Lagrangian Relaxation for Natural Language Decoding

by

Alexander M. Rush

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Author ..... Signature redacted
Department of Electrical Engineering and Computer Science
May 21, 2014

Certified by ... Signature redacted .....
Michael Collins
Visiting Associate Professor, MIT
Vikram S. Pandit Professor of Computer Science, Columbia University
Thesis Supervisor

Accepted by ..... Signature redacted .....
Leslie A. Kolodziejski
Chair of the Committee on Graduate Students

Lagrangian Relaxation for Natural Language Decoding

by

Alexander M. Rush

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2014, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

The major success story of natural language processing over the last decade has been the development of high-accuracy statistical methods for a wide-range of language applications. The availability of large textual data sets has made it possible to employ increasingly sophisticated statistical models to improve performance on language tasks. However, oftentimes these more complex models come at the cost of expanding the search-space of the underlying decoding problem. In this dissertation, we focus on the question of how to handle this challenge. In particular, we study the question of decoding in large-scale, statistical natural language systems. We aim to develop a formal understanding of the decoding problems behind these tasks and present algorithms that extend beyond common heuristic approaches to yield optimality guarantees.

The main tool we utilize, Lagrangian relaxation, is a classical idea from the field of combinatorial optimization. We begin the dissertation by giving a general background introduction to the method and describe common models in natural language processing. The body of the dissertation consists of six chapters. The first three chapters discuss relaxation methods for core natural language tasks : (1) examines the classical problem of parsing and part-of-speech tagging; (2) addresses the problem of language model composition in syntactic machine translation; (3) develops efficient algorithms for non-projective dependency parsing. The second set of chapters discuss methods that utilize relaxation in combination with other combinatorial techniques: (1) develops an exact beam-search algorithm for machine translation; (2) uses a parsing relaxation in a coarse-to-fine cascade. At the core of each chapter is a relaxation of a difficult combinatorial problem and the implementation of this algorithm in a large-scale system.

Thesis Supervisor: Michael Collins
Title: Visiting Associate Professor, MIT
Vikram S. Pandit Professor of Computer Science, Columbia University

Acknowledgments

Michael Collins was a wonderful advisor. I first took Mike's course as a undergraduate almost a decade ago, and I remember being transfixed by his combination of intellectual curiosity and formal rigor. As an advisor, he has displayed these same traits, encouraging me to explore exciting ideas while pushing me towards clear and rigorous explanation. I would also like to thank the other members of my committee, Regina Barzilay and Tommi Jaakkola, who both played a crucial role in shaping my interests during the time I spent at MIT. I am grateful as well to David Sontag and Terry Koo who acted as invaluable mentors in my first year of graduate school. Throughout graduate school, I have worked closely with my office-mate Yin-Wen Chang, and at Columbia, I have had the fortunate chance to further collaborate with Shay Cohen, Karl Stratos, Avner May, and Andrei Simion.

I benefitted greatly from experience as an intern during graduate school. Liang Huang hosted me at USC ISI and was a strong influence on the way I think about natural language. It was also a great opportunity for many interesting conversations with David Chiang and Kevin Knight. Slav Petrov hosted me at Google Research in New York and has since been a selfless mentor, advocate, and source of advice. Google was also an opportunity for useful discussions with many other researchers including Ryan McDonald, Hao Zhang, Michael Ringgaard, Terry Koo, Keith Hall, Kuzman Ganchev, Andre Martins, Yoav Goldberg, and Oscar Tackstrom.

I would also like to acknowledge my undergraduate advisor, and future colleague, Stuart Shieber for introducing me to the area of Natural Language Processing, and for never letting me know how difficult the problems I wanted to work on really were.

Thanks to my parents for surrounding me with both books and Basic and never making much of a distinction, my brother Noah for reminding me about the pleasure of a porch and a guitar, and Erica for making it so that even doing laundry is fun.

This dissertation is dedicated to Pop – these are the gizmos I've been playing with.

Contents

List of Mathematical Symbols

General

$\mathbb{R}$ the real numbers.

$\mathbb{Z}$ the natural numbers.

$\delta(\mathbf{i})$ indicator vector d with $d_i = 1$ and $d_j = 0$ for all $j \neq i$.

Decoding

$f(y)$, $g(z)$ scoring function for derivations, by convention $f : \mathcal{Y} \mapsto \mathbb{R}$ and $g : \mathcal{Z} \mapsto \mathbb{R}$.

$\mathcal{I}$, $\mathcal{J}$ the index set for derivations.

$\mathcal{Y} \subset \{0,1\}^{\mathcal{I}}$, $\mathcal{Z} \subset \{0,1\}^{\mathcal{J}}$ the set of feasible derivation structures.

$\theta \in \mathbb{R}^{\mathcal{I}}$, $\omega \in \mathbb{R}^{\mathcal{J}}$ the weight parameter of the scoring function, for f and g respectively.

Lagrangian Relaxation

$L(\lambda)$ the Lagrangian dual function.

A, C, b the matrices and vector used for linear constraints , $Ay = b$ or $Ay + Cz = b$.

$\lambda \in \mathbb{R}^p$ the vector of dual variables.

Subgradient Method

K the constant limit on number of subgradient descent iterations.

$\alpha_1 \ldots \alpha_K$ the rate parameters for subgradient descent.

Hypergraph

$\mathcal{V}$, $\mathcal{E}$ the set of vertices and hyperedges.

$\mathcal{V}^{(n)}$, $\mathcal{V}^{(t)}$ the non-terminal and terminal vertices respectively.

$h(e)$, $t(e)$ the head vertex and tail vertices of a hyperedge.

$v_0 \in \mathcal{V}$ the distinguished root vertex.

$\mathcal{Y}$ the set of valid hyperpaths.

Finite-State Automata

Σ the set of symbols in a finite-state automaton.

$\mathcal{F} \subset \mathcal{Q}$ the set of final states in a finite-state automaton.

$w \in \mathbb{R}^{\mathcal{Q} \times \Sigma}$ the transition weights.

$q_0 \in \mathcal{Q}$ a distinguished initial state.

$\mathcal{Q}$ the set of states in a finite-state automaton.

$\tau : \mathcal{Q} \times \Sigma \mapsto \mathcal{Q}$ the state transition function.

Dependency Parsing

* the special root symbol for dependency parsing.

$h \in \{0 \dots n\}$, $m \in \{1 \dots n\}$ naming convention for head and modifier indices.

$y_{|h}$, $z_{|h}$ the modifiers of head index h.

`<d>`, `</d>` boundary markers for head automata.

Part-of-Speech Tagging

`<t>`, `</t>` boundary markers for start and end of tag sequence.

$\mathcal{T}$ the set of part-of-speech tags.

Context-Free Grammars

$\mathcal{N}$ The set of non-terminal symbols in a CFG.

$\mathcal{G}$ the set of context-free rules.

Σ the set of terminal symbols in a CFG.

$n_0 \in \mathcal{N}$ the root symbol for the context-free grammar

$w \in \mathbb{R}^{\mathcal{G}}$ the weight vector for context-free rules.

Phrase-Based Translation

$\mathcal{P}$ the set of translation phrases.

Language Modeling

`<s>`, `</s>` boundary markers for language modeling.

Σ the lexicon.

Chapter 1

Introduction

In recent years, statistical methods have become the central tool of natural language processing (NLP). The wide availability of large textual data sets has made it possible to apply these methods to many core tasks. Broadly defined these techniques make up *statistical natural language processing*, and their application has led to state-of-the-art systems for many natural language challenges including performing machine translation, predicting syntactic structure, and extracting information from text.

The methods discussed in this thesis all fall into the category of statistical NLP. From a high-level, we will think of a statistical natural language system as having three components. First, at its core, there is a statistical *model* prescribing the structure of the underlying language phenomenon. Second, there is an *estimation* method for selecting parameters for the model based on available data. Finally we can make *predictions* that utilize these parameters for unseen examples.

While we will often utilize statistical models and estimation methods, the focus of this thesis is primarily on the prediction step. In particular, for a given input example we will focus on the problem of predicting the optimal output structure based on the model parameters. In NLP, this problem is often referred to as *decoding*.

More concretely the focus will be on solving combinatorial optimization problems of the form

$$y^* = \arg\max_{y \in \mathcal{Y}} f(y)$$

where

- $\mathcal{Y}$ is a discrete set of possible structures.
- f is a scoring function estimated from statistical evidence.
- y^* is the highest-scoring possible output structure.

To make this setup concrete, consider the task of machine translation. The decoding problem is defined for an input sentence in a foreign language. The set $\mathcal{Y}$ encodes all possible translations for this sentence. The function f scores each translation under a set of parameters that are estimated from data. And finally the output structure y^* represents the highest-scoring translation.

However, while it is often straightforward to define the decoding problem for a natural language task, it can often be very difficult to solve in practice. Consider the following examples of core problems in natural language processing:

A Statistical Model of Syntax: Dependency Parsing The goal of statistical dependency parsing is to recover the grammatical structure of a sentence in the form of dependency relationships. Each relationship is represented as a directed arc from one word in the sentence to its modifier; together the arcs form a complete parse. Figure 1-1(a) shows an example parse structure. Given an input sentence, a parser aims to produce the highest-scoring set of dependency relationships. The score of the parse is based on a scoring function f that is estimated from a large set of annotated sentences.

Dependency parsing is a critical first step for many natural language applications; however finding the highest-scoring dependency parse can be quite difficult. There are an exponential number of possible dependency structures in the length of the sentence, and for some commonly-used scoring functions finding the best structure is NP-hard. Figure 1-1(b) illustrates a set of possible output structures for a given sentence.

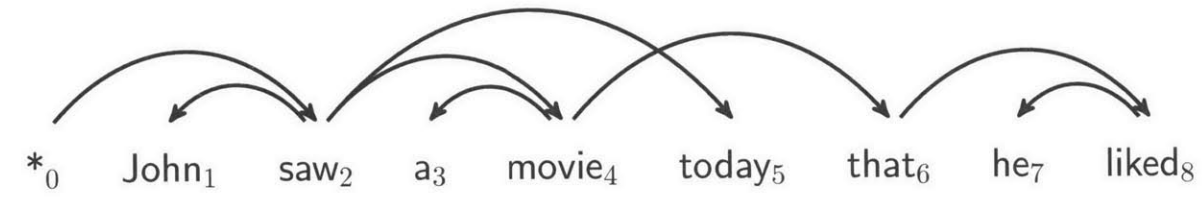
We will return to the problem of dependency parsing several times in this work, particularly the problem of finding the highest-scoring dependency parse for various models. We will see that different variants of the problem have different decoding challenges and require different types of decoding algorithms.

A Statistical Model of Translation: Syntax-Based Translation There are several different statistical models used for machine translation, each with different linguistic and computational properties. One model we will discuss in depth is known as *syntax-based machine translation*. While some models directly map the words of a source sentence to a target-language translation, *syntax-based translation* instead maps between the languages at the level of the syntactic structure. Figure 1-2(a) shows an example of a translation under a syntax-based model of translation known as Hiero (Chiang, 2007). This structure represents a *synchronous derivation*.

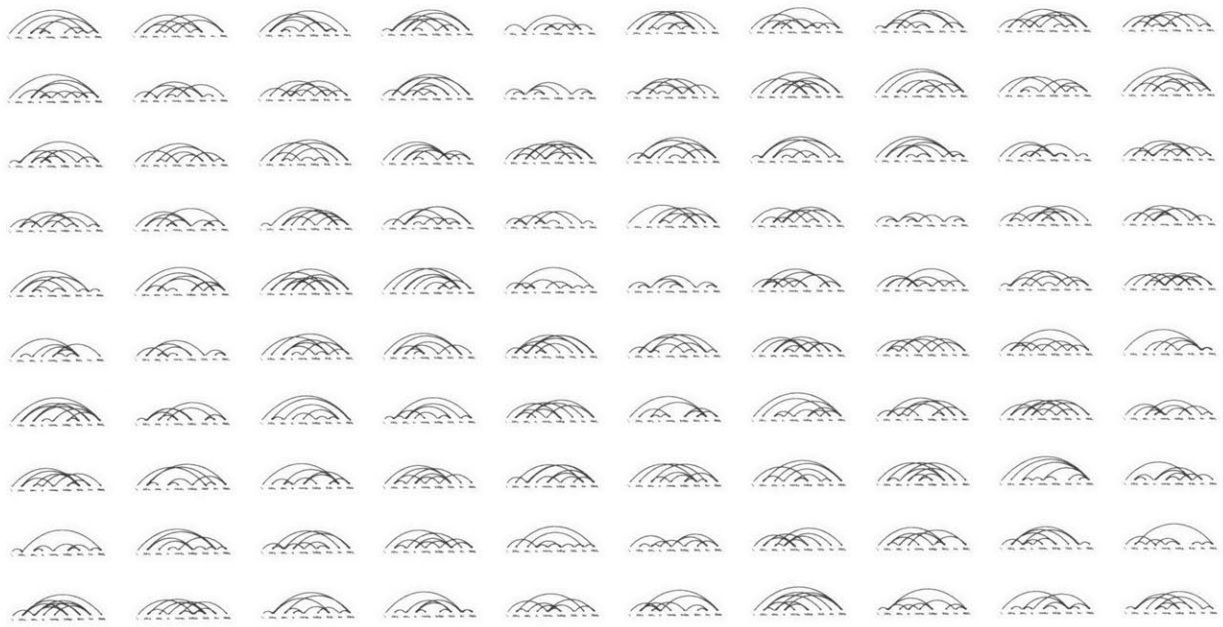
As with dependency parsing, finding the best synchronous derivation can be quite difficult in practice. A common strategy is to implicitly build up a *translation forest* which contains all possible synchronous derivations for a given sentence. This data structure can then be used to decode the highest-scoring translation. Figure 1-2(b) shows an example translation forest for an input sentence under a relatively simple model of translation.

In practice, these translation forests can become very large which makes exhaustive decoding quite slow. To handle this problem, researchers often employ heuristic algorithms to try to find a relatively high-scoring feasible translation from a forest.

A Statistical Model of Speech: Speech Recognition A final example is a model of speech recognition. Unlike the previous two examples, in speech processing the input does not consist of a sentence but of a speech signal, shown in Figure 1-3 as a *spectrogram*. The aim of speech recognition is to produce the highest-scoring transcription of the input signal into a sentence. To decode, we partition the signal into labeled segments known as *phonemes* which are then be converted to characters and to words. The parameters of this conversion come from a statistical model estimated from a large collection of text.

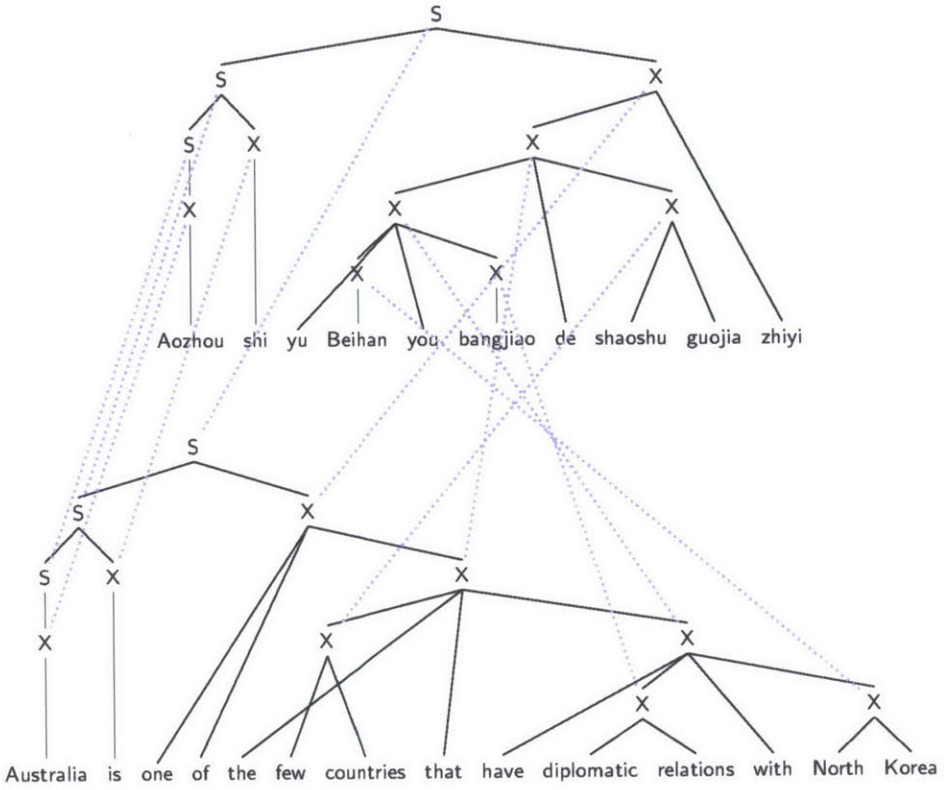


(a)

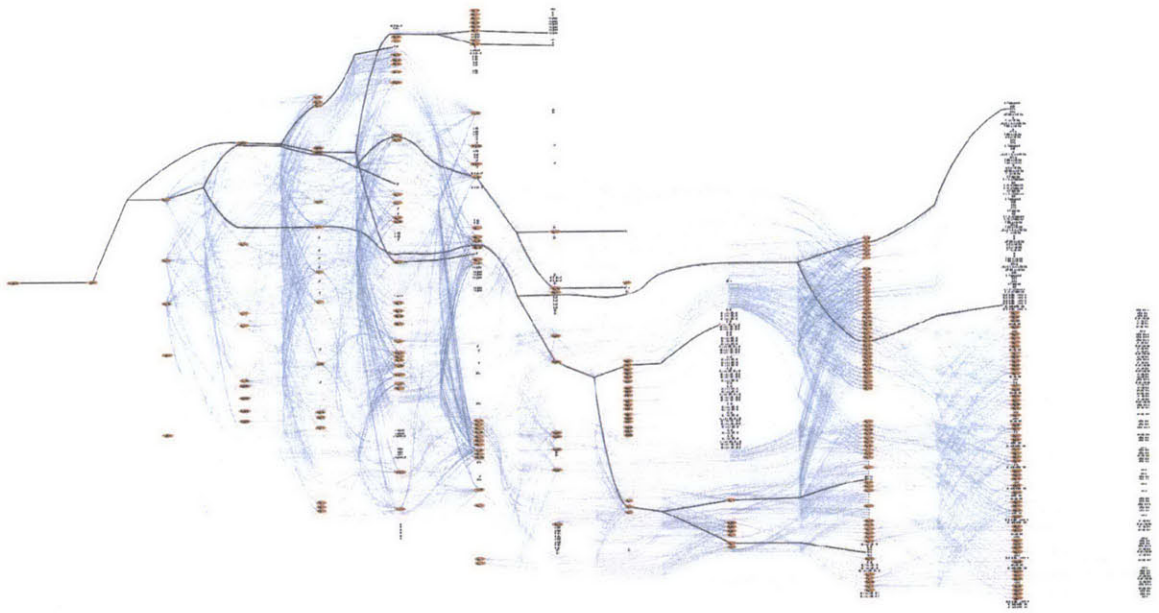


(b)

Figure 1-1: (a) An example sentence and its correct dependency parse. (b) A subset of the possible dependency parses for the example sentence.

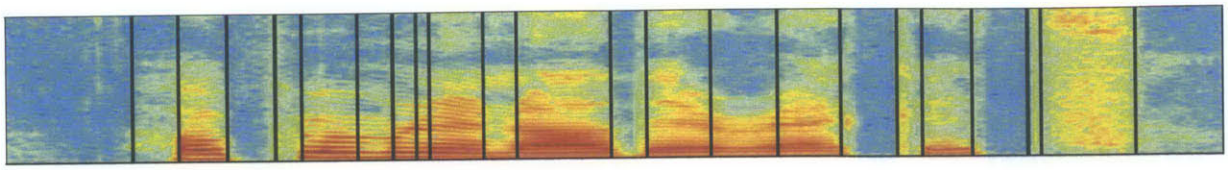


(a)

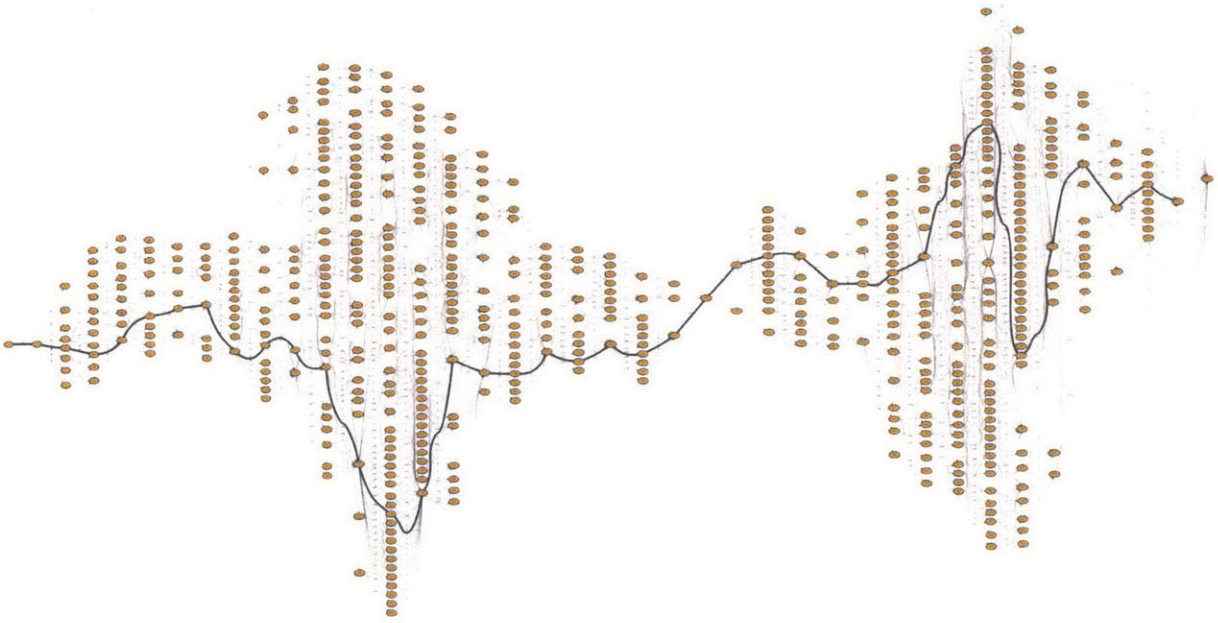


(b)

Figure 1-2: (a) An example of a synchronous derivation from Chinese-to-English from the Hiero translation system (Chiang, 2007). (b) A translation forest encoding all valid synchronous derivation for an input sentence in a Chinese-to-English translation task. The darker lines represent a single synchronous derivation from the set.



(a)



(b)

Figure 1-3: (a) A spectrogram representing an input speech signal. (b) A finite-state automaton encoding all feasible conversions from phonemes to English characters.

However even if we have the correct segmentation into phonemes there is still a large set of feasible sentences. Choosing the highest-scoring sentence can be a challenging decoding problem. Figure 1-3 shows the finite-state automaton (FSA) for this problem. Each path in the FSA represents one conversion from the given phonemes to characters. The goal is to find the highest-scoring conversion for this signal, as shown as a path through this automaton.

Challenges of Decoding When building natural language systems for problems like parsing, translation and speech recognition, we will be interested in three important criteria. We would like to have: (1) rich models of language that incorporate underlying linguistic struc-

ture, e.g. incorporating syntax into translation; (2) optimal predictions under these models, i.e. finding y^*; and (3) computationally efficient decoding.

Of course in practice it is difficult to satisfy all of these criteria. We will see that for many problems there is an inherent trade-off. As we develop richer, more accurate models of language we often greatly increase the underlying decoding complexity. To handle this issue, many large-scale natural language systems either use simple models of language or employ approximate inference.

However, even when decoding is very challenging in the worst-case, in practice we are interested in a specific subset of optimization problems, particularly those for real-world language instances under statistical models. For specific instances of these problems we may be able to efficiently find provably optimal solutions, even though the underlying combinatorial problem is challenging.

Lagrangian Relaxation The main tool we employ to confront this problem is a classical methodology from combinatorial optimization known as Lagrangian relaxation. This approach was first introduced by Held and Karp (1971) in a seminal series of papers focusing on the traveling-salesman problem.

The high-level idea of Lagrangian relaxation is to decompose a difficult decoding problem into one or more simpler problems that can be solved exactly using efficient algorithms. Under certain conditions, iteratively solving these simpler problems may give an optimal solution to the original problem. If the subproblems are chosen such that they can be solved very efficiently, this technique can give much faster algorithms for challenging decoding problems.

We will see across many empirical examples that this method has several important benefits:

- The underlying combinatorial algorithms employed are simple to implement and well-understood.
- The complete implementation can be very efficient: significantly faster than off-the-

shelf solvers and comparable in speed to heuristic algorithms.

- Unlike heuristic algorithms commonly used for decoding, this method can yield a *certificate of optimality* proving that it has produced an optimal solution.

In the coming chapters, we show how to use this classical method from combinatorial optimization in order to solve challenging decoding problems from modern statistical natural language processing. While the methodology is quite general, we will see that often the effectiveness of the technique depends on choosing relaxations that exploit the underlying structure of the particular natural language decoding problem. This relationship between the linguistic structure of the task of interest and the combinatorial nature of the decoding problem and relaxations forms the main theme of the work.

1.1 Overview

The thesis consists of the following chapters:

Part I: Preliminaries

Natural Language Decoding We begin by providing background for standard NLP decoding. This chapter covers: the combinatorial form of decoding, lattice decoding and the Viterbi algorithm, and general dynamic programming decoding with hypergraphs. It also includes examples from part-of-speech tagging, dependency parsing, and context-free parsing.

Theory of Lagrangian Relaxation We continue the background by describing the theory of Lagrangian relaxation. This chapter provides the formal background of Lagrangian relaxation as well as an important variant known as dual decomposition.

Part II: Relaxation Methods

Joint Syntactic Decoding This chapter explores methods for combining decoding problems. In particular we look at two classical joint problems in NLP decoding: joint part-of-speech tagging and syntactic parsing, and joint context-free parsing and dependency parsing. We show that dual decomposition can easily be used to combine different systems, resulting in a simple algorithm that can be significantly faster than classical approaches.

Syntax-Based Translation In addition to combining multiple problems, Lagrangian relaxation can also be applied to simplify difficult combinatorial problems. This chapter focuses on applying this method to the problem of syntax-based machine translation. We introduce a simple unconstrained version of the problem that effectively decouples two parts of the underlying model. We show that this version can be solved very efficiently and that it enables us to find optimal solutions to the original problem in practice.

Non-Projective Dependency Parsing We next turn to the problem higher-order non-projective dependency parsing, an important model in NLP that is NP-hard to decode. We develop a simple relaxation for this problem, involving standard NLP algorithms, and show how it fits in a dual decomposition algorithm. This method allows us to efficiently solve the problem and give guarantees of optimality for the vast majority of examples.

Part III: Relaxation Variants

Beam Search This chapter focuses on a standard NLP heuristic technique known as beam search. Generally beam search produces inexact solutions, but by combining it with Lagrangian relaxation we show that can find provably exact solutions much more efficiently than general-purpose solvers. We apply this method to two variants of translation decoding: phrase-based and syntax-based translation.

Projective Dependency Parsing Projective dependency parsing is a form of dependency parsing that is polynomial-time to decode even for higher-order models. However,

the standard exact algorithms for this task can be too slow in practice for large-scale data sets. We consider decoding using a linear-time relaxation of the problem that is orders of magnitude faster than standard decoding. While this algorithm is not exact, the relaxation has many benefits over heuristic approximations, and it is justified by empirical data. In practice this method produces major speed-ups while still maintaining very high accuracy.

We conclude by summarizing the results presented in these chapters and discussing future directions for the methods described in this thesis.

The aim of this chapter is to present a formal background for the topics discussed in the thesis. We also give some basic example problems that are central to natural language inference. For readers interested specifically in applications, this chapter can be safely skipped in favor of more concrete presentation later in the work and consulted as a reference. The following background topics are covered

Inference Inference problems in NLP can be represented as combinatorial optimization problems. We introduce global linear models notation for natural langauge inference. This notation is demonstrated for the problem of part-of-speech tagging.

Lagrangian Relaxation Lagrangian relaxation and dual decomposition are the mathematical tools at the heart of all the algorithms in this thesis. We present both these methods for general inference problems.

Hypergraphs Hypergraphs provide a general formalism for accurately and concisely expressing the dynamic programming algorithms, which are commonly used for solving inference problems. We introduce hypergraph notation and present an extended example of context-free parsing with hypergraphs.

Before starting we give a brief overview of notation.

1.2 Summary of Notation

By convention, unless otherwise noted, scalars, vectors, and sequences are written in lowercase or Greek letters, matrices are written in uppercase, and sets are written in script-case, e.g. $\mathcal{X}$. All vectors are assumed to be column vectors. We define finite sequences such as x with the notation, $x_1 \dots x_n$, where n is the length of the sequence. On occasion we use zero-indexed sets $x_0 \dots x_n$ for convenience.

We make heavy use of *index sets*. An index set consists of a discrete set $\mathcal{I}$. A vector $v \in \{0,1\}^{|\mathcal{I}|}$ is the vector representation of a subset of $\mathcal{I}$. For any index $i \in \mathcal{I}$, we define

the notation $v(i) = v_i$. For two vectors $u \in \mathbb{R}^{\mathcal{I}}, v \in \mathbb{R}^{\mathcal{I}}$ the dot-product $u^\top v$ is used interchangably with $\sum_{i \in \mathcal{I}} u(i)v(i)$. Additionally define the special function $\delta(i)$ as an indicator vector with $\delta(i)_i = 1$ and $\delta(i)_j = 0$ for all $j \in \mathcal{I}$ such that $j \neq i$.

In NLP it is common to use models that generate chains of symbols (i.e. words, tags, etc.) and require boundary conditions for start and end elements. When noted, we use the following convention: for a sequence of symbols $x_1 \dots x_n$, define x_i for $i \leq 0$ as a special start marker like `<x>` and x_{n+1} as a special end marker like `</x>`. In particular the markers we use are: for words (`<s>`, `</s>`), for tags (`<t>`, `</t>`), for dependencies (`<d>`, `</d>`).

In pseudocode, we often assume non-parameter arguments (such as graphs) are global variables. In `for` loops, if no order on enumeration is given then any ordering is considered valid.

Part I

Preliminaries

Chapter 2

Natural Language Decoding

Consider the process of an interpreter tasked with translating a sentence from Russian into English. Constructing the translation will involve thinking though the many possible meanings for each word and arranging them into many possible English sentences. As a speaker of both languages, she will have a notion of a good translation, based on some combination of its fidelity to the original Russian and its fluency as a sentence in English. She might further use this knowledge to rank various possible choices by their quality, and then try to select the best one.

The question of modeling this translation process is one researchers have contemplated since the early history of computer science. In a letter written in 1947, the mathematician Warren Weaver likened the translation problem to another core task in computer science:

> ... one naturally wonders if the problem of translation could conceivably be treated as a problem in cryptography. When I look at an article in Russian, I say: "This is really written in English, but it has been coded in some strange symbols. I will now proceed to decode." (Weaver, 1955)

Weaver views the Russian sentence as an encrypted ciphertext hiding within it the original English plaintext. The challenge of an automatic translation system is simply to reverse the encryption process. From Weaver's perspective, the act of automatic translation is an act of *decoding*.

Of course, if it is a code, natural language is quite a challenging one to break. Since the time of Weaver's letter the field of natural language processing has greatly evolved. In modern systems, the parameters of the "cryptography" problem underlying statistical translation are often estimated with millions of human translated documents and billions of words.

However if one looks past the scale of modern systems, the translation problem is not too estranged from Weaver's original formulation. The aim is still simply to find the highest-scoring translation (plaintext) for the input sentence (ciphertext) under a fixed model of the translation process.

In fact, variants of this decoding problem are central to most tasks in modern statistical natural language processing. From a high-level, the aim of decoding will be to find the highest-scoring structure out of a set of feasible possibilities. For each domain, the algorithms we use will look quite different, but the underlying decoding problem will take a common form. For example:

- For part-of-speech tagging, the input consists of a sentence, and the decoding problem is to recover the highest-scoring *sequence of tags* for this sentence.
- For syntactic parsing, the input consists of a sentence, and the decoding problem is to recover the highest-scoring *parse tree* for this sentence.
- For speech recognition, the input consists of a speech signal, and the decoding problem is to recover the highest-scoring *transcription* of this signal to text.

We begin by formally describing decoding as an optimization problem and then look at two concrete decoding examples in more detail.

2.1 Decoding as Optimization

In general, the *decoding problem* can be formulated as a combinatorial optimization problem with respect to a given input. The problem is specified by a set of possible output structures

$\mathcal{Y}$ and a scoring function f. The output structures are given by a combinatorial set $\mathcal{Y} \subset \{0,1\}^{\mathcal{I}}$, where $\mathcal{I}$ is an instance-specific finite index set. The scoring function, $f : \{0,1\}^{\mathcal{I}} \mapsto \mathbb{R}$, maps binary vectors over the index set to real-valued scores.

The aim is to find y^*, the highest-scoring member of the set $\mathcal{Y}$,

$$y^* = \arg\max_{y \in \mathcal{Y}} f(y)$$

Note that in general the number of possible output structures, $|\mathcal{Y}|$, may be exponential in the size of the input, and therefore it is intractable to enumerate and score each member of the set $\mathcal{Y}$.

An important special case of this formulation is the class of *linear decoding problems*, where the scoring function f is linear in its parameters y. For these problems, the scoring function f can be written as

$$f(y;\theta) = \sum_{i \in \mathcal{I}} \theta(i) y(i) = \theta^\top y$$

where parameter vector $\theta \in \mathbb{R}^{\mathcal{I}}$ is a score vector mapping each element in the index set to a score. We will see that it is often useful to specify problems in their linear form.

For linear decoding problems we will use the tuple $(f, \mathcal{I}, \mathcal{Y}, \theta)$ to refer to the scoring function, index set, structure set, and parameter vector of the problem. When there is an alternative inference problem to consider we will use the tuple $(g, \mathcal{J}, \mathcal{Z}, \omega)$ and define the decoding scoring function as

$$g(z) = \sum_{j \in \mathcal{J}} \omega(j) z(j) = \omega^\top z$$

where the terms are each analogous to those defined above.

Next we make these definitions concrete by looking at two important decoding problems in natural language processing: sequence decoding and dependency parsing.

2.1.1 Example 1: Part-of-Speech Tagging

We start with a fundamental problem in natural language processing, predicting the part-of-speech (POS) tags for a sentence. We can predict part-of-speech tags by using sequence decoding. In this task, the input may consist of a sentence like the "The dog walks to the park". The aim is to predict the best part-of-speech tags for each word in this sequence, e.g. "The/D dog/N walks/V to/P the/D park/N". Since the part-of-speech tag predicted for each word will depend on the surrounding tags, we define this problem in terms of decoding the best sequence of tags.

