begin{aligned}L(u)&=\max_{y\in\mathcal{Y},z\in\mathcal{Z}}L(u,y,z)\\&=\max_{y\in\mathcal{Y}}\left(f(y)+\sum_{i,t}u(i,t)y(i,t)\right)+\max_{z\in\mathcal{Z}}\left(g(z)-\sum_{i,t}u(i,t)z(i,t)\right)\end{aligned} \tag{5.16}$$

[6] We will introduce weighting parameters for $f(y)$ and $g(z)$ in §5.4.

$u^{(0)}(i,t) = 0$ for all $i \in \{1 \ldots L\}, t \in \mathcal{T}$

for $q = 1$ to Q **do**

$y^{(q)} = \arg\max_{y \in \mathcal{Y}} \left(f(y) + \sum_{i,t} u^{(q-1)}(i,t) y(i,t) \right)$ [Parsing]

$z^{(q)} = \arg\max_{z \in \mathcal{Z}} \left(g(z) - \sum_{i,t} u^{(q-1)}(i,t) z(i,t) \right)$ [Tagging]

if $y^{(q)}(i,t) = z^{(q)}(i,t)$ for all i,t

return $(y^{(q)}, z^{(q)})$

else

$u^{(q+1)} = u^{(q)}(i,t) - \delta^{(q)} \left(y^{(q)}(i,t) - z^{(q)}(i,t) \right)$

Figure 5.5: Dual decomposition algorithm for integrated parsing and tagging.

The dual objective solves each of the subproblems individually, each of which can be solved efficiently by itself. The dual objective (Equation 5.16) presents an upper bound that we want to make as tight as possible by solving the dual problem $\min_u L(u)$. We optimise the values of the $u(i,t)$ variables using the same algorithm as Rush et al. (2010) for their tagging and parsing problem (Figure 5.5).[7] At each iteration the algorithm finds $y^{(q)}$ and $z^{(q)}$ efficiently, it uses a step size $\delta^{(q)}$ decaying over time steps $q = 1, 2, \ldots, Q$.[8] An advantage of dual decomposition is that, on convergence, it recovers *exact* solutions to the combined problem. The algorithm converges if the $y^{(q)}$ and $z^{(q)}$ agree on all supertag assignments i.e. $y^{(q)}(i,t) = z^{(q)}(i,t)$ for all i,t. In this case, the Lagrangian multipliers cancel out

$$\sum_{i,t} u(i,t) \left(y(i,t) - z(i,t) \right) = 0 \tag{5.17}$$

and therefore $L(u,y,z) = f(y) + g(z)$ which means that we have a solution to the original optimisation problem in Equation 5.13. However, if the algorithm does not converge or we stop early, an approximation must be returned: following Rush et al. (2010) we used the highest scoring output of the parsing sub-model over all iterations i.e. for some q we choose $y^{(q')}$ such that

$$q' = \arg\max_{q' \leq q} f(y^{(q')}) \tag{5.18}$$

[7] The u terms can be interpreted as the *messages* from factors to variables (Sontag et al., 2010) and the resulting message passing algorithms are similar to the max-product algorithm (§2.3.6).

[8] We describe the setting of this parameter in §5.4.

5.4 Experiments

We use the C&C parser (Clark and Curran, 2007) and its supertagger (Clark, 2002). The baseline is the hybrid dependency model we described in §3.3.3 and our integrated model extends the hybrid model by adding the supertagger features; we add the default features of the C&C supertagger described in §3.3.1. The parser relies solely on the supertagger for pruning, using CKY for search over the pruned space. Training requires repeated calculation of feature expectations over packed charts of derivations. All models were trained with the parser's L-BFGS trainer (§3.4) except in the LBP training experiments (§5.4.4) where we used SGD. Our settings are identical to previous work (Clark and Curran, 2007) apart from the larger training beams we described in §3.4.[9]

Model combination. We combine the parser and the supertagger over the search space defined by the set of supertags within the supertagger beam (see Table 5.1), Restricting the search space to the supertagger beam avoids having to perform inference over the prohibitively large set of parses spanned by all supertags, which is intractable on current hardware. Hence at each beam setting, the model operates over the same search space as the baseline; the difference is that we score parses with the integrated model instead.

Piecewise training. The piecewise estimator (Sutton and McCallum, 2009) is an efficient way to train large models by decomposing the models into pieces which are trained individually and combined at test time. We separate our model into is original parts, the supertagging and parsing sub-model and combine their weights at test-time using either dual decomposition or belief propagation. Piecewise training is particularly well suited for Conditional Random Fields (§2.3.1) since it avoids having to explicitly calculate the global partition function Z (Sutton and McCallum, 2005b), which is expensive. Instead, each sub-model is normalised individually. The piecewise approximation over r parts of the graph gives an upper-bound on the original partition function Z (Sutton and McCallum, 2009; Cohn, 2007):

$$logZ \leq \sum_{r} logZ_r \tag{5.19}$$

where each sub-graph has its own partition function Z_r; proof can be found in Sutton and McCallum (2009).

[9] The larger beam settings only marginally improve accuracy for parsing with `AST`. However, for `Reverse` the wider beam settings are more effective since it uses wider beam settings more often.

We will now describe the training objective we use in detail. Our integrated model is based on the hybrid dependency model (§3.3.3) where each sentence in the training data is paired with a predicate-argument structure. Formally, we are given training data $\mathcal{D} = \{\langle x^{(i)}, y^{(i)}\rangle\}_{i=1}^{m}$, where $x^{(i)} \in \mathcal{X}$ is drawn from a set of possible input sentences, and each parse $y^{(i)} \in \mathcal{Y}(x^{(i)})$, a predicate-argument structure, is drawn from a set of possible parses, function $l(y)$ returns the supertags z for parse y, and z is in the set of all supertag sequences $\mathcal{Z}(x)$ for sentence x. We want to find the parameters θ of our model, where each $\lambda_k \in \theta$ is the weight of an associated feature $h_k(x,y)$ and $\theta = \theta_a \cup \theta_b$ is a concatenation of M parsing weights θ_a and N supertagging weights θ_b. Function $f_a(x,y)$ maps inputs to the vector $h_1(x,y),\ldots,h_M(x,y)$, and $f_b(x,l(y))$ maps inputs to the vector $h_{M+1}(x,l(y)),\ldots,h_{M+N}(x,l(y))$.

The objective we optimise is the sum of the log-likelihoods of the parsing factor $\ell_a(\mathcal{D};\theta)$ and the supertagging factor $\ell_b(\mathcal{D};\theta)$ as well as a Gaussian prior $G(\theta)$ over all parameters:

$$\ell_{PW}(\mathcal{D};\theta) = \ell_a(\mathcal{D};\theta_a) + \ell_b(\mathcal{D};\theta_b) - G(\theta) \tag{5.20}$$

$$\begin{aligned}\ell_{PW}(\mathcal{D};\theta) = &\sum_{i=1}^{m}\left[\log\frac{\sum_{d\in\Delta(y^{(i)})}\exp\{\theta_a^\top f_a(x^{(i)},\langle d,y^{(i)}\rangle)\}}{\sum_{\langle d',y'\rangle\in\Omega(x^{(i)})}\exp\{\theta_a^\top f_a(x^{(i)},\langle d',y'\rangle)\}}\right] \\ &+\sum_{i=1}^{m}\left[\theta_b^\top f_b(x^{(i)},l(y^{(i)})) - \log\sum_{z\in\mathcal{Z}(x^{(i)})}\exp\{\theta_b^\top f_b(x^{(i)},z)\}\right] - \sum_{\lambda_k\in\theta}\frac{\lambda_k^2}{2\sigma^2}\end{aligned} \tag{5.21}$$

where we use Equation 3.11 to define ℓ_a, which sums over all derivations d of a predicate-argument structure y. We denote $\Delta(y)$ as the set of all derivations leading to dependency structure y, and $\Omega(x)$ is the set of all dependency structures and derivations for x. For the supertagger we use a normal conditional log-likelihood formulation ℓ_b (Equation 2.16).

We only use the supertagger features to compute the supertagger-likelihood ℓ_b, and parser features are only used for the parser likelihood ℓ_a. The gradients for the parser features are computed such as in the dependency model (Equation 3.13) and the gradients for the supertagger features λ_k, where $M+1 \leq k \leq N+M$, are given as:

$$\frac{\partial}{\partial\lambda_k} = \sum_{i=1}^{m}\left[h_k(x^{(i)},l(y^{(i)})) - \sum_{z\in\mathcal{Z}(x^{(i)})}\frac{\exp\{\theta_b^\top f_b(x^{(i)},z)\}}{\sum_{z'\in\mathcal{Z}(x^{(i)})}\exp\{\theta_b^\top f_b(x^{(i)},z')\}}h_k(x^{(i)},z)\right] - \frac{\lambda_k}{\sigma_k^2}$$

Loopy Belief Propagation. For belief propagation we defined convergence when the message values did not change by a factor of more than 10^{-5}. We introduce a model

weighting parameter $0 < \gamma < 1$ which expresses the relative weight of the messages sent by the parsing sub-model; the supertagger weight is then simply $1 - \gamma$. The weighting changes the messages computed by the supertagger (Equations 5.10 and 5.11) to

$$F_i(t_i) = \sum_{t_{i-1}} F_{i-1}(t_{i-1}) s_i(t_{i-1}, t_i) e_{i-1}(t_{i-1}) \left(\underbrace{O(t_i)}_{\text{tree}} \right)^{\gamma} \tag{5.22}$$

$$B_i(t_i) = \sum_{t_{i+1}} B_{i+1}(t_{i+1}) s_{i+1}(t_i, t_{i+1}) e_{i+1}(t_{i+1}) \left(\underbrace{O(t_{i+1})}_{\text{tree}} \right)^{\gamma} \tag{5.23}$$

and the messages computed by the tree factor to

$$I(A_{i,j}) = \begin{cases} w(a_i \Rightarrow A_{i,j}) \left(\underbrace{F_i(A_{i,j}) B_i(A_{i,j}) e_i(A_{i,j})}_{\text{supertag messages}} \right)^{(1-\gamma)} & \text{if } j = i+1 \\ \sum_{k,B,C} I(B_{i,k}) I(C_{k,j}) w(BC \Rightarrow A) & i < k \leq j \end{cases} \tag{5.24}$$

We tuned the γ parameter on the development set and found that $\gamma = 0.5$ performed best.

Dual Decomposition. For dual decomposition we set the step size $\delta^{(q)}$ used by the algorithm in Figure 5.5 as follows: We start with $\delta^{(0)} = 0.5$ and define

$$\delta^{(q)} = \delta^{(0)} \times 2^{-\mu^{(q)}} \tag{5.25}$$

where $\mu^{(q)}$ is the number of times the Lagrangian $L(u^{(q')}) > L(u^{(q'-1)})$ for $q' \leq q$. The step size halves every time the dual increases from one iteration to the next.

Similar to belief propagation, we use a model weighting parameter $0 < \gamma < 1$ which expresses the relative weight of the parsing sub-model; the supertagger weight is $1 - \gamma$. This changes the updates of the dual decomposition algorithm in Figure 5.5 to

$$y^{(k)} = \arg\max_{y \in \mathcal{Y}} \left(f(y) + \left[\sum_{i,t} u^{(q-1)}(i,t) y(i,t) \right]^{\gamma} \right) \qquad \text{[Parsing]} \tag{5.26}$$

$$z^{(k)} = \arg\max_{z \in \mathcal{Z}} \left(g(z) - \left[\sum_{i,t} u^{(q-1)}(i,t) z(i,t) \right]^{(1-\gamma)} \right) \qquad \text{[Tagging]} \tag{5.27}$$

We tuned the parameter on the development set and found that a weighting of $\gamma = 0.4$ performed best.

Decoding Algorithms. For decoding we run belief propagation (BP) and dual decomposition (DD) for many iterations and irrespective of convergence we use the following decoding algorithms: For BP we recover a minimum-risk dependency structure

from the current marginals using the maximum dependency recall algorithm outlined in §3.3.3. For DD, we find the highest-scoring derivation using the Viterbi algorithm and we return the dependency-structure to which the derivation leads; at the last iteration of DD, we return the highest-scoring dependency-structure seen over all iterations or the solution at convergence. Note that the hybrid dependency model is defined in terms of dependency-structures but it is still possible to recover individual derivations. In preliminary experiments, we found that the accuracy difference between Viterbi decoding and maximum dependency recall decoding is minimal.

Coverage. The coverage of a parser is the percentage of sentences for which an analysis could be returned for a given data set. The coverage for *all* parsers in this and the next chapter is 99.22% on the development set (section 00 of CCGbank) and 99.63% on the test set (section 23). This is because every parser is subject to the same grammar and beam restrictions (§3.3.3) and we do not change the search space.

5.4.1 Parsing Accuracy

We first measure the accuracy of the piecewise trained model which is then decoded using belief propagation (BP) and dual decomposition (DD). Figure 5.6 shows the evolving accuracy of the models on the development set. In line with our oracle experiment, these results demonstrate that we can coax more accurate parses from the larger search space provided by the reverse setting; the influence of the supertagger features allow us to exploit this advantage.

One behavior we observe in the graph is that the DD results tend to incrementally improve in accuracy while the BP results quickly stabilise, mirroring the result of Smith and Eisner (2008). Accuracy for DD improves because it continues to find higher scoring parses at each iteration, and hence the results change. However for BP, even if the marginals have not converged, the minimum risk solution turns out to be fairly stable across successive iterations.