Sequence Decoding Sequence decoding assumes there is a fixed set of possible tags or labels, $\mathcal{T}$, for instance in part-of-speech tagging this set might include `N` (noun), `A` (adjective), `V` (verb), etc. An input instance consists of a sequence of length n, represented by tokens $s_1 \dots s_n$ where for $i \in \{1 \dots n\}$ the token s_i comes from a dictionary Σ, for instance Σ may be the words in a language. The goal of decoding is to find the highest-scoring tag sequence $t_1 \dots t_n$ for this input where for $i \in \{1 \dots n\}$ the tag t_i is in $\mathcal{T}$.

As a first pass, we consider the problem in its general form, i.e. we will allow an arbitrary scoring function over sequences. Define the index set $\mathcal{I}$ to indicate the tag selected for each word,

$$\mathcal{I} = \{(i, t) : i \in \{1 \dots n\}, t \in \mathcal{T}\}$$

Next define the set of output structures in terms of this index set. The set includes a linear constraint that enforces that there is exactly one tag selected per word

$$\mathcal{Y} = \{y \in \{0, 1\}^{\mathcal{I}} : \sum_{t \in \mathcal{T}} y(i, t) = 1 \text{ for all } i \in \{1 \dots n\}\}$$

The decoding problem is to find

$$y^* = \arg\max_{y \in \mathcal{Y}} f(y)$$

Note that under this definition there are $|\mathcal{Y}| = |\mathcal{T}|^n$ possible tag sequences, and we are

assuming an arbitrary scoring function f. Without further assumptions on the structure of f, this decoding problem is intractable to solve.

Trigram Tagging In natural language applications, a common simplification for this problem is to use *trigram tagging*. This model assumes that the score of a tag sequence factors into scores of contiguous triples of tags (t_{i-2}, t_{i-1}, t_i) with respect to an index i.

The trigram tagging problem is defined as

$$\arg\max_{t_1 \in \mathcal{T} \ldots t_n \in \mathcal{T}} \sum_{i=3}^{n} \theta(i, t_{i-2}, t_{i-1}, t_i)$$

where $\theta(i, t_{i-2}, t_{i-1}, t_i)$ is the score for the sub-sequence of tags t_{i-2}, t_{i-1}, t_i ending at position i in the sentence.

The same problem can be written as a linear decoding problem. Define an index set $\mathcal{I}$ as all triples of tags and their position

$$\mathcal{I} = \left\{(i, A, B, C) : i \in \{3 \ldots n\}, A, B, C \in \mathcal{T}\right\}$$

Each element of $\mathcal{I}$ indicates that a triple of tags A, B, C is used at positions $i-2, i-1$, and i respectively. Note that the index starts at position 3 since the element $(3, A, B, C)$ covers the first two words. The trigram scoring function shown above can be defined as a real-valued vector over this index set, $\theta \in \mathbb{R}^{\mathcal{I}}$.

The set of tagging sequences $\mathcal{Y} \subset \{0, 1\}^{\mathcal{I}}$ consists of all valid sequences indexed by trigrams. To formally specify this set we must ensure that the trigrams for a sequence overlap consistently. We take care to explicitly define these constraints as they will become important in the next section. A valid member of $\mathcal{Y}$ must have the following properties:

1. The sequence must include exactly one "start" trigram, $y(3, A, B, C) = 1$ for any $A, B, C \in \mathcal{T}$, and one "end" trigram, $y(n, A, B, C) = 1$ for any $A, B, C \in \mathcal{T}$. Formally these constraints can be written respectively as

$$\sum_{A,B,C \in \mathcal{T}} y(3, A, B, C) = 1, \quad \sum_{A,B,C \in \mathcal{T}} y(n, A, B, C) = 1$$

2. The trigram ending at position i must overlap with the trigram ending at position $i+1$, i.e. if at position i we have $y(i, A, B, C) = 1$ then at position $i+1$ we must have $y(i+1, B, C, D) = 1$ for some $D \in \mathcal{T}$, and vice versa. Formally this constraint is written as

$$\sum_{A \in \mathcal{T}} y(i, A, B, C) = \sum_{D \in \mathcal{T}} y(i+1, B, C, D)$$

for all tags pairs $B, C \in \mathcal{T}$ and positions $i \in \{3 \dots (n-1)\}$.

Putting these properties together gives a set of linear constraints on binary variables for expressing $\mathcal{Y}$,

$$\mathcal{Y} = \{y \in \{0,1\}^{\mathcal{I}} \quad : \quad \sum_{A,B,C \in \mathcal{T}} y(3, A, B, C) = 1,$$

$$\sum_{A,B,C \in \mathcal{T}} y(n, A, B, C) = 1,$$

$$\sum_{A \in \mathcal{T}} y(i, A, B, C) = \sum_{D \in \mathcal{T}} y(i+1, B, C, D) \quad \forall \quad i \in \{3 \dots (n-1)\}, B, C \in \mathcal{T}\}$$

The decoding problem over this set is to find

$$y^* = \arg\max_{y \in \mathcal{Y}} \sum_{i=3}^{n} \sum_{A,B,C \in \mathcal{T}} \theta(i, A, B, C) y(i, A, B, C) = \arg\max_{y \in \mathcal{Y}} \theta^\top y$$

While the size of the combinatorial set $\mathcal{Y}$ is still exponential in n, we will see that for trigram tagging, the highest-scoring member of $\mathcal{Y}$ can be found in $O(n|\mathcal{T}|^3)$ time. This sequence can be found using a dynamic programming algorithm known as the Viterbi algorithm, which can be derived directly from this constrained representation of $\mathcal{Y}$. This idea is explored in detail in Section 2.2.

Example 2.1.1 (A Valid Sequence). Consider an example input sentence `The dog walks in the park` with $n = 6$, and a simple tag set $\mathcal{T} = \{\texttt{D}, \texttt{N}, \texttt{V}, \texttt{P}\}$ corresponding to determiner, noun, verb, and preposition respectively.

With this tag set, one possible setting for $t_1 \ldots t_6$ is $t = \texttt{D N V P D N}$. For trigram tagging, the corresponding vector representation of $y \in \mathcal{Y}$ has the following non-zero values:

$$\begin{aligned} y(3, \ \texttt{D}, \ \texttt{N}, \ \texttt{V}) &= 1 \\ y(4, \ \texttt{N}, \ \texttt{V}, \ \texttt{P}) &= 1 \\ y(5, \ \texttt{V}, \ \texttt{P}, \ \texttt{D}) &= 1 \\ y(6, \ \texttt{P}, \ \texttt{D}, \ \texttt{N}) &= 1 \end{aligned}$$

where all other values are 0. It is straightforward to verify that this vector obeys each of the constraints of $\mathcal{Y}$ since it satisfies the boundary conditions and the trigrams overlap consistently.

Example 2.1.2 (Hidden Markov Model). A simple method for instantiating the trigram scores for trigram tagging is as log-probabilities from a second-order hidden Markov model (HMM). This example gives a brief derivation of this model.

An HMM is a probabilistic model for the joint probability of a tag and token sequence, i.e. if we define random vectors S and T for the tokens and the tags respectively, it models the joint probability

$$P(S_1 = s_1, \ldots S_n = s_n, T_1 = t_1, \ldots, T_n = t_n)$$

A second-order hidden Markov model makes the assumption that this joint probability can be approximated as

$$\begin{aligned} &P(S_1 = s_1, \ldots S_n = s_n, T_1 = t_1, \ldots, T_n = t_n) \ \approx \\ &\prod_{i=1}^{n} P(S_i = s_i | T_i = t_i) P(T_i = t_i | T_{i-2} = t_{i-2}, T_{i-1} = t_{i-1}) \end{aligned}$$

where we define t_i with $i \leq 0$ as a special start symbol `<t>` and $\mathcal{T}' = \mathcal{T} \cup \{\texttt{<t>}\}$.

We use two sets of parameters to model these probability distributions: emission parameters, $\rho \in \mathbb{R}^{\Sigma \times \mathcal{T}}$ for $P(S_i = s_i | T_i = t_i)$ and transition parameters $\tau \in \mathbb{R}^{\mathcal{T} \times \mathcal{T}' \times \mathcal{T}'}$ for $P(T_i = t_i | T_{i-2} = t_{i-2}, T_{i-1} = t_{i-1})$. The complete model has a total of $|\mathcal{T}||\Sigma| + |\mathcal{T} \times \mathcal{T}' \times \mathcal{T}'|$ parameters.

Sequence decoding with a second-order HMM model aims to produce the sequence with the highest probability, also known as maximum a posteriori (MAP) inference. We can write this as an optimization problem over the parameters

$$\underset{t_1 \in \mathcal{T}, \ldots t_n \in \mathcal{T}}{\arg\max} \prod_{i=1}^{n} \rho(s_i | t_i) \tau(t_i | t_{i-2}, t_{i-1})$$

This probabilistic inference problem can be expressed with the general formulation of trigram tagging. Set $\theta \in \mathbb{R}^{\mathcal{I}}$ to be the log-probabilities - the log changes the total score but not the arg max - from this HMM, i.e. for all $(i, A, B, C) \in \mathcal{I}$ and for a fixed input sequence $s_1 \ldots s_n$, let

$$\theta(i, A, B, C) = \log \begin{cases} \rho(s_i | C) \tau(C | A, B) & \text{if } 3 < i \leq n \\ \rho(s_1 | A) \rho(s_2 | B) \rho(s_3 | C) \tau(C | A, B) \tau(B | \texttt{<t>}, A) \tau(A | \texttt{<t>}, \texttt{<t>}) & \text{if } i = 3 \end{cases}$$

With this score vector the MAP inference problem can be written as a linear decoding problem

$$\underset{y \in \mathcal{Y}}{\arg\max} \sum_{(i,A,B,C) \in \mathcal{I}} y(i, A, B, C) \theta(i, A, B, C) = \underset{y \in \mathcal{Y}}{\arg\max}\, \theta^\top y$$

2.1.2 Example 2: Dependency Parsing

Syntactic parsing will be one of the key topics of this thesis. Many natural language applications rely on having a grammatical representation of an underlying sentence. There are several widely-used syntactic representations in NLP, each with different linguistic and computational properties. For a given representation, we will be interested in decoding the best

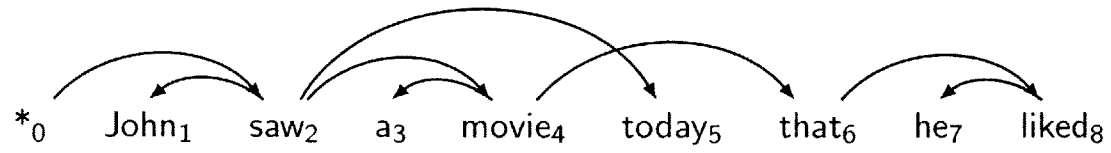


Figure 2-1: An example dependency parse for the sentence `John saw a movie today that he liked`. *Subscripts indicate the corresponding vertex index in $\mathcal{V}$. The symbol $*$ is used to indicate the root of the graph. Dependency arcs are drawn as arrows from the head word to its modifiers. Only the arcs in one directed spanning tree are shown.*

syntactic structure given a statistical model and parameters. In this section we introduce a simple, but widely used syntactic representation known as dependency parsing; Section 2.2.3 introduces another common formalism known as context-free parsing.

For the purpose of this section, a dependency parse is simply made up of a series of arcs that relate each word in a sentence to its modifiers. Figure 2-1 shows an example of a dependency parse over an input sentence. Each arc is said to indicate a *head-modifier* relationship. These arcs may indicate a relationship adhering to one of many different grammatical formalisms that use dependency structures as a (partial) representation; for brevity, we gloss over the underlying formalism and focus on the combinatorial aspects of the problem.

Dependency Trees As with sequence decoding the input to dependency parsing is a sequence of tokens, $s_1 \dots s_n$. The set of possible dependency parse trees is defined with respect to a fully-connected directed graph $\mathcal{D}$ for this sentence with vertices $\mathcal{V}$ and edges $\mathcal{E} = \mathcal{V} \times \mathcal{V}$. The graph is defined to have $\mathcal{V} = \{0 \dots n\}$ vertices, with one vertex for each token $i \in \{1 \dots n\}$ and a special *root* vertex 0. Each directed arc in this graph $(h, m) \in \mathcal{E}$ corresponds to a possible dependency relation indicating that head token s_h is modified by token s_m.

There are two widely-used models of dependency parsing, projective and non-projective. We start with the general non-projective case and later define projectivity. A non-projective dependency parse for a sentence is simply any *directed spanning tree*[1] over the graph $\mathcal{D}$.

[1]In the combinatorial optimization literature these are called *spanning arborescences*. We use the term

By convention, these trees must be rooted at vertex 0. Define the set $\mathcal{Y}$ to be all directed spanning trees for an input sentence.

The spanning tree constraint ensures two crucial syntactic properties: (1) all non-root tokens $m \in \{1 \dots n\}$ are used as a modifier exactly once, and (2) since there are no cycles between the arcs in the sentence, no word can modify its children or descendants.

It is known that there are an exponential number of possible directed spanning trees over a complete graph. Therefore to ensure that decoding is tractable, it is important to make further assumptions about the scoring function or the feasible structures for this problem.

We next look at two common choices for the scoring function f. Both of these scoring functions are based on the work of McDonald (2006). Interestingly, the two scoring functions will be quite similar, but we will see that the first is easy to decode, but the second is very difficult in general. We then discuss another model of the problem, projective dependency parsing, which will have different computational properties.

First-Order Dependency Parsing A first-order model of dependency parsing assumes that the scoring function f factors over each of the arcs in the tree. Since each token must modify exactly one word, we can define a function $h : (\mathcal{Y} \times \{1 \dots n\}) \mapsto \{0 \dots n\}$ to give the head index chosen by y for a given modifier index. For example in the parse tree from Figure 2-1, we have $h(y, 1) = 2$, $h(y, 2) = 0$, $h(y, 3) = 4$ etc. A first-order scoring function factors into terms for each of these head-modifier relationships, i.e.

$$f(y; \theta) = \sum_{m=1}^{n} \theta(h(y, m), m)$$

The linear decoding problem for this scoring function can be defined in terms of a simple index set. Let $\mathcal{I}$ be the index set over arcs defined as

$$\mathcal{I} = \{(h, m) : h \in \{0 \dots n\}, m \in \{1 \dots n\}, h \neq m\}$$

and let $\mathcal{Y} \subset \{0, 1\}^{\mathcal{I}}$ be the set of all directed spanning trees represented with these indices.

directed spanning trees which is more commonly used in natural language processing.

The highest-scoring parse is defined as

$$y^* = \arg\max_{y \in \mathcal{Y}} \sum_{(h,m) \in \mathcal{I}} y(h,m)\theta(h,m) = \arg\max_{y \in \mathcal{Y}} \theta^\top y$$

Now, crucially, because the scores factor over arcs, this optimization problem can be solved with a maximum directed spanning tree algorithm. These algorithms are well-studied and efficient (Chu & Liu, 1965; Edmonds, 1967). We will make direct use of a maximum directed spanning tree algorithm in Chapter 6.

Example 2.1.3 (A Dependency Parse with First-Order Scores). Figure 2-1 shows an example dependency parse for the sentence $\texttt{John}_1$ $\texttt{saw}_2$ $\texttt{a}_3$ $\texttt{movie}_4$ $\texttt{today}_5$ $\texttt{that}_6$ $\texttt{he}_7$ $\texttt{liked}_8$. The main word in the sentence is the verb `saw` which modifies the root symbol `*`. This verb `saw` is in turn modified on its left by the subject `John` and on its right by the direct object `movie` and by `today`. The score of these first four arcs is simply

$$f(y;\theta) = \theta(0,2) + \theta(2,1) + \theta(2,4) + \theta(2,5) + \ldots$$

The full parse shown is represented by a vector y that has the following indices set to 1:

$$\begin{aligned} &y(0,2) = 1, \quad y(2,1) = 1, \quad y(2,4) = 1, \quad y(4,3) = 1, \\ &y(2,5) = 1, \quad y(4,6) = 1, \quad y(6,8) = 1, \quad y(8,7) = 1 \end{aligned}$$

with all other indices set to 0.

Second-Order Dependency Parsing Unfortunately scoring each dependency arc individually, as in a first-order model, can be too strong of a modeling assumption. It has been shown that using higher-order models produces significantly more accurate predictions (McDonald, 2006). (These higher-order models will be the focus of Chapter 6 on non-projective parsing.)

Consider for instance a second-order model of dependency parsing. This model assigns a score to each arc based on the head token, the modifier token, *and* the previous modifier to-

ken, known as the *sibling* modifier. This extra token will give the scoring function additional context for assigning a score.

Define a sibling function $s : (\mathcal{Y} \times \{1 \dots n\}) \mapsto \{0 \dots n\} \cup \{\texttt{NULL}\}$ to give the previous word used as a modifier before the current modifier on the same side of its head. This function is defined as

$$s(y, m) = \begin{cases} \max \left\{ m' : h(y, m) < m' < m, h(y, m') = h(y, m) \right\} & \text{if } m > h(y, m) \\ \min \left\{ m' : h(y, m) > m' > m, h(y, m') = h(y, m) \right\} & \text{if } m < h(y, m) \end{cases}$$

By convention if m is the first modifier on its side, the function returns $s(y, m) = \texttt{NULL}$.

In a second-order model, the scoring vector is based on the sibling as well as the modifier

$$f(y) = \sum_{m=1}^{n} \theta(h(y, m), s(y, m), m)$$

As with trigram tagging, this optimization can be written as a linear decoding problem with an enriched index set. Define a new index set $\mathcal{I}$ as

$$\mathcal{I} = \begin{cases} (h, s, m) : & h \in \{0 \dots n\}, m \in \{1 \dots n\}, h \neq m, \\ & h < s < m \text{ or } m < s < h \text{ or } s = \texttt{NULL} \end{cases}$$

The set has one index for each possible head, modifier, and sibling modifier triple. The full decoding problem is $\arg\max_{y \in \mathcal{Y}} \sum_{(h,s,m) \in \mathcal{I}} y(h, s, m) \theta(h, s, m) = \arg\max_{y \in \mathcal{Y}} \theta^\top y$.

However, there is an important difference when switching to a second-order model of dependency parsing. Unlike the first-order decoding problem which could be solved in polynomial-time, for a scoring function that uses second-order information, the decoding problem is NP-hard. Adding this contextual information makes the model more accurate, at the cost of making decoding much more difficult.

Example 2.1.4 (Second-Order Dependency Parsing). Recall the dependency parse shown in

Figure 2-1. For a second-order model the first several scores in the graph are

$$f(y) = \theta(0, \texttt{NULL}, 2) + \theta(2, \texttt{NULL}, 1) + \theta(2, \texttt{NULL}, 4) + \theta(2, 4, 5) + \ldots$$

Note that for the first three arcs, the modifier chosen was the closest modifier to the head word on its side, and so by convention the sibling is set as NULL. For the fourth arc, there is a closer modifier on its side $\texttt{movie}_4$, which is set as the sibling.
The full vector y has the following indices set to 1:

$$\begin{array}{llll} y(0, \texttt{NULL}, 2) = 1, & y(2, \texttt{NULL}, 1) = 1, & y(2, \texttt{NULL}, 4) = 1, & y(2, 4, 5) = 1, \\ y(4, \texttt{NULL}, 3) = 1, & y(4, \texttt{NULL}, 6) = 1, & y(6, \texttt{NULL}, 8) = 1, & y(8, \texttt{NULL}, 7) = 1 \end{array}$$

All other indices are set to 0.

Projective Dependency Parsing For many languages, we can often make a further assumption about the underlying parse structure. This assumption reduces the number of valid parses and has the added advantage of making the underlying decoding problem significantly easier.

The assumption is that the dependency parse obeys a structural property known as projectivity.

Property 2.1.1 (Projectivity). *For any parse $y \in \mathcal{Y}$ and any vertex $v \in \mathcal{V}$, define the set $\mathcal{A}_v$ as v and its descendants in y*

$$\mathcal{A}_v = \{v\} \cup \{u : \textit{ there is a path from } v \textit{ to } u \textit{ in } y\}$$

A directed spanning tree is **projective** *if for all $v \in \mathcal{V}$ the set $\mathcal{A}_v$ is contiguous, i.e. $\mathcal{A} = \{i \ldots j\}$ for some $0 \leq i \leq j \leq n$.*

This property is also equivalent to having no "crossing" arcs, i.e. if the vertices are positioned in order no arcs will cross.

We will see in the next section assuming projectivity greatly simplifies higher-order decoding. Define the second-order projective dependency parsing problem with the same index set $\mathcal{I}$ as above and define the set $\mathcal{Y}' \subset \{0,1\}^{\mathcal{I}}$ as

$$\mathcal{Y}' = \{y : y \in \mathcal{Y} \text{ and } y \text{ is projective}\}$$

With this new constraint, the decoding problem, $\arg\max_{y \in \mathcal{Y}'} f(y)$, is no longer NP-hard and can be solved using dynamic programming. In fact the problem can be solved exactly in $O(n^3)$ time using a dynamic programming algorithm known as Eisner's algorithm (Eisner & Satta, 1999). We will revisit this algorithm in the next section.

Example 2.1.5 (Non-Projectivity). Consider the example sentence in Figure 2-1. Vertex 2, which corresponds to the word `saw` has descendant set $\mathcal{A}_2 = \{1,2,3,4,5,6,7,8\}$ which is contiguous. However, vertex 4, which corresponds to the word `movie` has descendant set $\mathcal{A}_4 = \{3,4,6\}$ which is non-contiguous and therefore violates projectivity. Another way to see this violation is to draw two of the arcs from this parse, $(4,6)$ and $(2,5)$. These arcs cross each other which indicates non-projectivity.

However, this example should not be taken as typical for English. This construction is an exception for the English language; on the other hand, for certain languages, most notably Czech, non-projectivity is common, and it is important to account for it in practice.

2.2 Hypergraphs and Dynamic Programming

Once a decoding problem has been specified, the aim is to produce the optimal output structure for this problem. Often times the optimal structure can be found by employing a combinatorial algorithm. There are a large collection of different combinatorial algorithms used in natural language decoding; we have already mentioned maximum directed spanning tree algorithms, and we will also discuss several others, for example undirected spanning tree algorithms and matching algorithms.

However, in practice the most widely-used class of decoding algorithms in NLP is the

class of dynamic programming (DP) algorithms. This section focuses on formal background for dynamic programming. For this section we assume the reader has a basic understanding of dynamic programming. We begin by working through an example and then formalizing dynamic programming using a class of directed hypergraphs. This formalism goes back to the work of Knuth (1977) and Martin et al. (1990). Hypergraphs have been used extensively in the NLP literature for parsing and translation (Klein & Manning, 2005; Chiang, 2007; Huang & Chiang, 2005).

Dynamic programming will be used for most of the problems discussed in this thesis including: the Viterbi algorithm for tagging, the CKY algorithm for parsing, and extensions to CKY for syntactic translation. Additionally, Chapter 7 discusses how beam search, used for phrase-based translation, can be viewed within a DP framework.

2.2.1 The Viterbi Algorithm for Sequence Decoding

As an extended example, we return to the sequence decoding problem introduced in Section 2.1.1, and particularly the problem of trigram tagging. Given a token sequence $s_1 \dots s_n$ the goal is to find the best tag sequence $t_1 \dots t_n$, from the set $\mathcal{T}$. We defined an index set for this model as

$$\mathcal{I} = \{(i, A, B, C) : i \in \{3 \dots n\}, A, B, C \in \mathcal{T}\}$$

Next we will consider extending this index set with elements

$$\mathcal{J} = \{(i, A, B) : i \in \{2 \dots n\}, A, B \in \mathcal{T}\}$$

where the new set of tag sequences is $\mathcal{Y} \subset \{0, 1\}^{\mathcal{I} \cup \mathcal{J}}$. We now reintroduce the sequence constraints for $\mathcal{Y}$ in a slightly different form

$$\mathcal{Y} = \left\{ y : \begin{array}{ll} 1 = \sum_{A,B \in \mathcal{T}} y(2, A, B), & \\ y(i-1, B, C) = \sum_{D \in \mathcal{T}} y(i, B, C, D) & \forall i \in \{3 \ldots n\}, B, C \in \mathcal{T}, \\ \sum_{A \in \mathcal{T}} y(i, A, B, C) = y(i, B, C) & \forall i \in \{3 \ldots n\}, B, C \in \mathcal{T}, \\ \sum_{A,B \in \mathcal{T}} y(n, A, B) = 1 & \end{array} \right\}$$

These constraints can be used to derive a graphical representation of the set $\mathcal{Y}$. Define a weighted directed graph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ with weights $\theta \in \mathbb{R}^{\mathcal{E}}$ where

- The set of vertices includes one vertex for each element of $\mathcal{J}$ as well as a source and sink vertex

$$\mathcal{V} = \{\texttt{Source}, \texttt{Sink}\} \cup \mathcal{J}$$

- For each variable $(i, A, B, C) \in \mathcal{I}$, there is an edge from vertex $(i-1, A, B)$ to (i, B, C) in $\mathcal{E}$. The edges have weight $\theta(i, A, B, C)$ where θ is just the score from trigram tagging.

- Additionally there is an edge from `Source` to $(2, A, B)$ for all $A, B \in \mathcal{T}$ and from (n, A, B) to `Sink` for all $A, B \in \mathcal{T}$. These edges have weight equal to 0 in θ.

This directed graph is known as the *lattice* representation of the set $\mathcal{Y}$. Because the word position i increases at each edge, the lattice is also a directed acyclic graph (which will be important for decoding).

Note also that the lattice is defined such that there is a bijection between paths in the graph and elements in $\mathcal{Y}$. Every path in the lattice can be converted to a valid tag sequence, and every tag sequence can be converted into a path.

Example 2.2.1 (A Tagging Lattice). Consider a simple trigram tagging problem. Let the input sentence be `The man walks the dog`, and the set of possible tags be $\mathcal{T} = \{\texttt{D}, \texttt{N}, \texttt{V}\}$. Applying the lattice construction described yields the directed graph $\mathcal{D}$ shown in Figure 2-2. One possible tag sequence $t =$ `D N V D N` is highlighted in the graph. This path corresponds to the sequence with the following variables set to 1

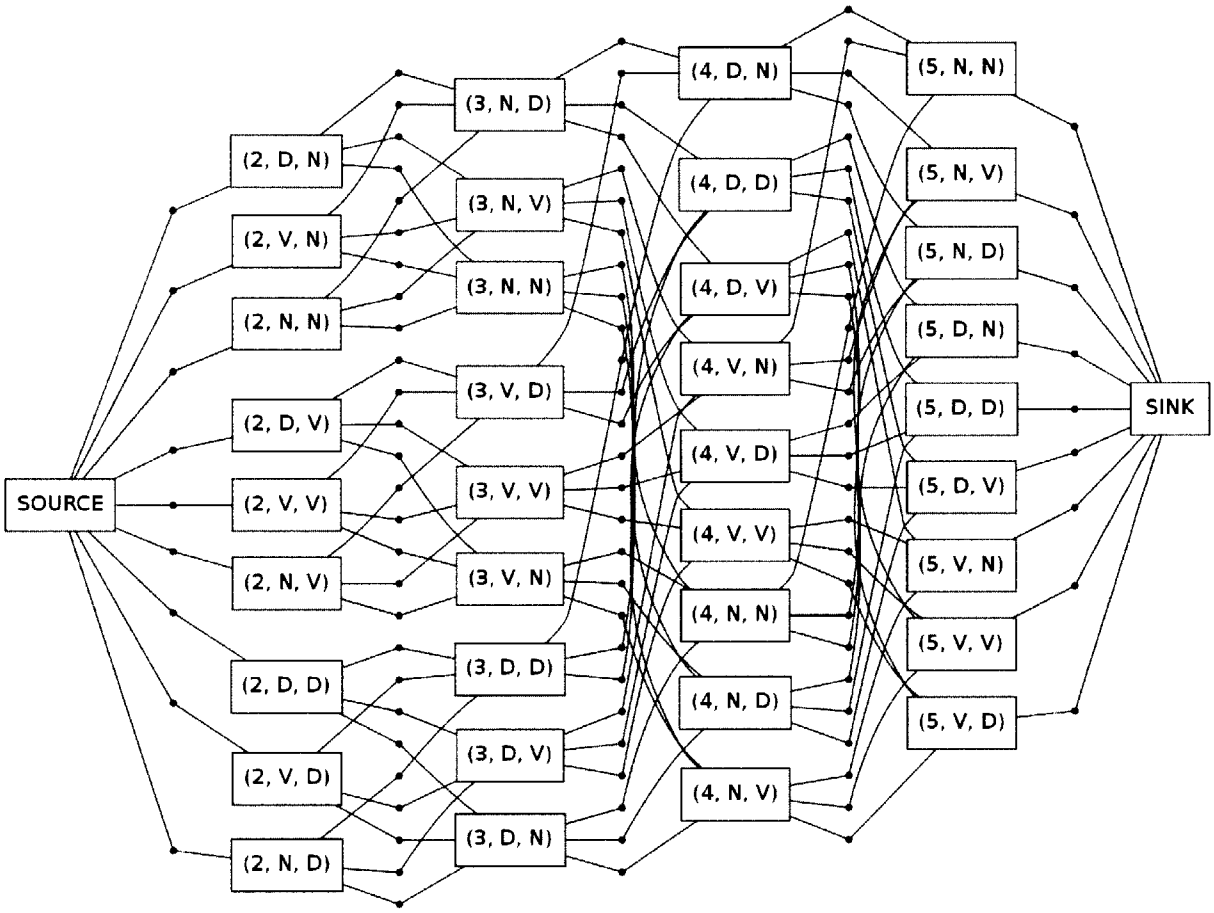


Figure 2-2: A lattice encoding the set of possible tag sequences as a directed acyclic graph with edges going from left-to-right. One possible tag sequence, **The/D man/N walks/V the/D dog/N**, *is highlighted in blue as a path through this graph.*

$$
\begin{aligned}
y(2,\ \texttt{D},\ \texttt{N}) &= 1 \\
y(3,\ \texttt{N}\ \ \texttt{V}) &= 1 \\
y(4,\ \texttt{V}\ \ \texttt{D}) &= 1 \\
y(5,\ \texttt{D}\ \ \texttt{N}) &= 1 \\
y(3,\ \texttt{D},\ \texttt{N}\ \ \texttt{V}) &= 1 \\
y(4,\ \texttt{N},\ \texttt{V}\ \ \texttt{D}) &= 1 \\
y(5,\ \texttt{V},\ \texttt{D}\ \ \texttt{N}) &= 1
\end{aligned}
$$

The Viterbi Algorithm Next consider finding paths in the lattice $\mathcal{D}$. Since there is a mapping between the index set $\mathcal{I}$ and the set of directed edges $\mathcal{E}$ and the weight vector is defined directly from the vector for trigram tagging, the decoding problem, $\arg\max_{y \in \mathcal{Y}} \theta^\top y$, is now simply a graph-search problem, and the sequence y^* is simply the highest-scoring path in $\mathcal{D}$.

Since the lattice is acyclic, the highest-scoring path – and by extension the highest-scoring tag sequence – can be found using the dynamic programming algorithm shown in Algorithm 1. This algorithm, known generally as the Viterbi algorithm, is one of the most celebrated algorithms in natural language processing and is central to tagging and speech recognition.

procedure TRIGRAMVITERBI($\mathcal{T}, \theta$)
  π[Source] $\leftarrow 0$
  for $B, C \in \mathcal{T}$ **do**
    $\pi[2, B, C] \leftarrow \pi$[Source]
  for $i = 3 \ldots n$ **do**
    for $B, C \in \mathcal{T}$ **do**
      $\pi[i, B, C] \leftarrow \max_{A \in \mathcal{T}} \theta(i, A, B, C) + \pi[i-1, A, B]$
  π[Sink] $\leftarrow \max_{B,C \in \mathcal{T}} \pi(n, B, C)$
  return π[Sink]

Algorithm 1: The Viterbi dynamic programming algorithm specialized for the case of trigram tagging.

The algorithm starts by initializing a dynamic programming *chart* π to 0 at the `Source`

position. The chart is indexed by a position in the sentence i and the tags at $i-1$ and i. It stores the score of the best partial path ending at that position with those tags. At each iteration of the algorithm, the chart is filled in by enumerating all possible previous tags A (at position $i-2$), computing their score, and checking if any are better than the current partial path. The last line returns the score at the `Sink` vertex.

procedure VITERBI($\mathcal{V}$, $\mathcal{E}$, θ)
  π[`Source`] $\leftarrow 0$
  bp[`Source`] $\leftarrow \epsilon$
  for $j \in \mathcal{V}$ in topological order **do**
    $\pi[j] \leftarrow \max_{i \in \mathcal{V}:(i,j) \in \mathcal{E}} \theta(i,j) + \pi[i]$
    $bp[j] \leftarrow \arg\max_{i \in \mathcal{V}:(i,j) \in \mathcal{E}} \theta(i,j) + \pi[i]$
  $p \leftarrow \langle \rangle$
  $c \leftarrow$ `Sink`
  while $bp[c] \neq \epsilon$ **do**
    $p \leftarrow \langle c, p_1, \ldots, p_n \rangle$
    $c \leftarrow bp[c]$
  return p

Algorithm 2: The general Viterbi dynamic programming algorithm for a directed acyclic graph with back-pointers.

This algorithm shows the `max` version of the dynamic program, i.e. it returns the highest score. In practice we usually require the `argmax` version as well that returns the highest-scoring structure. Fortunately we can recover the structure as well with a simple modification to store "back-pointers" along with the current score. Algorithm 2 shows the general Viterbi algorithm with back-pointers for a lattice.

2.2.2 Hypergraphs: Generalizing Lattices

While the Viterbi algorithm is perhaps the most widely-used dynamic programming algorithm for natural language decoding, many of the problems we will look at for tasks like parsing and translation will require more intricate dynamic programming algorithms. These algorithms are difficult to represent using lattices; however we will find it convenient to have

a similar graphical representation. In this section, we introduce a general form for dynamic programs known as a hypergraph.

Hypergraphs will give us a generalization of lattices for arbitrary dynamic programming algorithms. This section defines notation for specifying hypergraphs and shows how they can be used to define a set of feasible structures as well as dynamic programming algorithms for decoding. Throughout this work we return to this formalism to easily move between different representations of combinatorial sets.

Hypergraphs A directed, ordered hypergraph is a pair $(\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a set of vertices, and $\mathcal{E}$ is a set of directed hyperedges. Each hyperedge $e \in \mathcal{E}$ is a tuple $\langle \langle v_2 \dots v_{|v|} \rangle, v_1 \rangle$ where $v_i \in \mathcal{V}$ for $i \in \{1 \dots |v|\}$. The *head* of the hyperedge is $h(e) = v_1$. The *tail* of the hypergraph is the ordered sequence $t(e) = \langle v_2 \dots v_{|v|} \rangle$. The size of the tail $|t(e)|$ may vary across different edges, but $|t(e)| \geq 1$ and $|t(e)| \leq k$ for some constant k for all edges. We represent a directed graph as a directed hypergraph with $|t(e)| = 1$ for all edges $e \in \mathcal{E}$.