We next compare the algorithms against the baseline on our test set (Table 5.3). We find that the early stability of BP's performance generalises to the test set as does DD's improvement over several iterations. More importantly, we find that the applying our combined model using either algorithm *consistently* outperforms the baseline after only a few iterations. Overall, we improve the labelled F-measure by 1.1% and unlabelled F-measure by nearly 0.7% over the baseline. To the best of our knowledge, the results obtained with BP and DD are the best reported results on this task using gold

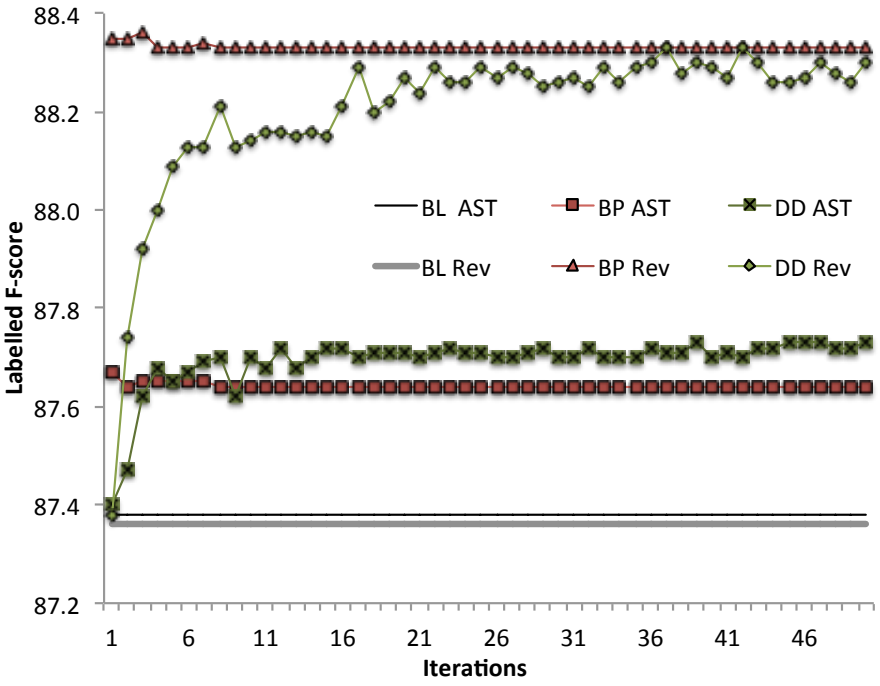


Figure 5.6: Labelled F-score of baseline (BL), belief propagation (BP), and dual decomposition (DD) on section 00, using either adaptive supertagging (AST) or reverse adaptive supertagging (Rev).

POS tags.

Next, we evaluate performance when using *automatic* part-of-speech tags as input to our parser and supertagger (Tables 5.4 and 5.5). This enables us to compare against the results of Fowler and Penn (2010), who trained the Petrov parser (Petrov et al., 2006) on CCGbank. We outperform them on all criteria. Hence our combined model represents the best CCG parsing results under any setting.

Finally, we revisit the oracle experiment of §5.2 using our combined models (Figure 5.7). Both show an improved relationship between model score and F-measure.

5.4.2 Algorithmic Convergence and Exactness

Figure 5.6 shows that parse accuracy converges after a few iterations. Do the *algorithms* converge? BP converges when the marginals do not change between iterations, and DD converges when both sub-models agree on all supertags. We measured the

	AST			Reverse		
	LF	UF	ST	LF	UF	ST
Baseline	87.73	93.09	94.33	87.65	93.06	94.01
C&C '07	87.64	93.00	94.32	-	-	-
$BP_{q=1}$	**88.25**	**93.33**	**94.60**	**88.86**	**93.75**	**94.84**
$BP_{q=25}$	88.22	93.32	94.58	88.84	93.75	94.81
$DD_{q=1}$	87.74	93.10	94.33	87.67	93.07	94.02
$DD_{q=25}$	88.14	93.24	94.59	88.80	93.68	94.82

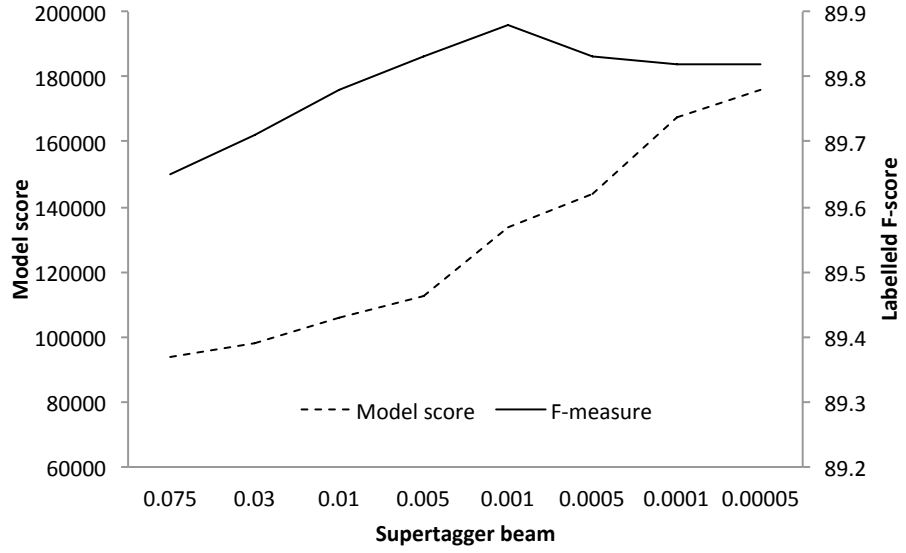
Table 5.3: Results on test set (section 23) for decoding the integrated model using dual decomposition (DD) or belief propagation (BP) for q iterations, evaluated by labelled and unlabelled F-score (LF/UF) and supertag accuracy (ST). We compare against the previous best result of Clark and Curran (2007); they do not report figures for the reverse condition. Our baseline is their model with wider training beams (§3.4).

	LF	LP	LR	UF	UP	UR
Baseline	85.53	85.73	85.33	91.99	92.20	91.77
Petrov I-5	85.79	86.09	85.50	92.44	92.76	92.13
$BP_{q=1}$	**86.45**	**86.75**	**86.17**	**92.60**	**92.92**	**92.29**
$DD_{q=25}$	86.35	86.65	86.05	92.52	92.85	92.20

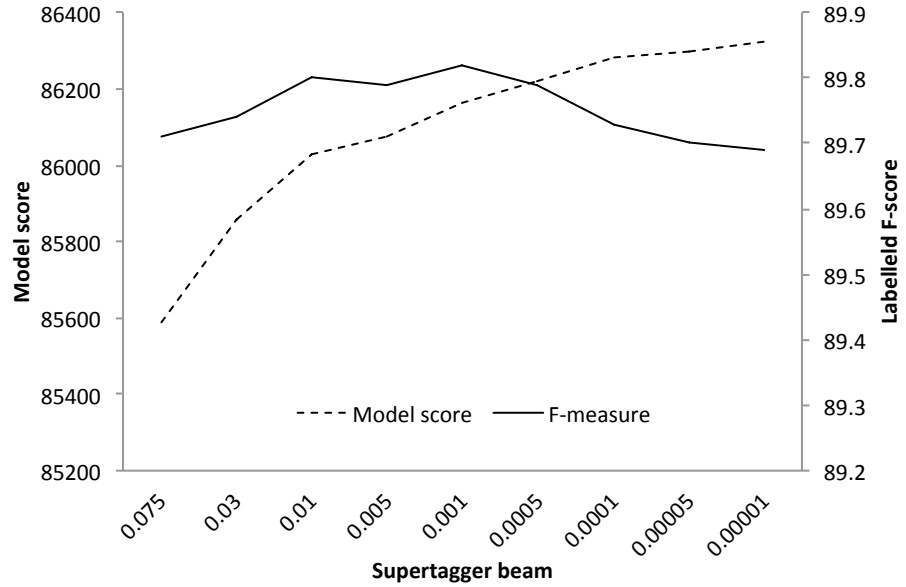
Table 5.4: Results on development set (section 00) with automatically assigned POS tags. *Petrov I-5* is based on the parser output of Fowler and Penn (2010); we evaluate on sentences for which *all* parsers returned an analysis (1834 sentences for section 00).

	LF	LP	LR	UF	UP	UR
Baseline	85.74	85.90	85.58	91.92	92.09	91.75
Petrov I-5	86.01	86.29	85.74	92.34	92.64	92.04
$BP_{q=1}$	**86.84**	**87.08**	**86.61**	**92.57**	**92.82**	**92.32**
$DD_{q=25}$	86.68	86.90	86.46	92.44	92.67	92.21

Table 5.5: Results on test set (section 23) with automatically assigned POS tags (cf. Table 5.4). We evaluate on sentences for which *all* parsers returned an analysis (2323 sentences for section 23).



(a) Belief Propagation



(b) Dual Decomposition

Figure 5.7: Comparison between model score and F-score for the integrated model using belief propagation (a) and dual decomposition (b); the results are based on the same data as Figure 5.1.

.

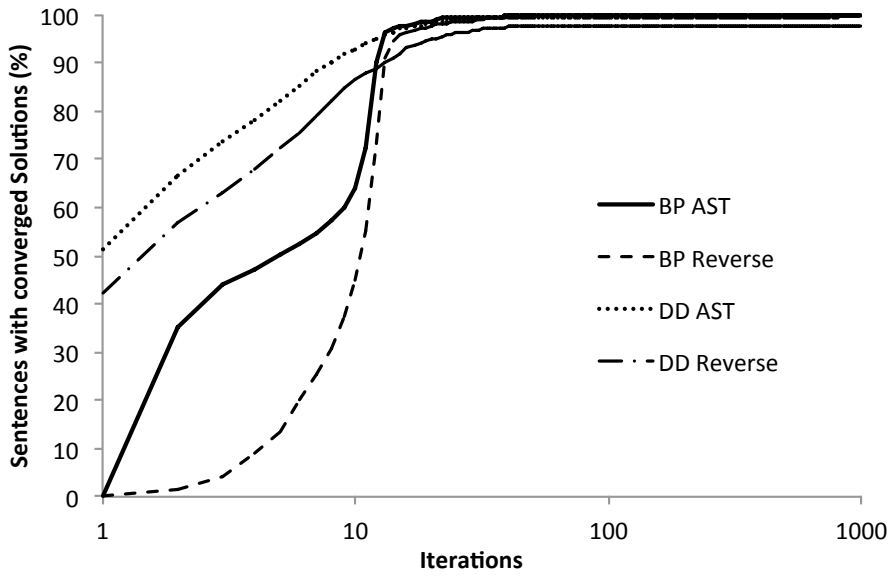


Figure 5.8: Proportion of sentences with converged solutions across iterations for belief propagation (BP) and dual decomposition (DD) with maximum $q = 1000$.

convergence of each algorithm under these criteria over 1000 iterations (Figure 5.8). Initially, DD converges much faster, while BP in the reverse condition converges quite slowly; however, BP overtakes DD after 13 iterations. This is interesting when contrasted with its behavior on parse accuracy–after one iteration, it has converged on only 1.5% of sentences, but its accuracy is already the highest at this point. Over the entire 1000 iterations, most sentences converge: all sentences for BP (both in AST and reverse) and all but 41 (2.1%) for DD in reverse (6 in AST).

We mentioned that DD can recover exact solutions when it converges (§5.3.2). Figure 5.8 shows that it can find exact solutions relatively quickly for our problem. Notably, about 42% of sentences converge after the first iteration of DD, at which point no Lagrangian constraints have been added to the subproblems. The number of sentences converged at the first iteration indicates how many exact solutions the parser can recover alone and how many exact solutions are due to DD.

For BP there is no guarantee that it can find exact solutions at all, even when the marginals converge (§2.3.6). But how often does BP find the exact solution? We evaluate the number of exact BP solutions in the reverse condition by comparing the BP parses after iterations $q = 1, 2, \ldots, 1000$ to DD solutions at $q = 1000$. At $q = 1000$, DD has found exact solutions for 98% of sentences (Figure 5.9), a good proxy to determine the exactness of the BP solutions. It turns out that nearly 91% of BP parses

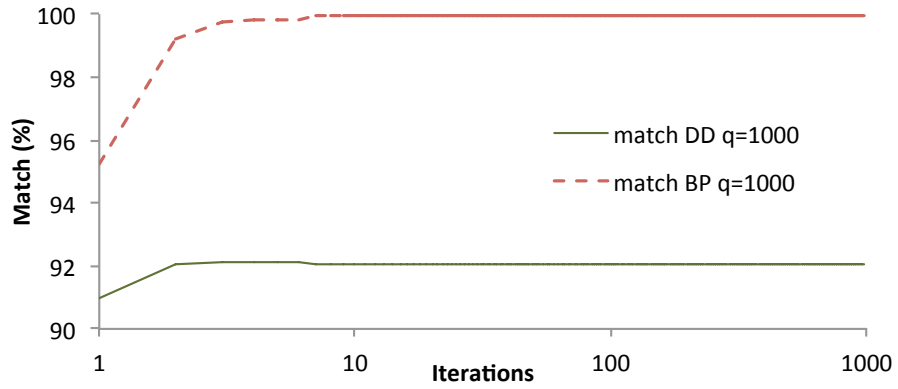


Figure 5.9: Matches between belief propagation (BP) solutions at iteration q and the dual decomposition (DD) solutions at $q = 1000$ as well as the BP solutions at $q = 1000$ in the reverse condition; the former is an indicator of the exactness of the BP solutions.

at $q = 1$ match the final DD solutions. This explains the high initial performance of BP (Figure 5.6) but it is also interesting to contrast this to its low algorithmic convergence rate of only about 1.5% at $q = 1$ (Figure 5.8).

Furthermore, the high degree of similar solutions returned by BP and DD is also interesting in terms of the decoding decision rules we used in each setting: For BP we rely on minimum-risk decoding and for DD we use Viterbi decoding (§2.3.3); however, the results show a very high overlap in the solutions they recover. A likely scenario is that the parse distribution is very peaked, resulting in very few high probability derivations that are recovered by both decision rules.

5.4.3 Parsing Speed

Because the C&C parser with AST is very fast, we wondered about the effect of BP and DD on speed for our model. We measured the runtime (§3.2.2) of the algorithms under the condition that we stopped at a particular iteration (Table 5.6). Although our models improve substantially over C&C, there is a significant cost in speed for the best result. The experiment also confirms that BP at $q = 1$ is more accurate than DD at a similar speed. Furthermore, BP can parse more accurately than the baseline at comparable speed.

	AST		Reverse	
	sent/sec	LF	sent/sec	LF
Baseline	65.6	87.38	5.8	87.36
$BP_{q=1}$	60.1	87.70	5.6	88.35
$BP_{q=5}$	45.2	87.70	4.6	88.33
$BP_{q=25}$	34.4	87.70	3.5	88.33
$DD_{q=1}$	63.8	87.40	5.8	87.38
$DD_{q=5}$	41.4	87.65	3.0	88.09
$DD_{q=25}$	32.2	87.71	1.9	88.29

Table 5.6: Parsing rate in sentences per second and labelled F-measure on section 00.

	AST			Reverse		
	LF	UF	ST	LF	UF	ST
Baseline	86.7	92.7	94.0	86.7	92.7	93.9
BP inf	86.8	92.8	94.1	87.3	93.2	94.3
BP train	86.4	92.6	93.9	85.7	92.2	93.2

Table 5.7: Results of training with SGD on approximate gradients from LPB on section 00. We test LBP in both inference and training (train) as well as in inference only (inf); a maximum number of 10 iterations is used both during training and testing.

5.4.4 Training the Integrated Model

In the experiments reported so far, the parsing and supertagging models were trained with the piecewise estimator. Although the outcome of these experiments was successful, we wondered if we could obtain further improvements by jointly training the integrated model using belief propagation.

Since the gradients produced by (loopy) BP are approximate, for these experiments we used a stochastic gradient descent (SGD) trainer (Bottou, 2003; see §3.4). There are two changes to our setup: We used the lower performing normal-form model (§3.3.3) and we used the stricter training beam settings from Clark and Curran (2007; §3.4) to make estimation on a single machine practical. At test time we used the Viterbi algorithm to decode the normal-form model. We also had to restrict the maximum number of BP iterations during training to 10, which resulted in early stopping for

some sentences.

Curiously, however, we found that the combined model does not perform as well when the parameters are trained with belief propagation (Table 5.7). One reason for the lower performance maybe the more restrictive training setup. Another may be that belief propagation over-fits the training data with our complex model (Sutton and McCallum, 2005b; Cohn, 2007). The piecewise estimator is likely to avoid this issue because it separates the two sub-models during training.

Training a model using DD would require a different optimisation algorithm based on Viterbi results (e.g. the perceptron) which is an interesting avenue for future work.

5.5 Conclusions

This chapter has introduced an integrated model of both supertagging and parsing. We have demonstrated that the interaction between supertagging and parsing can be better exploited in an integrated model rather than by chaining the two models in a pipeline. Our empirical comparison of BP and DD also complements the theoretically-oriented comparison of marginal- and margin-based variational approximations for parsing described by Martins et al. (2010).

We have shown that the aggressive pruning used in adaptive supertagging significantly harms the oracle performance of the parser, though it mostly prunes bad parses. Based on these findings, we combined parser and supertagger features into a single model. Using belief propagation and dual decomposition, we obtained more principled—and more accurate—approximations than a pipeline, improving both parsing as well as supertagging accuracy. Models combined using belief propagation achieve very good performance immediately, despite an initial convergence rate of just over 1%, while dual decomposition produces comparable results after several iterations, and algorithmically converges more quickly. Moreover, the initial solutions recovered by belief propagation are highly exact at over 90% despite the lack of formal guarantees. For our task we found that belief propagation achieved comparable accuracy to dual decomposition but in significantly less time. Our best result of 88.9% represents the state-of-the art in CCG parsing accuracy.

Chapter 6

Task-specific Optimisation

In Chapter 5 we have shown that a single integrated approach to supertagging and parsing improves accuracy for both tasks compared to the traditionally used pipeline strategy of separating them into subsequent steps. The integrated model is a Conditional Random Field (CRF; Lafferty et al., 2001), a choice that has been very successful for parsing models (Clark and Curran, 2007; Finkel et al., 2008). In practice, CRFs are usually trained by maximising the conditional log-likelihood (CLL) of the training data (§2.3.2). However, it is widely appreciated that optimising for task-specific metrics often leads to better performance on those tasks (Och, 2003).

In this chapter, we demonstrate that optimising each sub-model towards a separate task-specific metric, while combining them at test-time can further improve the accuracy of the integrated approach. For this purpose we adopt the softmax-margin (SMM) objective (Sha and Saul, 2006; Povey and Woodland, 2002; Gimpel and Smith, 2010a) (§6.1). In addition to retaining a probabilistic interpretation and optimising towards a loss function, it is also convex, making it straightforward to optimise. Gimpel and Smith (2010a) show that it can be easily implemented with a simple change to standard likelihood-based training, provided that the loss function decomposes over the predicted structure.

Unfortunately, the widely-used F-measure metric does not decompose over parses. Previous work has focused on approximations thereof, both for sequence models (Altun et al., 2003) and parsing models (Taskar et al., 2004). We introduce a novel dynamic programming algorithm that enables us to compute the exact quantities needed under the softmax-margin objective using F-measure as a loss (§6.2). We experiment with F-measure and several other metrics, including precision, recall, and decomposable approximations thereof. Our ability to optimise towards exact metrics enables

us to verify the effectiveness of more efficient approximations. We test the training procedures on the CCG parser of Clark and Curran (2007), obtaining substantial improvements under a variety of conditions. We then apply this method to our integrated model (Chapter 5) and optimise each sub-model towards a different loss. The improvements are additive, obtaining the best reported results on this task (§6.3).

6.1 Softmax-Margin Training

The softmax-margin objective modifies the standard likelihood objective for CRF training (§2.3.2) by reweighting each possible outcome of a training input according to its *risk*, which is simply the loss incurred on a particular example. This is done by incorporating the loss function directly into the linear scoring function of an individual example.

Formally, we are given m training pairs $(x^{(1)},y^{(1)})...(x^{(m)},y^{(m)})$, where each $x^{(i)} \in \mathcal{X}$ is drawn from the set of possible inputs, and each $y^{(i)} \in \mathcal{Y}(x^{(i)})$ is drawn from a set of possible instance-specific outputs. We want to learn the K parameters θ of a log-linear model, where each $\lambda_k \in \theta$ is the weight of an associated feature $h_k(x,y)$. The function $f(x,y)$ maps input/output pairs to the vector $h_1(x,y)...h_K(x,y)$, and our log-linear model assigns probabilities in the usual way.

$$p(y|x) = \frac{\exp\{\theta^\mathsf{T} f(x,y)\}}{\sum_{y' \in \mathcal{Y}(x)} \exp\{\theta^\mathsf{T} f(x,y')\}} \tag{6.1}$$

The conditional log-likelihood objective function is given by Equation 6.2 (Figure 6.1). Intuitively, this objective is minimised when the first term, corresponding to the sum of the weights of the features applying to the correct solution, is large and the second term, corresponding to the partition function, that is the sum of the weights of the features firing for all possible solutions, is small (§2.3.2). Hence, the optimiser tries to choose high weights on features applying to correct predictions and low or even negative weights for features firing on incorrect predictions.

Now consider a function $\ell(y,y')$ that returns the loss incurred by choosing to output y' when the correct output is y. The softmax-margin objective simply modifies the unnormalised, unexponentiated score $\theta^\mathsf{T} f(x,y')$ by adding $\ell(y,y')$ to it. This yields the objective function (Equation 6.3) and gradient computation (Equation 6.4) shown in Figure 6.1. Intuitively, this objective forces the optimiser to make weights on incorrect predictions even more negative due to the influence of the loss function.

$$\min_{\theta} \sum_{i=1}^{m} \left[-\theta^{\mathsf{T}} f(x^{(i)}, y^{(i)}) + \log \sum_{y \in \mathcal{Y}(x^{(i)})} \exp\{\theta^{\mathsf{T}} f(x^{(i)}, y)\} \right] \tag{6.2}$$

$$\min_{\theta} \sum_{i=1}^{m} \left[-\theta^{\mathsf{T}} f(x^{(i)}, y^{(i)}) + \log \sum_{y \in \mathcal{Y}(x^{(i)})} \exp\{\theta^{\mathsf{T}} f(x^{(i)}, y) + \ell(y^{(i)}, y)\} \right] \tag{6.3}$$

$$\frac{\partial}{\partial \lambda_k} = \sum_{i=1}^{m} \left[-h_k(x^{(i)}, y^{(i)}) + \sum_{y \in \mathcal{Y}(x^{(i)})} \frac{\exp\{\theta^{\mathsf{T}} f(x^{(i)}, y) + \ell(y^{(i)}, y)\}}{\sum_{y' \in \mathcal{Y}(x^{(i)})} \exp\{\theta^{\mathsf{T}} f(x^{(i)}, y') + \ell(y^{(i)}, y')\}} h_k(x^{(i)}, y) \right] \tag{6.4}$$

Figure 6.1: Conditional log-likelihood (Equation 6.2), Softmax-margin objective (Equation 6.3) and gradient (Equation 6.4).

This straightforward extension has several desirable properties. In addition to having a probabilistic interpretation, it can be shown to minimise a bound on expected risk, and it is convex (see Appendix A of (Gimpel and Smith, 2010b) for proof).

We can also see from Equation 6.4 that the only difference from standard CLL training is that we must compute feature expectations with respect to the cost-augmented scoring function. As Gimpel and Smith (2010a) discuss, if the loss function decomposes over the predicted structure, we can treat its decomposed elements as unweighted features that fire on the corresponding structures, and compute expectations in the normal way. In the case of our parser, where we compute expectations using the inside and outside algorithms (§2.3.4), a loss function decomposes if it can be incrementally computed over items or productions of a CKY chart. In other words, the loss function needs to be incrementally computable over the individual sub-trees of a derivation (Figure 6.2).

6.2 Loss Functions for Parsing

Ideally, we would like to optimise our parser towards a task-based evaluation. Our CCG parser is evaluated on labelled, directed dependency recovery using F-measure (Clark and Hockenmaier, 2002) as described in §3.2.1. Recall that under this evaluation we represent output y' and ground truth y as variable-sized sets of dependencies (see §2.1.2). We can then compute precision $P(y, y')$, recall $R(y, y')$, and F-measure

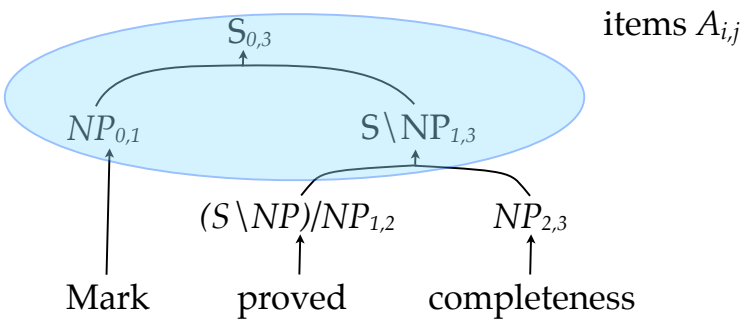


Figure 6.2: Hypergraph for a derivation for a simple sentence. Its loss needs to be incrementally computable over all parent-child sub-trees; each individual computation has only access to the local sub-tree. The sub-tree for $NP_{0,1}\ S\backslash NP_{1,3} \Rightarrow S_{0,3}$ is highlighted.

$F_1(y,y')$.

$$P(y,y') = \frac{|y \cap y'|}{|y'|} \tag{6.5}$$

$$R(y,y') = \frac{|y \cap y'|}{|y|} \tag{6.6}$$

$$F_1(y,y') = \frac{2PR}{P+R} = \frac{2|y \cap y'|}{|y| + |y'|} \tag{6.7}$$

These metrics are positively correlated with performance – they are *gain* functions. To incorporate them in the softmax-margin framework we reformulate them as loss functions by subtracting from one.

6.2.1 F-Score-Augmented Expectations with Exact Loss Functions

Unfortunately, none of these metrics decompose over parses. However, the individual statistics that are used to compute them *do* decompose, a fact we will exploit to devise an algorithm that computes the necessary expectations. This new algorithm is closely related to the dynamic program we used to compute oracle F-measure parses in §5.2.1. In this section we will first focus on exactly optimising F-measure at the *sentence-level* and then outline an algorithm for the corpus-level.