Each vertex $v \in \mathcal{V}$ is either a *non-terminal* or a *terminal* in the hypergraph. The set of non-terminals is $\mathcal{V}^{(n)} = \{v \in \mathcal{V} : h(e) = v \text{ for some } e \in \mathcal{E}\}$. Conversely, the set of terminals is defined as $\mathcal{V}^{(t)} = \mathcal{V} \setminus \mathcal{V}^{(n)}$.

We will further require that all hypergraphs used in this work also obey the following properties. These properties limit the set of hypergraphs considered to the set of valid DPs and will allow us to make claims about optimal structures. These definitions are based on Martin et al. (1990).

Property 2.2.1 (Acyclicity). *A directed hypergraph is acyclic if there exists a bijective function, $\sigma : \mathcal{V} \to \{1 \dots |\mathcal{V}|\}$, that orders each vertex in $\mathcal{V}$ with the property that for all $e \in \mathcal{E}$,*

$$\sigma(h(e)) > \sigma(v) \text{ for all } v \in t(e)$$

We refer to the vertex order of σ as a ***bottom-up*** *ordering.*

Property 2.2.2 (Connectedness). *Except for a distinguished root vertex $v_0 \in \mathcal{V}^{(n)}$, all non-terminal vertices $v \in \mathcal{V}^{(n)} \setminus \{v_0\}$ are in the tail of at least one hyperedge $e \in \mathcal{E}$, i.e. $v \in t(e)$.*

Additionally, we require that there exist a non-empty, *reference set* Γ associated with the hypergraph, and non-empty subsets $\Gamma[v] \subseteq \Gamma$ associated with each vertex $v \in \mathcal{V}$ such that the following property holds:

Property 2.2.3 (Reference Set Properties).

1. *Completeness; The reference set associated with the root must be the full reference set, i.e.*

$$\Gamma[v_0] = \Gamma$$

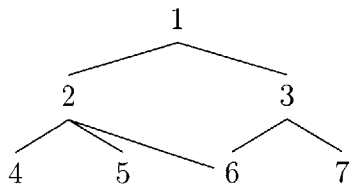
2. *Consistency; The reference sets must be consistent with the acyclic ordering σ, i.e. for all edges $e \in \mathcal{E}$,*

$$\Gamma[v] \subseteq \Gamma[h(e)] \text{ for all } v \in t(e)$$

3. *Disjointness; The reference sets associated with the tail of an edge must be disjoint, i.e. for all edges $e \in \mathcal{E}$,*

$$\Gamma[v] \cap \Gamma[u] = \emptyset \quad \text{for all } u, v \in t(e) \text{ with } u \neq v$$

These properties ensure that the tail vertices of each hyperedge do not share any descendants, and therefore that each hyperpath (which will be discussed in the next section) forms a valid tree. Without these properties, we can form a connected, acyclic hypergraph with non-tree hyperpaths. For instance consider a hypergraph with hyperedges $\langle\langle 2, 3\rangle, 1\rangle$, $\langle\langle 4, 5, 6\rangle, 2\rangle$, and $\langle\langle 6, 7\rangle, 3\rangle$ i.e.



This hypergraph is acyclic and connected, but the vertex 6 could appear multiple times in a single hyperpath. However, by the reference set properties, $\Gamma[6]$ must be non-empty, and

therefore, by Property 2, $\Gamma[2]$ and $\Gamma[3]$ must be non-disjoint, which means the hypergraph violates Property 3. Therefore this hypergraph does not satisfy the reference set properties.

These properties also allow us to formally define inside-outside relationships between all vertices in a hypergraph. In the parsing literature, *inside* refers to the items below the current item in the dynamic programming chart, whereas *outside* refers to items that are either above or independent of the current item. For a hypergraph we define these as follows:

Property 2.2.4 (Inside-Outside). *For vertices $u, v \in \mathcal{V}$, we say u is* inside *of v if*

- $\Gamma[u] \subset \Gamma[v]$ *or both* $\Gamma[u] = \Gamma[v]$ *and* $\sigma(u) < \sigma(v)$

Alternatively if u is outside *of v if*

- $\Gamma[u] \cap \Gamma[v] = \emptyset$ *or v is inside of u*

Otherwise u and v conflict, and cannot both appear in the same hyperpath.

For all lattices (hypergraphs with all tails of size 1), these reference set properties are trivially satisfied. We simply set $\Gamma = \{1\}$ and $\Gamma[v] = \{1\}$ for all $v \in \mathcal{V}$. Since all hyperedges are of the form $\langle \langle v_2 \rangle, v_1 \rangle$, Property 2 is satisfied with equality and Property 3 is satisfied because there is only one vertex in the tail. For lattices, since all vertices have $\Gamma[v] = \Gamma[u]$, the inside-outside relationship is given completely by σ and is called the *forward-backward* relationship.

For most other non-lattice examples in NLP, the dynamic program is defined over an ordered sequence $s_1 \dots s_n$ (such as a sentence). The reference set is naturally defined by setting $\Gamma[v]$ to the continuous span $\{i \dots j\}$ with $1 \leq i \leq j \leq n$, *covered* by a vertex v. The properties are satisfied by ensuring proper nesting of the spans. That is we can set $\Gamma = \{1 \dots n\}$ and ensure that for each hyperedge, the spans of the tail vertices $\Gamma[v_2] \dots \Gamma[v_{|v|}]$ are disjoint. We will show this explicitly for the case of context-free parsing.

Hyperpaths For lattices, we were interested in the set of paths corresponding to feasible sequences; for hypergraphs, we will analogously be interested in the set of hyperpaths corresponding to feasible structures.

For a hypergraph satisfying the properties above, define the set of hyperpaths as $\mathcal{Y} \subset \{0,1\}^{\mathcal{I}}$ with index set $\mathcal{I} = \mathcal{V} \cup \mathcal{E}$. That is for $y \in \mathcal{Y}$, $y(v) = 1$ if vertex v is used in the hyperpath, $y(v) = 0$ otherwise; and $y(e) = 1$ if hyperedge e is used in the hyperpath, $y(e) = 0$ otherwise. Because of the reference set properties, a valid hyperpath forms an ordered tree rooted at vertex v_0. Formally, a hyperpath satisfies the following constraints:

- The root vertex must be in the hyperpath, i.e.

$$y(v_0) = 1$$

- For every vertex $v \in \mathcal{V}^{(n)}$ visited in the hyperpath, $y(v) = 1$, there must be one hyperedge e entering the vertex, $y(e) = 1$ with $h(e) = v$. Conversely, for any vertex $v \in \mathcal{V}^{(n)}$ not visited, $y(v) = 0$, any edge e with $h(e) = v$ must have $y(e) = 0$. We write this linear constraint as

$$y(v) = \sum_{e \in \mathcal{E}: h(e) = v} y(e)$$

  for all non-terminal vertices $v \in \mathcal{V}^{(n)}$.

- For every visited vertex $v \in \mathcal{V}$ other than the root with $y(v) = 1$, there must be one leaving hyperedge, i.e. $e \in \mathcal{E}$ with $y(e) = 1$ and with $v \in t(e)$. Conversely, for every non-root vertex v not visited, $y(v) = 0$, no hyperedge $e \in \mathcal{E}$ with $y(e) = 1$ can have v as one if its children,. We write this constraint as,

$$y(v) = \sum_{e \in \mathcal{E}: v \in l(e)} y(e)$$

  for all non-root vertices $v \in \mathcal{V} \setminus \{v_0\}$.

The set of valid hyperpaths can be written as

$$\begin{aligned}\mathcal{Y} = \{y \in \{0,1\}^{\mathcal{I}} : y(v_0) &= 1, \\ y(v) &= \sum_{e \in \mathcal{E}: h(e)=v} y(e) \quad \forall\, v \in \mathcal{V}^{(n)}, \\ y(v) &= \sum_{e \in \mathcal{E}: v \in t(e)} y(e) \quad \forall\, v \in \mathcal{V} \setminus \{v_0\}\}\end{aligned}$$

The hypergraph decoding problem is to find the best hyperpath under a linear objective. Let $\theta \in \mathbb{R}^{\mathcal{V} \cup \mathcal{E}}$ be the weight vector for the hypergraph. (Often times the vertex weights will be 0.) This decoding problem is to find

$$\arg\max_{y \in \mathcal{Y}} \sum_{i \in \mathcal{I}} \theta(i) y(i) = \arg\max_{y \in \mathcal{Y}} \theta^\top y$$

This maximization can be computed for any weight vector and directed acyclic hypergraph in time $O(|\mathcal{E}|)$ using simple bottom-up dynamic programming—essentially the CKY algorithm. Algorithm 3 shows this algorithm.

procedure BESTPATHSCORE( )
  $\pi[v] \leftarrow \theta(v)$ for all $v \in \mathcal{V}^{(t)}$
  for $e \in \mathcal{E}$ in bottom-up order **do**
    $\langle \langle v_2 \dots v_k \rangle, v_1 \rangle \leftarrow e$
    $s \leftarrow \theta(e) + \theta(v_1) + \sum_{i=2}^{k} \pi[v_i]$
    if $s > \pi[v_1]$ **then** $\pi[v_1] \leftarrow s$
  return π

Algorithm 3: Dynamic programming algorithm for hypergraph decoding. Note that this version only returns the highest score: $\max_{y \in \mathcal{Y}} \theta^\top y$. *The optimal hyperpath can be found by including back-pointers.*

The hypergraph representation also makes it convenient to compute other properties of the underlying set in addition to the highest-scoring structure. For instance, we will see that it is useful to compute the *max-marginals* for various decoding problems. Define the

max-marginals for a hypergraph as a vector $M \in \mathbb{R}^{\mathcal{E}}$ defined as

$$M(e;\theta) = \max_{y \in \mathcal{Y}; y(e)=1} \theta^\top y$$

where $M(e)$ is the score of the highest-scoring hyperpath that includes hyperedge e. In Chapter 8, we make use of max-marginals to perform pruning.

Algorithm 4 shows an efficient method for computing the max-marginal vector for a given hypergraph. The algorithm is essentially a variant of the inside-outside parsing algorithm. It first computes the best path chart π, then computes an outside best path chart ρ. The max-marginals for vertices and hyperedges are set by combining these values.

procedure MAXMARGINALS
  $\pi \leftarrow$BESTPATH()
  $\rho \leftarrow$ OUTSIDE(π)
  for $e \in \mathcal{E}$ **do**
    $M(e) \leftarrow \rho[h(e)] + \sum_{v \in t(e)} \pi[t] + \theta(e)$
  return M
procedure OUTSIDE(π)
  $\rho[v_0] \leftarrow \theta(v_0)$
  for $e \in \mathcal{E}$ in top-down order **do**
    $\langle \langle v_2 \dots v_k \rangle, v_1 \rangle \leftarrow e$
    $t \leftarrow \theta(e) + \theta(v_1) + \sum_{i=2}^{k} \pi[v_i]$
    for $i = 2 \dots k$ **do**
      $s \leftarrow \rho[v_1] + t - \pi[v_i]$
      if $s > \rho[v_i]$ **then** $\rho[v_i] \leftarrow s$
  return ρ

Algorithm 4: Dynamic programming algorithm for computing the max-marginals of a hypergraph. Returns a vector $M \in \mathbb{R}^{\mathcal{E}}$ with the max-marginal values. The procedure OUTSIDE *is used to compute the best partial hyperpath including vertex v but not any vertices inside of v. Max-marginals are defined by combining the inside and outside values.*

Hypergraphs satisfying the properties given can be shown to represent arbitrary dynamic

programs (Martin et al., 1990). These same algorithms can be used for decoding context-free parses, projective dependency parses, and derivations for syntactic machine translation, as well as a generalization of lattice algorithms for speech alignment, part-of-speech tagging, and head-automata models for dependency parsing. We next consider dynamic programming algorithms for two parsing tasks.

2.2.3 Example 1: CFG Parsing

Earlier we looked at dependency-based representation of syntactic structure. We now consider a different syntactic representation, context-free grammars. Context-free grammars (CFGs) are one of the most widely used formalisms in natural language processing. Like dependency trees, context-free parse trees are often used as a first step for other language applications. We will also see that generalizations of context-free trees play an important role in problems like machine translation.

This section considers the problem of parsing with a context-free grammar in order to give an example of hypergraph decoding. We first define this decoding problem and then show how it can be represented using a hypergraph. Context-free parsing will be further applied in Chapter 4.

Weighted Context-Free Grammars Define a weighted context-free grammar as a 5-tuple $(\mathcal{N}, \Sigma, \mathcal{G}, r, w)$ with

- $\mathcal{N}$; the set of *non-terminal symbols*, for instance; the set of syntactic tags e.g. `NN`, `DT`, `VP`.
- Σ; the set of *terminal symbols*, often the words of the language e.g. `dog`, `cat`, `house`.
- $\mathcal{G}$; a set of context-free rules, consisting of a left-hand symbol $X \in \mathcal{N}$ and a sequence of right-hand symbols in $\mathcal{N}$ or Σ e.g. `NP` $\rightarrow$ `DT NN` or `NN` $\rightarrow$ `dog`.
- $w \in \mathbb{R}^{\mathcal{G}}$; a weight vector assigning a score to each rule.

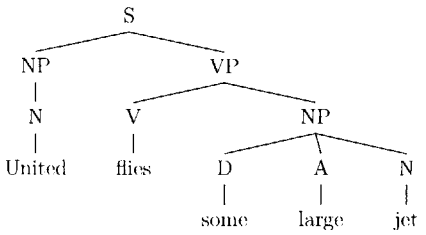


Figure 2-3: An example parse tree.

- $\eta_0 \in \mathcal{N}$; a distinguished root non-terminal symbol.

A valid context-free parse consists of a tree with non-terminal labels in $\mathcal{N}$, terminal labels in Σ, and root vertex labeled η_0. Each production in the tree must consist of an application of a rule from $\mathcal{G}$. For a parse tree y, the function $c(y, \text{X} \rightarrow \text{Y Z})$ denotes the number of times that rule $\text{X} \rightarrow \text{Y Z}$ is seen in the tree.

The score for a parse tree is the sum of scores for the rules it contains. As an example, consider the parse tree shown in Figure 2-3; for this tree, the scoring function yields

$$\begin{aligned} f(y) = \; & w(\texttt{S} \rightarrow \texttt{NP VP}) + w(\texttt{NP} \rightarrow \texttt{N}) + w(\texttt{N} \rightarrow \texttt{United}) \\ & + w(\texttt{VP} \rightarrow \texttt{V NP}) + w(\texttt{V} \rightarrow \texttt{flies}) + w(\texttt{NP} \rightarrow \texttt{D A N}) \\ & + w(\texttt{D} \rightarrow \texttt{some}) + + w(\texttt{A} \rightarrow \texttt{large}) + w(\texttt{N} \rightarrow \texttt{jet}) \end{aligned}$$

Again we remain agnostic to how the scores for individual context-free rules are defined. As one example, in a probabilistic context-free grammar, we would define $w(X \rightarrow w) = \log P(X \rightarrow w | X)$.

A weighted CFG can be used to produce the highest-scoring parse for a given sentence, this task is known as context-free parsing. Given a sentence $s_1 \ldots s_n$ where $s_i \in \Sigma$ for all $i \in \{1 \ldots n\}$, the decoding problem for this model is to find the best parse tree that produces this sentence. For simplicity, we limit the problem to CFGs in Chomsky normal form (CNF). That is we restrict the set of rules $\mathcal{G}$ to

$$\mathcal{G} \subset \{\mathrm{X} \to \mathrm{Y}\ \mathrm{Z} : X, Y, Z \in \mathcal{N}\} \cup \{\mathrm{X} \to \mathrm{s} : X \in \mathcal{N}, s \in \Sigma\}$$

Let $\mathcal{Y}$ be valid parse trees. Our goal is to find

$$\arg\max_{y \in \mathcal{Y}} \sum_{\mathrm{X} \to \mathrm{Y}\ \mathrm{Z} \in \mathcal{G}} w(\mathrm{X} \to \mathrm{Y}\ \mathrm{Z}) c(y, \mathrm{X} \to \mathrm{Y}\ \mathrm{Z}) + \sum_{\mathrm{X} \to \mathrm{s}} w(\mathrm{X} \to \mathrm{s}) c(y, \mathrm{X} \to \mathrm{s})$$

As with the other decoding problems seen so far, the number of trees in $\mathcal{Y}$ is exponential in the length of the sentence n. However the set of structures can be compactly represented by reformulating $\mathcal{Y}$ as a hypergraph.

Hypergraph Parsing Now for a given input sentence consider a hypergraph $(\mathcal{V}, \mathcal{E})$ where the set of hyperpaths $\mathcal{Y}$ corresponds to the parse trees under a context-free grammar. The hypergraph is specified by the following construction:

- For all hypergraph nonterminal vertices $v \in \mathcal{V}^{(n)}$, let v be defined as $v = (i, j, X)$ where i and j are a span of the source sentence with $1 \leq i < j \leq n$ and $X \in \mathcal{N}$ is a non-terminal symbol of the grammar.

- For all hypergraph terminal vertices $v \in \mathcal{V}^{(t)}$, let $v = (i, i, X)$ where $i \in \{1 \dots n\}$ is a position in the sentence, $X \in \mathcal{N}$ is a non-terminal symbol, and $X \to s_i$ is a rule in $\mathcal{G}$.

- There is a binary hyperedge $e \in \mathcal{E}$ with $e = \langle \langle v_2, v_3 \rangle, v_1 \rangle$, for each rule $\mathrm{X} \to \mathrm{Y}\ \mathrm{Z} \in \mathcal{G}$, head vertex $v_1 = (i, k, X)$ and tail vertices $v_2 = (i, j, Y)$ and $v_3 = (j + 1, k, Z)$ with $1 \leq i \leq j < k \leq n$. We refer to these hyperedges as $X(i, k) \to Y(i, j)\ Z(j + 1, k)$.

- The root vertex v_0 is $(1, n, \eta_0)$ where η_0 is the distinguished root symbol in the underlying context-free grammar.

The size of the final hypergraph is the number of the binary hyperedges $X(i, k) \to Y(i, j)\ Z(j + 1, k)$. For each rule in $\mathcal{G}$ there is at most one of these hyperedges for indices $1 \leq i \leq j \leq k \leq n$. This gives a total size of $|\mathcal{E}| = O(n^3 |\mathcal{G}|)$.

To express this problem as a hypergraph decoding problem, define the score vector θ based on the weights of the underlying grammar. For all hyperedges let

$$\theta(X(i,k) \rightarrow Y(i,j)\ Z(j+1,k)) = w(X \rightarrow Y\ Z) + \delta[i=j]w(Y \rightarrow s_i) + \delta[j+1=k]w(Z \rightarrow s_{j+1})$$

and let the weights of each vertex v be 0. Finding the highest-scoring hyperpath is linear in the number of hyperedges, so $O(n^3|\mathcal{G}|)$ is also the runtime complexity of CFG decoding.

We can also show that this hypergraph satisfies the necessary reference set properties. Define the set Γ as $\{1 \dots n\}$ and let $\Gamma[(i,j,X)] = \{i \dots j\}$ for all $(i,j,X) \in \mathcal{V}$. The following properties hold:

- Completeness; By the root definition $\Gamma[v_0] = \Gamma[(1,n,\eta_0)] = \Gamma$.
- Consistency; For all hyperedges, $X(i,k) \rightarrow Y(i,j)\ Z(j+1,k)$, by the hyperedge definition $\Gamma[(i,j,Y)] \subset \Gamma[(i,k,X)]$ and $\Gamma[(j+1,k,Z)] \subset \Gamma[(i,k,X)]$.
- Disjointness; For all hyperedges, $X(i,k) \rightarrow Y(i,j)\ Z(j+1,k)$, by the hyperedge definition $\Gamma[(i,j,Y)] \cap \Gamma[(j+1,k,Z)] = \emptyset$.

Example 2.2.2 (Context-Free Parsing). Consider a specific CFG in Chomsky normal form. Define the grammar $(\mathcal{N}, \Sigma, \mathcal{G}, r, w)$ with a set of non-terminal symbols,

$$\mathcal{N} = \{\texttt{S}, \texttt{NP}, \texttt{PP}, \texttt{VP}, \texttt{N}, \texttt{D}, \texttt{P}, \texttt{V}\}$$

a set of terminal symbols,

$$\Sigma = \{\texttt{the}, \texttt{man}, \texttt{walks}, \texttt{dog}, \texttt{park}\}$$

a set of nonterminal rules and weights,

Rules $\mathrm{X} \rightarrow \mathrm{Y}\ \ \mathrm{Z} \in \mathcal{G}$	$w(\mathrm{X} \rightarrow \mathrm{Y}\ \ \mathrm{Z})$
S → NP VP	1
VP → V NP	3
VP → VP PP	2
NP → NP PP	5
PP → P NP	4

a set of terminal rules and weights,

Rules $\mathrm{X} \rightarrow \mathrm{Y} \in \mathcal{G}$	$w(\mathrm{X} \rightarrow \mathrm{Y})$
N → man	2
N → dog	3
N → park	4
P → in	2
D → the	1
V → walks	5

and root symbol $\eta_0 = \mathtt{S}$.

Now consider parsing an input sentence, `The man walks the dog in the park`, with $n = 8$. This sentence is a toy example of prepositional phrase attachment ambiguity. The prepositional phrase `in the park` could modify `dog` to describe the type of dog or modify `walk` to describe where the walking takes place. The two possible parses $y^{(1)}$ and $y^{(2)}$ have scores

$$\begin{aligned} f(y^{(1)}) \quad = \quad & w(\mathtt{PP} \rightarrow \mathtt{P}\ \ \mathtt{NP}) + w(\mathtt{NP} \rightarrow \mathtt{NP}\ \ \mathtt{PP}) + w(\mathtt{VP} \rightarrow \mathtt{V}\ \ \mathtt{NP}) \\ & + w(\mathtt{NP} \rightarrow \mathtt{D}\ \ \mathtt{N}) + w(\mathtt{NP} \rightarrow \mathtt{D}\ \ \mathtt{N}) + w(\mathtt{NP} \rightarrow \mathtt{D}\ \ \mathtt{N}) \\ & + w(\mathtt{S} \rightarrow \mathtt{NP}\ \ \mathtt{VP}) + \ldots \end{aligned}$$

and

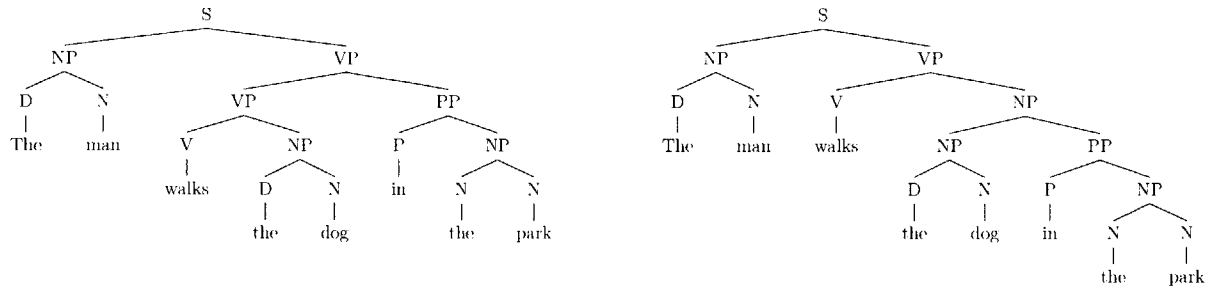


Figure 2-4: Two possible trees for the sentence ***The/D man/N walks/V the/D dog/N in/P the/D park/N.***

$$
\begin{aligned}
f(y^{(2)}) \;=\; & w(\texttt{VP} \rightarrow \texttt{V NP}) + w(\texttt{PP} \rightarrow \texttt{P NP}) + w(\texttt{VP} \rightarrow \texttt{VP PP}) \\
& + w(\texttt{NP} \rightarrow \texttt{D N}) + w(\texttt{NP} \rightarrow \texttt{D N}) + w(\texttt{NP} \rightarrow \texttt{D N}) \\
& + w(\texttt{S} \rightarrow \texttt{NP VP}) + \ldots
\end{aligned}
$$

where for both trees we have elided the identical pre-terminal scores.

The two parse trees are shown in Figure 2-4. Note that with a more realistic grammar there would be many more parses for a sentence of this length.

In order to represent the different parse structures, we construct a hypergraph containing all possible trees. The hypergraph for this problem can be constructed mechanically from applying the rules given above; the hypergraph for the problem is shown in Figure 2-5.

Dynamic programming can then be used to find the best hyperpath y^* in this hypergraph using Algorithm 3. Figure 2-5 also shows this hyperpath overlaid on the hypergraph itself. Once a hyperpath is found, it is can be used to reconstruct the highest-scoring parse tree.

2.2.4 Example 2: Projective Dependency Parsing

When discussing dependency parsing, we made a distinction between two important variants: non-projective and projective structures. In the general non-projective case, a dependency tree may be any directed spanning tree over a sentence. For second-order scoring functions,

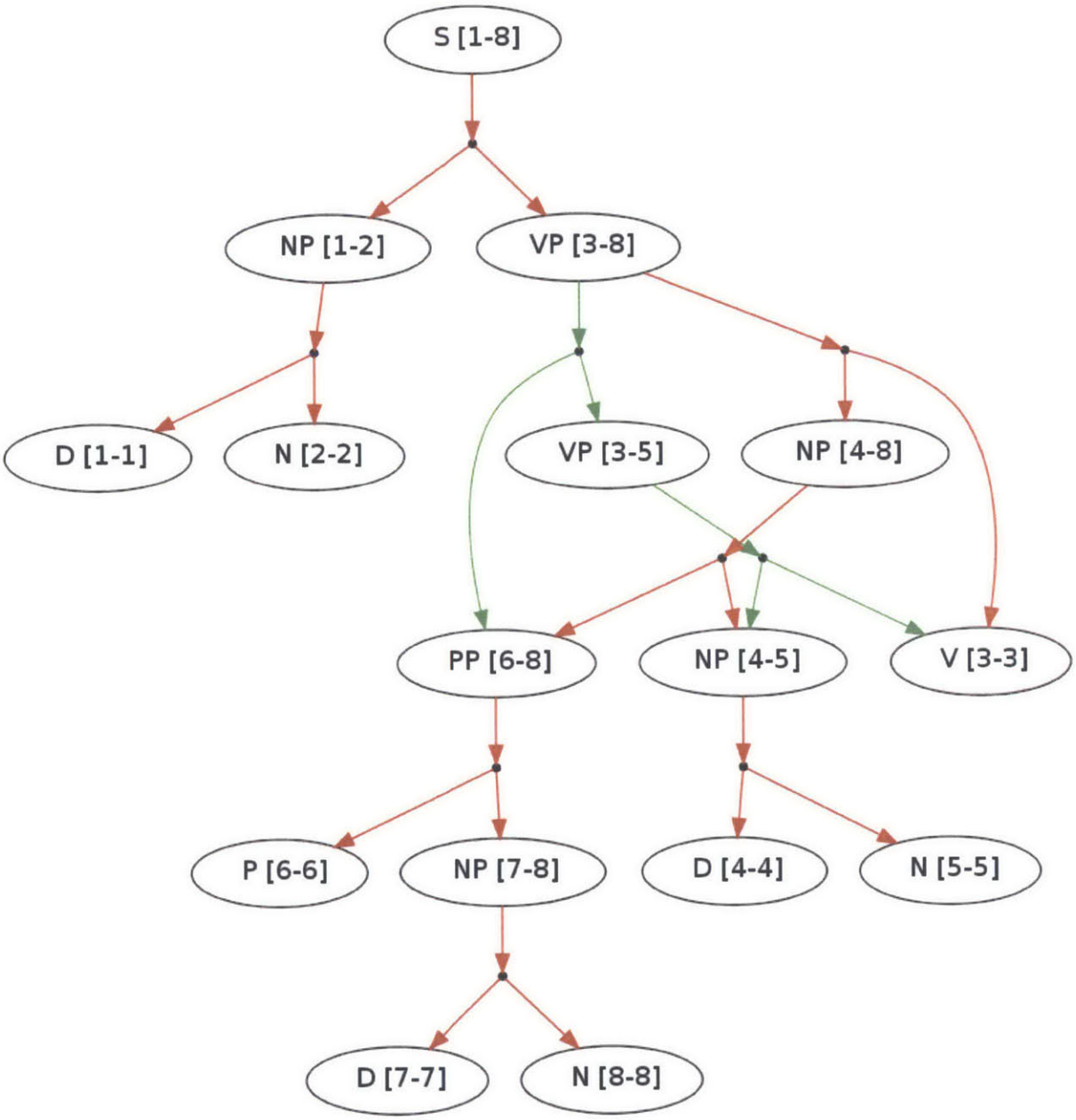


Figure 2-5: The hypergraph for the sentence **The/D man/N walks/V the/D dog/N in/P the/D park/N**. *This data structure encodes all possible parses for this sentence under the given grammar (we omit hyperedges that are not part of a valid hyperpath). Each dot represents a hyperedge, its parent vertex is its head, it child vertices are its tail. Vertices are labeled by the nonterminal symbol and span. The hyperedges used in parse* $y^{(2)}$ *are shown in red.*

finding the highest scoring structure from this set is NP-hard. However, for many languages, the majority of dependency structures are projective, i.e. they obey the contiguous substructure property shown in Property 2.1.1.

Projectivity is important because it is a sufficient restriction for deriving a dynamic programming algorithm for dependency parsing. Not only does it allow us to use dynamic programming for decoding, but it also allows us to employ higher-order scoring functions and still maintain efficient decoding, even though the equivalent non-projective problem is NP-hard.

The main DP algorithm for projective dependency parsing is known as Eisner's algorithm (Eisner & Satta, 1999). For a sentence of length n both first- and second-order projective dependency parsing can be decoded in $O(n^3)$ time. In this section we give a description of Eisner's algorithm using the hypergraph formalism. We will make direct use of the specifics of this algorithm in Chapter 8.

Eisner's Algorithm Recall that the main assumption of projective dependency parsing was that for any valid parse there exist sets $\mathcal{A}_h$ for each vertex $h \in \{0 \dots n\}$ in the directed graph $\mathcal{D}$ indicating the contiguous span covered by h. Eisner's algorithm additionally makes a further assumption about the scoring function

Property 2.2.5 (Split-Head Assumption). *Define two sub-spans of the set $\mathcal{A}_h$: $\mathcal{A}_h^{\leftarrow} = \{u : u \leq h\}$ and $\mathcal{A}_h^{\rightarrow} = \{u : u \geq h\}$ indicating the left and right descendents respectively. A scoring function f obeys the split-head assumption if for a subtree y rooted at h it factors into two parts i.e. $f(y) = f(y_{\{\mathcal{A}_h^{\leftarrow}\}}) + f(y_{\{\mathcal{A}_h^{\rightarrow}\}})$.*

There are a range of interesting scoring functions that obey this property. In general, the algorithm applies to a broad class of scoring functions known of head-automata models (Alshawi, 1996). We will return to head-automata models in Chapter 6. For simplicity we describe the algorithm for the specific case of a first-order scoring function.

Informally, for each token h, the algorithm incrementally and independently builds up $\mathcal{A}_h^{\leftarrow}$ and $\mathcal{A}_h^{\rightarrow}$. We focus on the right facing sequence, $\mathcal{A}_h^{\rightarrow} = \{h \dots e\}$ for some position e.

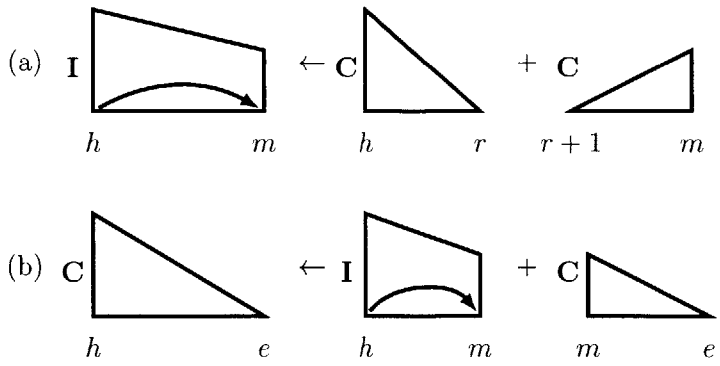


Figure 2-6: Schematic representation of Eisner's algorithm using "triangles and trapezoids". Rule (a) merges an adjacent right complete item and left complete item to form a right incomplete item. Rule (b) merges an adjacent right incomplete item with a right complete item to form an extended right complete item.

At any point, this sequence may be in two states: *incomplete* `I` and or *complete* `C`. If the sequence is incomplete, then e has just been added as a direct modifier of h, and the next step is to find $\mathcal{A}_e^{\rightarrow}$ and to complete the sequence. If the sequence is complete, we have just extended $\mathcal{A}_h^{\rightarrow}$ with some $\mathcal{A}_e^{\rightarrow}$ and we can now look for more modifiers.

Figure 2-6 shows a schematic diagram of the algorithm. Incomplete items are drawn as trapezoids and complete items are drawn as triangles. At each step we either apply merge two complete items and create a dependency, or merge an incomplete item with a complete item to create a complete item. The diagram only shows the right-facing rules, there is a symmetric set of rules for left-facing items.

We can explicitly specify this algorithm as a hypergraph

- A vertex $v \in \mathcal{V}$ is a tuple (t, d, i, j) where t is the type – either complete, `C`, or incomplete, `I` – d is a direction, either left or right, and i, j is a span over the sentence. The set of vertices is defined as

$$\mathcal{V} = \{(t, d, i, j) : t \in \{\texttt{C}, \texttt{I}\}, d \in \{\leftarrow, \rightarrow\}, 0 \leq i \leq j \leq n\}$$

- A hyperedge is an instance of one of two rules. It may either create a new dependency or complete an incomplete structure. The two rules are shown graphically in Figure 2-6.

– The first rule creates an arc by merging a complete right-facing item and a complete left-facing item.

$$\begin{aligned}\mathcal{E}_1 \;=\; & \left\{\langle(\mathtt{C},\rightarrow,h,r),(\mathtt{C},\leftarrow,r+1,m)\rangle,(\mathtt{I},\rightarrow,h,m)\rangle : 0 \le h \le r < m \le n\right\} \cup \\ & \left\{\langle(\mathtt{C},\rightarrow,m,r),(\mathtt{C},\leftarrow,r+1,h)\rangle,(\mathtt{I},\leftarrow,m,h)\rangle : 0 \le m \le r < h \le n\right\}\end{aligned}$$

These hyperedges correspond to adding an arc from h to m, and have weight $\theta(h,m)$.

– The second rule completes an incomplete item by merging it with an adjacent complete item in the same direction.

$$\begin{aligned}\mathcal{E}_2 \;=\; & \left\{\langle(\mathtt{I},\rightarrow,h,m),(\mathtt{C},\rightarrow,m,e)\rangle,(\mathtt{C},\rightarrow,h,e)\rangle : 0 \le h < m \le e \le n\right\} \cup \\ & \left\{\langle(\mathtt{C},\leftarrow,e,m),(\mathtt{I},\leftarrow,m,h)\rangle,(\mathtt{C},\leftarrow,e,h)\rangle : 0 \le e \le m < h \le n\right\}\end{aligned}$$

These hyperedges are structural and have weight 0.