Note since y is fixed for every sentence, F_1 is a function of two integers: $|y \cap y'|$, representing the number of correct dependencies in y'; and $|y'|$, representing the total number of dependencies in y', which we will denote as n and d, respectively. Each pair $\langle n,d \rangle$ leads to a different value of F_1. Importantly, both n and d decompose over parses. This is illustrated in Figure 6.3 which shows that we *could* compute an exact

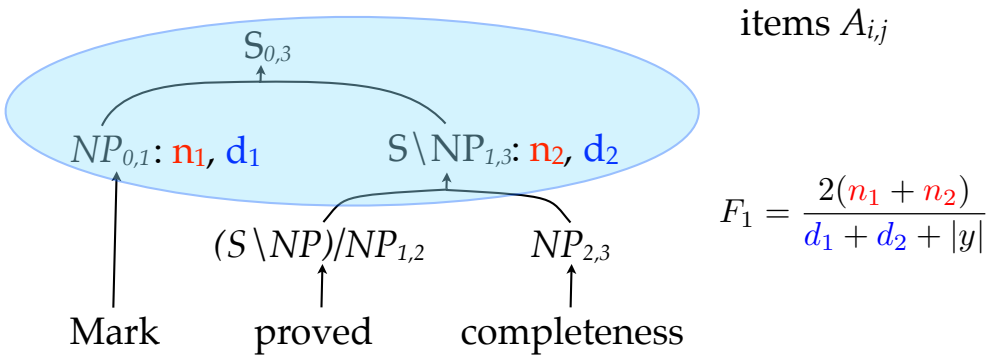


Figure 6.3: Illustration of how sentence-level F-measure can decompose over parses by combining the dependency counts n and d of individual sub-structures.

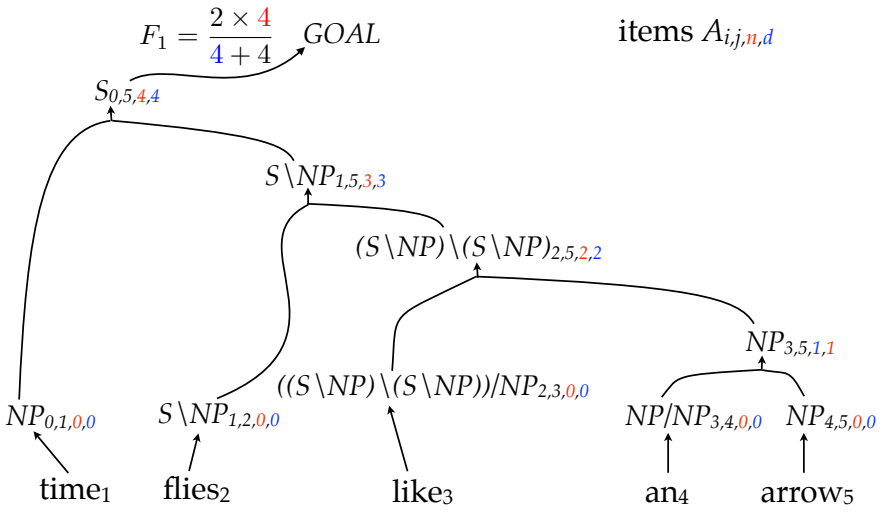
sentence-level F-measure if we had $\langle n,d\rangle$ pairs at every item $A_{i,j}$; we use notation $A_{i,j} : n,d$ to denote an $\langle n,d\rangle$ pair incident at $A_{i,j}$.[1]

The key idea will be to treat F_1 as a *non-local feature* of the parse, dependent on values n and d. To compute expectations we split each item in an otherwise usual inside-outside computation by all pairs $\langle n,d\rangle$ incident at that item. This is similar to the algorithm in §5.2.1 which *maximised* over n given d, whereas the new algorithm keeps track of all possible n and d.

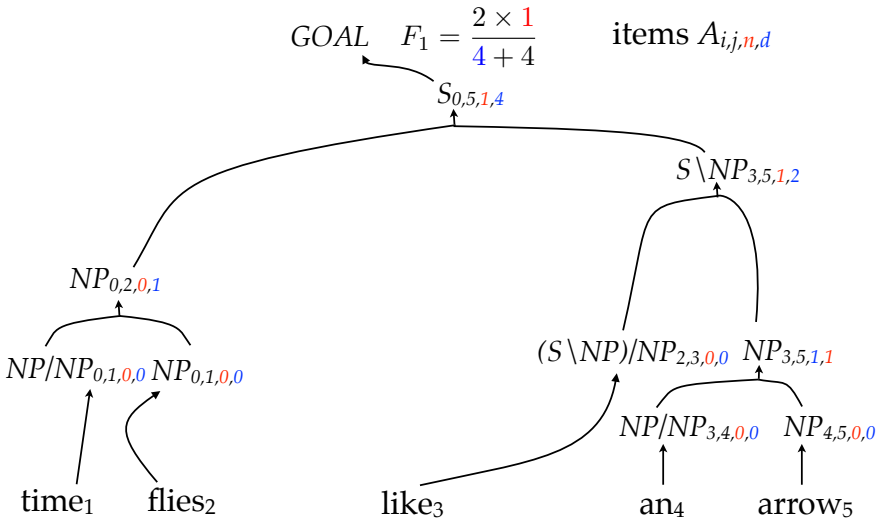
Formally, our goal will be to compute expectations over the sentence $a_1 ... a_L$. In order to abstract away from the particulars of CCG we present the algorithm in relatively familiar terms as a variant of the classic inside-outside algorithm (Baker 1979; §2.3.4). The classic algorithm associates inside probability $I(A_{i,j})$ and outside probability $O(A_{i,j})$ with $A_{i,j}$. The expectation of A spanning positions i through j is simply $I(A_{i,j})O(A_{i,j})/I(GOAL)$.

Our algorithm extends these computations to state-split items $A_{i,j,n,d}$. Using functions $n_+(\cdot)$ and $d_+(\cdot)$ to respectively represent the number of correct and total dependencies introduced by a parsing action, we present our algorithm in Figure 6.6. The final inside equation and the initial outside equation incorporate the loss function for all derivations having a particular F-score, enabling us to obtain the desired expectations. A simple modification of the goal equations enables us to optimise precision, recall or a weighted F-measure. Figure 6.4 illustrates the inside pass of this algorithm.

[1] Note that this notation only denotes that there are n correct dependencies and d dependencies overall for a parse leading to $A_{i,j}$, the items $A_{i,j,n,d}$ introduced later are for a different dynamic program.



(a) Correct parse



(b) Incorrect parse

Figure 6.4: Illustration of the inside pass for a correct (a) and an incorrect (b) parse using the state-split algorithm in Figure 6.6. The items in the hypergraph keep track of the dependencies introduced and the loss is computed when transitioning to the GOAL state using the F-measure function F_1. The graphs simplify the computation when transitioning to the GOAL state, which is actually $I(S_{0,L,n,d})(1 - F_1)$, as in Figure 6.6.

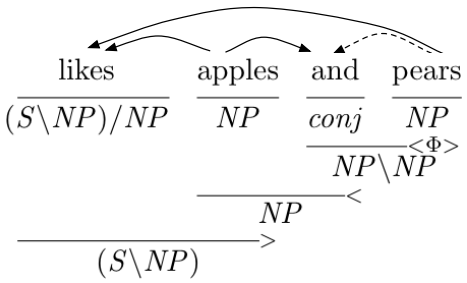


Figure 6.5: Example of flexible dependency realisation in CCG: The C&C parser (Clark and Curran, 2007) creates dependencies arising from coordination once all conjuncts are found and treats "and" as the syntactic head of coordinations. The coordination rule (Φ) does not yet establish the dependency "and - pears" (dotted line); it is the backward application ($<$) in the larger span, "apples and pears", that establishes it, together with the dependency "and - pears". CCG also deals with unbounded dependencies which potentially lead to more dependencies than words (Steedman, 2000); in this example a unification mechanism creates the dependencies "likes - apples" and "likes - pears" in the forward application ($>$). For further examples and a more detailed explanation of the mechanism as used in the C&C parser refer to Clark et al. (2002).

To analyse the complexity of this algorithm, we must ask: how many pairs $\langle n, d \rangle$ can be incident at each item? A CCG parser does not necessarily return one dependency per word (§2.1.2) or see Figure 6.5, so d is not necessarily equal to the sentence length L as it might be in many dependency parsers, though it is still bounded by $O(L)$. However, it is sufficiently uncommon that d is much larger than L and so we expect all parses of a sentence, good or bad, to have close to L dependencies; hence we expect the range of d to be constant on average. Furthermore, n will be bounded from below by zero and from above by $\min(|y|, |y'|)$. Hence the set of all possible F-measures for all possible parses is bounded by $O(L^2)$. Following McAllester (1999), we can see from inspection of the free variables in Fig. 6.6 that the algorithm requires worst-case $O(L^7)$ time complexity, and worse-case $O(L^4)$ space complexity.

F-Measure-Augmented Expectations at the Corpus Level. In order to compute exact corpus-level expectations for softmax-margin using F-measure, we need to add an additional transition before reaching the *GOAL* item in our original program. However, in order to reach it, we must parse every sentence in the corpus, associating statistics of aggregate $\langle n, d \rangle$ pairs for the entire training set in intermediate symbols $\Gamma^{(1)} ... \Gamma^{(m)}$

$$I(A_{i,i+1,n,d}) = w(a_{i+1} \Rightarrow A) \quad \text{iff } n = n_+(a_{i+1} \Rightarrow A),\ d = d_+(a_{i+1} \Rightarrow A)$$

$$I(A_{i,j,n,d}) = \sum_{k,B,C} \sum_{\substack{\{n',n'':n'+n''+n_+(BC\Rightarrow A)=n\},\\ \{d',d'':d'+d''+d_+(BC\Rightarrow A)=d\}}} I(B_{i,k,n',d'})I(C_{k,j,n'',d''})w(BC \Rightarrow A)$$

$$I(GOAL) = \sum_{n,d} I(S_{0,L,n,d}) \left(1 - \frac{2n}{d+|y|}\right)$$

$$O(S_{0,L,n,d}) = \left(1 - \frac{2n}{d+|y|}\right)$$

$$O(A_{i,j,n,d}) = \sum_{k,B,C} \sum_{\substack{\{n',n'':n'-n''-n_+(AB\Rightarrow C)=n\},\\ \{d',d'':d'-d''-d_+(AB\Rightarrow C)=d\}}} O(C_{i,k,n',d'})I(B_{j,k,n'',d''})w(AB \Rightarrow C) +$$

$$\sum_{k,B,C} \sum_{\substack{\{n',n'':n'-n''-n_+(BA\Rightarrow C)=n\},\\ \{d',d'':d'-d''-d_+(BA\Rightarrow C)=d\}}} O(C_{k,j,n',d'})I(B_{k,i,n'',d''})w(BA \Rightarrow C)$$

Figure 6.6: State-split inside and outside recursions for computing softmax-margin with F-measure.

with the following inside recursions.

$$
\begin{aligned}
I(\Gamma^{(1)}_{n,d}) &= I(S^{(1)}_{0,|x^{(1)}|,n,d}) \\
I(\Gamma^{(\ell)}_{n,d}) &= \sum_{\substack{\{n',n'':n'+n''=n\},\\ \{d',d'':d'+d''=d\}}} I(\Gamma^{(\ell-1)}_{n',d'}) I(S^{(\ell)}_{0,|x^{(l)}|,n'',d''}) \\
I(GOAL) &= \sum_{n,d} I(\Gamma^{(m)}_{n,d}) \left(1 - \frac{2n}{d+|y|}\right)
\end{aligned}
$$

where $|x^i|$ is the number of words in sentence x^i. Outside recursions follow straightforwardly. Implementation of this algorithm would require substantial distributed computation or external data structures, so we did not attempt it.

6.2.2 Approximate Loss Functions

We will also consider approximate but more efficient alternatives to our exact algorithms. The idea is to use cost functions which only utilise statistics available within the current local structure. We borrow the approximate precision loss function used by Taskar et al. (2004) which tracks constituent errors in a context-free parser; in our setting it simply tracks dependency errors. Furthermore, we devise two more cost functions to approximate recall and F-measure on CCG dependency structures.