The full set of hyperedges is $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$

- The terminal vertices of the hypergraph are the complete items spanning a single word.

$$\mathcal{V}^{(t)} = \left\{(\mathtt{C},d,i,i) : i \in \{1\ldots n\}, d \in \{\leftarrow,\rightarrow\}\right\}$$

- The root vertex is a complete item covering the entirety of the sentence $(\mathtt{C},\rightarrow,0,n)$.

Both sets $\mathcal{E}_1$ and $\mathcal{E}_2$ have size $O(n^3)$ which gives the runtime of the dynamic programming algorithm.

The same algorithm can be further extended to include higher-order scoring functions. Somewhat surprisingly the highest-scoring second-order parse can also be found in $O(n^3)$ time. Additionally a similar technique also extends to a third-order parsing function that incorporates grandparent arcs, i.e. the head of the head of a modifier, as well as siblings. This scoring function can be decoded in $O(n^4)$ time (Koo & Collins, 2010).

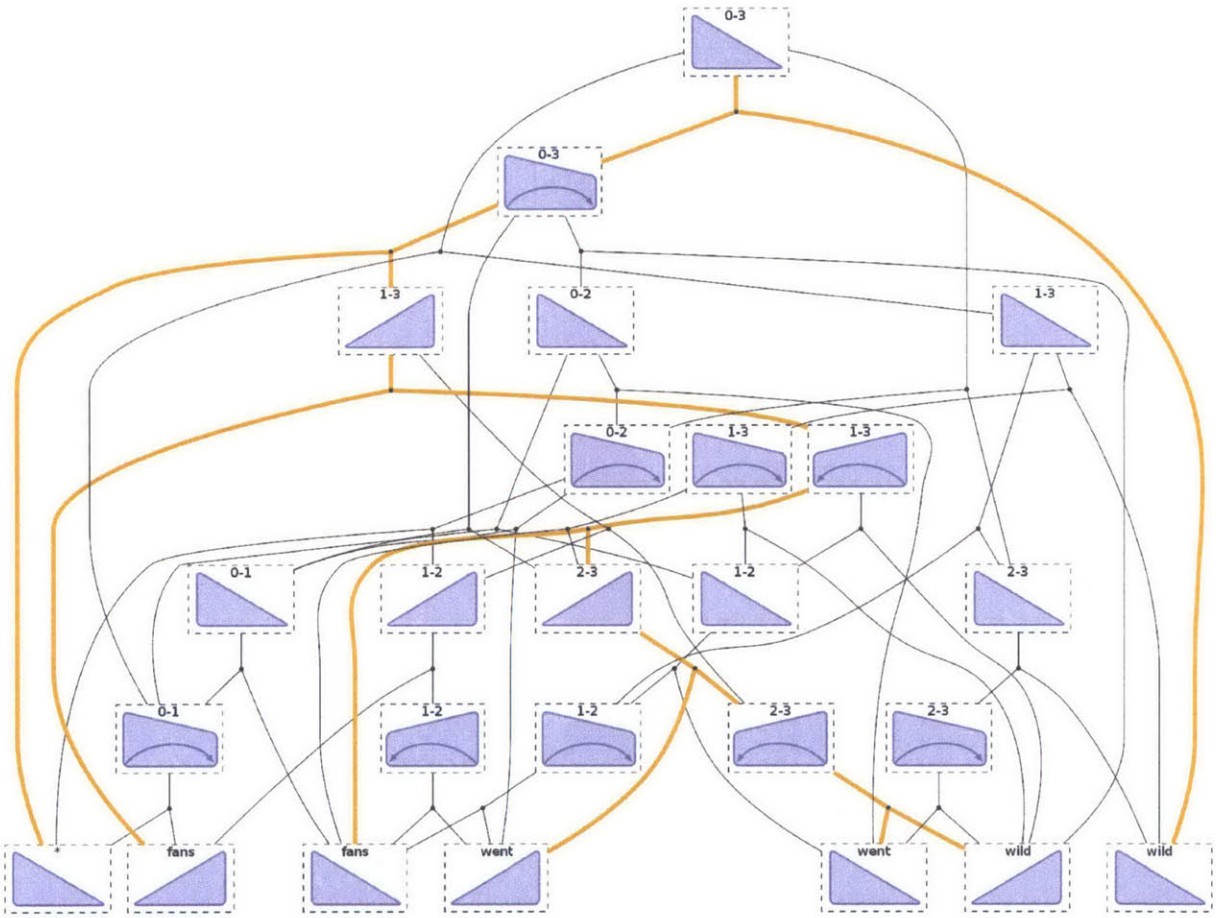


Figure 2-7: A hypergraph encoding all possible projective parses for the sentence **fans went wild**. *The orange highlighted lines represent one possible hyperpath through the hypergraph.*

Example 2.2.3 (Dependency Parse). Consider the hypergraph for a simple sentence `Fans went wild`. This hypergraph is shown in Figure 2-7.

The hyperpath highlighted in orange is one parse structure for this sentence. The hyperpath corresponds to a dependency parse with the arcs.

$$(*_0 \rightarrow \texttt{wild}_3), (\texttt{wild}_3 \rightarrow \texttt{went}_2), (\texttt{wild}_3 \rightarrow \texttt{fans}_1)$$

It passes through vertices

$(\texttt{C}, \rightarrow, 0, 0)$	$(\texttt{C}, \leftarrow, 1, 1)$	$(\texttt{C}, \rightarrow, 1, 1)$	$(\texttt{C}, \leftarrow, 2, 2)$	$(\texttt{C}, \rightarrow, 2, 2)$
$(\texttt{C}, \leftarrow, 3, 3)$	$(\texttt{C}, \rightarrow, 3, 3)$	$(\texttt{I}, \leftarrow, 2, 3)$	$(\texttt{C}, \leftarrow, 2, 3)$	$(\texttt{I}, \leftarrow, 1, 3)$
$(\texttt{C}, \leftarrow, 1, 3)$	$(\texttt{I}, \rightarrow, 0, 3)$	$(\texttt{C}, \rightarrow, 0, 3)$		

2.2.5 Hypergraphs and Finite-State Automata

We conclude the background section by discussing a final core formalism for natural language tasks: finite-state automata (FSA). Like context-free grammars, FSAs play an important role for formulating natural language models. FSAs are not a direct focus of this work, but they will be a useful descriptive tool. In this section we give a brief formal introduction to FSAs and then describe an important classical result for the combination of a finite-state automaton and a directed hypergraph. We will return to this result and FSAs generally several times.

FSAs Define a weighted, deterministic finite-state automaton as a 6-tuple $(\Sigma, \mathcal{Q}, q_0, \mathcal{F}, \tau, w)$ where

- Σ; the underlying symbol alphabet.
- $\mathcal{Q}$; a finite set of states.
- $q_0 \in \mathcal{Q}$; a distinguished initial state.

- $\mathcal{F} \subset \mathcal{Q}$; a distinguished set of final states
- $\tau : \mathcal{Q} \times \Sigma \mapsto \mathcal{Q}$; a deterministic state transition function.
- $w : \mathbb{R}^{\mathcal{Q} \times \Sigma}$; a set of transition weights.

This definition is based on Hopcroft et al. (2001). Note that we do not allow ϵ transitions, i.e. transitions only occur on an input symbol from Σ.

An FSA can be applied to an input sequence consisting of symbols $\sigma_1 \ldots \sigma_n$ where for all $i \in \{1 \ldots n\}$ the symbol σ_i comes from alphabet Σ. For an input sequence there is a corresponding state sequence $q_1 \ldots q_n$, where for $i \in \{1 \ldots n\}$ state q_i is from $\mathcal{Q}$, is defined recursively from the start state q_0 as

$$q_1 = \tau(q_0, \sigma_1), q_2 = \tau(q_1, \sigma_2), \ldots q_n = \tau(q_{n-1}, \sigma_n)$$

The input sequence is said to be *valid* if $q_n \in \mathcal{F}$.

The weight of the sequence is the sum of the weight of each of the transitions used. We overload the symbol w as both a transition score and the weight of an input sequence, defined as

$$w(\sigma_1, \ldots, \sigma_n) = \sum_{i=1}^{n} w(q_{i-1}, \sigma_i)$$

There are many other useful properties and operations involving FSAs, including finding intersections, optimal paths, and minimal state sets. For brevity we limit our description to scoring paths and direct interested research to the text of Hopcroft et al. (2001) for a general treatment and to the work of Mohri (1997) for applications in natural language processing.

Example 2.2.4 (Trigram Tagging). Consider again the problem of trigram tagging over a set of tags $\mathcal{T}$. In this example we consider representing the state of a trigram tagger using a finite-state automaton.

The main idea is that each finite-state sequence will correspond to a possible set of triples of the sentence. Define the alphabet to be the set of tags $\Sigma = \mathcal{T}$, and define the set of states

$\mathcal{Q}$ as the previous two tags

$$\mathcal{Q} = \{(B, C) : B, C \in \mathcal{T}\} \cup \{(\texttt{<t>}, B) : B \in \mathcal{T}\} \cup \{(\texttt{<t>}, \texttt{<t>})\}$$

where <t> is a boundary tag.

The start state is set to be the boundary tags before seeing any input

$$q_0 = (\texttt{<t>}, \texttt{<t>})$$

and the final state set is all valid non-initial pairs

$$\mathcal{F} = \{(A, B) : A, B \in \mathcal{T}\}$$

finally the transition function simply extends the current state

$$\tau(\sigma, q) = \tau(C, (A, B)) = (B, C)$$

The size of the state set is $|\mathcal{Q}| = O(|\mathcal{T}|^2)$.

For now, we will ignore the weight w. Note that the states have no information about the position in the sentence. This information will come from intersecting the FSA with an ordered structure, either with a sequence or – as we will see in the next section – a hypergraph.

FSA and Hypergraph Intersection Our main application of FSAs will be to explore the intersection of a weighted directed hypergraph $(\mathcal{V}, \mathcal{E})$ with a weighted finite-state automaton. This technique is a classical method for combining word-level models with grammatical models, and also generalizes to combining sequence models with hypergraph structures.

More concretely, given a hypergraph and an FSA, we will be interested in jointly maximizing the score of a hyperpath and the FSA score of symbols associated with the fringe of the hyperpath. Assume that we have a symbol labeling function $l : \mathcal{V}^{(t)} \mapsto \Sigma$ that maps each terminal vertex of the hypergraph to a symbol in the FSA alphabet.

Recall that a hyperpath $y \in \mathcal{Y}$ is an ordered tree, and therefore its terminal vertices form an ordered sequence. Let the fringe of the tree be the sequence $v_1 \dots v_m$ where $v_i \in \mathcal{V}^{(t)}$ for $i \in \{1 \dots m\}$. Furthermore, for a given hyperpath, define its *symbol sequence* as the ordered sequence of symbols at its fringe, i.e. $l(v_1), \dots, l(v_m)$.

Next, define the function $g(y)$ to give the score of the symbol sequence under a finite-state automaton.

$$g(y) = w(l(v_1) \dots l(v_m))$$

Given a weighted hypergraph, a finite-state automaton, and a labeling function l, the intersected hypergraph-FSA decoding problem is

$$\arg\max_{y \in \mathcal{Y}} \theta^\top y + g(y)$$

The decoding problem maximizes the score of the best hyperpath as well as the score of the fringe symbol sequence under a finite-state automaton. This intersection problem comes up very often in natural language processing. We will use it for two very different applications: intersecting parse structures with tagging models and intersecting a translation model with a language model.

A celebrated early result in NLP is the construction of Bar-Hillel for the intersection of a FSA and a CFG (Bar-Hillel et al., 1964). This construction is more general, but for simplicity we consider the special case of a directed acyclic hypergraph and an FSA.

Theorem 2.2.1. *The intersection of a weighted directed acyclic hypergraph $(\mathcal{V}, \mathcal{E})$ and a weighted finite-state automaton yields a new directed acyclic hypergraph $(\mathcal{V}', \mathcal{E}')$ where*

$$\arg\max_{y \in \mathcal{Y}} \theta^\top y + g(y) = \arg\max_{y \in \mathcal{Y}'} \theta'^\top y$$

The new hypergraph size has size $|\mathcal{E}'| = O(|\mathcal{E}||\mathcal{Q}|^{m+1})$ where m is the maximum tail size $m = \max_{e \in \mathcal{E}} |t(e)|$.

The construction is straightforward. The main idea is to augment each vertex in $v \in \mathcal{V}$

with each possible entering state q and leaving state q' for $q, q' \in \mathcal{Q}$, which we write as $(v, q, q') \in \mathcal{V}'$. For terminal vertices, we must also ensure that the FSA transitions from q to q' given the symbol associated with the vertex. For hyperedges we must ensure consistency among the tails. Define the new hypergraph $(\mathcal{V}', \mathcal{E}')$ as:

- For each terminal vertex $v \in \mathcal{V}^{(t)}$ and states $q, q' \in \mathcal{Q}$ such that $\tau(q, l(v)) = q'$, define a vertex $(v, q, q') \in \mathcal{V}^{(t)'}$. Let the weight of this vertex be $\theta'(v, q, q') = \theta(v) + w(q, l(v))$.

- For each non-terminal vertex $v \in \mathcal{V}^{(n)}$ and states $q, q' \in \mathcal{Q}$, define a vertex $(v, q, q') \in \mathcal{V}^{(t)'}$. Let the weight of this vertex be $\theta'(v, q, q') = \theta(v)$.

- For each $e \in \mathcal{E}$ with $t(e) = v_2 \dots v_k$ and for states $q_1, \dots, q_k$, define an edge $e' \in \mathcal{E}'$ with head $(h(e), q_0, q_m)$ and tail $(v_2, q_1, q_2), (v_2, q_1, q_2) \dots, (v_k, q_{k-1}, q_k)$. Let the weight of this edge be $\theta'(e) = \theta(e)$.

Finally we add a new root vertex v_0' with unary hyperedges to each of the vertices in of

$$\{(v_0, q_0, q) \in \mathcal{V}' : q \in \mathcal{F}\}$$

.

In the worst case, for each hyperedge in the original graph there will be $|\mathcal{Q}|^k$ new hyperedges, one for each $q_0 \dots q_k$ where k is the largest tail size. In practice though we will only be concerned with the case where $m = 2$, and therefore $O(|\mathcal{E}||\mathcal{Q}|^3)$.

Example 2.2.5 (CFG Parsing and Trigram Tagging). Recall that we constructed a finite-state automaton for trigram tagging $(\Sigma, \mathcal{Q}, q_0, \mathcal{F}, \tau, w)$. The FSA has a symbol alphabet defined to be the set of possible $\Sigma = \mathcal{T}$.

Consider the context-free parsing hypergraph from Section 2.2.3. With this FSA in mind, define a labeling of the hypergraph for context-free parsing to simply be

$$l(v) = l(\langle i, i, X \rangle) = X$$

Under this definition the label sequence $l(v_1) \ldots l(v_m)$ simply corresponds to the tag sequence implied by the parse structure. Furthermore for this hypergraph the sequence length m is the same as the sentence length $m = n$.

Recall that this hypergraph has a set of vertices and hyperedges defined as

$$\mathcal{V} = \{\langle X, i, j\rangle : X \in \mathcal{N}, 1 \leq i \leq j \leq n\}$$

$$\mathcal{E} = \{\mathrm{X}(i,k) \rightarrow \mathrm{Y}(i,j)\ \mathrm{Z}(j+1,k) : (\mathrm{X} \rightarrow \mathrm{Y}\ \mathrm{Z}) \in \mathcal{G}, 1 \leq i \leq j < k \leq n\}$$

And therefore running dynamic programming over this hypergraph has runtime complexity $O(|\mathcal{G}|n^3)$.

Now assume that we have a finite-state automaton that we want to *intersect* with the original grammar. Assuming the automaton has a set of states $\mathcal{Q}$, applying the Bar-Hillel intersection gives new hyperedges

$$\mathcal{V}' = \{\langle X_{q,q'}, i, j\rangle : X \in \mathcal{N}, 1 \leq i \leq j \leq n, q, q' \in \mathcal{Q}\}$$

$$\mathcal{E}' = \{\mathrm{X}_{q_1,q_3}(i,k) \rightarrow \mathrm{Y}_{q_1,q_2}(i,j)\ \mathrm{Z}_{q_2,q_3}(j+1,k) \ : \ (\mathrm{X} \rightarrow \mathrm{Y}\ \mathrm{Z}) \in \mathcal{G}, 1 \leq i \leq j < k \leq n, \ q_1, q_2, q_3 \in \mathcal{Q}\ \}$$

After intersection, we can count the number of hyperedges to see that there $|\mathcal{E}| = O(|\mathcal{G}|n^3|\mathcal{Q}|^3)$, which implies that dynamic programming can find the best derivation in $O(|\mathcal{G}|n^3|\mathcal{Q}|^3)$ time.

In the case of tagging, we have seen in Section 2.2.1 and particularly Figure 2-2, that a trigram tagging model can be represented as an automaton with $|\mathcal{T}|^2$ states, where each state represents the previous two tags. After intersection, this yields an $O(|\mathcal{G}|n^3|\mathcal{T}|^6)$ time algorithm.

The addition of the tagging model leads to a multiplicative factor of $|\mathcal{T}|^6$ in the runtime

of the parser, which is a very significant decrease in efficiency (it is not uncommon for $|\mathcal{T}|$ to take values of say 5 or 50, giving values for $|\mathcal{T}|^6$ larger than 15,000 or 15 million).

In contrast, the dual decomposition algorithm which we will describe in Chapter 4 takes $O(K(|\mathcal{G}|n^3 + |\mathcal{T}|^3 n))$ time for this same joint decoding problem, where K is the number of iterations required for convergence; in experiments, K is often a small number. This is a very significant improvement in runtime over the Bar-Hillel et al. method.

2.3 Conclusion

In this chapter we described the general decoding setup that we will use for each of the natural language tasks described in this thesis and discussed examples from tagging, context-free parsing, and dependency parsing. We then gave a formal description of dynamic programming using hypergraphs for solving these decoding problems. Finally we introduced FSAs and showed how these can be intersected with hypergraphs; however we also saw that this intersection led to a very inefficient decoding approach. We noted that the dual decomposition algorithm for this problem is often much more efficient in practice. Before describing this particular method, we first describe the formal background of Lagrangian relaxation and dual decomposition in the next chapter.

Chapter 3

Theory of Lagrangian Relaxation

In the last chapter we saw that natural language decoding can be written as an optimization problem of the form

$$\arg\max_{y \in \mathcal{Y}} f(y)$$

For some tasks this optimization problem can be efficiently solved using combinatorial algorithms such as maximum directed spanning tree – as for first-order dependency parsing – or by dynamic programming – as for trigram tagging or context-free parsing.

However there are many important decoding problems in NLP that are challenging to efficiently solve, and there are even some, like non-projective, second-order dependency parsing, that are NP-hard.

We will be interested in solving specific instances of these problems exactly; even if the general problem is intractable in the worst case. The tool we employ is a classical technique from combinatorial optimization known as Lagrangian relaxation. From a high level, this method allows us to relax a difficult decoding problem into a simpler optimization problem. We can then use standard combinatorial algorithms for solving this relaxed problem, and even provide provable bounds on the solution to the original decoding problem.

This section provides a formal overview of this method. Lagrangian relaxation has a long history in the combinatorial optimization literature, going back to the seminal work of Held and Karp (1971), who derive a relaxation algorithm for the traveling salesman problem.

Initial work on Lagrangian relaxation for decoding statistical models focused on the MAP problem in Markov random fields (Komodakis et al., 2007, 2011).

An important special case of Lagrangian relaxation is dual decomposition (DD), where two or more combinatorial algorithms are used. It will be useful to consider it as a variant of Lagrangian relaxation that makes use of multiple combinatorial algorithms, together with a set of linear constraints that are again incorporated using Lagrange multipliers.

In this background section we introduce Lagrangian relaxation and dual decomposition as general techniques, provide a collection of useful theorems for these approaches, and discuss connections to subgradient optimization. This section is largely adapted from the tutorial of Rush and Collins (2012).

3.1 Lagrangian Relaxation

To apply Lagrangian relaxation we assume that we have a combinatorial optimization problem over a discrete set $\mathcal{Y} \subset \{0,1\}^{\mathcal{I}}$, defined as

$$\arg\max_{y \in \mathcal{Y}} f(y) = \arg\max_{y \in \mathcal{Y}} \theta^\top y$$

Additionally, we will assume that this problem is difficult to solve directly.

The first step will be to choose a discrete set $\mathcal{Y}' \subset \{0,1\}^{\mathcal{I}}$ that has the following properties:

- $\mathcal{Y} \subset \mathcal{Y}'$. Hence $\mathcal{Y}'$ contains all vectors found in $\mathcal{Y}$, and in addition contains some vectors that are not in $\mathcal{Y}$.

- For any value of $\theta \in \mathbb{R}^{\mathcal{I}}$, we can easily find $\arg\max_{y \in \mathcal{Y}'} \theta^\top y$. By "easily" we mean that this problem is significantly easier to solve than the original problem.

- Finally, we assume that

$$\mathcal{Y} = \{y : y \in \mathcal{Y}' \text{ and } Ay = b\} \tag{3.1}$$

for some constraint matrix $A \in \mathbb{R}^{p \times |\mathcal{I}|}$ and vector $b \in \mathbb{R}^p$. The condition $Ay = b$ specifies p linear constraints on y. We will assume that the number of constraints, p, is polynomial in the size of the input.

The implication here is that the linear constraints $Ay = b$ need to be added to the set $\mathcal{Y}'$, but these constraints considerably complicate the decoding problem. Instead of incorporating them as hard constraints, we will deal with these constraints using Lagrangian relaxation.

We introduce a vector of Lagrange multipliers, $\lambda \in \mathbb{R}^p$. The Lagrangian is

$$L(\lambda, y) = \theta^\top y + \lambda^\top (Ay - b)$$

This function combines the original objective function $\theta^\top y$, with a second term that incorporates the linear constraints and the Lagrange multipliers. The Lagrangian dual objective is

$$L(\lambda) = \max_{y \in \mathcal{Y}'} L(\lambda, y)$$

and the dual problem is to find

$$\min_{\lambda \in \mathbb{R}^p} L(\lambda)$$

A common approach—which will be used in most algorithms we present—is to use the subgradient method to minimize the dual. We set the initial Lagrange multiplier values to be $\lambda^{(1)} = 0$. For $k = 1, 2, \ldots$ we then perform the following steps:

1. Compute the current optimal structure

$$y^{(k)} = \arg\max_{y \in \mathcal{Y}'} L(\lambda^{(k)}, y) \tag{3.2}$$

2. Update the Lagrange multipliers based on $y^{(k)}$

$$\lambda^{(k+1)} = \lambda^{(k)} - \alpha_k (Ay^{(k)} - b) \tag{3.3}$$

where $\alpha_k > 0$ is the step size at the k'th iteration.

A crucial point is that $y^{(k)}$ can be found efficiently, because

$$\arg\max_{y \in \mathcal{Y}'} L(\lambda^{(k)}, y) = \arg\max_{y \in \mathcal{Y}'} \theta^\top y + \lambda^{(k)\top}(Ay - b) = \arg\max_{y \in \mathcal{Y}'} \theta'^\top y$$

where $\theta' = \theta + A^\top \lambda^{(k)}$. Hence the Lagrange multiplier terms are easily incorporated into the objective function.

We can now state the following theorem:

Theorem 3.1.1. *The following properties hold for Lagrangian relaxation:*

a). For any $\lambda \in \mathbb{R}^p$, $L(\lambda) \geq \max_{y \in \mathcal{Y}} \theta^\top y$.

b). Under a suitable choice of the step sizes α_k, $\lim_{k \to \infty} L(\lambda^{(k)}) = \min_\lambda L(\lambda)$.

c). Define $y^{(\lambda)} = \arg\max_{y \in \mathcal{Y}'} L(\lambda, y)$. *If there exists a* λ *such that* $Ay^{(\lambda)} = b$, *then* $y^{(\lambda)} = \arg\max_{y \in \mathcal{Y}'} \theta^\top y$ *(i.e.,* $y^{(\lambda)}$ *is optimal).*

In particular, in the subgradient algorithm described above, if for any k *we have* $Ay^{(k)} = b$, *then* $y^{(k)} = \arg\max_{y \in \mathcal{Y}} \theta^\top y$.

Part (a) of the theorem states that the dual value provides an upper bound on the score for the optimal solution. This part is known as weak duality and is shown formally in Theorem 3.3.1. Part (b) states that the subgradient method successfully minimizes this upper bound. An appropriate rate sequence and proof will be given in Theorem 3.3.2. Part (c) states that if we ever reach a solution $y^{(k)}$ that satisfies the linear constraints, then we have solved the original optimization problem.

3.2 Dual Decomposition

Dual decomposition is a variant of Lagrangian relaxation that leverages the observation that often times decoding problems can be expressed as two or more sub-problems tied together with linear constraints that enforce some notion of agreement between solutions to

the different problems. These sub-problems can often be solved efficiently using combinatorial algorithms. Several of the decoding algorithms derived in this thesis are based on this method.

In dual decomposition, we assume that we have a discrete set $\mathcal{Y} \subset \{0,1\}^{\mathcal{I}}$ where $\mathcal{I}$ is a problem-specific index set. Each vector $y \in \mathcal{Y}$ has an associated score

$$f(y) = \theta^\top y$$

where θ is a vector in $\mathbb{R}^{\mathcal{I}}$. In addition, we assume a second model $\mathcal{Z} \subset \{0,1\}^{\mathcal{J}}$, with each vector $z \in \mathcal{Z}$ having an associated score

$$g(z) = \omega^\top z$$

where $\omega \in \mathbb{R}^{\mathcal{J}}$. The combinatorial problem of interest is to find

$$\begin{aligned} \underset{y \in \mathcal{Y}, z \in \mathcal{Z}}{\arg\max} \quad & \theta^\top y + \omega^\top z & (3.4) \\ \text{such that} \quad & Ay + Cz = b & (3.5) \end{aligned}$$

where the constraints are specified with $A \in \mathbb{R}^{p \times |\mathcal{I}|}$, $C \in \mathbb{R}^{p \times |\mathcal{J}|}$, $b \in \mathbb{R}^p$.

The goal is to find the optimal *pair* of derivations under the linear constraints specified by $Ay + Cz = b$. In practice, the linear constraints specify consistency between y and z: often they specify that the two vectors are in some sense "in agreement".

For convenience, we define the following sets:

$$\begin{aligned} \mathcal{W} &= \{(y,z) : y \in \mathcal{Y}, z \in \mathcal{Z}, Ay + Cz = b\} \\ \mathcal{W}' &= \{(y,z) : y \in \mathcal{Y}, z \in \mathcal{Z}\} \end{aligned}$$

where the first set enforces the constraints and the second does not. It follows that our

decoding problem is to find

$$\arg\max_{(y,z)\in\mathcal{W}} \theta^\top y + \omega^\top z \tag{3.6}$$

Next, we make the following assumptions:

- For any value of $\theta \in \mathbb{R}^{\mathcal{I}}$, we can easily find $\arg\max_{y\in\mathcal{Y}} \theta^\top y$. Furthermore, for any value of $\omega \in \mathbb{R}^{\mathcal{J}}$, we can easily find $\arg\max_{z\in\mathcal{Z}} \omega^\top b$. It follows that for any $\theta \in \mathbb{R}^{\mathcal{I}}$, $\omega \in \mathbb{R}^{\mathcal{J}}$, we can easily find

  $$(y^*, z^*) = \arg\max_{(y,z)\in\mathcal{W}'} \theta^\top y + \omega^\top z \tag{3.7}$$

  by setting

  $$y^* = \arg\max_{y\in\mathcal{Y}} \theta^\top y, \quad z^* = \arg\max_{z\in\mathcal{Z}} \omega^\top z$$

  Note that Eq. 3.7 is closely related to the problem in Eq. 3.6, but with $\mathcal{W}$ replaced by $\mathcal{W}'$ (i.e., the linear constraints $Ay + Cz = b$ have been dropped). By "easily" we again mean that these optimization problems are significantly easier to solve than our original problem in Eq. 3.6.

Our goal involves optimization of a linear objective, over the finite set $\mathcal{W}$, as given in Eq. 3.6. We can efficiently find the optimal value over a set $\mathcal{W}'$ such that $\mathcal{W}$ is a subset of $\mathcal{W}'$, and $\mathcal{W}'$ has dropped the linear constraints $Ay + Cz = b$.

The dual decomposition algorithm is then derived to exploit this assumption. As with Lagrangian relaxation, we introduce a vector of Lagrange multipliers, $\lambda \in \mathbb{R}^p$. The Lagrangian is now

$$L(\lambda, y, z) = \theta^\top y + \omega^\top z + \lambda^\top (Ay + Cz - b)$$

and the Lagrangian dual objective is

$$L(\lambda) = \max_{(y,z)\in\mathcal{W}'} L(\lambda, y, z)$$

and the dual problem is

$$\min_{\lambda\in\mathbb{R}^p} L(\lambda)$$

Many algorithms can be used to find this minimum. Again we use a subgradient algorithm for finding $\min_{\lambda \in \mathbb{R}^p} L(\lambda)$. We initialize the Lagrange multipliers to $\lambda^{(1)} = 0$. For $k = 1, 2, \ldots$ we perform the following steps:

$$(y^{(k)}, z^{(k)}) = \arg\max_{(y,z) \in \mathcal{W}'} L(\lambda^{(k)}, y, z)$$

followed by

$$\lambda^{(k+1)} = \lambda^{(k)} - \alpha_k (A y^{(k)} + C z^{(k)} - b)$$

where each $\alpha_k > 0$ is a stepsize. The pseudo-code is shown in Algorithm 5.

```
Set λ^(1) = 0⃗
for k = 1 to K do
    y^(k) ← arg max_{y∈𝒴} (θ + A^⊤ λ^k)^⊤ y
    z^(k) ← arg max_{z∈𝒵} (ω + C^⊤ λ^(k))^⊤ z
    if A y^(k) + C z^(k) = b then
        return (y^(k), z^(k))
    else
        λ^(k+1) ← λ^(k) − α_k (A y^(k) + C z^(k) − b)
```

Algorithm 5: General pseudo-code for dual decomposition.

Note that the solutions $y^{(k)}, z^{(k)}$ at each iteration are found easily, because it can be verified that

$$\arg\max_{(y,z) \in \mathcal{W}'} L(\lambda^{(k)}, y, z) = \left(\arg\max_{y \in \mathcal{Y}} \theta'^{\top} y, \arg\max_{z \in \mathcal{Z}} \omega'^{\top} z, \right)$$

where $\theta' = \theta + A^{\top} \lambda^{(k)}$ and $\omega' = \omega + C^{\top} \lambda^{(k)}$. Thus the dual decomposes into two easily solved maximization problems.

3.3 Formal Properties of Dual Decomposition

We now give some formal properties of the dual decomposition approach described in the previous section. We first describe three important theorems regarding the algorithm, and then describe connections between this algorithm and subgradient optimization methods. Note that in this section we focus on dual decomposition, but the same theorems can also be derived for Lagrangian relaxation with minor modifications.

3.3.1 Three Theorems

Recall that the problem we are attempting to solve is defined as

$$\begin{array}{ll} \arg\max_{y \in \mathcal{Y}, z \in \mathcal{Z}} & \theta^\top y + \omega^\top z \\ \text{such that} & Ay + Cz = b \end{array}$$

The first step will be to introduce the Lagrangian for this problem. We introduce a vector of Lagrange multipliers $\lambda \in \mathbb{R}^p$ for this equality constraint. The Lagrangian is now

$$L(\lambda, y, z) = \theta^\top y + \omega^\top z + \lambda^\top (Ay + Cz - b)$$

and the Lagrangian dual objective is

$$L(\lambda) = \max_{(y,z) \in \mathcal{W}'} L(\lambda, y, z)$$

Under the assumptions described above, the dual value $L(\lambda)$ for any value of λ can be calculated efficiently: we simply compute the two max's, and sum them. Thus the dual decomposes in a very convenient way into two efficiently solvable sub-problems.

Finally, the dual problem is to minimize the dual objective, that is, to find

$$\min_{\lambda} L(\lambda)$$

We will see shortly that Algorithm 5 is a subgradient algorithm for minimizing the dual objective.

Define (y^*, z^*) to be the optimal solution to optimization problem 3.5. The first theorem is as follows:

Theorem 3.3.1. *For any value of* λ,

$$L(\lambda) \geq \theta^\top y^* + \omega^\top z^*$$

Hence $L(\lambda)$ provides an upper bound on the score of the optimal solution. The proof is simple:

Proof:

$$\begin{aligned} L(\lambda) &= \max_{y \in \mathcal{Y}, z \in \mathcal{Z}} L(\lambda, y, z) && (3.8) \\ &\geq \max_{y \in \mathcal{Y}, z \in \mathcal{Z}: Ay + Cz = b} L(\lambda, y, z) && (3.9) \\ &= \max_{y \in \mathcal{Y}, z \in \mathcal{Z}: Ay + Cz = b} \theta^\top y + \omega^\top (z) && (3.10) \\ &= \theta^\top y^* + \omega^\top z^* && (3.11) \end{aligned}$$

Eq. 3.9 follows because by adding the constraints $Ay + Cz = b$, we are optimizing over a smaller set of (y, z) pairs, and hence the max cannot increase. Eq. 3.10 follows because if $Ay + Cz = b$, we have

$$\lambda^\top (Ay + Cz - b) = 0$$

and hence $L(\lambda, y, z) = \theta^\top y + \omega^\top z$. Finally, Eq. 3.11 follows through the definition of y^* and z^*. $\square$

The property that $L(\lambda) \geq \theta^\top y^* + \omega^\top z^*$ for any value of λ is often referred to as *weak duality*. The value of $\inf_\lambda L(\lambda) - (\theta^\top y^* + \omega^\top z^*)$ is often referred to as the *duality gap* or the *optimal duality gap* (see for example Boyd & Vandenberghe, 2004).

Note that obtaining an upper bound on $\theta^\top y^* + \omega^\top z^*$ (providing that it is relatively tight)

can be a useful goal in itself. First, upper bounds of this form can be used as admissible heuristics for search methods such as A* or branch-and-bound algorithms. Second, if we have some method that generates a potential solution (y, z), we immediately obtain an upper bound on how far this solution is from being optimal, because

$$(\theta^\top y^* + \omega^\top z^*) - (\theta^\top y + \omega^\top z) \leq L(\lambda) - (\theta^\top y + \omega^\top z)$$

Hence if $L(\lambda) - (\theta^\top y + \omega^\top z)$ is small, then $(\theta^\top y^* + \omega^\top z^*) - (\theta^\top y + \omega^\top z)$ must be small. See Appendix A for more discussion.

Our second theorem states that the algorithm in Algorithm 5 successfully converges to $\min_\lambda L(\lambda)$. Hence the algorithm successfully converges to the tightest possible upper bound given by the dual. The theorem is as follows:

Theorem 3.3.2. *Consider the algorithm in Figure 5. For any sequence $\alpha_1, \alpha_2, \alpha_3, \ldots$ such that $\alpha_k > 0$ for all $k \geq 1$, and*

$$\lim_{k \to \infty} \alpha_k = 0 \quad \textit{and} \quad \sum_{k=1}^{\infty} \alpha_k = \infty,$$

we have

$$\lim_{k \to \infty} L(\lambda^{(k)}) = \min_\lambda L(\lambda)$$

Proof: See the work of Shor (1985). See also Appendix A. □

Our algorithm is actually a *subgradient method* for minimizing $L(\lambda)$: we return to this point in Section 3.3.2. For now though, the important point is that our algorithm successfully minimizes $L(\lambda)$.

Our final theorem states that if we ever reach agreement during the algorithm, we are guaranteed to have the optimal solution. We first need the following definitions:

Definition 3.3.1. *For any value of λ, define*

$$y^{(\lambda)} = \arg\max_{y \in \mathcal{Y}} (\theta + A^\top \lambda)^\top y$$

and

$$z^{(\lambda)} = \arg\max_{z \in \mathcal{Z}} (\omega + C^\top \lambda)^\top z$$

The theorem is then:

Theorem 3.3.3. *If there exists a λ such that*

$$Ay^{(\lambda)} + Cz^{(\lambda)} = b$$

then

$$\theta^\top y^{(\lambda)} + \omega^\top z^{(\lambda)} = \theta^\top y^* + \omega^\top z^*$$

i.e., $(y^{(\lambda)}, z^{(\lambda)})$ is optimal.

Proof: We have, by the definitions of $y^{(\lambda)}$ and $z^{(\lambda)}$,

$$\begin{aligned} L(\lambda) &= \theta^\top y^{(\lambda)} + \omega^\top z^{(\lambda)} + \lambda^\top (Ay + Cz - b) \\ &= \theta^\top y^{(\lambda)} + \omega^\top z^{(\lambda)} \end{aligned}$$

where the second equality follows because $Ay + Cz = b$. But $L(\lambda) \geq \theta^\top y^* + \omega^\top z^*$ for all values of λ, hence

$$\theta^\top y^{(\lambda)} + \omega^\top z^{(\lambda)} \geq \theta^\top y^* + \omega^\top z^*$$

Because y^* and z^* are optimal, we also have

$$\theta^\top y^{(\lambda)} + \omega^\top z^{(\lambda)} \leq \theta^\top y^* + \omega^\top z^*$$

hence we must have

$$\theta^\top y^{(\lambda)} + \omega^\top z^{(\lambda)} = \theta^\top y^* + \omega^\top z^*$$

□

Theorems 3.3.2 and 3.3.3 refer to quite different notions of convergence of the dual decomposition algorithm. For the remainder of this work, to avoid confusion, we will explicitly use the following terms:

- *d-convergence* (short for "dual convergence") will be used to refer to convergence of the dual decomposition algorithm to the minimum dual value: that is, the property that $\lim_{k\to\infty} L(\lambda^{(k)}) = \min_\lambda L(\lambda)$. By theorem 3.3.2, assuming appropriate step sizes in the algorithm, we **always** have d-convergence.

- *e-convergence* (short for "exact convergence") refers to convergence of the dual decomposition algorithm to a point where $Ay + Cz = b$. By theorem 3.3.3, if the dual decomposition algorithm e-converges, then it is guaranteed to have provided the optimal solution. However, the algorithm is **not** guaranteed to e-converge.

3.3.2 Subgradients and the Subgradient Method

The proof of d-convergence, as defined in Theorem 3.3.2, relies on the fact that the algorithm in Figure 5 is a subgradient algorithm for minimizing the dual objective $L(\lambda)$. Subgradient algorithms are a generalization of gradient-descent methods; they can be used to minimize convex functions that are non-differentiable. This section describes how the algorithm in Figure 5 is derived as a subgradient algorithm.

Recall that $L(\lambda)$ is defined as follows:

$$\begin{aligned} L(\lambda) &= \max_{y\in\mathcal{Y}, z\in\mathcal{Z}} L(\lambda, y, z) \\ &= \max_{y\in\mathcal{Y}} \left(\theta^\top y + \lambda^\top A y\right) + \max_{z\in\mathcal{Z}} \left(\omega^\top z + \lambda^\top C z\right) - \lambda^\top b \end{aligned}$$

and that our goal is to find $\min_\lambda L(\lambda)$.

First, we note that $L(\lambda)$ has the following properties:

- $L(\lambda)$ is a convex function. That is, for any $\lambda^{(1)} \in \mathbb{R}^p$, $\lambda^{(2)} \in \mathbb{R}^p$, $u \in [0,1]$,

$$L(u\lambda^{(1)} + (1-u)\lambda^{(2)}) \leq uL(\lambda^{(1)}) + (1-u)L(\lambda^{(2)})$$

  (The proof is simple: see Appendix B.1.)

- $L(\lambda)$ is not differentiable. In fact, it is easily shown that it is a piecewise linear function.

The fact that $L(\lambda)$ is not differentiable means that we cannot use a gradient descent method to minimize it. However, because it is nevertheless a convex function, we can instead use a subgradient algorithm. The definition of a subgradient is as follows:

Definition 3.3.2 (Subgradient). *A subgradient of a convex function $L : \mathbb{R}^p \to \mathbb{R}$ at λ is a vector $\gamma^{(\lambda)}$ such that for all $v \in \mathbb{R}^p$,*

$$L(v) \geq L(\lambda) + \gamma^{(\lambda)} \cdot (v - \lambda)$$

The subgradient $\gamma^{(\lambda)}$ is a tangent at the point λ that gives a lower bound to $L(\lambda)$: in this sense it is similar[1] to the gradient for a convex but differentiable function.[2] The key idea in subgradient methods is to use subgradients in the same way that we would use gradients in gradient descent methods. That is, we use updates of the form

$$\lambda' = \lambda - \alpha\gamma^{(\lambda)}$$

where λ is the current point in the search, $\gamma^{(\lambda)}$ is a subgradient at this point, $\alpha > 0$ is a step size, and λ' is the new point in the search. Under suitable conditions on the stepsizes α (e.g., see theorem 3.3.2), these updates will successfully converge to the minimum of $L(\lambda)$.

So how do we calculate the subgradient for $L(\lambda)$? It turns out that it has a very convenient form. As before (see definition 3.3.1), define $y^{(\lambda)}$ and $z^{(\lambda)}$ to be the arg max's for the two

[1] More precisely, for a function $L(\lambda)$ that is convex and differentiable, then its gradient at any point λ is a subgradient at λ.

[2] It should be noted, however, that for a given point λ, there may be more than one subgradient: this will occur, for example, for a piecewise linear function at points where the gradient is not defined.

maximization problems in $L(\lambda)$. If we define the vector $\gamma^{(\lambda)}$ as

$$\gamma^{(\lambda)} = Ay^{(\lambda)} + Cz^{(\lambda)} - b$$

then it can be shown that $\gamma^{(\lambda)}$ is a subgradient of $L(\lambda)$ at λ. The updates in the algorithm in Figure 5 take the form

$$\lambda' = \lambda - \alpha(Ay^{(\lambda)} + Cz^{(\lambda)} - b)$$

and hence correspond directly to subgradient updates.

See Appendix B.2 for a proof that the subgradients take this form, and Appendix B.3 for a proof of convergence for the subgradient optimization method.

3.4 Related Work

Lagrangian relaxation and related methods have been used in combinatorial optimization, operations research, and machine learning. In this section, we give a summary of relevant related work from these areas.

Combinatorial Optimization Lagrangian relaxation (LR) is a widely used method in combinatorial optimization, going back to the seminal work of Held and Karp (1971) on the traveling salesman problem. Held and Karp propose relaxing the set of valid tours of an undirected graph to the set of *1-trees*, consisting of spanning trees over the graph with an additional edge between vertex 1 and any other vertex. Lagrange multipliers are then used to encourage the highest-scoring 1-tree to be a tour. The relaxation is illustrated in Figure 3-1.

For more background, see the work of Lemaréchal (2001) and Fisher (1981) for surveys of LR methods, and the textbook of Korte and Vygen (2008) on combinatorial optimization. Decomposing linear and integer linear programs is also a fundamental technique in the optimization community (Dantzig & Wolfe, 1960; Everett III, 1963).

There is a very direct relationship between LR algorithms and linear programming relax-

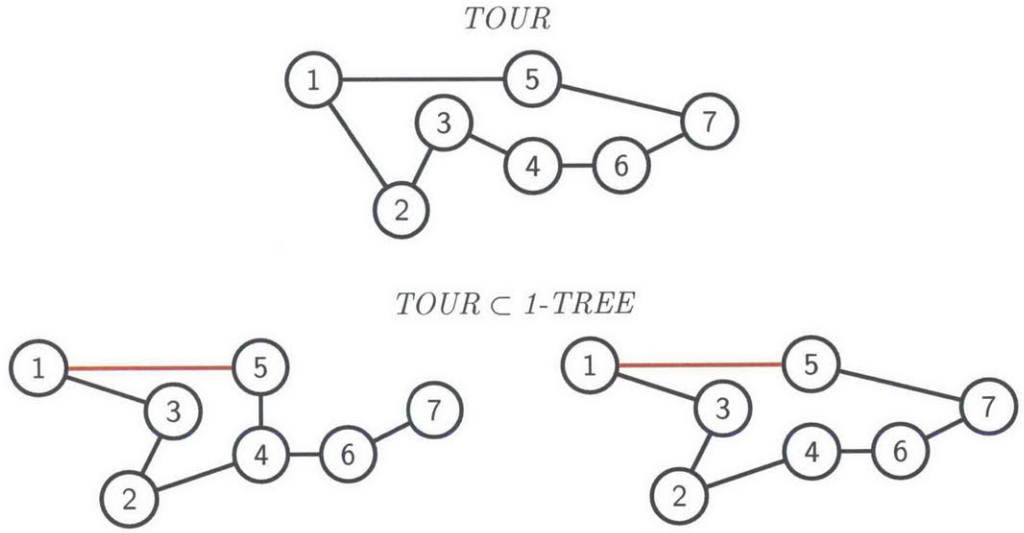


Figure 3-1: The Held and Karp (1971) relaxation of the traveling salesman problem. The approach is to relax the set TOUR *into a less constrained set* 1-TREE. *The highest-scoring 1-tree can be found with a maximum spanning tree algorithm. Lagrange multipliers encourage the highest-scoring 1-tree to be a valid tour.*

ations of combinatorial optimization problems; again, see the textbook of Korte and Vygen.

Belief Propagation, and Linear Programming Relaxations for Inference in MRFs

There has been a large amount of research on the MAP inference problem in Markov random fields (MRFs), which consists of a combinatorial decoding problem over a globally-normalized distribution. For tree-structured MRFs, max-product belief propagation (max-product BP) (Pearl, 1988) gives exact solutions. Max-product BP is a form of dynamic programming, which generalizes the Viterbi algorithm.

For general MRFs where the underlying graph may contain cycles, the MAP problem is NP-hard: this has led researchers to consider a number of approximate inference algorithms. Early work considered loopy variants of max-product BP (see for example Felzenszwalb & Huttenlocher, 2006, for the application of loopy max-product BP to problems in computer vision); however, these methods are heuristic, lacking formal guarantees.

More recent work has considered methods based on linear programming (LP) relaxations of the MAP problem. See the work of Yanover, Meltzer, and Weiss (2006), or Section 1.6

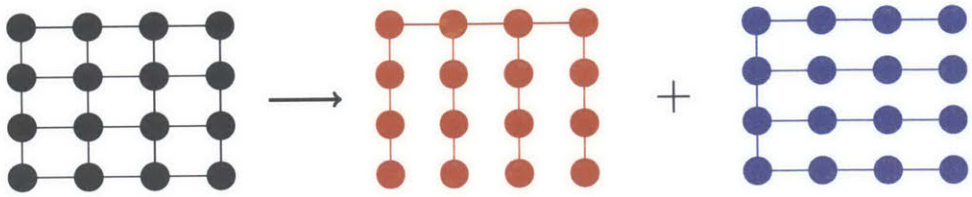


Figure 3-2: Sketch of a relaxation from Komodakis et al. (2007) for MAP inference in a grid structured Markov random field. The relaxation decomposes a loopy MRF into two interleaved tree-structured problems that can be solved exactly with belief propagation.

of the work of Sontag, Globerson, and Jaakkola (2010), for a description. Methods based on LP relaxations have the benefit of stronger guarantees than loopy belief propagation. Inference is cast as an optimization problem, for example the problem of minimizing a dual. Since the dual problem is convex, convergence results from convex optimization and linear programming can be leveraged directly. One particularly appealing feature of these methods is that certificates of optimality can be given when the exact solution to the MAP problem is found.

Komodakis et al. (2007, 2011) describe a dual decomposition method that provably optimizes the dual of an LP relaxation of the MAP problem using a subgradient method, illustrated in Figure 3-2. This work is a crucial reference for this tutorial. (Note that in addition, Johnson, Malioutov, & Willsky, 2007, also describes LR methods for inference in MRFs.)

Combinatorial Algorithms in Belief Propagation A central idea in the algorithms we describe is the use of combinatorial algorithms other than max-product BP. This idea is closely related to earlier work on the use of combinatorial algorithms within belief propagation, either for the MAP inference problem (Duchi et al., 2007b), or for computing marginals (Smith & Eisner, 2008a). These methods generalize loopy BP in a way that allows the use

of combinatorial algorithms. Again, we argue that methods based on Lagrangian relaxation are preferable to variants of loopy BP, as they have stronger formal guarantees.

Linear Programs for Decoding in Natural Language Processing Dual decomposition and Lagrangian relaxation are closely related to integer linear programming (ILP) approaches, and linear programming relaxations of ILP problems. Several authors have used integer linear programming directly for solving challenging problems in NLP. Germann, Jahr, Knight, Marcu, and Yamada (2001) use ILP to test the search error of a greedy phrase-based translation system on short sentences. Roth and Yih (2005a) formulate a constrained sequence labeling problem as an ILP and decode using a general-purpose solver. Lacoste-Julien, Taskar, Klein, and Jordan (2006) describe a quadratic assignment problem for bilingual word alignment and then decode using an ILP solver. Both the work of Riedel and Clarke (2006a) and Martins, Smith, and Xing (2009a) formulates higher-order non-projective dependency parsing as an ILP. Riedel and Clarke decode using an ILP method where constraints are added incrementally. Martins et al. solve an LP relaxation and project to a valid dependency parse. Like many of these works, the method presented in this tutorial begins with an ILP formulation of the inference problem; however, instead of employing a general-purpose solver we aim to speed up inference by using combinatorial algorithms that exploit the underlying structure of the problem.

Part II

Relaxation Methods

Chapter 4

Joint Syntactic Decoding

[This chapter is adapted from joint work with David Sontag, Michael Collins, and Tommi Jaakkola entitled "On Dual Decomposition and Linear Programming Relaxations for Natural Language Processing" (Rush et al., 2010)]

Natural language is a complex and interconnected system. While it is common to pose natural language decoding problems as isolated tasks—e.g. predict the best part-of-speech tags, predict the best parse for a sentence—in practice different elements of language are highly coupled. The interplay between different prediction tasks motivates a joint decoding approach.

This chapter focuses on algorithms for joint decoding of syntactic models. In particular we consider decoding problems of the form

$$\arg\max_{y \in \mathcal{Y}, z \in \mathcal{Z}} f(y) + g(z) \text{ such that } y \text{ and } z \text{ agree}$$

for some definition of agreement. These problems can often be quite challenging to solve, even in the case when each of the individual decoding problems are easy. In particular we will look at two examples of joint decoding problems. We begin our discussion by exploring a model that combines two classical problems in natural language syntax: part-of-speech tagging and syntactic parsing. This problem provides a concrete framework for the relaxation algorithms discussed in the previous chapter. The second example is a problem that combines context-

free parsing with a rich dependency parsing model.

The central method of this chapter is dual decomposition (DD). We first give a formal definition of the problem, describe motivation for the problem and previous approaches, and then describe the dual decomposition algorithms. We then connect this approach to the formal properties described in the previous chapter. These algorithms act as concrete examples of the general DD approach.

4.1 Decoding for Parsing and Tagging

We now describe the individual decoding models for parsing and tagging. We take some care to set up notation that will allow us to make a clear connection between the decoding problems and their formal properties.

Our first model is for weighted CFG parsing. We assume a context-free grammar in Chomsky normal form with a set of non-terminals $\mathcal{N}$. We also assume that the set of part-of-speech (POS) tags $\mathcal{T}$ is contained within the set of nonterminals $\mathcal{T} \subset \mathcal{N}$. The grammar $\mathcal{G}$ contains only rules of the form $X \rightarrow Y\ Z$, where $X \in \mathcal{N} \setminus \mathcal{T}$ and $Y, Z \in \mathcal{N}$, and $X \rightarrow s$ where $X \in \mathcal{T}$ and $s \in \Sigma$ is word from dictionary Σ. It is simple to relax this assumption to give a less constrained grammar. For a more detailed description of context-free grammars see background Section 2.2.3.

Given a sentence with n words, $s_1, s_2, \ldots s_n$, a parse tree is made up of a set of rule productions of the form $\langle X \rightarrow Y\ Z, i, k, j \rangle$ where $X, Y, Z \in \mathcal{N}$, and $1 \leq i \leq k < j \leq n$. Each rule production represents the use of CFG rule $X \rightarrow Y\ Z$ where non-terminal X spans words $s_i \ldots s_j$, non-terminal Y spans words $s_i \ldots s_k$, and non-terminal Z spans words $s_{k+1} \ldots s_j$. There are $O(|\mathcal{G}|n^3)$ such rule productions. Each parse tree corresponds to a subset of these rule productions, of size $n - 1$, that forms a well-formed parse tree.[1]

[1] We do not require rules of the form $X \rightarrow s_i$ in this representation, as they are redundant: specifically, a rule production $\langle X \rightarrow Y\ Z, i, k, j \rangle$ implies a rule $X \rightarrow s_i$ iff $i = k$, and $Y \rightarrow s_j$ iff $j = k + 1$.

Define the index set for CFG parsing as

$$\mathcal{I} = \{\langle X \rightarrow Y\ Z, i, k, j\rangle \ : \ X \in \mathcal{N} \setminus \mathcal{T},\ Y, Z \in \mathcal{N},\ 1 \leq i \leq k < j \leq n\}$$

Each parse tree is a vector $y = \{y(r) : r \in \mathcal{I}\}$, with $y(r) = 1$ if rule r is in the parse tree, and $y_r = 0$ otherwise. We use $\mathcal{Y}$ to denote the set of all valid parse-tree vectors; the set $\mathcal{Y}$ is a subset of $\{0,1\}^{\mathcal{I}}$ (not all binary vectors correspond to valid parse trees).

In addition, we assume a vector $\theta \in \mathbb{R}^{\mathcal{I}}$ that specifies a weight for each rule production.[2] The optimal parse tree is

$$y^* = \arg\max_{y \in \mathcal{Y}} \ \theta^\top y$$

where $\theta^\top y = \sum_{r \in \mathcal{I}} y(r)\theta(r)$ is the inner product between θ and y.

We will use similar notation for other problems. As a second example, in POS tagging the task is to map a sentence of n words $s_1 \ldots s_n$ to a tag sequence $t_1 \ldots t_n$, where each t_i is chosen from a set $\mathcal{T}$ of possible tags. We assume a trigram tagger, where a tag sequence is represented through indices (i, A, B, C) where $A, B, C \in \mathcal{T}$, and $i \in \{3 \ldots n\}$. Each production represents a transition where C is the tag of word s_i, and (A, B) are the previous two tags. The index set for tagging is

$$\mathcal{J} = \{(i, A, B, C) \ : \ A, B, C \in \mathcal{T},\ 3 \leq i \leq n\}$$

Note that we do not need transitions for $i = 1$ or $i = 2$, because the transition $(3, A, B, C)$ specifies the first three tags in the sentence.[3]

Each tag sequence is represented as a vector $z \in \{0,1\}^{\mathcal{J}}$, and we denote the set of valid tag sequences, a subset of $\{0,1\}^{\mathcal{J}}$, as $\mathcal{Z}$. Given a parameter vector $\omega \in \mathbb{R}^{\mathcal{J}}$, the optimal tag sequence is $\arg\max_{z \in \mathcal{Z}} \omega^\top z$. For a longer discussion of trigram tagging see background

[2] We do not require parameters for rules of the form $X \rightarrow s$, as they can be folded into rule production parameters. E.g., under a PCFG we define $\theta(X \rightarrow Y\ Z, i, k, j) = \log p(X \rightarrow Y\ Z \mid X) + \delta_{i,k} \log p(Y \rightarrow s_i | Y) + \delta_{k+1,j} \log p(Z \rightarrow s_j | Z)$ where $\delta_{x,y} = 1$ if $x = y$, 0 otherwise.

[3] As one example, in an HMM, the parameter $\theta(3, A, B, C)$ would be $\log p(A|\texttt{<t>}, \texttt{<t>}) + \log p(B|\texttt{<t>}, A) + \log p(C|A, B) + \log p(s_1|A) + \log p(s_2|B) + \log p(s_3|C)$, where <t> is the tag start symbol. The full HMM derivation is discussed in Section 2.1.2.

Section 2.1.1.

As a modification to the above approach, we will find it convenient to introduce extended index sets for both the CFG parsing and POS tagging examples. For the CFG case we define the extended index set to be $\mathcal{I}' = \mathcal{I} \cup \mathcal{I}_{\text{uni}}$ where

$$\mathcal{I}_{\text{uni}} = \{(i, t) : i \in \{1 \dots n\}, t \in \mathcal{T}\}$$

Here each pair (i, t) represents word s_i being assigned the tag t. Thus each parse-tree vector y will have additional (binary) components $y(i, t)$ specifying whether or not word i is assigned tag t. Note that this representation is redundant, since a parse tree determines a unique tagging for a sentence: more explicitly, for any $i \in \{1 \dots n\}$, $Y \in \mathcal{T}$, the following linear constraint holds:

$$\begin{aligned} y(i, Y) \quad = \quad & \sum_{k=i+1}^{n} \sum_{X,Z \in \mathcal{N}} y(X \rightarrow Y\ Z, i, i, k) + \\ & \sum_{k=1}^{i-1} \sum_{X,Z \in \mathcal{N}} y(X \rightarrow Z\ Y, k, i-1, i) \end{aligned}$$

We apply the same extension to the tagging index set, effectively mapping trigrams down to unigram assignments, again giving an over-complete representation. The extended index set for tagging is given by $\mathcal{J}' = \mathcal{J} \cup \mathcal{I}_{\text{uni}}$.

From here on we will make exclusive use of extended index sets for CFG parsing and trigram tagging. We use the set $\mathcal{Y}$ to refer to the set of valid parse structures under the extended representation; each $y \in \mathcal{Y}$ is a binary vector of length $|\mathcal{I}'|$. We similarly use $\mathcal{Z}$ to refer to the set of valid tag structures under the extended representation. We assume parameter vectors for the two problems, $\theta \in \mathbb{R}^{\mathcal{I}'}$ and $\omega \in \mathbb{R}^{\mathcal{J}'}$.

4.2 Two Joint Decoding Examples

This section describes the dual decomposition approach for two decoding problems in NLP. The first problem we consider is phrase-structure parsing with integrated part-of-speech tagging. The second problem concerns the integration of phrase-structure and dependency parsing. In both cases exact decoding algorithms exist, through methods that intersect dynamic programs, but the algorithms that we derive are considerably more efficient.

4.2.1 Integrated Parsing and Trigram Tagging

We now describe the dual decomposition approach for integrated parsing and trigram tagging. First, define the set $\mathcal{W}$ as follows:

$$\begin{aligned}\mathcal{W} = \{(y, z) \ : \ & y \in \mathcal{Y}, z \in \mathcal{Z}, \\ & y(i,t) = z(i,t) \text{ for all } (i,t) \in \mathcal{I}_{\text{uni}}\}\end{aligned} \tag{4.1}$$

Hence $\mathcal{W}$ is the set of all (y, z) pairs that agree on their part-of-speech assignments. The integrated parsing and trigram tagging problem is then to solve

$$\max_{(y,z)\in\mathcal{W}} \theta^\top y + \omega^\top z \tag{4.2}$$

This problem is equivalent to

$$\max_{y\in\mathcal{Y}} \theta^\top y + \omega^\top l(y)$$

where $l : \mathcal{Y} \mapsto \mathcal{Z}$ is a function that maps a parse tree y to its set of trigrams $z = l(y)$. The benefit of the formulation in Eq. 4.2 is that it makes explicit the idea of maximizing over all pairs (y, z) under a set of agreement constraints $y(i,t) = z(i,t)$—this intuition will be central to the algorithms in this chapter.

With this in mind, we note that we have efficient methods for the decoding problems of tagging and parsing alone, and that our combined objective almost separates into these two independent problems. In fact, if we drop the $y(i,t) = z(i,t)$ constraints from the

optimization problem, the problem splits into two parts, which can each be solved relatively easily using dynamic programming:

$$(y^*, z^*) = (\arg\max_{y \in \mathcal{Y}} \theta^\top y, \arg\max_{z \in \mathcal{Z}} \omega^\top z)$$

Set $\lambda^{(1)}(i,t) = 0$ for all $(i,t) \in \mathcal{I}_{\text{uni}}$
for $k = 1$ to K **do**
  $y^{(k)} \leftarrow \arg\max_{y \in \mathcal{Y}} \left(\theta^\top y + \sum_{i,t} \lambda^{(k)}(i,t) y(i,t) \right)$
  $z^{(k)} \leftarrow \arg\max_{z \in \mathcal{Z}} \left(\omega^\top z - \sum_{i,t} \lambda^{(k)}(i,t) z(i,t) \right)$
  if $y^{(k)}(i,t) = z^{(k)}(i,t)$ for all $(i,t) \in \mathcal{I}_{\text{uni}}$ **then**
    return $(y^{(k)}, z^{(k)})$
  else
    $\lambda^{(k+1)}(i,t) \leftarrow \lambda^{(k)}(i,t) - \alpha_k (y^{(k)}(i,t) - z^{(k)}(i,t))$ for all $(i,t) \in \mathcal{I}_{\text{uni}}$

Algorithm 6: The algorithm for integrated parsing and tagging. The parameters $\alpha_k > 0$ for $k = 1 \ldots K$ specify step sizes for each iteration. The two arg max *problems can be solved using dynamic programming algorithms described in the background chapter.*

Dual decomposition exploits this idea; it results in the method given in Algorithm 6. The algorithm optimizes the combined objective by repeatedly solving the two sub-problems separately—that is, it directly solves the harder optimization problem using an existing CFG parser and trigram tagger. After each iteration the algorithm adjusts the weights $\lambda(i,t)$; these updates modify the objective functions for the two models, encouraging them to agree on the same POS sequence. As discussed in the previous chapter, the variables $\lambda(i,t)$ are Lagrange multipliers enforcing agreement constraints, and the algorithm corresponds to a subgradient method for optimization of a dual function. The algorithm is easy to implement: all that is required is a decoding algorithm for each of the two models, and simple additive updates to the Lagrange multipliers enforcing agreement between the two models.

Next, consider the efficiency of the algorithm. Each iteration of the algorithm requires decoding under each of these two models. If the number of iterations k is relatively small, the

algorithm can be much more efficient than using a classical dynamic programming approach. Assuming a context-free grammar in Chomsky normal form, and a trigram tagger with tags from $\mathcal{T}$, the CKY parsing algorithm takes $O(|\mathcal{G}|n^3)$ time, and the Viterbi algorithm for tagging takes $O(|\mathcal{T}|^3 n)$ time. Thus the total running time for the dual decomposition algorithm is $O(k(|\mathcal{G}|n^3 + |\mathcal{T}|^3 n))$ where k is the number of iterations required for convergence.

In contrast, the classical dynamic programming approach involves intersecting a context-free grammar with a finite-state tagger following the construction of Bar-Hillel et al. (1964). This approach is described in detail in background Section 2.2.5 and is guaranteed to produce an optimal solution to the problem. However this method results in an algorithm with running time of $O(|\mathcal{G}||\mathcal{T}|^6 n^3)$. The dual decomposition algorithm results in an *additive* cost for incorporating a tagger (a $|\mathcal{T}|^3 n$ term is added into the runtime), whereas the construction of Bar-Hillel et al. results in a much more expensive *multiplicative* cost (a $|\mathcal{T}|^6$ term is multiplied into the runtime).

4.2.2 Integrating Two Lexicalized Parsers

Our second example problem is the integration of a phrase-structure parser with a higher-order dependency parser. The goal is to add higher-order features to phrase-structure parsing without greatly increasing the complexity of decoding.

First, we define an index set for second-order projective dependency parsing. The second-order parser considers first-order dependencies, as well as sibling and grandparent second-order dependencies (e.g., see Carreras (2007)). We assume that $\mathcal{J}$ is an index set containing all such dependencies. For convenience we define an extended index set that makes explicit use of first-order dependencies, $\mathcal{J}' = \mathcal{J} \cup \mathcal{I}_{\text{arc}}$, where

$$\mathcal{I}_{\text{arc}} = \{(h, m) : h \in \{0 \ldots n\}, m \in \{1 \ldots n\}, h \neq m\}$$

Here (h, m) represents a dependency with head s_h and modifier s_m ($h = 0$ corresponds to the root symbol in the parse). We use $\mathcal{Z} \subseteq \{0, 1\}^{\mathcal{J}'}$ to denote the set of valid projective

dependency parses.

The second model we use is a lexicalized CFG. Each symbol in the grammar takes the form $X(h)$ where $X \in \mathcal{N}$ is a non-terminal, and $h \in \{1 \dots n\}$ is an index specifying that s_h is the head of the constituent. Rule productions take the form $\langle X(h) \rightarrow Y(b)\ Z(c), i, k, j \rangle$ where $b \in \{i \dots k\}$, $c \in \{(k+1) \dots j\}$, and h is equal to b or c, depending on whether X receives its head-word from its left or right child. Each such rule implies a dependency (h, b) if $h = c$, or (h, c) if $h = b$. Define the index set $\mathcal{I}$ to be

$$\mathcal{I} = \big\{ \langle X(h) \rightarrow Y(b)\ Z(c), i, k, j \rangle \ : \ X, Y, Z \in \mathcal{N},\ 1 \leq i \leq k < j \leq n, \\ b \in \{i \dots k\}, c \in \{(k+1) \dots j\}, h \in \{b, c\} \big\}$$

We take $\mathcal{I}' = \mathcal{I} \cup \mathcal{I}_{\text{arc}}$ to be the extended index set. We define $\mathcal{Y} \subseteq \{0,1\}^{\mathcal{I}'}$ to be the set of valid parse trees.

The integrated parsing problem is then to find

$$(y^*, z^*) = \arg\max_{(y,z) \in \mathcal{W}} \theta^\top y + \omega^\top z \tag{4.3}$$

$$\text{where} \quad \mathcal{W} = \{(y, z) : y \in \mathcal{Y}, z \in \mathcal{Z}, \\ y(h, m) = z(h, m) \text{ for all } (h, m) \in \mathcal{I}_{\text{arc}} \ \} \tag{4.4}$$

This problem has a very similar structure to the problem of integrated parsing and tagging, and we can derive a similar dual decomposition algorithm. The Lagrange multipliers λ are a vector in $\mathbb{R}^{\mathcal{I}_{\text{arc}}}$ enforcing agreement between dependency assignments. Algorithm 7 is identical to Algorithm 6, but with $\mathcal{I}_{\text{uni}}$ replaced with $\mathcal{I}_{\text{arc}}$. The algorithm only requires decoding algorithms for the two models, together with simple updates to the Lagrange multipliers.

Example 4.2.1 (A Run of the Algorithm). We now give an example run of the algorithm for combined CFG parsing and part-of-speech tagging. For simplicity, we will assume that the step size α_k is equal to 1 for all iterations k. We take the input sentence to be `United flies`

Set $\lambda^{(1)}(h,m) = 0$ for all $(h,m) \in \mathcal{I}_{\text{arc}}$
for $k = 1$ to K **do**
$$y^{(k)} \leftarrow \arg\max_{y \in \mathcal{Y}} \left(\theta^\top y + \sum_{h,m \in \mathcal{I}_{\text{arc}}} \lambda^{(k)}(h,m) y(h,m) \right)$$
$$z^{(k)} \leftarrow \arg\max_{z \in \mathcal{Z}} \left(\omega^\top z - \sum_{h,m \in \mathcal{I}_{\text{arc}}} \lambda^{(k)}(h,m) z(h,m) \right)$$
if $y^{(k)}(h,m) = z^{(k)}(h,m)$ for all $(h,m) \in \mathcal{I}_{\text{arc}}$ **then**
return $(y^{(k)}, z^{(k)})$
else
$\lambda^{(k+1)}(h,m) \leftarrow \lambda^{(k)}(h,m) - \alpha_k(y^{(k)}(h,m) - z^{(k)}(h,m))$ for all $(h,m) \in \mathcal{I}_{\text{arc}}$

Algorithm 7: The algorithm for integrated constituency and dependency parsing. The arg max *problems can again be solved using dynamic programming.*

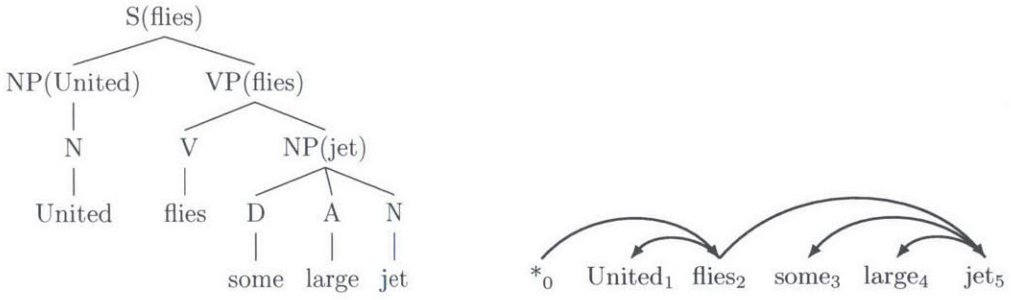
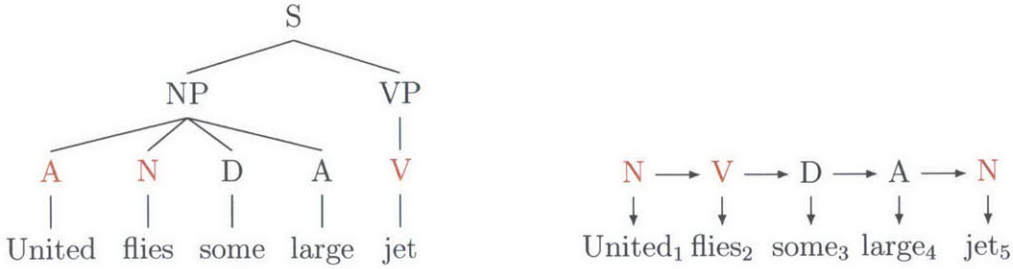


Figure 4-1: A lexicalized parse tree, and a dependency structure.