Let $T(y)$ be the set of parsing actions required to build parse y. Our decomposable approximation to precision simply counts the number of incorrect dependencies using the local dependency counts, $n_+(\cdot)$ and $d_+(\cdot)$.

$$DecP(y) = \sum_{t \in T(y)} d_+(t) - n_+(t) \tag{6.8}$$

To compute our approximation to recall we require the number of gold dependencies, $c_+(\cdot)$, that should be introduced by a particular parsing action. A gold dependency is due to be recovered by a parsing action if its head lies within one child span and its dependent within the other child span. This yields a decomposed approximation to recall that counts the number of missed dependencies.

$$DecR(y) = \sum_{t \in T(y)} c_+(t) - n_+(t) \tag{6.9}$$

Unfortunately, the flexible handling of dependencies in CCG complicates our formulation of c_+, rendering it slightly more approximate. The unification mechanism of CCG

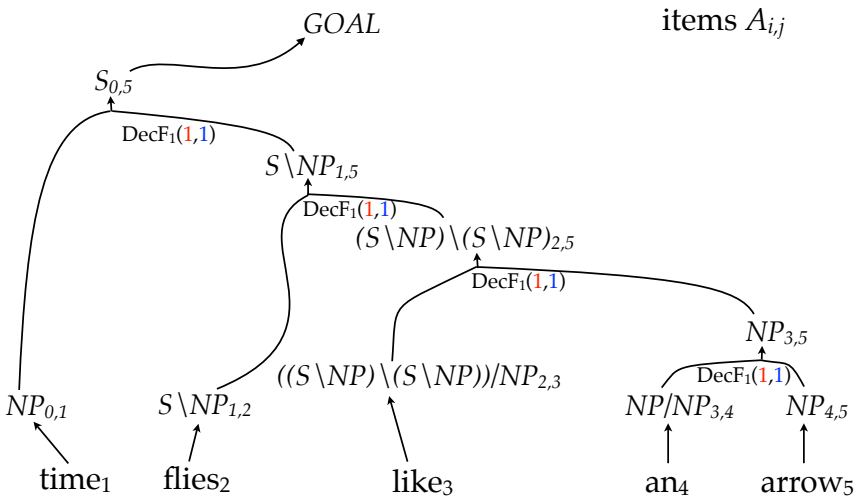


Figure 6.7: Illustration of the inside pass when computing expectations with approximate F-measure $DecF1$. The inside value of every item is augmented by the loss function $DecF1$.

sometimes causes dependencies to be realised later in the derivation, at a point when both the head and the dependent are in the same span, violating the assumption we use to compute c_+ (see again Figure 6.5). Exceptions like this can cause mismatches between n_+ and c_+. We set $c_+ = n_+$ whenever $c_+ < n_+$ to account for these occasional discrepancies. This assumption is not correct but it allows us to compute the decomposed recall loss function.

Finally, we obtain a decomposable approximation to F-measure.

$$DecF1(y) = DecP(y) + DecR(y) \tag{6.10}$$

Figure 6.7 illustrates the inside pass when computing expectations for a single derivation. In contrast to the exact sentence-level loss functions (§6.2.1), the approximate loss functions do not take into account the entire structure, e.g., the parse or corpus, but rather only local dependency counts incident at a particular item. For a parse with incorrect dependencies, the exact loss functions will augment the marginals of all items part of the parse *equally*. However, the local loss functions will focus entirely on the incorrect items, rewarding the correct items and penalising only the incorrect ones.

6.3 Experiments

We begin by evaluating task-specific training for parsing alone and then apply it to our integrated model (Chapter 5). In particular, we first evaluate a simple baseline approach which uses the standard likelihood objective function (§6.3.1). We then evaluate the performance of our exact algorithms for F-measure, precision and recall (§6.3.2). Next we compare the exact algorithms against approximations (§6.3.3). Finally, we apply task-specific training methods to our integrated supertagging and parsing model (§6.3.4).

Our experimental setup is identical to Chapter 5 which follows §3.3.3: We use the C&C parser (Clark and Curran, 2007) and its supertagger (Clark, 2002). Our baseline is the hybrid dependency model of Clark and Curran (2007), which contains features over both normal-form derivations and CCG dependencies. Training requires calculation of feature expectations over packed charts of derivations and we trained our models with the parser's L-BFGS trainer (§3.4). Similar to before we test with both adaptive supertagging (AST) and reverse adaptive supertagging (Reverse) beam settings (§5.1). The coverage of our parsers is again identical for *all* parsers since we do not vary the training and test beams (99.22% on section 00 and 99.63% on section 23).

6.3.1 Training with Maximum F-measure Parses

So far we discussed how to optimise towards task-specific metrics via changing the training objective. In our first experiment we change the *data* on which we optimise CLL. This is a baseline to our later experiments, attempting to achieve the same effect by simpler means. Specifically, we use the algorithm of Huang (2008), introduced in §5.2.1, to generate oracle F-measure parses for each sentence. Updating towards these oracle parses corrects the reachability problem in standard CLL training which prevents the use of all training data.

To make CLL training feasible, the training forests are pruned using the supertagger. Unfortunately, pruning sometimes removes the correct parse required by the objective function. Clark and Curran (2007) mitigate the effects of pruning by restoring gold supertags that have been removed. However, this still only results in a training data utilisation of 91.5% (see §3.4). A potential disadvantage of restoring pruned supertags is to bias the model by training it in an idealised setting not available at test time. Using oracle parses corrects this bias while permitting nearly 99% training data utilisation. The labelled F-score of the oracle parses is 98.1%. Despite expectations

of our method improving accuracy, the results (Table 6.1) show no effect. However, it does serve as a useful baseline.

	LF	LP	LR	UF	UP	UR	Data Util (%)
Baseline	87.40	87.85	86.95	93.11	93.59	92.63	91.5%
Max-F Parses	87.46	87.95	86.98	93.09	93.61	92.57	98.7%
CCGbank+Max-F	87.45	87.96	86.94	93.09	93.63	92.55	98.9%

Table 6.1: Performance on section 00 of CCGbank when comparing models trained with treebank-parses (Baseline) and maximum F-score parses (Max-F) using adaptive supertagging as well as a combination of CCGbank and Max-F parses. Evaluation is based on labelled and unlabelled F-measure (LF/UF), precision (LP/UP) and recall (LR/UR).

6.3.2 Training with the Exact Algorithm

We first tested our assumptions about the feasibility of training with our exact algorithm by measuring the amount of state-splitting. Figure 6.8 plots the average number of splits per item against the relative span-frequency, i.e., the word span size of an item; the statistics are based on a typical set of training forests containing nearly 7.6 billion items.[2] The increase in the number of splits is polynomial in the span size but similarly does the number of spans decrease in the number of splits. Hence the small number of items with a high number of splits is balanced by a large number of spans with only a few splits: The highest number of splits per span observed with our setup was 4888 but we find that the average number of splits lies at 44. Encouragingly, this enables experimentation in all but very large scale settings.

Figure 6.9 shows the distribution of n and d pairs across all split-states in the training corpus; since n, the number of correct dependencies, over d, the number of all recovered dependencies, is precision, the graph shows that only a minority of states have either very high or very low precision. The range of values suggests that the softmax-margin criterion will have an opportunity to substantially modify the expec-

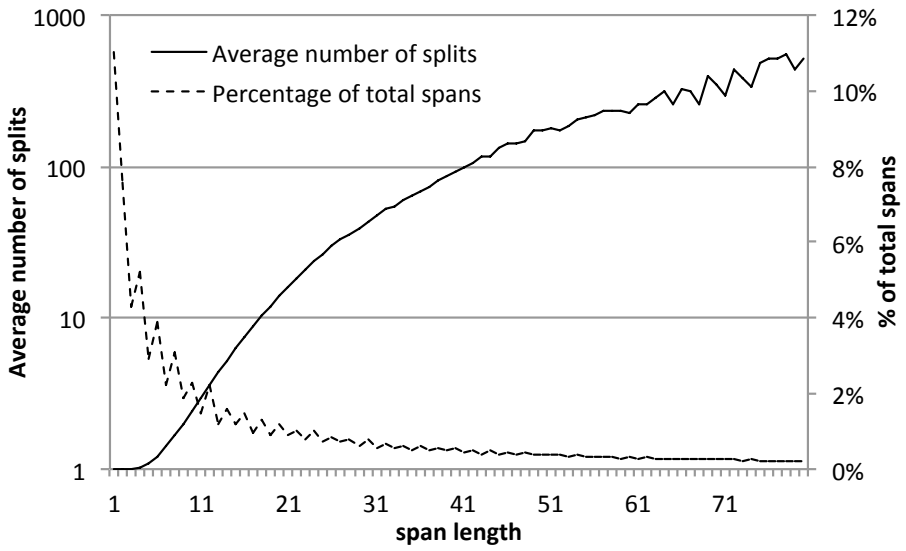


Figure 6.8: Average number of state-splits per span length as introduced by a sentence-level F-measure loss function. The statistics are averaged over the training forests generated using the settings described in §6.3.

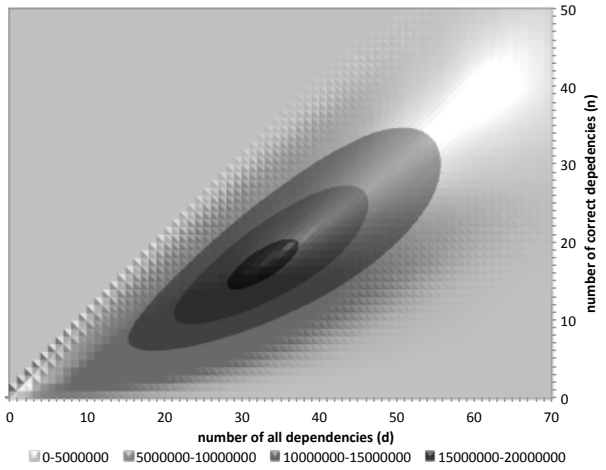


Figure 6.9: Distribution of states with d dependencies of which n are correct in the training forests.

tations, hopefully to good effect.

We next turn to the question of optimisation with these algorithms. Due to the significant computational requirements, we used the computationally less intensive normal-form model of Clark and Curran (2007) as well as their more restrictive training beam settings (see §3.3.3). We train on all sentences of the training set as above and test with AST.

In order to provide greater control over the influence of the loss function, we introduce a multiplier τ, which simply amends the second term of the objective function (6.3) to:

$$\log \sum_{y \in Y(x^i)} \exp\{\theta^\top f(x^i, y) + \tau \times \ell(y^i, y)\}$$

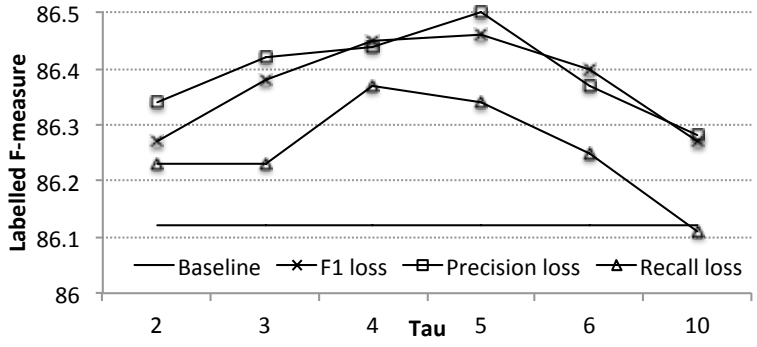
Figure 6.10 plots performance of the exact loss functions across different settings of τ on various evaluation criteria, for models restricted to at most 3000 items per chart at training time to allow rapid experimentation with a wide parameter set. Even in this constrained setting, it is encouraging to see that each loss function performs best on the criteria it optimises. The precision-trained parser also does very well on F-measure; this is because the baseline parser has better precision accuracy than recall accuracy.

6.3.3 Exact vs. Approximate Loss Functions

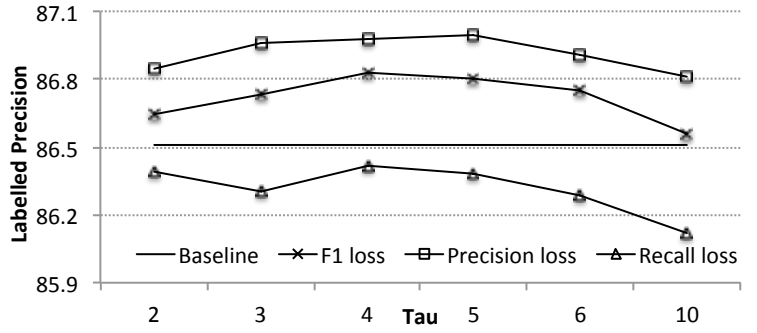
With these results in mind, we conducted a comparison of parsers trained using our exact and approximate loss functions. Table 6.3 and Table 6.4 compare their performance head to head when restricting training chart sizes to 100,000 items per sentence, the largest setting our computing resources allowed us to experiment with. The results confirm that the loss-trained models improve over a likelihood-trained baseline, and furthermore that the exact loss functions seem to have the best performance. However, the approximations are extremely competitive to their exact counterparts. Moreover, the high efficiency of approximate loss functions makes them attractive for experimentation on a larger-scale.

Training time with exact loss functions increases only 30-fold compared to standard CLL training (Table 6.2), despite much higher worst-case theoretical complexity (§6.2.1). On the other hand, memory or space requirements only triple, mostly due to an efficient implementation; there is no significant change in either time or space for

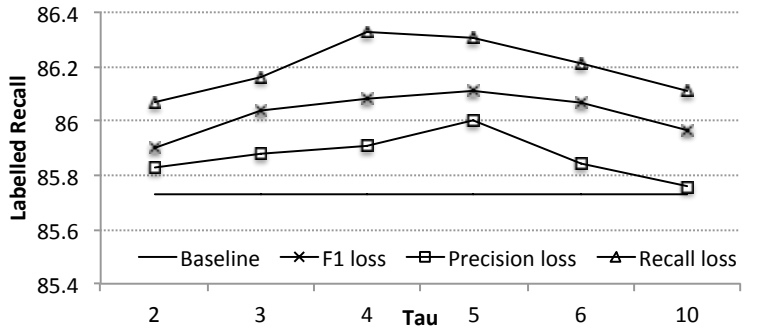
[2] We reported a lower figure in Auli and Lopez (2011c) due to an error in the quantification of the forest-sizes, however, the analysis based on the forests remains unchanged.