`some large jet`. Initially, the algorithm sets $\lambda(i,t) = 0$ for all (i,t). For our example, decoding with these initial weights leads to the two hypotheses



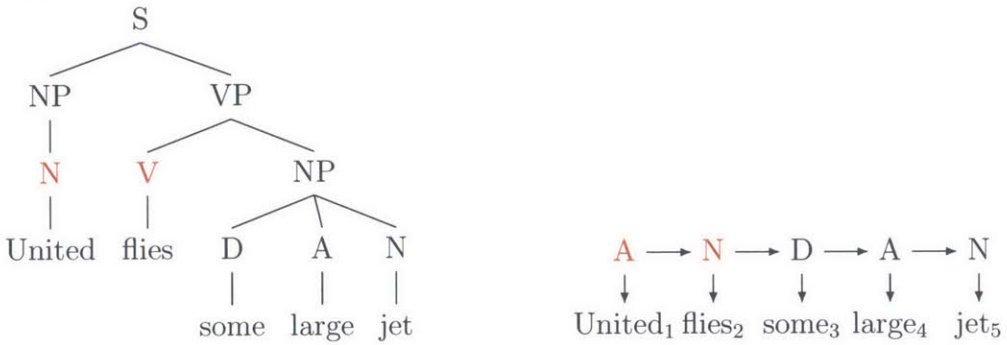
These two structures have different part-of-speech tags at three positions, highlighted in red; thus the two structures do not agree. We then update the $\lambda(i,t)$ variables based on

these differences, giving new values as follows:

$$\lambda(1, \mathrm{A}) = \lambda(2, \mathrm{N}) = \lambda(5, \mathrm{V}) = -1$$

$$\lambda(1, \mathrm{N}) = \lambda(2, \mathrm{V}) = \lambda(5, \mathrm{N}) = 1$$

Any $\lambda(i, t)$ values not shown still have value 0. We now decode with these new $\lambda(i, t)$ values, giving structures



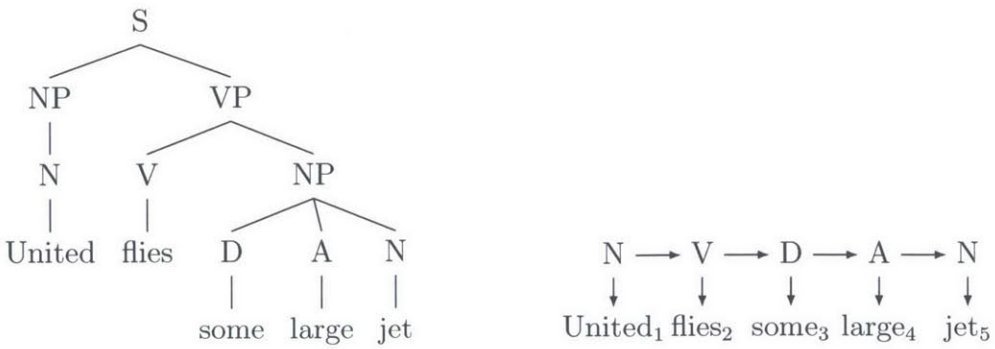
Again, differences between the structures are shown in red. We update the $\lambda(i, t)$ values to obtain new values as follows:

$$\lambda(5, \mathrm{N}) = -1$$

$$\lambda(5, \mathrm{V}) = 1$$

with all other $\lambda(i, t)$ values being 0. (Note that the updates reset $\lambda(1, \mathrm{A})$, $\lambda(1, \mathrm{N})$, $\lambda(2, \mathrm{N})$ and $\lambda(2, \mathrm{V})$ back to zero.)

We decode again, with the new $\lambda(i, t)$ values; this time, the two structures are



These two structures have identical sequences of part-of-speech tags, and the algorithm terminates, with the guarantee that the solutions are optimal.

4.3 Relationship to the Formal Definition

It is easily verified that the approach we have described is an instance of the dual decomposition framework described in Chapter 3. For instance, for combined part-of-speech tagging and parsing, the set $\mathcal{Y}$ is the set of all parses for the input sentence; the set $\mathcal{Z}$ is the set of all part-of-speech sequences for the input sentence. The constraints

$$y(i,t) = z(i,t)$$

for all (i,t) can be encoded through linear constraints

$$Ay + Cz = b$$

We set $b = \vec{0}$ and make suitable choices for $A \in \mathbb{R}^{p \times |\mathcal{I}'|}$ and $C \in \mathbb{R}^{p \times |\mathcal{J}'|}$ where $p = n \times |\mathcal{T}|$. Informally, we select A to project y from $\mathcal{Y}$ to $\{0,1\}^{\mathcal{I}_{\text{uni}}}$ and C to project z from $\mathcal{Z}$ to $\{0,1\}^{\mathcal{I}_{\text{uni}}}$ and to negate the vector.

Similarly we can show that the example of lexicalized context-free parsing and dependency parsing can be represented in the general form.

4.4 Marginal Polytopes and LP Relaxations

The dual decomposition algorithm can also be shown to be solving a linear programming relaxation of the original problem. To make the connection to linear programming, we first introduce the idea of *marginal polytopes*. We then give a precise statement of the LP relaxations that are being solved by making direct use of marginal polytopes. In Section 3.3 we will prove that the example algorithms solve these LP relaxations.

4.4.1 Marginal Polytopes

For a finite set $\mathcal{Y}$, define the set of all distributions over elements in $\mathcal{Y}$ as

$$\Delta = \left\{\alpha \in \mathbb{R}^{\mathcal{Y}} : \alpha_y \geq 0, \sum_{y \in \mathcal{Y}} \alpha_y = 1\right\}$$

Each $\alpha \in \Delta$ gives a vector of marginals, $\mu = \sum_{y \in \mathcal{Y}} \alpha_y y$, where $\mu \in \mathbb{R}^{\mathcal{I}}$ can be interpreted as the probability that $y(r) = 1$ for some $r \in \mathcal{I}$ and for a y selected at random from the distribution α.

The set of all possible marginal vectors, known as the *marginal polytope*, is defined as follows:

$$\mathcal{M} = \{\mu \in \mathbb{R}^{\mathcal{I}} : \exists \alpha \in \Delta \text{ such that } \mu = \sum_{y \in \mathcal{Y}} \alpha_y y\}$$

$\mathcal{M}$ is also frequently referred to as the *convex hull* of $\mathcal{Y}$, written as $\text{conv}(\mathcal{Y})$. We will use the notation $\text{conv}(\mathcal{Y})$ in the remainder.

For an arbitrary set $\mathcal{Y}$, the marginal polytope $\text{conv}(\mathcal{Y})$ can be quite complex.[4] However, Martin et al. (1990) show that for a very general class of dynamic programming problems, the implied marginal polytope can be expressed in the form

$$\text{conv}(\mathcal{Y}) = \{\mu \in \mathbb{R}^{\mathcal{I}} : D\mu = f, \mu \geq 0\} \tag{4.5}$$

where D is a $p \times m$ matrix, f is vector in $\mathbb{R}^p$, and the value p is linear in the size of a hypergraph representation of the dynamic program. Note that D and f specify a set of p linear constraints.

We now give an explicit description of the resulting constraints for CFG parsing:[5] similar constraints arise for other dynamic programming algorithms for parsing, for example the algorithms of Eisner (2000). The exact form of the constraints, and the fact that they

[4] For any finite set $\mathcal{Y}$, $\text{conv}(\mathcal{Y})$ can be expressed as $\{\mu \in \mathbb{R}^k : A\mu \leq b\}$ where A is a matrix of dimension $p \times |\mathcal{I}|$, and $b \in \mathbb{R}^p$ (e.g., Korte and Vygen (2008), page 65). However the value for p depends on the set $\mathcal{I}$, and can be exponential in size.

[5] Taskar et al. (2004) describe the same set of constraints, but without proof of correctness or reference to Martin et al. (1990).

$$\sum_{\substack{X,Y,Z\in\mathcal{N}\\ k=\{1\ldots(n-1)\}}} \mu(X \rightarrow Y\ Z, 1, k, n) = 1, \tag{4.6}$$

$$\forall X \in \mathcal{N}, 1 \leq i < j \leq n, (i,j) \neq (1,n): \tag{4.7}$$
$$\sum_{\substack{Y,Z\in\mathcal{N}\\ k\in\{i\ldots(j-1)\}}} \mu(X \rightarrow Y\ Z, i, k, j) = \sum_{\substack{Y,Z\subset\mathcal{N}\\ k\in\{1\ldots(i-1)\}}} \mu(Y \rightarrow Z\ X, k, i-1, j) + \sum_{\substack{Y,Z\in\mathcal{N}\\ k\in\{(j+1)\ldots n\}}} \mu(Y \rightarrow X\ Z, i, j, k),$$

$$\forall Y \in \mathcal{T}, i \in \{1 \ldots n\}:$$
$$\mu(i, Y) = \sum_{\substack{X,Z\in\mathcal{N}\\ k\in\{(i+1)\ldots n\}}} \mu(X \rightarrow Y\ Z, i, i, k) + \sum_{\substack{X,Z\in\mathcal{N}\\ k\in\{1\ldots(i-1)\}}} \mu(X \rightarrow Z\ Y, k, i-1, i) \tag{4.8}$$

Figure 4-2: The linear constraints defining the marginal polytope for CFG parsing for $\mu \in \mathbb{R}^{\mathcal{I}'}$ with $\mu \geq 0$. These constraints can be derived directly from the hypergraph encoding of context-free grammars described in the background chapter.

are polynomial in number, is not essential for the formal results in this chapter. However, a description of the constraints gives valuable intuition for the structure of the marginal polytope.

The constraints are given in Figure 4-2. To develop some intuition, consider the case where the variables $\mu(r)$ are restricted to be binary: hence each binary vector μ specifies a parse tree. The second constraint in Eq. 4.6 specifies that exactly one rule must be used at the top of the tree. The set of constraints in Eq. 4.7 specify that for each production of the form $\langle X \rightarrow Y\ Z, i, k, j \rangle$ in a parse tree, there must be exactly one production higher in the tree that generates (X, i, j) as one of its children. The constraints in Eq. 4.8 enforce consistency between the $\mu(i, Y)$ variables and rule variables higher in the tree. Note that the constraints in Eqs.(4.6–4.8) can be written in the form $A\mu = b$, $\mu \geq 0$, as in Eq. 4.5.

Under these definitions, we have the following:

Theorem 4.4.1. *Define $\mathcal{Y}$ to be the set of all CFG parses, as defined in Section 4.2. Then*

$$\text{conv}(\mathcal{Y}) = \{\mu \in \mathbb{R}^{\mathcal{I}} : \mu \text{ satisifies Eqs.(4.6–4.8)}\}$$

$$\sum_{A,B,C \in \mathcal{T}} \nu(3, A, B, C) = 1, \quad \sum_{A,B,C \in \mathcal{T}} \nu(n, A, B, C) = 1,$$

$$\forall \ i \in \{3 \ldots (n-1)\}, B, C \in \mathcal{T} : \quad \sum_{A \in \mathcal{T}} \nu(i, A, B, C) = \sum_{D \in \mathcal{T}} \nu(i+1, B, C, D),$$

$$\forall \ i \in \{3 \ldots (n-1)\}, C \in \mathcal{T} : \quad \nu(i, C) = \sum_{A,B \in \mathcal{T}} \nu(i, A, B, C),$$

$$\forall A \in \mathcal{T} : \quad \nu(1, A) = \sum_{B,C \in \mathcal{T}} \nu(3, A, B, C),$$
$$\forall B \in \mathcal{T} : \quad \nu(2, B) = \sum_{A,C \in \mathcal{T}} \nu(3, A, B, C)$$

Figure 4-3: The linear constraints defining the marginal polytope for trigram POS tagging for $\nu \in \mathbb{R}^{\mathcal{J}'}$ with $\nu \geq 0$.

Proof: This theorem is a special case of Martin et al. (1990), theorem 2.

The marginal polytope for tagging, $\text{conv}(\mathcal{Z})$, can also be expressed using linear constraints as in Eq. 4.5. This follows from well-known results for graphical models (Wainwright & Jordan, 2008), or from the hypergraph construction. See Figure 4-3 for the full set of constraints.

As a final point, the following theorem gives an important property of marginal polytopes, which we will use at several points in this paper:

Theorem 4.4.2. *(Korte and Vygen (2008), page 66.) For any set $\mathcal{Y} \subseteq \{0,1\}^k$, and for any vector $\theta \in \mathbb{R}^k$,*

$$\max_{y \in \mathcal{Y}} \theta^\top y = \max_{\mu \in \text{conv}(\mathcal{Y})} \theta^\top \mu$$

The theorem states that for a linear objective function, maximization over a discrete set

$\mathcal{Y}$ can be replaced by maximization over the convex hull $\text{conv}(\mathcal{Y})$. The problem

$$\max_{\mu \in \text{conv}(\mathcal{Y})} \theta^\top \mu$$

is a linear programming problem.

For parsing, this theorem implies that:

1. Weighted CFG parsing can be framed as a linear programming problem, of the form $\max_{\mu \in \text{conv}(\mathcal{Y})} \theta^\top \mu$, where $\text{conv}(\mathcal{Y})$ is specified by a polynomial number of linear constraints.

2. Conversely, dynamic programming algorithms such as the CKY algorithm can be considered to be oracles that efficiently solve LPs of the form $\max_{\mu \in \text{conv}(\mathcal{Y})} \theta^\top \mu$.

Similar results apply for the POS tagging case.

In the next section we prove that Algorithm 6 solves the problem in Eq 4.10. A similar result holds for the algorithm in Section 4.2.2: it solves a relaxation of Eq. 4.4, where $\mathcal{W}$ is replaced by

$$\begin{aligned} \mathcal{Q} = \{(\mu, \nu) \quad : \quad & \mu \in \text{conv}(\mathcal{H}), \nu \in \text{conv}(\mathcal{D}), \\ & \mu(i,j) = \nu(i,j) \text{ for all } (i,j) \in \mathcal{I}_{\text{first}}\} \end{aligned}$$

4.4.2 Linear Programming Relaxations

We now describe the LP relaxations that are solved by the example algorithms in Section 4.2. We begin with Algorithm 6.

Recall that the feasible set of both of the combinatorial problems is defined as

$$\mathcal{W} \quad = \quad \{(y,z) : y \in \mathcal{Y}, z \in \mathcal{Z}, Ay + Cz = b\}$$

The original optimization problem was to find $\max_{(y,z) \in \mathcal{W}} \theta^\top y + \omega^\top z$ (see Eq. 4.2). By

theorem 4.4.2, this is equivalent to solving

$$\max_{(\mu,\nu)\in \text{conv}(\mathcal{W})} \theta^\top \mu + \omega^\top \nu \tag{4.9}$$

To approximate this problem, we first define

$$\begin{aligned} \mathcal{Q} = \{(\mu,\nu) \quad : \quad & \mu \in \text{conv}(\mathcal{Y}), \nu \in \text{conv}(\mathcal{Z}), \\ & A\mu + C\nu = b\} \end{aligned}$$

This definition is very similar to the definition of $\mathcal{W}$, but with $\mathcal{Y}$ and $\mathcal{Z}$ replaced by $\text{conv}(\mathcal{Y})$ and $\text{conv}(\mathcal{Z})$. We then define the following problem:

$$\max_{(\mu,\nu)\in \mathcal{Q}} \theta^\top \mu + \omega^\top \nu \tag{4.10}$$

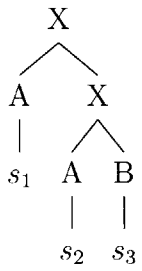
$\mathcal{Q}$ is a set that is defined by a finite set of linear constraints; hence this problem is a linear program. Eq. 4.10 is a *relaxation* of the problem in Eq. 4.9, in that we have replaced $\text{conv}(\mathcal{W})$ with $\mathcal{Q}$, and $\text{conv}(\mathcal{W})$ is a subset of $\mathcal{Q}$.

To see this, note that any point in $\mathcal{W}$ is clearly in $\mathcal{Q}$. It follows that any point in $\text{conv}(\mathcal{W})$ is also in $\mathcal{Q}$, because $\mathcal{Q}$ is a convex set.[6] The set $\text{conv}(\mathcal{W})$ is usually a strict subset of $\mathcal{Q}$, that is, the outer-bound includes points that are not in the feasible set.

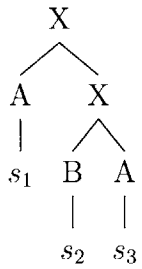
Example 4.4.1 (Fractional Solutions). We now give an example of a member (μ,ν) of the set $\mathcal{Q}\backslash\text{conv}(\mathcal{W})$, corresponding to a fractional extreme point of the polytope $\mathcal{Q}$ for joint part-of-speech tagging and parsing. Note that constructing such examples is non-trivial. We found this one by repeatedly solving the primal LP relaxation with random objectives. Recall that the constraints in $\mathcal{Q}$ specify that $\mu \in \text{conv}(\mathcal{Y})$, $\nu \in \text{conv}(\mathcal{Z})$, and $\mu(i,t) = \nu(i,t)$ for all $(i,t) \in \mathcal{I}_{\text{uni}}$. Given that $\mu \in \text{conv}(\mathcal{Y})$, it must be a convex combination of 1 or more members of $\mathcal{Y}$; a similar property holds for ν. We define the example as follows. There are two possible parts of speech, A and B, and an additional non-terminal symbol X. The

[6] This is easily verified, as $\mathcal{Q}$ is defined by linear constraints.

sentence is of length 3, s_1 s_2 s_3. Consider the case where ν is a convex combination of two tag sequences, each with weight 0.5, namely s_1/A s_2/A s_3/A and s_1/A s_2/B s_3/B. Take μ to be a convex combination of two parses, with weight 0.5 each, namely:



and



It can be verified that we have $\mu(i,t) = \nu(i,t)$ for all (i,t)—the marginals for single tags for μ and ν agree—even though both μ and ν are fractional.

To demonstrate that this fractional solution is a vertex of the polytope, we now give parameter values that give this fractional solution as the arg max of the decoding problem. For the tagging model, set $\theta(3, \text{A}, \text{A}, \text{A}) = \theta(3, \text{A}, \text{B}, \text{B}) = 0$, and all other parameters to be some negative value. For the parsing model, set $\theta(\text{X} \rightarrow \text{A} \ \ \text{X}, 1, 1, 3) = \theta(\text{X} \rightarrow \text{A} \ \text{B}, 2, 2, 3) = \theta(\text{X} \rightarrow \text{B} \ \ \text{A}, 2, 2, 3) = 0$, and all other rule parameters to be negative. Under these settings it can be verified that the fractional solution has value 0, while all integral solutions have a negative value, hence the arg max must be fractional.

LP Relaxations LP relaxations based on outer bounds of marginal polytopes have been applied in many papers on approximate inference in graphical models, with some success. In general, the solution to Eq. 4.10 may be in $\mathcal{Q}$ but not in $\text{conv}(\mathcal{W})$, in which case it will be fractional. However in several empirical settings the relaxation in Eq. 4.10 turns out to be tight, in that the solution is often integral (i.e., it is in $\mathcal{W}$). In these cases solving the

relaxed LP *exactly* solves the original problem of interest.

The formal properties for dual decomposition show that

$$\min_{\lambda \in \mathbb{R}^p} L(\lambda) = \max_{(\mu,\nu) \in \mathcal{Q}} \theta^\top \mu + \omega^\top \nu$$

The problem

$$(\mu^*, \nu^*) = \arg\max_{(\mu,\nu) \in \mathcal{Q}} \theta^\top \mu + \omega^\top \nu$$

is a linear programming problem, and $L(\lambda)$ is the dual of this linear program. In the case that (μ^*, ν^*) is also a member of $\mathcal{W}$, dual decomposition will produce the optimal solution to the original problem.

4.5 Related Work

The methods described in this chapter are related to several other lines of work in graphical models and natural language processing.

Variational inference (and linear programming) in graphical models has relied extensively on the idea of marginal polytopes, see for example Wainwright and Jordan (2008). Several papers have explored LP relaxations as a framework for deriving alternative algorithms to loopy belief propagation (LBP) (e.g., Globerson and Jaakkola (2007) and Sontag et al. (2008)).

Our work is similar in spirit to methods that incorporate combinatorial solvers within LBP, either for MAP inference (Duchi et al., 2007a) or for computing marginals (Smith & Eisner, 2008b). Our method aims to solve the MAP inference problem, but unlike LBP it has relatively clear convergence guarantees, in terms of solving an LP relaxation.

Other work has considered LP or integer linear programming (ILP) for inference in NLP (Martins et al., 2009b; Riedel & Clarke, 2006b; Roth & Yih, 2005b). This work uses general-purpose LP or ILP solvers. Our method has the advantage that it leverages underlying structure arising from LP formulations of NLP problems: we will see that dynamic programming

algorithms such as CKY can be considered to be very efficient solvers for particular LPs. In dual decomposition, these LPs and their efficient solvers—can be embedded within larger LPs corresponding to more complex inference problems.

Note also that Klein and Manning (2002) describe a method for combination of a dependency parser with a constituent-based parser, where the score for an entire structure is again the sum of scores under two models. In this approach an A* algorithm is developed, where admissible estimates within the A* method can be computed efficiently using separate decoding under the two models. There are interesting connections between the A* approach and the dual decomposition algorithm.

4.6 Experiments

4.6.1 Integrated Phrase-Structure and Dependency Parsing

Our first set of experiments considers the integration of Model 1 of Collins (2003) (a lexicalized phrase-structure parser, from here on referred to as Model 1),[7] and the second-order discriminative dependency parser of Koo et al. (2008). The decoding problem for a sentence is to find

$$y^* = \arg\max_{y \in \mathcal{Y}} \left(f(y) + \eta g(l(y)) \right) \tag{4.11}$$

where $\mathcal{Y}$ is the set of all lexicalized phrase-structure trees for a sentence; $f(y)$ is the score (log probability) under Model 1; $g(y)$ is the score under Koo et al. (2008) for the dependency structure implied by y; and $\eta > 0$ is a parameter dictating the relative weight of the two models. This problem is similar to the second example in Section 4.2; a very similar dual decomposition algorithm to that described in Section 4.2.2 can be derived.

We used the Penn Wall Street Treebank (Marcus et al., 1993) for the experiments, with Sections 2-21 for training, Section 22 for development, and Section 23 for testing. The parameter η was chosen to optimize performance on the development set.

[7] We use a reimplementation that is a slight modification of Collins Model 1, with very similar performance, and which uses the TAG formalism of Carreras et al. (2008).

Itn.	1	2	3	4	5-10	11-20	20-50	**
Dep	43.5	20.1	10.2	4.9	14.0	5.7	1.4	0.4
POS	58.7	15.4	6.3	3.6	10.3	3.8	0.8	1.1

Table 4.1: Convergence results for Section 23 of the WSJ Treebank for the dependency parsing and POS experiments. Each column gives the percentage of sentences whose exact *solutions were found in a given range of subgradient iterations. ** is the percentage of sentences that did not converge by the iteration limit (K=50).*

	Precision	Recall	F_1	Dep
Model 1	88.4	87.8	88.1	91.4
Koo08 Baseline	89.9	89.6	89.7	93.3
DD Combination	91.0	90.4	90.7	93.8

Table 4.2: Performance results for Section 23 of the WSJ Treebank. Model 1: a reimplementation of the generative parser of (Collins, 2002). Koo08 Baseline: Model 1 with a hard restriction to dependencies predicted by the discriminative dependency parser of (Koo et al., 2008). DD Combination: a model that maximizes the joint score of the two parsers. Dep shows the unlabeled dependency accuracy of each system.

We used the following step size in our experiments. First, we initialized α_0 to equal 0.5, a relatively large value. Then we defined $\alpha_k = \alpha_0 * 2^{-\eta_k}$, where η_k is the number of times that $L(\lambda^{(k')}) > L(\lambda^{(k'-1)})$ for $k' \leq k$. This learning rate drops at a rate of $1/2^t$, where t is the number of times that the dual increases from one iteration to the next.

We ran the dual decomposition algorithm with a limit of $K = 50$ iterations. Note again that the dual decomposition algorithm returns an exact solution if agreement occurs in Algorithm 7; we found that of 2416 sentences in Section 23, this occurs for 2407 (99.6%) sentences. Table 4.1 gives statistics showing the number of iterations required for convergence. Over 80% of the examples converge in 5 iterations or fewer; over 90% converge in 10 iterations or fewer. For the sentences that do not converge to an exact solution, we return the highest-scoring valid structure seen, i.e.

$$\arg\max_{y \in \{y^{(1)} \ldots y^{(K)}\}} f(y) + \eta g(l(y))$$

We compare the accuracy of the dual decomposition approach to two baselines: first, Model 1; and second, a naive integration method that enforces the hard constraint that

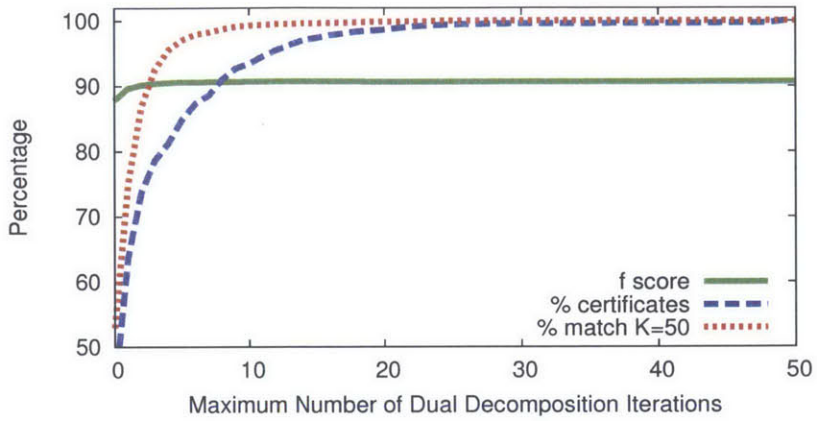


Figure 4-4: Performance on the parsing task assuming a fixed number of iterations K. f-score: accuracy of the method. % certificates: percentage of examples for which a certificate of optimality is provided. % match: percentage of cases where the output from the method is identical to the output when using $K = 50$.

Model 1 must only consider dependencies seen in the first-best output from the dependency parser. Table 4.2 shows all three results. The dual decomposition method gives a significant gain in precision and recall over the naive combination method, and boosts the performance of Model 1 to a level that is close to some of the best single-pass parsers on the Penn treebank test set. Dependency accuracy is also improved over the Koo et al. (2008) model, in spite of the relatively low dependency accuracy of Model 1 alone.

Figure 4-4 shows performance of the approach as a function of K, the maximum number of iterations of dual decomposition. For this experiment, for cases where the method has not converged for $k \leq K$, the output from the algorithm is chosen to be the $y^{(k)}$ for $k \leq K$ that maximizes the objective function in Eq. 4.11. The graphs show that values of K less than 50 produce almost identical performance to $K = 50$, but with fewer cases giving certificates of optimality (with $K = 10$, the f-score of the method is 90.69%; with $K = 5$ it is 90.63%).

Figure 4-5 shows the percentage of dependencies for which the outputs under the two models, $y^{(k)}$ and $z^{(k)}$, agree, as a function of k. If the method has converged on some example after some number k' of iterations, we take the disagreement rate for $k \geq k'$ to be 0 in this figure. The figure shows that the level of agreement between the two models decreases rapidly.

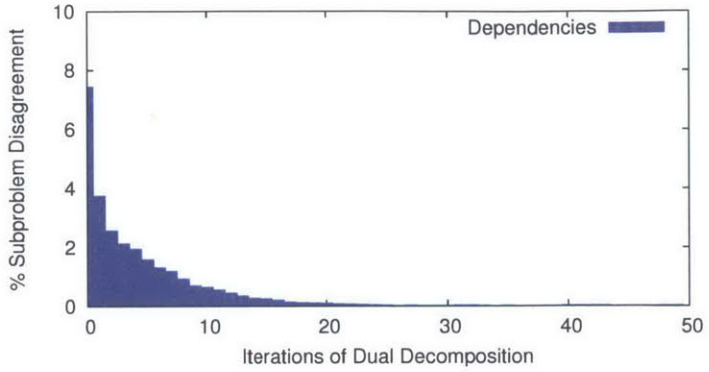


Figure 4-5: The average percentage of dependencies that disagreed at each iteration of dual decomposition.

	Precision	Recall	F_1	POS Acc
Fixed Tags	88.1	87.6	87.9	96.7
DD Combination	88.7	88.0	88.3	97.1

Table 4.3: Performance results for Section 23 of the WSJ. Model 1 (Fixed Tags): a baseline parser initialized to the best tag sequence of from the tagger of Toutanova and Manning (2000). DD Combination: a model that maximizes the joint score of parse and tag selection.

4.6.2 Integrated Phrase-Structure Parsing and Trigram tagging

In a second experiment, we used dual decomposition to integrate the Model 1 parser with the Stanford max-ent trigram POS tagger (Toutanova & Manning, 2000), using a very similar algorithm to that described in Section 4.2.1. We use the same training/dev/test split as in Section 4.6.1.

We ran the algorithm with a limit of $K = 50$ iterations. Out of 2416 test examples, the algorithm found an exact solution in 98.9% of the cases. Table 4.1 gives statistics showing the speed of convergence for different examples: over 94% of the examples converge to an exact solution in 10 iterations or fewer. In terms of accuracy, we compare to a baseline approach of using the first-best tag sequence as input to the parser. The dual decomposition approach gives 88.3 F1 measure in recovering parse-tree constituents, compared to 87.9 for the baseline.

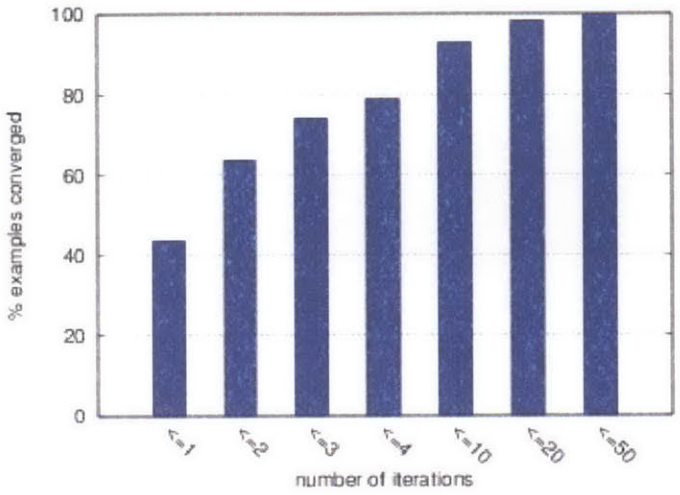


Figure 4-6: Convergence results for the integration of a lexicalized probabilistic context-free grammar, and a discriminative dependency parsing model. We show the percentage of examples where an exact solution is returned by the algorithm, versus the number of iterations of the algorithm.

4.7 Conclusion

This chapter introduced dual-decomposition algorithms for decoding joint syntactic models. The method finds exact solutions on over 99% of examples, with over 90% of examples e-converging in 10 iterations or less. Figure 4-6 shows a histogram of the number of examples that have e-converged, versus the number of iterations of the algorithm. The method gives significant gains in parsing accuracy over the model of Collins (1997), and significant gains over a baseline method that simply forces the lexicalized CFG parser to have the same dependency structure as the first-best output from the dependency parser.

More generally, the methods introduced in this chapter demonstrate their effectiveness of Lagrangian relaxation algorithms for NLP problems, particularly those that would traditionally be solved using intersections of dynamic programs (e.g., Bar-Hillel et al. (1964)). The problems described use simple agreement constraints to combine multiple models. These constraints act as concrete examples of the general dual decomposition approach. In the next two chapters we extend this idea to problems with more sophisticated constraints that encode richer linguistic properties.

Chapter 5

Syntax-Based Machine Translation

We now turn from syntactic parsing to machine translation. The two tasks are quite different; however we will see that the two problems have important similarities. Previously we intersected a parsing model with a finite-state tagging model, in this chapter we combine a syntax-based translation model with a finite-state language model.

Statistical machine translation aims to produce the highest-scoring translation for a given source sentence. There are many different statistical models of the translation process. In this thesis, we will focus two types of translation: (i) phrase-based translation, which is discussed in Chapter 7, and (ii) syntax-based translation which is the focus of this chapter and of Chapter 7.

For syntax-based systems, the translation derivation often takes the form of a *synchronous* parse tree over the input source sentence and a target sentence. From any synchronous derivation produced, the translation itself can be read off from this structure. An example synchronous derivation is shown in Figure 5-1.

We begin by introducing the decoding optimization problem for syntax-based translation, and then give a Lagrangian relaxation algorithm for this problem.

Note that this chapter relies heavily on the hypergraph formalism described in background Section 2.2, and we assume the reader has some familiarity with the specifics of the formalism.

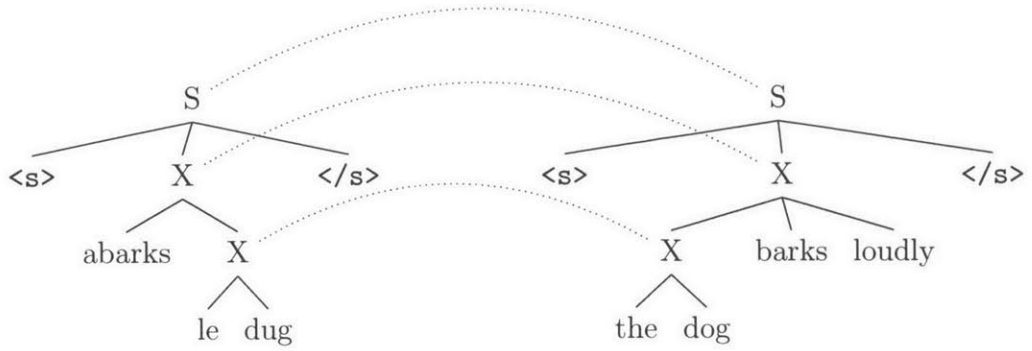


Figure 5-1: A toy example translation derivation in a synchronous context-free grammar. The source sentence is `abarks le dug` *and the target is* `the dog barks loudly`*. The dotted lines show the mapping between the non-terminals of the two sentences.*

5.1 Translation Hypergraphs

Translation with most syntax-based systems (e.g., (Chiang, 2005; Marcu et al., 2006; Shen et al., 2008; Huang & Mi, 2010)) can be implemented as a two-step process. The first step is to take an input sentence in the source language, and from this to create a hypergraph (sometimes called a translation forest) that represents the set of possible translations (strings in the target language) and derivations under the grammar. The second step is to integrate an n-gram language model with this hypergraph. For example, in the system of (Chiang, 2005), the hypergraph is created as follows: first, the source side of the synchronous grammar is used to create a parse forest over the source language string. Second, transduction operations derived from synchronous rules in the grammar are used to create the target-language hypergraph. Chiang's method uses a synchronous context-free grammar, but the hypergraph formalism is applicable to a broad range of other grammatical formalisms, for example dependency grammars (e.g., (Shen et al., 2008)).