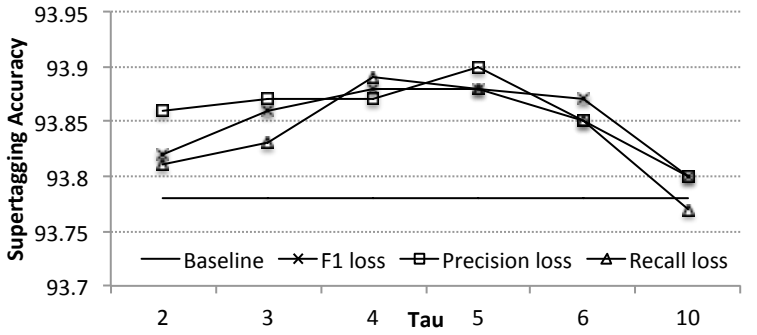
(a)



(b)



(c)



(d)

Figure 6.10: Performance of exact cost functions optimising F-measure, precision and recall in terms of (a) labelled F-measure, (b) precision, (c) recall and (d) supertag accuracy across various settings of τ on the development set.

	Time (h)	Space (Gb)
CLL	0.38	56
DecF1	0.41	56
F1	11.4	169

Table 6.2: Time and space requirements for standard conditional log-likelihood (CLL) training versus softmax-margin with approximate (DecF1) and exact F-measure loss functions (F1). Time is based on the duration of optimisation when using 64 Intel-Xeon cores of a compute-cluster; this is not the accumulative training time. However, space is accumulated across all cores and it is equal for both CLL and F1 because the additional statistics require so little space that no additional memory needed to be allocated.

training with approximate loss functions compared to standard CLL training.

Table 6.5 and Table 6.6 show the performance of the approximate losses with the hybrid dependency model and the usual training beams (§3.4). One striking result is that the softmax-margin trained models coax more accurate parses from the larger search space, in contrast to the likelihood-trained models. Our best loss model improves the labelled F-measure by over 0.9% on the test set.

How does an F-measure optimised model perform on *exact match*? The training methods we presented may result in poor accuracy on other measures than the one it has been specifically optimised. Table 6.7 confirms that this is not the case.

6.3.4 Combination with Integrated Supertagging and Parsing

As a final experiment, we embed our loss-trained model into the integrated model described in Chapter 5. We use the same experimental setup as in Chapter 5 with the piecewise estimator and at test time we use belief propagation for inference. For softmax-margin, we combine a parsing model trained with F1 and a supertagger trained with Hamming loss (SA), i.e., given the true tag-sequence $t_1 \dots t_L$ and a predicted tag-

	LF	LP	LR	UF	UP	UR
CLL	86.76	87.16	86.36	92.73	93.16	92.30
DecP	87.18	**87.93**	86.44	92.93	93.73	92.14
DecR	87.31	87.55	87.07	93.00	93.26	92.75
DecF1	87.27	87.78	86.77	93.04	93.58	92.50
P	87.25	87.85	86.66	92.99	**93.63**	92.36
R	**87.34**	87.51	**87.16**	92.98	93.17	**92.80**
F1	**87.34**	87.74	86.94	**93.05**	93.47	92.62

Table 6.3: Development set performance of exact and approximate loss functions and the baseline, conditional log-likelihood (CLL): Approximate losses are decomposable precision (DecP), recall (DecR) and F-measure (DecF1); exact losses are exact precision (P), recall (R) and F-measure (F1). Evaluation is based on labelled and unlabelled F-measure (LF/UF), precision (LP/UP) and recall (LR/UR).

	LF	LP	LR	UF	UP	UR
CLL	87.46	87.80	87.12	92.85	93.22	92.49
DecP	87.75	**88.34**	87.17	93.04	**93.66**	92.43
DecR	87.57	87.71	87.42	92.92	93.07	92.76
DecF1	87.69	88.10	87.28	93.04	93.48	92.61
P	**87.76**	88.23	87.30	**93.06**	93.55	92.57
R	87.57	87.62	**87.51**	92.92	92.98	**92.86**
F1	87.71	88.01	87.41	93.02	93.34	92.70

Table 6.4: Test set performance of exact and approximate loss functions and conditional log-likelihood (cf. Table 6.3).

sequence $t'_1 \dots t'_L$ we have

$$SA(y,y') = \tau \times \sum_{i=1}^{L} |\delta(t_i, t'_i)| \tag{6.11}$$

where $\delta(x,y)$ is the Kronecker-delta returning 1 if $x = y$ and 0 otherwise; τ is the multiplier we introduced in §6.3.2. Table 6.8 and Table 6.9 show the results: we observe a gain of up to 1.5% in labelled F-measure and 0.9% in unlabelled F-measure on the test set. The loss functions prove their robustness by improving the more accurate

	AST			Reverse		
	LF	UF	ST	LF	UF	ST
CLL	87.38	93.08	94.21	87.36	93.13	93.99
DecP	87.35	92.99	94.25	87.75	93.25	94.22
DecR	87.48	93.00	94.34	87.70	93.16	94.30
DecF1	**87.67**	**93.23**	**94.39**	**88.12**	**93.52**	**94.46**

Table 6.5: Development set performance of decomposed loss functions in large-scale training setting. Evaluation is based on labelled and unlabelled F-measure (LF/UF) and supertag accuracy (ST).

	AST			Reverse		
	LF	UF	ST	LF	UF	ST
CLL	87.73	93.09	94.33	87.65	93.06	94.01
DecP	**88.10**	93.26	94.51	88.51	93.50	94.39
DecR	87.66	92.83	94.38	87.77	92.91	94.22
DecF1	88.09	**93.28**	**94.50**	**88.58**	**93.57**	**94.53**

Table 6.6: Test set performance of decomposed loss functions in large-scale training setting (cf. Table 6.5).

	section 00 (dev)				section 23 (test)			
	AST		Reverse		AST		Reverse	
	LM	UM	LM	UM	LM	UM	LM	UM
CLL	36.93	38.20	36.93	38.25	37.66	39.28	37.99	39.62
DecF1	**37.78**	**38.94**	**38.46**	**39.62**	**38.03**	**39.41**	**39.07**	**40.33**

Table 6.7: Performance in terms of labelled and unlabelled exact match (LM/UM).

	AST			Reverse		
	LF	UF	ST	LF	UF	ST
CLL	87.38	93.08	94.21	87.36	93.13	93.99
BP	87.67	93.26	94.43	88.35	93.72	94.73
+DecF1	**87.90**	**93.40**	**94.52**	**88.58**	**93.88**	**94.79**
+SA	87.73	93.28	94.49	88.40	93.71	94.75

Table 6.8: Development set performance of integrated parsing and supertagging with belief propagation (BP); using decomposed-F1 (DecF1) as parser-loss function and supertag-accuracy (SA) as loss function for the supertagger.

	AST			Reverse		
	LF	UF	ST	LF	UF	ST
CLL	87.73	93.09	94.33	87.65	93.06	94.01
BP	88.25	93.33	94.60	88.86	93.75	94.84
+DecF1	88.32	93.32	94.66	89.15	93.89	94.98
+SA	**88.47**	**93.48**	**94.71**	**89.25**	**93.98**	**95.01**

Table 6.9: Test set performance of integrated parsing and supertagging with belief propagation (cf. Table 6.8).

integrated models up to 0.4% in labelled F-score.

Table 6.10 and Table 6.11 shows results with automatic part-of-speech tags and a direct comparison with the Petrov parser trained on CCGbank (Fowler and Penn, 2010), which we outperform on all metrics.

Finally, we compare the models proposed in this thesis in terms of efficiency and accuracy on the test set (Figure 6.11). The results can be clustered into two main groups: Parsing with adaptive supertagging (AST) and parsing with reverse adaptive supertagging (Rev). The former is preferable when speed is important and when less accurate parses can be tolerated. Reverse AST is preferable when accuracy is paramount and when lower speed is acceptable. Generally, the softmax-margin-trained integrated model using belief propagation (BP+SMM) offers the best tradeoff between

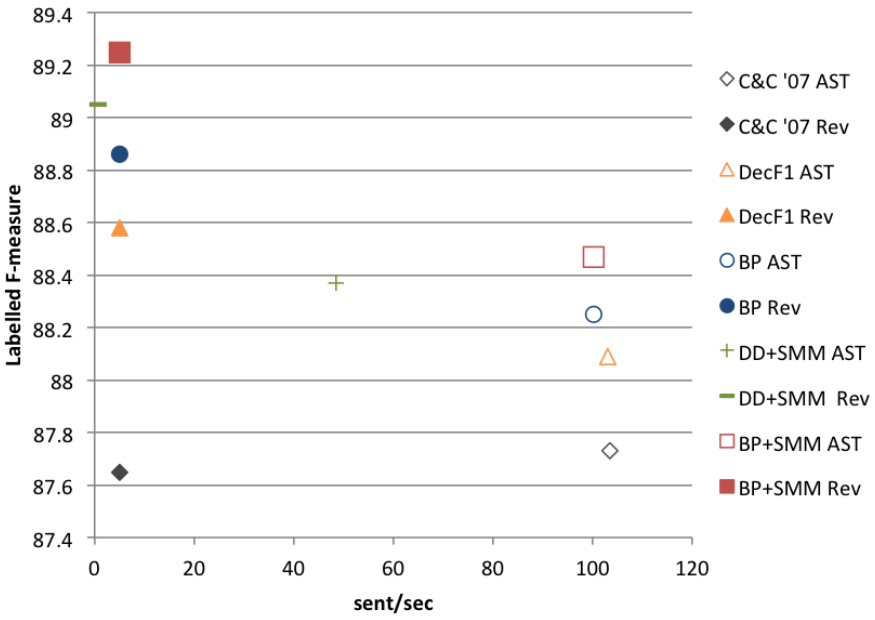


Figure 6.11: Accuracy versus efficiency for various models proposed in this thesis with adaptive supertagging (AST) and reverse AST (Rev) on the test set. We show our baseline model (Clark and Curran 2007; C&C '07), parsing models trained with approximate F1 (DecF1), belief propagation (BP), dual decomposition with softmax-margin (DD+SMM) and belief propagation with softmax-margin (BP+SMM). All BP models were run for at most one iteration, they are slower than the baseline because of the additional overhead of computing the sequence-model probabilities in the combined model. DD models were generally run for at most $q = 25$ but DD-SMM AST was only run for at most $q = 10$ because performance already peaked earlier in the dev set.

	LF	LP	LR	UF	UP	UR
CLL	85.53	85.73	85.33	91.99	92.20	91.77
Petrov I-5	85.79	86.09	85.50	92.44	92.76	92.13
BP	86.45	86.75	86.17	92.60	92.92	92.29
+DecF1	**86.73**	**87.07**	**86.39**	**92.79**	**93.16**	**92.43**
+SA	86.51	86.86	86.16	92.60	92.98	92.23

Table 6.10: Development set performance on automatically assigned POS tags. *Petrov I-5* is based on the parser output of Fowler and Penn (2010); evaluation is based on sentences for which all parsers returned an analysis (1834 sentences for section 00).

	LF	LP	LR	UF	UP	UR
CLL	85.74	85.90	85.58	91.92	92.09	91.75
Petrov I-5	86.01	86.29	85.73	92.34	92.64	92.04
BP	86.84	87.08	86.61	92.57	92.82	92.32
+DecF1	87.08	87.37	86.78	92.68	93.00	92.37
+SA	**87.20**	**87.50**	**86.90**	**92.76**	**93.08**	**92.44**

Table 6.11: Test set performance on automatically assigned POS tags (cf. Table 6.10); evaluation is based on sentences for which all parsers returned an analysis (2323 sentences for section 23).

speed and accuracy for each of the two groups. BP+SMM is closely followed by a softmax-margin-trained dual decomposition model (DD+SMM), which performs well in the reverse setting but which is significantly slower in the AST setting. The graph also clearly shows how the F-measure loss-trained parser (DecF1) improves significantly over the baseline (C&C), particularly so in the reverse setting.

6.4 Conclusions

In this chapter we demonstrated that training the individual components of our integrated supertagging and parsing model towards task-specific metrics improves accuracy on both tasks. The softmax-margin objective is simple and also very effective in training individual log-linear parsers. We have shown that it is possible to compute exact sentence-level losses under standard parsing metrics, not only approximations

(Taskar et al., 2004). This enabled us to analyse the effectiveness of these approximations, and it turns out that they are excellent substitutes for exact loss functions. Indeed, the approximate losses are as easy to use as standard conditional log-likelihood.

Empirically, softmax-margin training improves parsing performance across the board, beating the state-of-the-art CCG parsing model of Clark and Curran (2007) by up to 0.9% labelled F-measure. This method does not sacrifice accuracy on other evaluation metrics such as exact match, in fact, we even achieve an improvement of 1.0%/0.7% labelled and unlabelled exact match. It also proves robust, improving our already strong integrated supertagging and parsing model. In particular, we found that for applications where speed is crucial, adaptive supertagging with softmax-margin and belief propagation is most effective, improving accuracy by up to 0.7% labelled F-measure over the baseline C&C parser, while parsing at virtually the same speed of about 100 sentences per second. When accuracy is more important, then the reverse search setting is preferable but improvements come at the cost of lower speed. Using this setting, our final result of 89.3%/94.0% labelled and unlabelled F-measure is the best result reported for CCG parsing accuracy to date, beating the original C&C baseline by up to 1.5%.