Recall that a hypergraph is a pair $(\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \{1, 2, \ldots, |\mathcal{V}|\}$ is a set of vertices, and $\mathcal{E}$ is a set of hyperedges. A single distinguished vertex is taken as the root of the hypergraph; without loss of generality we take this vertex to be $v_0 = 1$. Each hyperedge $e \in \mathcal{E}$ is a tuple $\langle \langle v_2, \ldots, v_k \rangle, v_1 \rangle$ where $v_1 \in \mathcal{V}$, and $v_i \in \{2 \ldots |\mathcal{V}|\}$ for i in $\{2 \ldots k\}$. The vertex v_1 is referred to as the *head* of the edge. The ordered sequence $\langle v_2, \ldots, v_k \rangle$ is referred to as the *tail* of the

edge; in addition, we sometimes refer to $v_2, \ldots v_k$ as the *children* of the edge. The number of children k may vary across different edges, but $k \geq 1$ for all edges (i.e., each edge has at least one child). We will use $h(e)$ to refer to the head of an edge e, and $t(e)$ to refer to the tail.

We will assume that the hypergraph is acyclic and satisfies the reference set properties described in Martin et al. (1990). For the full definition see the description in the background chapter.

Each vertex $v \in \mathcal{V}$ is either a *non-terminal* in the hypergraph, or a *leaf*. The set of non-terminals is

$$\mathcal{V}^{(n)} = \{v \in \mathcal{V} : \exists e \in \mathcal{E} \quad \text{such that} \quad h(e) = v\}$$

Conversely, the set of terminals or leaves is defined as

$$\mathcal{V}^{(t)} = \{v \in \mathcal{V} : \nexists e \in \mathcal{E} \quad \text{such that} \quad h(e) = v\}$$

We also assume that each vertex $v \in \mathcal{V}$ has a label $l(v)$. The labels for leaf vertices in syntax-based translation will be *words*, and will be important in defining strings and language model scores for those strings. The labels for non-terminal nodes will not be important for results in this chapter.[1]

We now turn to derivations, also known as hyperpaths. Define an *index set* $\mathcal{I} = \mathcal{V} \cup \mathcal{E}$. A derivation is represented by a vector $y = \{y(r) : r \in \mathcal{I}\}$ where $y(v) = 1$ if vertex v is used in the derivation, $y(v) = 0$ otherwise (similarly $y(e) = 1$ if edge e is used in the derivation, $y(e) = 0$ otherwise). Thus y is a vector in $\{0, 1\}^{\mathcal{I}}$. We use $\mathcal{Y}$ to refer to the set of valid derivations. The set $\mathcal{Y}$ is a subset of $\{0, 1\}^{\mathcal{I}}$ (not all members of $\{0, 1\}^{\mathcal{I}}$ will correspond to valid derivations). The set of valid derivations is also defined formally in the background chapter.

Each derivation y in the hypergraph will imply an ordered sequence of leaves $v_1 \ldots v_n$. We use $s(y)$ to refer to this sequence. The target *sentence* associated with the derivation is

[1] They might for example be non-terminal symbols from the grammar used to generate the hypergraph.

then $l(v_1) \ldots l(v_n)$.

In a weighted hypergraph problem, we assume a parameter vector $\theta = \{\theta(r) : r \in \mathcal{I}\}$. The score for any derivation is $f(y) = \theta^\top y = \sum_{r \in \mathcal{I}} \theta(r) y(r)$. Simple bottom-up dynamic programming—essentially the CKY algorithm—can be used to find $y^* = \arg\max_{y \in \mathcal{Y}} f(y)$ under these definitions.

The focus of this chapter will be to solve problems involving the integration of a p'th order language model with a hypergraph. In these problems, the score for a derivation is modified to be

$$f(y) = \sum_{r \in \mathcal{I}} \theta(r) y(r) + \sum_{i=p}^{n} \theta(v_{i-p+1}, v_{i-p+2}, \ldots, v_i) \tag{5.1}$$

where $v_1 \ldots v_n = s(y)$. The $\theta(v_{i-p+1}, \ldots, v_i)$ parameters score n-grams of length p. These parameters are typically defined by a language model, for example with $p = 3$ we would have $\theta(v_{i-2}, v_{i-1}, v_i) = \log p(l(v_i) | l(v_{i-2}), l(v_{i-1}))$. The problem is then to find $y^* = \arg\max_{y \in \mathcal{Y}} f(y)$ under this definition.

Throughout this chapter we make the following assumption when using a bigram language model:

Assumption 5.1.1. *(*Bigram start/end assumption.*) For any derivation y, with leaves $s(y) = v_1, v_2, \ldots, v_n$, it is the case that: (1) $v_1 = 2$ and $v_n = 3$; (2) the leaves 2 and 3 cannot appear at any other position in the strings $s(y)$ for $y \in \mathcal{Y}$; (3) $l(2) =$ <s> where <s> is the start symbol in the language model; (4) $l(3) =$ </s> where </s> is the end symbol.*

This assumption allows us to incorporate language model terms that depend on the start and end symbols. It also allows a clean solution for boundary conditions (the start/end of strings).[2]

Example 5.1.1 (Example of a Hypergraph). Figure 5-2 shows an example hypergraph $(\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = \{1 \ldots |\mathcal{V}|\}$. Figure 5-3 shows an example derivation from the hypergraph, the

[2]The assumption generalizes in the obvious way to k'th order language models: e.g., for trigram models we assume that $v_1 = 2$, $v_2 = 3$, $v_n = 4$, $l(2) = l(3) =$ <s>, $l(4) =$ </s>.

Hyperedge $e \in \mathcal{E}$	$\theta(e)$
$1 \rightarrow 2\ 4\ 3$	-1
$4 \rightarrow 5\ 6\ 7$	2
$4 \rightarrow 6\ 5$	0.5
$4 \rightarrow 6\ 5\ 7$	3
$5 \rightarrow 8\ 9$	-4
$5 \rightarrow 10\ 11$	2.5

$\mathcal{V}^{(t)} = \{2, 3, 6, 7, 8, 9, 10, 11\}$
$\mathcal{V}^{(n)} = \{1, 4, 5\}$

l(2) = <s>
l(3) = </s>
l(6) = barks
l(7) = loudly
l(8) = the
l(9) = dog
l(10) = a
l(11) = cat

Figure 5-2: An example hypergraph, with the set of vertices $\mathcal{V} = \{1, 2, \ldots 11\}$. *We show: (1) the set of hyperedges* $\mathcal{E}$, *with weights* θ ; *(2) the set of non-terminals, and the set of leaves,* $\mathcal{V}^{(t)}$; *(3) the labels for the leaves in the hypergraph. Note that for readability we use the format* $v_1 \rightarrow v_2 v_3 \ldots v_k$ *for the hyperedges, rather than* $\langle (v_2, v_3, \ldots, v_k), v_1 \rangle$. *For example,* $1 \rightarrow 2\ 4\ 3$ *is used to denote the hyperedge* $\langle (2, 4, 3), 1 \rangle$.

associated sequence of labels for this derivation, and the sentence corresponding to the sequence of labels. The full set of sentences derived by the hypergraph is as follows:

<s> the dog barks loudly </s>

<s> a cat barks loudly </s>

<s> barks the dog </s>

<s> barks a cat </s>

<s> barks the dog loudly </s>

<s> barks a cat loudly </s>

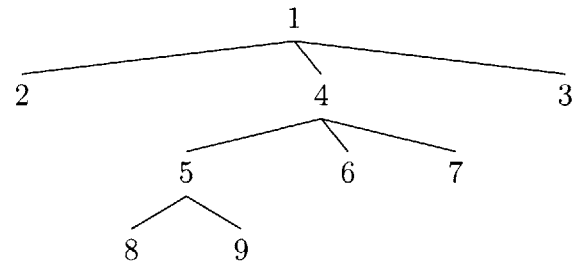
Note that the hypergraph satisfies Assumption 5.1.1 (the bigram start/end assumption). This is essentially because the only hyperedge that has the root vertex 1 as its head is

$$1 \rightarrow 2\ 4\ 3$$

which ensures that all derivations y have vertex 2 as the first leaf in $s(y)$, and vertex 3 as the last leaf in $s(y)$.

Example 5.1.2 (Synchronous Context-Free Grammars). We now give an example of how a hypergraph can be constructed using a synchronous context-free grammar (SCFG). Specif-

$y =$



$$l(y) = \langle 2, 8, 9, 6, 7, 3 \rangle$$
$$t = \texttt{<s> the dog barks loudly </s>}$$

Figure 5-3: An example of a valid derivation, y, from the hypergraph in Figure 5-2. Let $v_1, v_2 \ldots v_n$ be the sequence of leaves in the derivation, and $l(v_1) \ldots l(v_n)$ is the string defined by the derivation. The score for the derivation is the sum of weights for the hyperedges used, that is, $-1 + 2 - 4 = -3$. Under a bigram language model the derivation would have score $f(y) = -3 + \theta(l(2), l(8)) + \theta(l(8), l(9)) + \theta(l(9), l(6)) + \theta(l(6), l(7)) + \theta(l(7), l(3))$, where $\theta(u, v)$ is the language model score for the bigram u, v.

ically, we will give a sketch of the steps involved when a synchronous CFG is applied to a source-language string. Our description is informal; see Chiang (2005) for a formal description.

We will assume that the source language sentence is

<s> abarks le dug </s>

and that the synchronous CFG is as follows:

Rule $g \in \mathcal{G}$	$w(g)$
S → <s> X </s>, <s> X </s>,	-1
X → abarks X, X barks loudly	2
X → abarks X, barks X	0.5
X → abarks X, barks X loudly	3
X → le dug, the dog	-4
X → le dug, a cat	2.5

Each rule in the SCFG includes a non-terminal on the left hand side of the rule (e.g., X). The right hand side of each rule specifies a sequence of terminal and/or non-terminal symbols in the source language (e.g., `abarks` X), paired with a sequence of terminal and/or non-terminal symbols in the target language (e.g., X `barks loudly`).[3]

In the first step, we extract a context-free grammar from the synchronous CFG, by simply discarding the rule weights and the target side of the rules. Multiple instances of the same context-free rule are collapsed to a single rule instance. The resulting CFG is as follows:

$$\text{S} \rightarrow \texttt{<s>}\ \text{X}\ \texttt{</s>}$$
$$\text{X} \rightarrow \texttt{abarks}\ \text{X}$$
$$\text{X} \rightarrow \text{le}\ \texttt{dug}$$

The resulting grammar can be applied to the source-language string, the result being itself an (unweighted) hypergraph (essentially a context-free chart representing all parse trees for the sentence). In this example, the result is as follows:

$$\text{S}(1,5) \rightarrow \texttt{<s>}\ \text{X}(2,4)\ \texttt{</s>}$$
$$\text{X}(2,4) \rightarrow \texttt{abarks}\ \text{X}(3,4)$$
$$\text{X}(3,4) \rightarrow \text{le}\ \texttt{dug}$$

Note that we have allowed strings, such as S(1,5), or `abarks`, to be vertices in the hypergraph (as opposed to integers in the range $1 \ldots |\mathcal{V}|$). The strings consist of either a non-terminal in the CFG, with an associated span (e.g., X(2,4) represents non-terminal X spanning words 2 to 4 inclusive), or a terminal symbol (e.g., `dug`). The hypergraph created for this example is very simple, allowing only one derivation.

The final step is to apply the synchronous rules to transform this hypergraph to a new hypergraph—a "translation forest"—which encodes the set of possible strings in the target languge. As one example, if we apply the synchronous rule

[3]Note that the SCFG formalism of Chiang (Chiang, 2005) would normally require non-terminals on the right hand side of each rule to contain indices. For example, a well-formed rule would be S → <s> X_1 </s>, <s> X_1 </s>. The indices (a single index, 1, in this case) define a one-to-one mapping between non-terminals in the source and target sides of a rule. Our SCFG has at most one non-terminal in the source/target language side of the rule, hence these indices are unnecessary, and for brevity are omitted.

$$X \rightarrow \texttt{abarks}\ X,\ X\ \texttt{barks loudly}$$

which has weight 2, to the hyperedge

$$X(2,4) \rightarrow \texttt{abarks}\ X(3,4)$$

we end up with a new hyperedge

$$X(2,4) \rightarrow X(3,4)\ \texttt{barks}(2,4)\ \texttt{loudly}(2,4)$$

which has a weight equal to 2 (i.e., the weight for the new hyperedge is equal to the weight for the synchronous rule used to create the new hyperedge).

Note that terminal symbols such as `barks(2,4)` include a span as well as a word: the span (in this case `(2,4)`), is taken directly from the parent symbol. The inclusion of spans is important, because it allows different occurrences of the same target-language word to be differentiated.

As another example, if we apply the synchronous rule

$$X \rightarrow \texttt{abarks}\ X,\ \texttt{barks}\ X$$

which has weight 0.5, to the hyperedge

$$X(2,4) \rightarrow \texttt{abarks}\ X(3,4)$$

we end up with a new hyperedge

$$X(2,4) \rightarrow \texttt{barks}(2,4)\ X(3,4)$$

which has a weight equal to 0.5.

If we apply the full set of synchronous rules to the source-language hypergraph, we end up with the final hypergraph, as follows:

Hyperedge $e \in \mathcal{E}$	$\theta(e)$
S(1,5) → <s> X(2,4) </s>	-1
X(2,4) → X(3,4) barks(2,4) loudly(2,4)	2
X(2,4) → barks(2,4) X(3,4)	0.5
X(2,4) → barks(2,4) X(3,4) loudly(2,4)	3
X(3,4) → the(3,4) dog(3,4)	-4
X(3,4) → a(3,4) cat(3,4)	2.5

This is identical to the hypergraph in Figure 5-2, if we map S(1,5) to 1, <s> to 2, </s> to 3, X(2,4) to 4, and so on.

5.2 A Simple Lagrangian Relaxation Algorithm

We now give a Lagrangian relaxation algorithm for integration of a hypergraph with a bigram language model, in cases where the hypergraph satisfies the following simplifying assumption:

Assumption 5.2.1 (The strict ordering assumption). *For any two leaves v and w, it is either the case that: 1) for all derivations y such that v and w are both in the sequence $l(y)$, v precedes w; or 2) for all derivations y such that v and w are both in $l(y)$, w precedes v.*

Thus under this assumption, the relative ordering of any two leaves is fixed. This assumption is overly restrictive:[4] the next section describes an algorithm that does not require this assumption. However deriving the simple algorithm will be useful in developing intuition, and will lead directly to the algorithm for the unrestricted case.

5.2.1 A Sketch of the Algorithm

At a high level, the algorithm is as follows. We introduce Lagrange multipliers $u(v)$ for all $v \in \mathcal{V}^{(t)}$, with initial values set to zero. The algorithm then involves the following steps:

[4] It is easy to come up with examples that violate this assumption: for example a hypergraph with edges $\langle\langle 4, 5\rangle, 1\rangle$ and $\langle\langle 5, 4\rangle, 1\rangle$ violates the assumption. The hypergraphs found in translation frequently contain alternative orderings such as this.

1. For each leaf v, find the previous leaf w that maximizes the score $\theta(w, v) + u(w)$ (call this leaf $\alpha^*(v)$, and define $\alpha(v) = \theta(\alpha^*(v), v) + u(\alpha^*(v))$).

2. Find the highest scoring derivation using dynamic programming over the original (non-intersected) hypergraph, with leaf nodes having weights $\theta(v) + \alpha(v) - u(v)$.

3. If the output derivation from step 2 has the same set of bigrams as those from step 1, then we have an exact solution to the problem. Otherwise, the Lagrange multipliers $u(v)$ are modified in a way that encourages agreement of the two steps, and we return to step 1.

Steps 1 and 2 can be performed efficiently; in particular, we avoid the classical dynamic programming intersection, instead relying on dynamic programming over the original, simple hypergraph.

5.2.2 A Formal Description

We now give a formal description of the algorithm. Define $\mathcal{B} \subseteq \mathcal{V}^{(t)} \times \mathcal{V}^{(t)}$ to be the set of all ordered pairs $\langle v, w \rangle$ such that there is at least one derivation y with v directly preceding w in $s(y)$. Extend the bit-vector y to include variables $y(v, w)$ for $\langle v, w \rangle \in \mathcal{B}$ where $y(v, w) = 1$ if leaf v is followed by w in $s(y)$, 0 otherwise. We redefine the index set to be $\mathcal{I} = \mathcal{V} \cup \mathcal{E} \cup \mathcal{B}$, and define $\mathcal{Y} \subseteq \{0, 1\}^{\mathcal{I}}$ to be the set of all possible derivations. Under Assumptions 5.1.1 and 5.2.1 above, $\mathcal{Y} = \{y : y \text{ satisfies constraints } \mathbf{C0}, \mathbf{C1}, \mathbf{C2}\}$ where the constraint definitions are:

- **(C0)** The $y(v)$ and $y(e)$ variables form a derivation in the hypergraph.

- **(C1)** For all $v \in \mathcal{V}^{(t)}$ such that $v \neq 2$,

$$y(v) = \sum_{w : \langle w, v \rangle \in \mathcal{B}} y(w, v)$$

- **(C2)** For all $v \in \mathcal{V}^{(t)}$ such that $v \neq 3$,

$$y(v) = \sum_{w:\langle v,w\rangle \in \mathcal{B}} y(v,w)$$

C1 states that each leaf in a derivation has exactly one incoming bigram, and that each leaf not in the derivation has 0 incoming bigrams; **C2** states that each leaf in a derivation has exactly one outgoing bigram, and that each leaf not in the derivation has 0 outgoing bigrams.[5]

The score of a derivation is now $f(y) = \theta^\top y$, i.e.,

$$f(y) = \sum_v \theta(v)y(v) + \sum_e \theta(e)y(e) + \sum_{\langle v,w\rangle \in \mathcal{B}} \theta(v,w)y(v,w)$$

where $\theta(v,w)$ are scores from the language model. Our goal is to compute $y^* = \arg\max_{y\in\mathcal{Y}} f(y)$.

Next, define $\mathcal{Y}'$ as

$$\mathcal{Y}' = \{y : y \text{ satisfies constraints } \mathbf{C0} \text{ and } \mathbf{C1}\}$$

In this definition we have dropped the **C2** constraints. To incorporate these constraints, we use Lagrangian relaxation, with one Lagrange multiplier $u(v)$ for each constraint in **C2**. The Lagrangian is

$$L(u,y) = f(y) + \sum_v u(v)\left(y(v) - \sum_{w:\langle v,w\rangle \in \mathcal{B}} y(v,w)\right) \quad = \quad \beta^\top y$$

where $\beta(v) = \theta(v) + u(v)$, $\beta(e) = \theta(e)$, and $\beta(v,w) = \theta(v,w) - u(v)$.

The dual problem is to find $\min_u L(u)$ where

$$L(u) = \max_{y\in\mathcal{Y}'} L(u,y)$$

[5]Recall that according to the bigram start/end assumption the leaves 2/3 are reserved for the start/end of the sequence $s(y)$, and hence do not have an incoming/outgoing bigram.

Initialization: Set $u^{(1)}(v) = 0$ for all $v \in \mathcal{V}^{(t)}$
Algorithm: For $k = 1 \ldots K$:

- $y^{(k)} = \arg\max_{y \in \mathcal{Y}'} L(u^{(k)}, y)$
- **If** $y^{(k)}$ satisfies constraints **C2**, **return** $y^{(k)}$,
  Else

$$\forall v \in \mathcal{V}^{(t)}, u^{(k+1)}(v) \leftarrow u^{(k)}(v) - \alpha_k \left(y^{(k)}(v) - \sum_{w:\langle v,w \rangle \in \mathcal{B}} y^{(k)}(v,w) \right)$$

Figure 5-4: A simple Lagrangian relaxation algorithm. $\alpha_k > 0$ is the step size at iteration k.

Figure 5-4 shows a *subgradient* method for solving this problem. At each point the algorithm finds $y^{(k)} = \arg\max_{y \in \mathcal{Y}'} L(u^{(k)}, y)$, where $u^{(k)}$ are the Lagrange multipliers from the previous iteration. If $y^{(k)}$ satisfies the **C2** constraints in addition to **C0** and **C1**, then it is returned as the output from the algorithm. Otherwise, the multipliers $u(v)$ are updated. Intuitively, these updates encourage the values of $y(v)$ and $\sum_{w:\langle v,w \rangle \in \mathcal{B}} y(v,w)$ to be equal; formally, these updates correspond to subgradient steps.

The main computational step at each iteration is to compute $\arg\max_{y \in \mathcal{Y}'} L(u^{(k)}, y)$ This step is easily solved, as follows (we again use $\beta(v)$, $\beta(e)$ and $\beta(v_1, v_2)$ to refer to the parameter values that incorporate Lagrange multipliers):

- For all $v \in \mathcal{V}^{(t)}$, define

$$\alpha^*(v) = \arg\max_{w:\langle w,v \rangle \in \mathcal{B}} \beta(w,v)$$

  and $\alpha(v) = \beta(\alpha^*(v), v)$. For all $v \in \mathcal{V}^{(n)}$ define $\alpha(v) = 0$.

- Using dynamic programming, find values for the $y(v)$ and $y(e)$ variables that form a valid derivation, and that maximize

$$f'(y) = \sum_v (\beta(v) + \alpha(v)) y(v) + \sum_e \beta(e) y(e)$$

- Set $y(v, w) = 1$ iff $y(w) = 1$ and $\alpha^*(w) = v$.

The critical point here is that through our definition of $\mathcal{Y}'$, which ignores the **C2** constraints, we are able to do efficient search as just described. In the first step we compute the highest scoring incoming bigram for each leaf v. In the second step we use conventional dynamic programming over the hypergraph to find an optimal derivation that incorporates weights from the first step. Finally, we fill in the $y(v, w)$ values. Each iteration of the algorithm runs in $O(|\mathcal{E}| + |\mathcal{B}|)$ time.

As we have discussed in Chapter 3, there are important formal results for this algorithm:

1. For any value of u, $L(u) \geq f(y^*)$ (hence the dual value provides an upper bound on the optimal primal value).

2. Under an appropriate choice of the step sizes α_k, the subgradient algorithm is guaranteed to converge to the minimum of $L(u)$ (i.e., we will minimize the upper bound, making it as tight as possible).

3. If at any point the algorithm in Figure 5-4 finds a $y^{(k)}$ that satisfies the **C2** constraints, then this is guaranteed to be the optimal primal solution.

Unfortunately, this algorithm may fail to produce a good solution for hypergraphs where the strict ordering constraint does not hold. In this case it is possible to find derivations y that satisfy constraints **C0**, **C1**, **C2**, but which are invalid. As one example, consider a derivation with $s(y) = 2, 4, 5, 3$ and $y(2, 3) = y(4, 5) = y(5, 4) = 1$. The constraints are all satisfied in this case, but the bigram variables are invalid (e.g., they contain a cycle).

Example 5.2.1 (A Run of the Simple Algorithm). We give an extended example run of the simple algorithm at the end of this chapter in Section 5.8.

5.3 The Full Algorithm

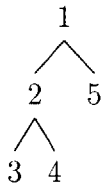
We now describe our full algorithm, which does not require the strict ordering constraint. In addition, the full algorithm allows a trigram language model. We first give a sketch, and

then give a formal definition.

5.3.1 A Sketch of the Algorithm

A crucial idea in the new algorithm is that of *paths* between leaves in hypergraph derivations. Previously, for each derivation y, we had defined $s(y) = v_1, v_2, \ldots, v_n$ to be the sequence of leaves in y. In addition, we will define $g(y) = p_0, v_1, p_1, v_2, p_2, v_3, p_3, \ldots, p_{n-1}, v_n, p_n$ where each p_i is a path in the derivation between leaves v_i and v_{i+1}. The path traces through the non-terminals that are between the two leaves in the tree.

As an example, consider the following derivation (with hyperedges $\langle\langle 2, 5\rangle, 1\rangle$ and $\langle\langle 3, 4\rangle, 2\rangle$):



For this example $g(y)$ is $\langle 1\downarrow, 2\downarrow\rangle$ $\langle 2\downarrow, 3\downarrow\rangle$ $\langle 3\downarrow\rangle$, 3, $\langle 3\uparrow\rangle$ $\langle 3\uparrow, 4\downarrow\rangle$ $\langle 4\downarrow\rangle$, 4, $\langle 4\uparrow\rangle$ $\langle 4\uparrow, 2\uparrow\rangle$ $\langle 2\uparrow, 5\downarrow\rangle$ $\langle 5\downarrow\rangle$, 5, $\langle 5\uparrow\rangle$ $\langle 5\uparrow, 1\uparrow\rangle$. States of the form $\langle a\downarrow\rangle$ and $\langle a\uparrow\rangle$ where a is a leaf appear in the paths respectively before/after the leaf a. States of the form $\langle a, b\rangle$ correspond to the steps taken in a top-down, left-to-right, traversal of the tree, where down and up arrows indicate whether a node is being visited for the first or second time (the traversal in this case would be $1, 2, 3, 4, 2, 5, 1$).

The mapping from a derivation y to a path $g(y)$ can be performed using the algorithm in figure 5-5. For a given derivation y, define $\mathcal{E}(y) = \{y : y(e) = 1\}$, and use $\mathcal{E}(y)$ as the set of input edges to this algorithm. The output from the algorithm will be a set of states $\mathcal{S}$, and a set of directed edges $\mathcal{T}$, which together fully define the path $g(y)$.

In the simple algorithm, the first step was to predict the previous leaf for each leaf v, under a score that combined a language model score with a Lagrange multiplier score (i.e., compute $\arg\max_w \beta(w, v)$ where $\beta(w, v) = \theta(w, v) + u(w)$). In this section we describe an algorithm that for each leaf v again predicts the previous leaf, but in addition predicts the full *path* back to that leaf. For example, rather than making a prediction for leaf 5 that it

- **Input:** A set $\mathcal{E}$ of hyperedges.
- **Output:** A directed graph $(\mathcal{S}, \mathcal{T})$ where $\mathcal{S}$ is a set of vertices, and $\mathcal{T}$ is a set of edges.
- **Step 1: Creating $\mathcal{S}$:** Define $\mathcal{S} = \cup_{e \in \mathcal{E}} \mathcal{S}_e$ where $\mathcal{S}_e$ is defined as follows. Assume $e = \langle \langle v_2, \ldots, v_k \rangle, v_1 \rangle$. Include the following states in $\mathcal{S}_e$:
  1. $\langle v_1 \downarrow, v_1 \downarrow \rangle$ and $\langle v_k \uparrow, v_1 \uparrow \rangle$.
  2. $\langle v_j \uparrow, v_{j+1} \downarrow \rangle$ for $j = 1 \ldots k-1$ (if $k = 1$ then there are no such states).
  3. In addition, for any v_j for $j = 1 \ldots k$ such that $v_j \in \mathcal{V}^{(t)}$, add the states $\langle v_j \downarrow \rangle$ and $\langle v_j \uparrow \rangle$.
- **Step 2: Creating $\mathcal{T}$:** $\mathcal{T}$ is formed by including the following directed arcs:
  1. Add an arc from $\langle a, b \rangle \in \mathcal{S}$ to $\langle c, d \rangle \in \mathcal{S}$ whenever $b = c$.
  2. Add an arc from $\langle a, b \downarrow \rangle \in \mathcal{S}$ to $\langle c \downarrow \rangle \in \mathcal{S}$ whenever $b = c$.
  3. Add an arc from $\langle a \uparrow \rangle \in \mathcal{S}$ to $\langle b \uparrow, c \rangle \in \mathcal{S}$ whenever $a = b$.

Figure 5-5: Algorithm for constructing a directed graph $(\mathcal{S}, \mathcal{T})$ from a set of hyperedges $\mathcal{E}$.

should be preceded by leaf 4, we would also predict the path $\langle 4 \uparrow \rangle \langle 4 \uparrow, 2 \uparrow \rangle \ \langle 2 \uparrow, 5 \downarrow \rangle \langle 5 \downarrow \rangle$ between these two leaves. Lagrange multipliers will be used to enforce consistency between these predictions (both paths and previous words) and a valid derivation.

5.3.2 A Formal Description

We first use the algorithm in figure 5-5 with the entire set of hyperedges, $\mathcal{E}$, as its input. The result is a directed graph $(\mathcal{S}, \mathcal{T})$ that contains *all possible paths* for valid derivations in $(\mathcal{V}, \mathcal{E})$ (it also contains additional, ill-formed paths). We then introduce the following definition:

Definition 5.3.1. A trigram path p *is* $p = \langle v_1, p_1, v_2, p_2, v_3 \rangle$ *where: a)* $v_1, v_2, v_3 \in \mathcal{V}^{(t)}$*; b)* p_1 *is a path (sequence of states) between nodes* $\langle v_1 \uparrow \rangle$ *and* $\langle v_2 \downarrow \rangle$ *in the graph* $(\mathcal{S}, \mathcal{T})$*; c)* p_2 *is a path between nodes* $\langle v_2 \uparrow \rangle$ *and* $\langle v_3 \downarrow \rangle$ *in the graph* $(\mathcal{S}, \mathcal{T})$*. We define* $\mathcal{P}$ *to be the set of all trigram paths in* $(\mathcal{S}, \mathcal{T})$*.*

The set $\mathcal{P}$ of trigram paths plays an analogous role to the set $\mathcal{B}$ of bigrams in our previous

algorithm.

We use $v_1(p), p_1(p), v_2(p), p_2(p), v_3(p)$ to refer to the individual components of a path p. In addition, define $\mathcal{S}_N$ to be the set of states in $\mathcal{S}$ of the form $\langle a, b \rangle$ (as opposed to the form $\langle c \downarrow \rangle$ or $\langle c \uparrow \rangle$ where $c \in \mathcal{V}^{(t)}$).

We now define a new index set, $\mathcal{I} = \mathcal{V} \cup \mathcal{E} \cup \mathcal{S}_N \cup \mathcal{P}$, adding variables $y(s)$ for $s \in \mathcal{S}_N$, and $y(p)$ for $p \in \mathcal{P}$. If we take $\mathcal{Y} \subset \{0,1\}^{\mathcal{I}}$ to be the set of valid derivations, the optimization problem is to find $y^* = \arg\max_{y \in \mathcal{Y}} f(y)$, where $f(y) = \theta^\top y$, that is,

$$f(y) = \sum_v \theta(v) y(v) + \sum_e \theta(e) y(e) + \sum_s \theta(s) y(s) + \sum_p \theta(p) y(p)$$

In particular, we might define $\theta(s) = 0$ for all s, and $\theta(p) = \log p(l(v_3(p)) | l(v_1(p)), l(v_2(p)))$ where $p(w_3 | w_1, w_2)$ is a trigram probability.

The set $\mathcal{P}$ is large (typically exponential in size): however, we will see that we do not need to represent the $y(p)$ variables explicitly. Instead we will be able to leverage the underlying structure of a path as a sequence of states.

The set of valid derivations is $\mathcal{Y} = \{y : y \text{ satisfies constraints } \mathbf{D0}\text{–}\mathbf{D6}\}$ where the constraints are shown in Figure 5-6. **D1** simply states that $y(s) = 1$ iff there is exactly one edge e in the derivation such that $s \in \mathcal{S}_e$. Constraints **D2**–**D4** enforce consistency between leaves in the trigram paths, and the $y(v)$ values. Constraints **D5** and **D6** enforce consistency between states seen in the paths, and the $y(s)$ values.

The Lagrangian relaxation algorithm is then derived in a similar way to before. Define

$$\mathcal{Y}' = \{y : y \text{ satisfies constraints } \mathbf{D0}\text{–}\mathbf{D2}\}$$

We have dropped the **D3**–**D6** constraints, but these will be introduced using Lagrange multipliers. The resulting Lagrangian is shown in figure 5-6, and can be written as $L(y, \lambda, \gamma, u, v) = \beta^\top y$ where

$$\beta(v) = \theta(v) + \lambda(v) + \gamma(v)$$

- **D0.** The $y(v)$ and $y(e)$ variables form a valid derivation in the original hypergraph.
- **D1.** For all $s \in \mathcal{S}_N$, $y(s) = \sum_{e:s\in\mathcal{S}_e} y(e)$ (see Figure 5-5 for the definition of $\mathcal{S}_e$).
- **D2.** For all $v \in \mathcal{V}^{(t)}$, $y(v) = \sum_{p:v_3(p)=v} y(p)$
- **D3.** For all $v \in \mathcal{V}^{(t)}$, $y(v) = \sum_{p:v_2(p)=v} y(p)$
- **D4.** For all $v \in \mathcal{V}^{(t)}$, $y(v) = \sum_{p:v_1(p)=v} y(p)$
- **D5.** For all $s \in \mathcal{S}_N$, $y(s) = \sum_{p:s\in p_1(p)} y(p)$
- **D6.** For all $s \in \mathcal{S}_N$, $y(s) = \sum_{p:s\in p_2(p)} y(p)$
- Lagrangian with Lagrange multipliers for **D3**–**D6**:

$$\begin{aligned} L(y,\lambda,\gamma,u,v) = \theta^\top y \;&+ \sum_v \lambda(v)(y(v) - \sum_{p:v_2(p)=v} y(p)) \\ &+ \sum_v \gamma(v)(y(v) - \sum_{p:v_1(p)=v} y(p)) \\ &+ \sum_s u(s)(y(s) - \sum_{p:s\in p_1(p)} y(p)) \\ &+ \sum_s v(s)(y(s) - \sum_{p:s\in p_2(p)} y(p)) \end{aligned}$$

Figure 5-6: Constraints **D0**–**D6**, *and the Lagrangian.*

Initialization: Set $\lambda^{(1)} = 0, \gamma^{(1)} = 0, u^{(1)} = 0, v^{(1)} = 0$

Algorithm: For $k = 1 \ldots K$:

- $y^{(k)} = \arg\max_{y \in \mathcal{Y}'} L(y, \lambda^{(k)}, \gamma^{(k)}, u^{(k)}, v^{(k)})$
- If $y^{(k)}$ satisfies the constraints **D3**–**D6**, return $y^{(k)}$, else:
  - $\forall v \in \mathcal{V}^{(t)}$, $\lambda^{(k+1)}(v) \leftarrow \lambda^{(k)}(v) - \alpha_k(y^{(k)}(v) - \sum_{p:v_2(p)=v} y^{(k)}(p))$
  - $\forall v \in \mathcal{V}^{(t)}$, $\gamma^{(k+1)}(v) \leftarrow \gamma^{(k)}(v) - \alpha_k(y^{(k)}(v) - \sum_{p:v_1(p)=v} y^{(k)}(p))$
  - $\forall s \in \mathcal{S}_N$, $u^{(k+1)}(s) \leftarrow u^{(k)}(s) - \alpha_k(y^{(k)}(s) - \sum_{p:s \in p_1(p)} y^{(k)}(p))$
  - $\forall s \in \mathcal{S}_N$, $v^{(k+1)}(s) \leftarrow v^{(k)}(s) - \alpha_k(y^{(k)}(s) - \sum_{p:s \in p_2(p)} y^{(k)}(p))$

Figure 5-7: The full Lagrangian relaxation algorithm. $\alpha_k > 0$ is the step size at iteration k.

$$\beta(s) = \theta(s) + u(s) + v(s)$$

$$\beta(p) = \theta(p) - \lambda(v_2(p)) - \gamma(v_1(p)) - \sum_{s \in p_1(p)} u(s) - \sum_{s \in p_2(p)} v(s)$$

.