Chapter 7

Conclusions and Future Work

Natural language processing is about *modelling* tasks dealing with human language. Modern approaches usually rely on probabilistic models which require *optimisation*. Predicting solutions with the learned models demands efficient *search*. This thesis has attacked all of these areas for the task of parsing with Combinatory Categorial Grammar (CCG; Steedman 2000). We made the following contributions:

- We have introduced a new and principled approach to supertagging and parsing which results in the most accurate CCG parser in the literature to date, raising parsing accuracy on CCGbank from 87.7% to 89.3% labelled F-measure with gold part-of-speech tags.

- We demonstrated that the interaction between supertagging and parsing can be better exploited in an integrated model rather than simply using the supertagger to prune the parser search space.

- We presented the first empirical comparison of dual decomposition and loopy belief propagation on a structured prediction task. We demonstrated that their accuracy is nearly identical in our setting. Moreover, and despite a lack of formal guarantees, we found that the vast majority of initial solutions obtained by belief propagation are exact, whereas the same level of exactness requires many iterations with dual decomposition.

- We presented novel dynamic programs which can compute loss functions that exactly correspond to F-measure. Furthermore, we demonstrated that approximations to exact sentence-level losses are excellent substitutions. These approximate loss functions are as easy to use as standard conditional log-likelihood

while substantially improving accuracy.

- We showed that, while improving efficiency, supertagging can significantly lower the upper bound on parsing accuracy.

- We demonstrated the viability of A* as a search algorithm for parsing with more expressive grammar formalisms such as CCG, and what we believe to be the first evaluation of A* parsing on the more realistic but more stringent metric of CPU time.

7.1 Future Work

An obvious extension to the work in this thesis is to add POS-tagging into our integrated model in order to have a single model for all tasks involved in CCG parsing. POS tagging is another sequence modelling task whose decisions influence both supertagging and parsing. Curran et al. (2006) demonstrated that combined POS-tagging and supertagging substantially increases supertagging accuracy and it is most likely that a model integrating all steps also improves parsing accuracy. Curran et al. (2006) equip the supertagger with real-valued features for the POS-tagger posteriors, although there has been work on combining multiple sequence models using loopy belief propagation (Sutton et al., 2004) in line with our approach. A single model may use these methods for sequence models within our integrated model.

Currently, and for purely practical reasons, the supertagger only models frequent lexical categories with reportedly minimal effect on parsing coverage (Clark and Curran, 2007). One could imagine a principled approach to decide which categories to model by deriving a grammar that is optimal for both parsing and supertagging. Such a method should capture interesting predicate-argument relations but at the same time enforce sparsity over the set of lexical categories. Grammar induction methods from the tree substitution grammar literature such as Cohn et al. (2010) may provide a starting point in this setting.

In Chapter 4 we showed that gains with A* do not come as easily for CCG as for context-free grammars. We experimented with heuristics that were successful for context-free grammars (Klein and Manning, 2003; Pauls and Klein, 2009a) and simple grammar projections that removed different levels of lexicalisation. Future work may explore the idea of collapsing actual CCG categories into coarser classes to obtain grammar projections similar to those of Petrov et al. (2006).

Chapter 6 presented algorithms for optimising F-measure for parsing models. We experimented with sentence-level F-measure and gave the algorithm for computing F-measure augmented expectations on the corpus-level but did not experiment with it. This algorithm requires larger computing resources but with adequate scaling, experimentation is likely to be feasible.

The efficient inference algorithms presented in Chapter 5 could be used to add further higher-order parsing features to our model. Essentially, such features are defined over properties of structures outside the current local sub-tree, such as grandparent-features. This would be the first application of such methods to a CCG parser and would be in line with Smith and Eisner (2008) who demonstrated the viability of loopy belief propagation in a dependency parser, or Koo et al. (2010) who used dual decomposition for the same task.

All presented methods in this thesis are general and future work may apply our integrated parsing and tagging approach to other grammars with large non-terminal sets such as the context-free parser of Petrov et al. (2006). Similar to our setting, this parser has thousands of non-terminals but uses a tree-based coarse-to-fine search. An alternative strategy may be to use a sequence model to select a high-precision set of lexical types to make search more efficient. Furthermore, the two models can be integrated with the methods presented in this thesis. Equally, Lexicalised Tree Adjoining Grammars (Schabes, 1992) have also very large sets of supertags and are a further candidate for integrated supertagging and parsing.

Finally, our algorithms for optimising F-measure carry over to phrase-structure grammars without modification. In this setting, the algorithm simply keeps track of constituents instead of dependencies and thus can be used to optimise constituency-based F-measure, recall or precision.

Bibliography

Ajdukiewicz, K. (1935). Die Syntaktische Konnexität. In McCall, S., editor, *Polish Logic 1920-1939*, pages 207–231. Oxford University Press. Translated from Studia Philosophica, 1, 1-27.

Altun, Y., Johnson, M., and Hofmann, T. (2003). Investigating Loss Functions and Optimization Methods for Discriminative Learning of Label Sequences. In *Proc. of EMNLP*, pages 145–152, Sapporo, Japan.

Auli, M. (2009). CCG-based Models for Statistical Machine Translation. First-Year PhD Report. Available from `http://homepages.inf.ed.ac.uk/s0453934`.

Auli, M. and Lopez, A. (2011a). A Comparison of Loopy Belief Propagation and Dual Decomposition for Integrated CCG Supertagging and Parsing. In *Proc. of ACL*, pages 470–480, Portland, Oregon, USA.

Auli, M. and Lopez, A. (2011b). Efficient CCG Parsing: A* versus Supertagging. In *Proc. of ACL*, pages 1577–1585, Portland, Oregon, USA.

Auli, M. and Lopez, A. (2011c). Training a Log-Linear Parser with Loss Functions via Softmax-Margin. In *Proc. of EMNLP*, pages 333–343, Edinburgh, Scotland, UK.

Baker, J. K. (1979). Trainable Grammars for Speech Recognition. In *Proc. of the Spring Conference of the Acoustical Society of America*, pages 547–550, Boston, MA, USA.

Bangalore, S. and Joshi, A. K. (1999). Supertagging: An Approach to Almost Parsing. *Computational Linguistics*, 25(2):238–265.

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

Bar-Hillel, Y., Perles, M., and Shamir, E. (1964). On Formal Properties of simple Phrase Structure Grammars. *Language and Information: Selected Essays on their Theory and Application*, pages 116–150.

Birch, A., Osborne, M., and Koehn, P. (2007). CCG Supertags in Factored Statistical Machine Translation. In *Proc. of WMT*, pages 9–16, Prague, Czech Republic.

Bos, J. (2005). Towards Wide-Coverage Semantic Interpretation. In *Proc. of IWCS*, pages 42–53, Tilburg, Netherlands.

Bos, J., Clark, S., Steedman, M., Curran, J. R., and Hockenmaier, J. (2004). Wide-Coverage Semantic Representations from a CCG Parser. In *Proc. of COLING*, pages 1240–1246, Geneva, Switzerland.

Bottou, L. (2004). Stochastic learning. In Bousquet, O. and von Luxburg, U., editors, *Advanced Lectures in Machine Learning*, Lecture Notes in Artificial Intelligence, pages 146–168. Springer Verlag, Berlin.

Brodal, G. S. (1996). Worst-case efficient priority queues. In *SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 52–58, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Chang, Y. and Collins, M. (2011). Exact Decoding of Phrase-based Translation Models through Lagrangian Relaxation. In *Proc. of EMNLP*, pages 26–37, Edinburgh, Scotland, UK.

Charniak, E. (2000). A Maximum-Entropy-Inspired Parser. In *Proceedings of the First Conference of the North American Chapter of the Association for Computational Linguistics*, pages 132–139, Seattle, WA, USA.

Chelba, C. and Jelinek, F. (1998). Exploiting Syntactic Structure for Language Modeling. In *Proc. of ACL-COLING*, pages 225–231, Montreal, Canada.

Church, K. and Patil, R. (1982). Coping with Syntactic Ambiguity or How to Put the Block in the Box on the Table. *Computational Linguistics*, 8(3–4):139–149.

Clark, S. (2002). Supertagging for Combinatory Categorial Grammar. In *Proceedings of the 6th International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+6)*, pages 19–24, Venice, Italy.

Clark, S., Copestake, A., Curran, J. R., Zhang, Y., Herbelot, A., Haggerty, J., Ahn, B.-G., Wyk, C. V., Roesner, J., Kummerfeld, J., and Dawborn, T. (2009). Large-Scale Syntactic Processing: Parsing the Web. Final Report of the JHU CLSP Workshop, Johns Hopkins University.

Clark, S. and Curran, J. R. (2003). Log-linear Models for Wide-Coverage CCG Parsing. In *Proc. of EMNLP*, pages 97–104, Sapporo, Japan.

Clark, S. and Curran, J. R. (2004a). Parsing the WSJ using CCG and Log-Linear Models. In *Proc. of ACL*, pages 104–111, Barcelona, Spain.

Clark, S. and Curran, J. R. (2004b). The Importance of Supertagging for Wide-Coverage CCG Parsing. In *Proc. of COLING*, pages 282–288, Geneva, Switzerland.

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

Clark, S. and Hockenmaier, J. (2002). Evaluating a Wide-Coverage CCG Parser. In *Proc. of LREC*, pages 60–66, Las Palmas, Spain.

Clark, S., Hockenmaier, J., and Steedman, M. (2002). Building Deep Dependency Structures with a Wide-Coverage CCG Parser. In *Proc. of ACL*, pages 327–334, Philadelphia, PA, USA.

Cohn, T. (2007). *Scaling Conditional Random Fields for Natural Language Processing*. PhD thesis, University of Melbourne.

Cohn, T., Blunsom, P., and Goldwater, S. (2010). Inducing Tree-Substitution Grammars. *Journal of Machine Learning Research*, 11:3053–3096.

Collins, M. (2003). Head-Driven Statistical Models for Natural Language Parsing. *Computational Linguistics*, 29(4):589–637.

Curran, J. R., Clark, S., and Vadas, D. (2006). Multi-Tagging for Lexicalized-Grammar Parsing. In *Proc. of COLING/ACL*, pages 697–704, Sydney, Australia.

Dantzig, G. B. and Wolfe, P. (1960). Decomposition Principle for Linear Programs. *Operations Research*, 8(1):101–111.

Darroch, J. N. and Ratcliff, D. (1972). Generalized iterative scaling for log-linear models. *Annals of Mathematical Statistics*, 43(5):1470–1480.

Della Pietra, S. A., Della Pietra, V. J., and Lafferty, J. D. (1997). Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:380–393.

DeNero, J. and Macherey, K. (2011). Model-Based Aligner Combination Using Dual Decomposition. In *Proc. of ACL*, pages 420–429, Portland, OR, USA.

Dreyer, M. and Eisner, J. (2009). Graphical Models over Multiple Strings. In *Proc. of EMNLP*, pages 101–110, Suntec, Singapore.

Dyer, C. and Resnik, P. (2010). Context-Free Reordering, Finite-State Translation. In *Proc. of HLT-NAACL*, pages 858–866, Los Angeles, CA, USA.

Dyer, C. J., Muresan, S., and Resnik, P. (2008). Generalizing Word Lattice Translation. In *Proc. of ACL*, pages 1012–1020, Columbus, OH, USA.

Eisner, J. (1996). Efficient Normal-Form Parsing for Combinatory Categorial Grammar. In *Proc. of ACL*, pages 79–86, Santa Cruz, CA, USA.

Felzenszwalb, P. F. and McAllester, D. (2007). The Generalized A* Architecture. *Journal of Artificial Intelligence Research*, 29:153–190.

Finkel, J. R., Kleeman, A., and Manning, C. D. (2008). Feature-based, Conditional Random Field Parsing. In *Proc. of ACL-HLT*, pages 959–967, Columbus, OH, USA.

Finkel, J. R. and Manning, C. D. (2009). Joint Parsing and Named Entity Recognition. In *Proc. of NAACL*, pages 326–334, Boulder, CO, USA.

Finkel, J. R., Manning, C. D., and Ng, A. Y. (2006). Solving the Problem of Cascading Errors: Approximate Bayesian Inference for Linguistic Annotation Pipelines. In *Proc. of EMNLP*, pages 618–626, Sydney, Australia.

Fowler, T. A. D. and Penn, G. (2010). Accurate Context-free Parsing with Combinatory Categorial Grammar. In *Proc. of ACL*, pages 335–344, Uppsala, Sweden.

Galley, M., Graehl, J., Knight, K., Marcu, D., DeNeefe, S., Wang, W., and Thayer, I. (2006). Scalable Inference and Training of Context-Rich Syntactic Translation Models. In *Proc. of ACL*, pages 961–968, Sydney, Australia.

Galley, M., Hopkins, M., Knight, K., and Marcu, D. (2004). What's in a translation rule? In *Proc. of HLT-NAACL*, pages 273–280, Boston, MA, USA.

Gimpel, K. and Smith, N. A. (2010a). Softmax-margin CRFs: Training Log-Linear Models with Cost Functions. In *Proc. of NAACL-HLT*, pages 733–736, Los Angeles, CA, USA.

Gimpel, K. and Smith, N. A. (2010b). Softmax-margin training for structured log-linear models. Technical Report CMU-LTI-10-008, Carnegie Mellon University.

Goodman, J. (1996). Parsing Algorithms and Metrics. In *Proc. of ACL*, pages 177–183, Santa Cruz, CA, USA.

Goodman, J. (1999). Semiring Parsing. *Computational Linguistics*, 25(4):573–605.

Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, USA.

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

Hassan, H., Hearne, M., and Way, A. (2007). Supertagged Phrase-based Statistical Machine Translation. In *Proc. of ACL*, pages 288–295, Prague, Czech Republic.

Hockenmaier, J. (2001). Statistical Parsing for CCG with simple Generative Models. In *Proc. of ACL Student Research Workshop*, pages 7–12, Toulouse, France.

Hockenmaier, J. (2003a). *Data and Models for Statistical Parsing with Combinatory Categorial Grammar*. PhD thesis, University of Edinburgh.

Hockenmaier, J. (2003b). Parsing with Generative Models of Predicate-Argument Structure. In *Proc. of ACL*, pages 359–366, Sapporo, Japan.

Hockenmaier, J. and Steedman, M. (2002). Generative models for statistical parsing with Combinatory Categorial Grammar. In *Proc. of ACL*, Philadelphia, PA, USA.

Hockenmaier, J. and Steedman, M. (2005). CCGbank: User's Manual. Technical report, University of Pennsylvania.

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

Huang, L. (2008). Forest Reranking: Discriminative Parsing with Non-Local Features. In *Proc. of ACL-HLT*, pages 586–594, Columbus, OH, USA.

Ihler, A. T., III, J. W. F., Willsky, A. S., and Chickering, M. (2005). Loopy Belief Propagation: Convergence and Effects of Message Errors. *Journal of Machine Learning Research*, 6:905–936.

Jiang, W., Huang, L., Liu, Q., and Lü, Y. (2008). A Cascaded Linear Model for Joint Chinese Word Segmentation and Part-of-Speech Tagging. In *Proc. of ACL-HLT*, pages 897–904, Columbus, OH, USA.

Joshi, A. K. and Bangalore, S. (1994). Disambiguation of Super Parts of Speech [or Supertags]: Almost Parsing. In *Proc. of COLING*, pages 154–160, Kyoto, Japan.

Kasami, T. (1965). An Efficient Recognition and Syntax Analysis Algorithm for Context-Free Languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA, USA.

Klein, D. and Manning, C. D. (2001). Parsing and Hypergraphs. In *Proceedings of the 7th International Workshop on Parsing Technologies (IWPT-2001)*, pages 123–134, Beijing, China.

Klein, D. and Manning, C. D. (2002). A* Parsing: Fast Exact Viterbi Parse Selection. Technical report, Stanford University.

Klein, D. and Manning, C. D. (2003). A* Parsing: Fast Exact Viterbi Parse Selection. In *Proc. of HLT-NAACL*, pages 119–126, Edmonton, Canada.

Knuth, D. E. (1977). A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6:1–5.

Komodakis, N., Paragios, N., and Tziritas, G. (2007). MRF Optimization via Dual Decomposition: Message-passing revisited. In *Proc. of Int. Conf. on Computer Vision*, pages 1–8, Rio de Janeiro, Brazil.

Koo, T., Rush, A. M., Collins, M., Jaakkola, T., and Sontag, D. (2010). Dual Decomposition for Parsing with Non-Projective Head Automata. In *In Proc. EMNLP*, pages 1288–1298, Cambridge, MA, USA.

Kruijff, G.-J. M., Lison, P., Trevor, B., Jacobsson, H., and Hawes, N. (2007). Incremental, Multi-level Processing for Comprehending Situated Dialogue in Human-Robot Interaction. In Lopes, L. S., Belpaeme, T., and Cowley, S. J., editors, *Symposium on Language and Robots (LangRo 2007)*, Aveiro, Portugal.

Kschischang, F. R., Frey, B. J., and Loeliger, H.-A. (1998). Factor Graphs and the Sum-Product Algorithm. *IEEE Transactions on Information Theory*, 47:498–519.

Kulesza, A. and Taskar, B. (2010). Structured Determinantal Point Processes. In *Proc. of NIPS*, pages 1171–1179, Vancouver, British Columbia, Canada.

Kummerfeld, J. K., Rosener, J., Dawborn, T., Haggerty, J., Curran, J. R., and Clark, S. (2010). Faster Parsing by Supertagger Adaptation. In *Proc. of ACL*, pages 345–355, Uppsala, Sweden.

Kwiatkowski, T., Zettlemoyer, L. S., Goldwater, S., and Steedman, M. (2010). Inducing Probabilistic CCG Grammars from Logical Form with Higher-order Unification. In *Proc. of EMNLP*, pages 1223–1233, Cambridge, MA, USA.

Kwiatkowski, T., Zettlemoyer, L. S., Goldwater, S., and Steedman, M. (2011). Lexical Generalization in CCG Grammar Induction for Semantic Parsing. In *Proc. of EMNLP*, pages 1512–1523, Edinburgh, Scotland, UK.

Lafferty, J., McCallum, A., and Pereira, F. (2001). Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proc. of ICML*, pages 282–289.

Lopez, A. (2009). Translation as Weighted Deduction. In *Proc. of EACL*, pages 532–540, Athens, Greece.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):314–330.

Martins, A. F. T., Smith, N. A., Xing, E. P., Aguiar, P. M. Q., and Figueiredo, M. A. T. (2010). Turbo Parsers: Dependency Parsing by Approximate Variational Inference. In *Proc. of EMNLP*, pages 33–44, Cambridge, MA, USA.

McAllester, D. (1999). On the Complexity Analysis of Static Analyses. In *Proc. of Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 312–329, Venice, Italy.

McAllester, D., Collins, M., and Pereira, F. (2008). Case-factor Diagrams for Structured Probabilistic Modeling. *Journal of Computer and System Sciences*, 74(1):84–96.

McCallum, A., Freitag, D., and Pereira, F. (2001). Maximum Entropy Markov Models for Information Extraction and Segmentation. In *Proc. of ICML*, pages 591–598, Stanford, CA, USA.

McDonald, R. (2006). *Discriminative Learning and Spanning Tree Algorithms for Dependency Parsing*. PhD thesis, University of Pennsylvania.

Mi, H., Huang, L., and Liu, Q. (2008). Forest-Based Translation. In *Proc. of ACL-HLT*, pages 192–199, Columbus, OH, USA.

Nasr, A. and Rambow, O. (2004). Supertagging and Full Parsing. In *Proc. of TAG*, pages 56–63, Vancouver, British Columbia, Canada.

Nivre, J., Hall, J., and Nilsson, J. (2006). MaltParser: A Data-Driven Parser-Gnerator for Dependency Parsing. In *Proc. of LREC*, pages 2216—2219.

Och, F. J. (2003). Minimum Error Rate Training in Statistical Machine Translation. In *Proc. of ACL*, pages 160–167, Sapporo, Japan.

Pauls, A. and Klein, D. (2009a). Hierarchical Search for Parsing. In *Proc. of HLT-NAACL*, pages 557–565, Boulder, CO, USA.

Pauls, A. and Klein, D. (2009b). k-best A* Parsing. In *Proc. of ACL-IJCNLP*, pages 958–966, Suntec, Singapore.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, CA, USA.

Pereira, F. C. N. and Warren, D. H. D. (1983). Parsing as Deduction. In *Proc. of ACL*, pages 137–144, Cambridge, MA, USA.

Petrov, S., Barrett, L., Thibaux, R., and Klein, D. (2006). Learning Accurate, Compact, and Interpretable Tree Annotation. In *Proc. of ACL*, pages 433–440, Sydney, Australia.

Povey, D. and Woodland, P. (2002). Minimum Phone Error and I-smoothing for Improved Discriminative Training. In *Proc. of ICASSP*, pages 105–108, Orlando, FL, USA.

Ratnaparkhi, A. (1996). A maximum entropy model for part-of-speech tagging. In *Proc. of EMNLP*, pages 133–142.

Riedel, S. and Clarke, J. (2006). Incremental Integer Linear Programming for Non-projective Dependency Parsing. In *Proc. of EMNLP*, pages 129–137, Sydney, Australia.

Riedel, S. and McCallum, A. (2011). Fast and Robst Joint Models for Biomedical Event Extraction. In *Proc. of EMNLP*, pages 1–12, Edinburgh, Scotland, UK.

Roark, B. and Hollingshead, K. (2009). Linear Complexity Context-Free Parsing Pipelines via Chart Constraints. In *Proc. of HLT-NAACL*, pages 647–655, Boulder, CO, USA.

Rush, A. M. and Collins, M. (2011a). A Tutorial on Dual Decomposition and Lagrangian Relaxation for Inference in Natural Language Processing. unpublished.

Rush, A. M. and Collins, M. (2011b). Exact Decoding of Syntactic Translation Models through Lagrangian Relaxation. In *Proc. of ACL*, pages 72–82, Portland, Oregon, USA.

Rush, A. M., Sontag, D., Collins, M., and Jaakkola, T. (2010). On Dual Decomposition and Linear Programming Relaxations for Natural Language Processing. In *In Proc. EMNLP*, pages 1–11, Cambridge, MA, USA.

Sarkar, A. (2000). Practical Experiments in Parsing using Tree Adjoining Grammars. In *Proc. of TAG+5*, pages 193–198, Paris, France.

Sarkar, A. (2010). Combining Supertagging with Lexicalized Tree-Adjoining Grammar Parsing. In Bangalore, S. and Joshi, A., editors, *Supertagging: Using Complex Lexical Descriptions in Natural Language Processing*. MIT Press, Boston, MA, USA.

Sato, T. (2007). Inside-Outside Probability Computation for Belief Propagation. In *Proc. of IJCAI*, pages 2605–2610, Hyderabad, India.

Schabes, Y. (1992). Stochastic Lexicalized Tree-Adjoining Grammars. In *Proc. of CL*, pages 425–432, Nantes, France.

Sha, F. and Saul, L. K. (2006). Large Margin Hidden Markov Models for Automatic Speech Recognition. In *Proc. of NIPS*, pages 1249–1256, Vancouver, British Columbia, Canada.

Shieber, S. M., Schabes, Y., and Pereira, F. C. N. (1995). Principles and Implementation of Deductive Parsing. *Journal of Logic Programming*, 24(1–2):3–36.

Smith, D. A. and Eisner, J. (2008). Dependency Parsing by Belief Propagation. In *Proc. of EMNLP*, pages 145–156, Honolulu, Hawaii, USA.

Smith, N. A. (2006). *Novel Estimation Methods for Unsupervised Discovery of Latent Structure in Natural Language Text*. PhD thesis, Johns Hopkins University.

Smyth, P., Heckerman, D., and Jordan, M. (1997). Probabilistic Independence Networks for Hidden Markov Probability Models. *Neural computation*, 9(2):227–269.

Sontag, D., Globerson, A., and Jaakkola, T. (2010). Introduction to Dual Decomposition. In Sra, S., Nowozin, S., and Wright, S. J., editors, *Optimization for Machine Learning*. MIT Press.

Steedman, M. (2000). *The Syntactic Process*. MIT Press, Cambridge, MA, USA.

Sutton, C. and McCallum, A. (2005a). Joint Parsing and Semantic Role Labelling. In *Proc. of CoNLL*.

Sutton, C. and McCallum, A. (2005b). Piecewise Training for Undirected Models. In *Conference on Uncertainty in Artificial Intelligence*, pages 568–575, Edinburgh, Scotland, UK.

Sutton, C. and McCallum, A. (2009). Piecewise Training for Structured Prediction. *Machine Learning*, 77:165–194.

Sutton, C. and McCallum, A. (2011). An Introduction to Conditional Random Fields. *Foundations and Trends in Machine Learning. To appear.*

Sutton, C., Rohanimanesh, K., and McCallum, A. (2004). Dynamic Conditional Random Fields: Factorized Probabilistic Models for Labeling and Segmenting Sequence Data. In *Proc. of ICML*.

Taskar, B., Klein, D., Collins, M., Koller, D., and Manning, C. (2004). Max-Margin Parsing. In *Proc. of EMNLP*, pages 1–8, Barcelona, Spain.

Vijay-Shanker, K., Weir, D., and Joshi, A. K. (1987). Characterizing structural descriptions produced by various grammatical formalisms. In *Proc. of ACL*, pages 104–111, Stanford, CA, USA.

Viterbi, A. J. (1967). Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269.

White, M. and Baldridge, J. (2003). Adapting Chart Realization to CCG. In *Proc. of the Ninth European Workshop on Natural Language Generation*, Budapest, Hungary.

Wood, M. M. (1993). *Categorial Grammar*. Linguistic Theory Guides. Routledge, London.

Yedidia, J., Freeman, W., and Weiss, Y. (2001). Generalized Belief Propagation. In *Proc. of NIPS*, pages 689–695, Vancouver, British Columbia, Canada.

Zettlemoyer, L. S. and Collins, M. (2007). Online Learning of Relaxed CCG Grammars for Parsing to Logical Form. In *Proc. of EMNLP-CoNLL*, pages 678–687, Prague, Czech Republic.

Zhang, Y. and Clark, S. (2011). Shift-Reduce CCG Parsing. In *Proc. of ACL*, pages 683–692, Portland, OR, USA.