The dual is $L(\lambda, \gamma, u, v) = \max_{y \in \mathcal{Y}'} L(y, \lambda, \gamma, u, v)$; figure 5-7 shows a subgradient method that minimizes this dual. The key step in the algorithm at each iteration is to compute $\arg\max_{y \in \mathcal{Y}'} L(y, \lambda, \gamma, u, v) = \arg\max_{y \in \mathcal{Y}'} \beta^\top y$ where β is defined above. Again, our definition of $\mathcal{Y}'$ allows this maximization to be performed efficiently, as follows:

1. For each $v \in \mathcal{V}^{(t)}$, define $\alpha^*(v) = \arg\max_{p:v_3(p)=v} \beta(p)$, and $\alpha(v) = \beta(\alpha^*(v))$. (i.e., for each v, compute the highest scoring trigram path ending in v.)

2. Find values for the $y(v)$, $y(e)$ and $y(s)$ variables that form a valid derivation, and that maximize

$$f'(y) = \sum_v (\beta(v) + \alpha(v))y(v) + \sum_e \beta(e)y(e) + \sum_s \beta_s y(s)$$

3. Set $y(p) = 1$ iff $y(v_3(p)) = 1$ and $p = \alpha^*(v_3(p))$.

The first step involves finding the highest scoring incoming trigram path for each leaf v. This step can be performed efficiently using the Floyd-Warshall all-pairs shortest path algorithm (Floyd, 1962) over the graph $(\mathcal{S}, \mathcal{T})$; the details are given in the appendix. The second step involves simple dynamic programming over the hypergraph $(\mathcal{V}, \mathcal{E})$ (it is simple to integrate the $\beta(s)$ terms into this algorithm). In the third step, the path variables $y(p)$ are filled in.

5.3.3 Properties

We now describe some important properties of the algorithm:

Efficiency. The main steps of the algorithm are:

1. construction of the graph $(\mathcal{S}, \mathcal{T})$
2. at each iteration, dynamic programming over the hypergraph $(\mathcal{V}, \mathcal{E})$
3. at each iteration, all-pairs shortest path algorithms over the graph $(\mathcal{S}, \mathcal{T})$.

Each of these steps is vastly more efficient than computing an exact intersection of the hypergraph with a language model.

Exact solutions. By usual guarantees for Lagrangian relaxation, if at any point the algorithm returns a solution $y^{(k)}$ that satisfies constraints **D3**–**D6**, then $y^{(k)}$ exactly solves the problem in Eq. 5.1.

Upper bounds. At each point in the algorithm, $L(\lambda^{(k)}, \gamma^{(k)}, u^{(k)}, v^{(k)})$ is an upper bound on the score of the optimal primal solution, $f(y^*)$. Upper bounds can be useful in evaluating the quality of primal solutions from either our algorithm or other methods such as cube pruning.

Simplicity of implementation. Construction of the $(\mathcal{S}, \mathcal{T})$ graph is straightforward. The other steps—hypergraph dynamic programming, and all-pairs shortest path– are widely known algorithms that are simple to implement.

5.4 Tightening the Relaxation

The algorithm that we have described minimizes the dual function $L(\lambda, \gamma, u, v)$. By usual results for Lagrangian relaxation (e.g., see (Korte & Vygen, 2008)), L is the dual function for a particular LP relaxation arising from the definition of $\mathcal{Y}'$ and the additional constaints **D3**–**D6**. In some cases the LP relaxation has an integral solution, in which case the algorithm will return an optimal solution $y^{(k)}$.[6] In other cases, when the LP relaxation has a fractional solution, the subgradient algorithm will still converge to the minimum of L, but the primal solutions $y^{(k)}$ will move between a number of solutions.

We now describe a method that incrementally adds hard constraints to the set $\mathcal{Y}'$, until the method returns an exact solution. For a given $y \in \mathcal{Y}'$, for any v with $y(v) = 1$, we can recover the previous two leaves (the trigram ending in v) from either the path variables $y(p)$, or the hypergraph variables $y(e)$. Specifically, define $v_{-1}(v, y)$ to be the leaf preceding v in the trigram path p with $y(p) = 1$ and $v_3(p) = v$, and $v_{-2}(v, y)$ to be the leaf two positions before v in the trigram path p with $y(p) = 1$ and $v_3(p) = v$. Similarly, define $v'_{-1}(v, y)$ and $v'_{-2}(v, y)$ to be the preceding two leaves under the $y(e)$ variables. If the method has not converged, these two trigram definitions may not be consistent. For a consistent solution, we require $v_{-1}(v, y) = v'_{-1}(v, y)$ and $v_{-2}(v, y) = v'_{-2}(v, y)$ for all v with $y(v) = 1$. Unfortunately, explicitly enforcing all of these constraints would require exhaustive dynamic programming over the hypergraph using the (Bar-Hillel et al., 1964) method, something we wish to avoid.

Instead, we enforce a weaker set of constraints, which require far less computation. Assume some function $\pi : \mathcal{V}^{(t)} \rightarrow \{1, 2, \ldots q\}$ that partitions the set of leaves into q different partitions. Then we will add the following constraints to $\mathcal{Y}'$:

$$\begin{aligned} \pi(v_{-1}(v, y)) &= \pi(v'_{-1}(v, y)) \\ \pi(v_{-2}(v, y)) &= \pi(v'_{-2}(v, y)) \end{aligned}$$

[6] Provided that the algorithm is run for enough iterations for convergence.

for all v such that $y(v) = 1$. Finding $\arg\max_{y \in \mathcal{Y}'} \theta^\top y$ under this new definition of $\mathcal{Y}'$ can be performed using the construction of (Bar-Hillel et al., 1964), with q different lexical items (for brevity we omit the details). This is efficient if q is small.[7]

The remaining question concerns how to choose a partition π that is effective in tightening the relaxation. To do this we implement the following steps:

1. Run the subgradient algorithm until L is close to convergence.
2. Run the subgradient algorithm for m further iterations, keeping track of all pairs of leaf nodes that violate the constraints (i.e., pairs $a = v_{-1}(v, y)/b = v'_{-1}(v, y)$ or $a = v_{-2}(v, y)/b = v'_{-2}(v, y)$ such that $a \neq b$).
3. Use a graph coloring algorithm to find a small partition that places all pairs $\langle a, b \rangle$ into separate partitions.
4. Continue running Lagrangian relaxation, with the new constraints added. We expand π at each iteration to take into account new pairs $\langle a, b \rangle$ that violate the constraints.

In related work, Sontag et al. (2008) describe a method for decoding in Markov random fields where additional constraints are chosen to tighten an underlying relaxation. Other relevant work in NLP includes Tromble and Eisner (2006), Riedel and Clarke (2006d). Our use of partitions π is related to previous work on coarse-to-fine decoding for machine translation (Petrov et al., 2008).

5.5 Related Work

A variety of approximate decoding algorithms have been explored for syntax-based translation systems, including cube-pruning (Chiang, 2007; Huang & Chiang, 2007), left-to-right decoding with beam search (Watanabe et al., 2006; Huang & Mi, 2010), and coarse-to-fine methods (Petrov et al., 2008).

[7] In fact in our experiments we use the original hypergraph to compute admissible outside scores for an exact A* search algorithm for this problem. We have found the resulting search algorithm to be very efficient.

Recent work has developed decoding algorithms based on finite state transducers (FSTs). Iglesias et al. (Iglesias et al., 2009) show that exact FST decoding is feasible for a phrase-based system with limited reordering (the MJ1 model (Kumar & Byrne, 2005)), and de Gispert et al. (de Gispert et al., 2010) show that exact FST decoding is feasible for a specific class of hierarchical grammars (shallow-1 grammars). Approximate search methods are used for more complex reordering models or grammars. The FST algorithms are shown to produce higher scoring solutions than cube-pruning on a large proportion of examples.

5.6 Experiments

We report experiments on translation from Chinese to English, using the tree-to-string model described in (Huang & Mi, 2010). We use an identical model, and identical development and test data, to that used by Huang and Mi.[8] The translation model is trained on 1.5M sentence pairs of Chinese-English data; a trigram language model is used. The development data is the newswire portion of the 2006 NIST MT evaluation test set (616 sentences). The test set is the newswire portion of the 2008 NIST MT evaluation test set (691 sentences).

We ran the full algorithm with the tightening method described in Section 5.4. We ran the method for a limit of 200 iterations, hence some examples may not terminate with an exact solution. Our method gives exact solutions on 598/616 development set sentences (97.1%), and 675/691 test set sentences (97.7%).

In cases where the method does not converge within 200 iterations, we can return the best primal solution $y^{(k)}$ found by the algorithm during those iterations. We can also get an upper bound on the difference $f(y^*) - f(y^{(k)})$ using $\min_k L(u^{(k)})$ as an upper bound on $f(y^*)$. Of the examples that did not converge, the worst example had a bound that was 1.4% of $f(y^{(k)})$ (more specifically, $f(y^{(k)})$ was -24.74, and the upper bound on $f(y^*) - f(y^{(k)})$ was 0.34).

Figure 5-8 gives information on decoding time for our method and two other exact decoding methods: integer linear programming (using constraints **D0**–**D6**), and exhaustive

[8] We thank Liang Huang and Haitao Mi for providing us with their model and data.

Time	%age (LR)	%age (DP)	%age (ILP)	%age (LP)
0.5s	37.5	10.2	8.8	21.0
1.0s	57.0	11.6	13.9	31.1
2.0s	72.2	15.1	21.1	45.9
4.0s	82.5	20.7	30.7	63.7
8.0s	88.9	25.2	41.8	78.3
16.0s	94.4	33.3	54.6	88.9
32.0s	97.8	42.8	68.5	95.2
Median time	0.79s	77.5s	12.1s	2.4s

Figure 5-8: Results showing percentage of examples that are decoded in less than t seconds, for $t = 0.5, 1.0, 2.0, \ldots, 32.0$. LR = Lagrangian relaxation; DP = exhaustive dynamic programming; ILP = integer linear programming; LP = linear programming (LP does not recover an exact solution). The (I)LP experiments were carried out using Gurobi, a high-performance commercial-grade solver.

dynamic programming using the construction of (Bar-Hillel et al., 1964). Our method is clearly the most efficient, and is comparable in speed to state-of-the-art decoding algorithms.

We also compare our method to cube pruning (Chiang, 2007; Huang & Chiang, 2007). We reimplemented cube pruning in C++, to give a fair comparison to our method. Cube pruning has a parameter, b, dictating the maximum number of items stored at each chart entry. With $b = 50$, our decoder finds higher scoring solutions on 50.5% of all examples (349 examples), the cube-pruning method gets a strictly higher score on only 1 example (this was one of the examples that did not converge within 200 iterations). With $b = 500$, our decoder finds better solutions on 18.5% of the examples (128 cases), cube-pruning finds a better solution on 3 examples. The median decoding time for our method is 0.79 seconds; the median times for cube pruning with $b = 50$ and $b = 500$ are 0.06 and 1.2 seconds respectively.

Our results give a very good estimate of the percentage of search errors for cube pruning. A natural question is how large b must be before exact solutions are returned on almost all examples. Even at $b = 1000$, we find that our method gives a better solution on 95 test examples (13.7%).

Figure 5-8 also gives a speed comparison of our method to a linear programming (LP)

solver that solves the LP relaxation defined by constraints **D0**–**D6**. We still see speed-ups, in spite of the fact that our method is solving a harder problem (it provides integral solutions). The Lagrangian relaxation method, when run without the tightening method of Section 5.4, is solving a dual of the problem being solved by the LP solver. Hence we can measure how often the tightening procedure is absolutely necessary, by seeing how often the LP solver provides a fractional solution. We find that this is the case on 54.0% of the test examples: the tightening procedure is clearly important. Inspection of the tightening procedure shows that the number of partitions required (the parameter q) is generally quite small: 59% of examples that require tightening require $q \leq 6$; 97.2% require $q \leq 10$.

5.7 Conclusion

We have described a Lagrangian relaxation algorithm for exact decoding of syntactic translation models, and shown that it is significantly more efficient than other exact algorithms for decoding tree-to-string models. There are a number of possible ways to extend this work. Our experiments have focused on tree-to-string models, but the method should also apply to Hiero-style syntactic translation models (Chiang, 2007). Additionally, our experiments used a trigram language model, however the constraints in Figure 5-6 generalize to higher-order language models. Finally, our algorithm recovers the 1-best translation for a given input sentence; it should be possible to extend the method to find k-best solutions.

The method described here also shows several further extensions on standard Lagrangian relaxation. We showed how to structure the relaxation in order to derive subproblems that can be solved efficiently with combinatorial methods, e.g. the CKY algorithm and all-pairs shortest path. We also gave a problem-specific method for tightening by generating additional constraints based on observed violations. We will also explore different method for solving challenging syntactic translation problems in Chapter 7.

5.8 Appendix: A Run of the Simple Algorithm

Figure 5-2 shows a hypergraph that satisfies the strict ordering assumption. We will now illustrate how the Lagrangian relaxation algorithm operates on this hypergraph.

First, it is simple to verify that the set $\mathcal{B}$ for this hypergraph is

$$\mathcal{B} = \{\langle 2,8\rangle, \langle 2,10\rangle, \langle 8,9\rangle, \langle 10,11\rangle, \langle 9,6\rangle, \langle 11,6\rangle, \langle 6,7\rangle, \langle 7,3\rangle\}$$

(recall that $\mathcal{B}$ is defined to be the set of all pairs $\langle v, w\rangle$ such that leaf v directly precedes leaf w in at least one derivation under the hypergraph).

For readability in this section, we will often replace the leaf indices $2, 3, 6, 7, 8, 9, 10$ with their labels (words) `<s>`, `</s>`, `barks`, `loudly`, `the`, `dog`, `a`, and `cat` respectively. We can do this safely because for this particular hypergraph the function l from leaf indices to words is one-to-one (i.e., $l(v) \neq l(w)$ if $v \neq w$). It is important to remember however that in the general case different leaves which have the same label (word) are treated separately by the algorithm. Under this convention, we can also write the set $\mathcal{B}$ as

$$\begin{aligned}\mathcal{B} = \{&(\texttt{<s>}, \texttt{the}), (\texttt{<s>}, \texttt{a}), (\texttt{the}, \texttt{dog}), (\texttt{a}, \texttt{cat}), (\texttt{dog}, \texttt{barks}),\\ &(\texttt{cat}, \texttt{barks}), (\texttt{barks}, \texttt{loudly}), (\texttt{loudly}, \texttt{</s>})\}\end{aligned}$$

Recall that $\mathcal{Y}'$ consists of structures that satisfy the **C0** constraints (i.e., that the $y(v)$ and $y(e)$ variables form a valid derivation) together with the **C1** constraints. We will find it convenient to depict members of $\mathcal{Y}'$ as trees. As an example, consider a structure $y \in \mathcal{Y}'$ such that

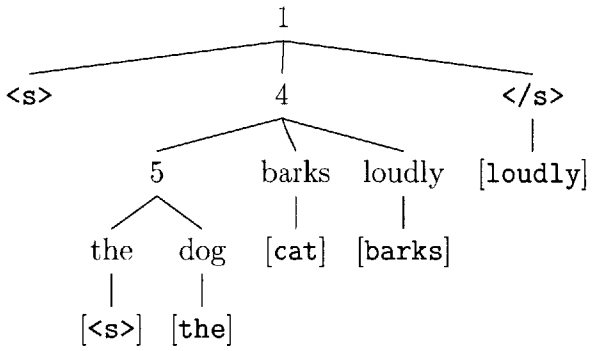
$$y(\langle 2,4,3\rangle, 1) = y(\langle 5,6,7\rangle, 4) = y(\langle 8,9\rangle, 5) = 1$$

$$y(\texttt{<s>}) = y(4) = y(\texttt{</s>}) = y(1) = y(5) = y(\texttt{barks}) = y(\texttt{loudly}) = y(\texttt{the}) = y(\texttt{dog}) = 1$$

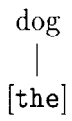
$$y(\texttt{<s>}, \texttt{the}) = y(\texttt{the}, \texttt{dog}) = y(\texttt{cat}, \texttt{barks}) = y(\texttt{barks}, \texttt{loudly}) = y(\texttt{loudly}, \texttt{</s>}) = 1$$

with all other variables equal to 0. It is easy to verify that this structure satisfies the **C0**

and **C1** constraints. The tree for this structure is



This tree shows the derivation implied by the values for the $y(v)$ and $y(e)$ variables. In addition, it includes productions of the form $w \rightarrow [v]$ for any $(v, w) \in \mathcal{B}$ such that $y(v, w) = 1$. For example, the fact that $y(\texttt{the}, \texttt{dog}) = 1$ leads to the production



This production represents the fact that the leaf `dog` has effectively chosen the leaf `the` as its previous word, under the $y(v, w)$ variables.

Note that this structure is a member of $\mathcal{Y}'$, but it is not a member of $\mathcal{Y}$. This is because the production

implies that the leaf `barks` should be preceded by the leaf `cat` in the tree, but it is instead preceded by the leaf `dog`. More formally, the structure is not a member of $\mathcal{Y}$ because two constraints in **C2** are not satisfied: we have $y(\texttt{cat}, \texttt{barks}) = 1$, but $y(\texttt{cat}) = 0$, hence $y(\texttt{cat}) \neq \sum_{\langle \text{cat}, v \rangle \in \mathcal{B}} y(\texttt{cat}, v)$; and we have $y(\texttt{dog}) = 1$ but $y(\texttt{dog}, v) = 0$ for all v, hence $y(\texttt{dog}) \neq \sum_{\langle \text{dog}, v \rangle \in \mathcal{B}} y(\texttt{dog}k, v)$.

We now turn to the Lagrangian relaxation algorithm. We will find it useful to make further use of the "extended" hypergraph with rules $w \rightarrow [v]$ for $(v, w) \in \mathcal{B}$. This will give an alternative, but nonetheless equivalent, presentation of the algorithm. Recall that the crucial step at each iteration of the Lagrangian relaxation algorithm is to calculate

$$\arg\max_{y \in \mathcal{Y}'} L(u, y) = \arg\max_{y \in \mathcal{Y}'} \left(f(y) - \sum_v u(v) y(v) + \sum_v u(v) \sum_{w : \langle v, w \rangle \in \mathcal{B}} y(v, w) \right)$$

where $f(y) = \sum_v \theta(v) y(v) + \sum_e \theta(e) y(e)$. This is equivalent to finding

$$\arg\max_{y \in \mathcal{Y}'} \left(f(y) + \sum_{\langle v, w \rangle \in \mathcal{B}} y(v, w)(u(v) - u(w)) \right) = \arg\max_{y \in \mathcal{Y}'} \beta^\top y$$

where $\beta(e) = \theta(e)$, $\beta(v) = \theta(v)$, and $\beta(v, w) = \theta(v, w) + u(v) - u(w)$. This follows because for any $y \in \mathcal{Y}'$, we have $y(v) = \sum_{w : \langle w, v \rangle \in \mathcal{B}} y(w, v)$ (by constraints **C1**). Hence

$$-\sum_v u(v) y(v) + \sum_v u(v) \sum_{w : \langle v, w \rangle \in \mathcal{B}} y(v, w) = \sum_{\langle v, w \rangle \in \mathcal{B}} y(v, w)(u(v) - u(w))$$

It follows that

$$\arg\max_{y \in \mathcal{Y}'} L(u, y)$$

can be calculated by dynamic programming over the extended hypergraph, where each rule of the form

$$w \rightarrow [v]$$

receives a score

$$\beta(v, w) = \theta(v, w) + u(v) - u(w)$$

The weights on the vertices $v \in \mathcal{V}$ and the edges $e \in \mathcal{E}$ in the original hypergraph remain

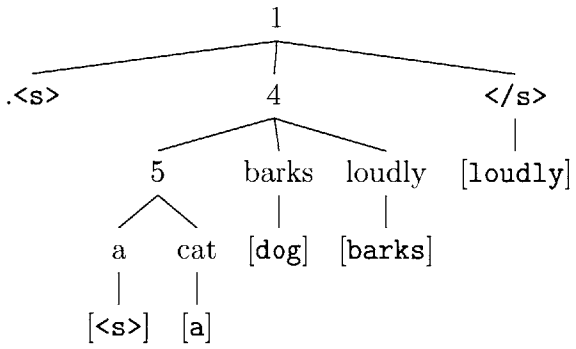
unchanged, as $\beta(v) = \theta(v)$, and $\beta(e) = \theta(e)$. The Lagrange multipliers $u(v)$ only effect the $\beta(v, w)$ parameters, now associated with the $w \rightarrow [v]$ rules. The advantage of this view of the algorithm is that everything is cast in terms of dynamic programming over hypergraphs.

We now give an example of the Lagrangian relaxation algorithm on the hypergraph in Figure 5-2. We will assume that $\theta(v, w) = 0$ for all $(v, w) \in \mathcal{B}$, with one exception, namely that $\theta(\texttt{cat}, \texttt{barks}) = -7$, so that the bigram (`cat`, `barks`) is penalized.

The algorithm in Figure 5-4 sets the initial Lagrange multiplier values $u^{(1)}(v)$ equal to 0. It then finds

$$y^{(1)} = \arg\max_{y \in \mathcal{Y}'} L(u^{(1)}, y)$$

This is achieved by dynamic programming over the extended hypergraph, where each rule $w \rightarrow [v]$ for $(v, w) \in \mathcal{B}$ has weight $\theta(v, w) + u^{(1)}(v) - u^{(1)}(w) = \theta(v, w) + 0 - 0 = \theta(v, w)$. It can be verified that the resulting structure, $y^{(1)}$, is



This structure is ill-formed—it does not satisfy the **C2** constraints—because the leaf `barks` specifies that the bigram (`dog`, `barks`) should be present, when in fact the leaf `cat`, not the leaf `dog`, directly precedes the leaf `barks`. We find the following violations of the **C2** constraints:

$$y(\texttt{cat}) = 1 \neq \sum_{w:(\texttt{cat},w) \in \mathcal{B}} y(\texttt{cat}, w) = 0$$

$$y(\texttt{dog}) = 0 \neq \sum_{w:(\texttt{dog},w) \in \mathcal{B}} y(\texttt{dog}, w) = 1$$

We then perform the updates

$$\forall v \in \mathcal{V}^{(t)}, u^{(2)}(v) = u^{(1)}(v) - \alpha_1 \left(y^{(1)}(v) - \sum_{w:\langle v,w\rangle \in \mathcal{B}} y^{(1)}(v,w) \right)$$

For simplicity, assume that the stepsize α_k is equal to 1: the new Lagrange multiplier values are then

$$u^{(1)}(dog) = 1, \quad u^{(1)}(cat) = -1$$

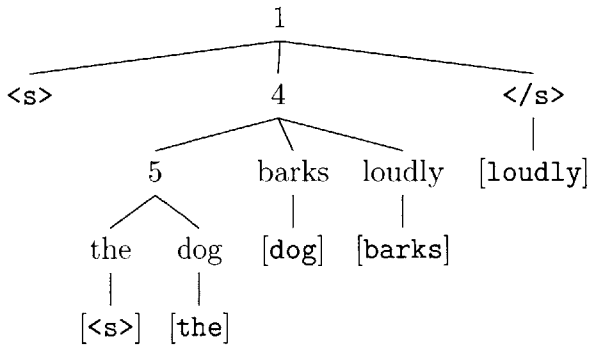
with $u^{(1)}(v) = 0$ for all values of v other than `dog` or `cat`.

With these new Lagrange multiplier values, any rule $w \rightarrow [v]$ with v and/or w equal to `dog` or `cat` will have its weight modified. Specifically, the rules `dog` $\rightarrow$ [`the`] and `barks` $\rightarrow$ [`cat`] have their weighted incremented by a value of 1 (and are hence encouraged), and the rules `cat` $\rightarrow$ [a] and `barks` $\rightarrow$ [`dog`] have their weights decremented by a value of 1 (and are hence discouraged).

The next step in the algorithm is to find

$$y^{(2)} = \arg\max_{y \in \mathcal{Y}'} L(u^{(2)}, y)$$

It can be verified that this time, the structure $y^{(2)}$ is



The structure $y^{(2)}$ is different from $y^{(1)}$, essentially because the new Lagrange multipliers encourage the rule `dog` $\rightarrow$ [`the`] (which is now seen in $y^{(2)}$), and discourage the rule `cat` $\rightarrow$ [a]

(which was in $y^{(1)}$, but is not in $y^{(2)}$). The **C2** constraints are satisified by $y^{(2)}$, and this structure is returned by the algorithm.

In summary, under this view of the algorithm, we implement the following steps:

- Create an "extended" hypergraph which contains the original set of hyperedges in $\mathcal{E}$, and in addition rules $w \rightarrow [v]$ for $(v, w) \in \mathcal{B}$. Each rule $w \rightarrow [v]$ has a weight $\theta(v, w) + u(v) - u(w)$, where $\theta(v, w)$ is the original bigram language modeling score, and $u(v)$ for $v \in \mathcal{V}^{(t)}$ is a Lagrange multiplier.

- At each iteration of the subgradient algorithm, find the highest scoring derivation in the extended hypergraph under the current Lagrange multiplier values. If the derivation is a member of $\mathcal{Y}$—i.e., it satisfies the **C2** constraints—then return it, with the guarantee that it is optimal. Otherwise, perform updates to the Lagrange multipliers and iterate.

Chapter 6

Non-Projective Dependency Parsing

[This chapter is adapted from joint work with Terry Koo, Michael Collins, Tommi Jaakkola and David Sontag entitled "Dual Decomposition for Parsing with Non-Projective Head Automata" (Koo et al., 2010)]

The focus of this chapter is decoding for higher-order, non-projective dependency parsing. We have seen that dependency parsing is an important form of syntactic parsing that aims to recover the head-modifier relationships of a sentence. In Chapter 2 we introduced the decoding problem for dependency parsing and talked about the challenge of finding optimal structures. In Chapter 4 we briefly discussed a model using projective dependency parsing, and in Chapter 8 we will return to the projective problem.

This chapter focuses on non-projective dependency parsing. Informally a non-projective parse allows dependency arcs that may cross each other, for example see Figure 6-1. While this phenomenon is relatively rare in English, it is quite common in languages with free word-order such as Czech and Dutch.

The distinction between projective and non-projective parses is critically important for decoding. If a parser optimizes over the set of projective parses, the decoding problem can often be solved with dynamic programming as discussed in the background chapter (Eisner & Satta, 1999; McDonald, 2006; Koo & Collins, 2010). However if we allow non-projective dependency parses the story becomes more complicated. If we use a simple, arc-factored

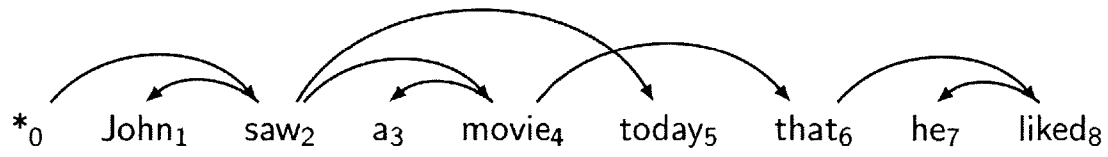


Figure 6-1: Example of a (non-projective) dependency parse.

model then there is a fast non-DP decoding algorithm. If we use higher-order models, for instance sibling models, the problem has been shown to be NP-hard (McDonald & Pereira, 2006; McDonald & Satta, 2007).

This chapter explores this problem of decoding for higher-order, non-projective, dependency parsing. We begin by formally introducing non-projective parsing and presenting a basic higher-order model known as a sibling model. We then give an initial dual-decomposition algorithm for this problem. Next we discuss higher-order models that extend the sibling-model. Finally we present experimental results using the system.

6.1 Dependency Parsing

We begin by reviewing the formal description of dependency parsing. These definitions focus on the elements of dependency parsing that are necessary for the algorithms discussed in this chapter. For a more complete overview of dependency parsing, see the summary given in the background.

The input for dependency parsing is a sentence consisting of tokens $s_1 \dots s_n$. Each token has a corresponding vertex in a fully-connected directed graph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$, which additionally includes a special root vertex 0 for a total set $\mathcal{V} = \{0 \dots n\}$. The set of parse structures is defined as all *directed spanning trees* over this graph, where a valid directed tree must be rooted at vertex 0, i.e. all arcs point away from the root. An example dependency parse is shown in Figure 6-1.

Define the set of parses as $\mathcal{Y} \subset \{0,1\}^{\mathcal{I}}$, where the index set $\mathcal{I}$ includes an element for

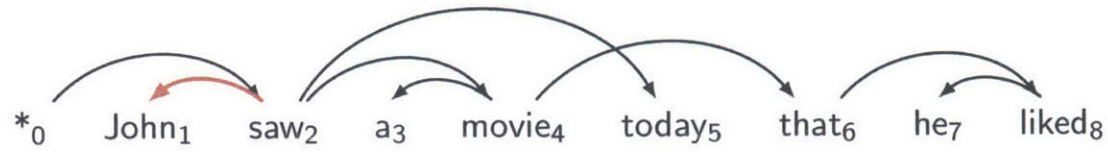


Figure 6-2: Example of a dependency parse with highlighted arc from **saw** *to* **John** *indicated as* $y(2,1) = 1$.

each possible directed arc in this graph, i.e.

$$\mathcal{I} = \{(h, m) : h \in \{0 \dots n\}, m \in \{1 \dots n\}\}$$

A parse $y \in \mathcal{Y}$ is an indicator vector where $y(h, m) = 1$ if a dependency with head word h and modifier m is in the parse, $y(h, m) = 0$ otherwise. Figure 6-2 gives an example of an indicator over this set.

We will first consider a linear scoring function over this set. Define the scoring function $f : \mathcal{Y} \mapsto \mathbb{R}$ where $f(y; \theta) = \theta^\top y$ and parameter vector $\theta \in \mathbb{R}^{\mathcal{I}}$ which includes a score for each possible dependency arc (h, m). Scoring functions of this form are referred to as *first-order* or alternatively *arc-factored.* They yield a decoding problem of the form

$$y^* = \arg\max_{y \in \mathcal{Y}} \theta^\top y \tag{6.1}$$

McDonald, Pereira, Ribarov, and Hajič (2005) were the first to define models of this form and show that the optimal parse tree y^* can be found efficiently using maximum directed spanning algorithms (Chu & Liu, 1965; Edmonds, 1967).

6.2 Higher-Order Dependency Parsing

Unfortunately, as we will see in experimental results, first-order models make too strong an assumption about the underlying dependency structure. Models that include higher-order information, in addition to the arcs themselves, often score significantly higher than basic first-order models.

In this section, we extend the model described to include higher-order information, in particular to include the score of sibling modifiers into the objective. While this will help improve accuracy, including this extra information makes the non-projective decoding problem NP-hard.

Consider now an extension for including this additional sibling information. It will be convenient to first define the following notation. Given an vector $y \in \{0,1\}^{\mathcal{I}}$, define the notation $y_{|h}$ to indicate a sub-vector where $y_{|h}(m) \triangleq y(h,m)$ for all $m \in \{1 \dots n\}$. This notation acts as an indicator vector for the words that modify the head index h. Note that the vectors $y_{|h}$ for $h \in \{0 \dots n\}$ form a partition of the full vector y.

For example, assume the parse in Figure 6-1 is y. The word $\texttt{he}_7$ has no modifiers, so $y_{|7}$ is the zero vector. The word $\texttt{movie}_4$ has two modifiers so $y_{|4}(3) = 1$ and $y_{|4}(6) = 1$, while other values are 0. The word $\texttt{saw}_2$ has three modifiers so $y_{|2}(1) = 1$, $y_{|2}(4) = 1$, and $y_{|2}(5) = 1$ as shown in Figure 6-3.

In this section, we will be interested in scoring functions of the form $g : \{0,1\}^{\mathcal{I}} \mapsto \mathbb{R}$ that in contrast to the previously defined function f may not be linear in the input. Instead, we assume that g takes the form

$$g(y) = \sum_{h=0}^{n} g_h(y_{|h})$$

Thus g decomposes into a sum of terms. where each g_h considers modifiers to the h'th word alone. Note in the general case, finding

$$y^* = \arg\max_{y \in \mathcal{Y}} g(y)$$

under this definition of $g(y)$ is an NP-hard problem.

However for certain definitions of g_h, it is possible to efficiently compute the best modifiers for a fixed head. Let $\mathcal{Z}_h$ refer to the set of all possible modifiers for word h: specifically, $\mathcal{Z}_h = \{0,1\}^{n-1}$ for $h \in \{1 \dots n\}$. We will assume that we have an efficient algorithm for

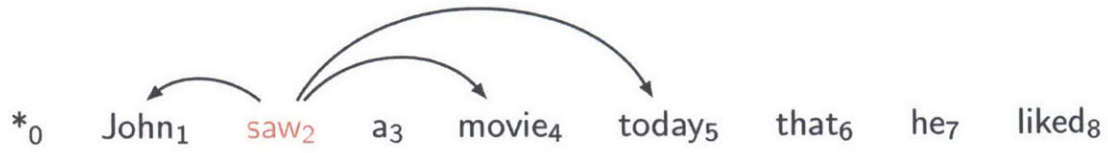


Figure 6-3: One possible setting of the modifiers for the word **saw**. *These modifiers are be indicated by the vector* $y_{|2}$.

solving

$$\arg\max_{z_{|h} \in \mathcal{Z}_h} g_h(z_{|h})$$

Under this assumption, we can also efficiently compute

$$z^* = \arg\max_{z \in \mathcal{Z}} g(z) = \arg\max_{z \in \mathcal{Z}} \sum_{h=0}^{n} g_h(z_{|h}) \tag{6.2}$$

where $\mathcal{Z} = \left\{z : z_{|h} \in \mathcal{Z}_h, h \in \{0 \dots n\}\right\} = \{0,1\}^{\mathcal{I}}$ is the set of *directed subgraphs* over $\mathcal{D}$. We can solve this decoding problem by simply computing

$$z^*_{|h} = \arg\max_{z_{|h} \in \mathcal{Z}_h} g_h(z_{|h})$$

for all $h \in \{0 \dots n\}$. Eq. 6.2 in an approximation to Eq. 6.1, where we have replaced $\mathcal{Y}$ with $\mathcal{Z}$. We will make direct use of this technique in the dual-decomposition parsing algorithm.

Note that $\mathcal{Y} \subseteq \mathcal{Z}$, and in all but trivial cases, $\mathcal{Y}$ is a strict subset of $\mathcal{Z}$. This means that $\mathcal{Z}$ contains structures that are not valid dependency parses. For example, a structure $z \in \mathcal{Z}$ could have $z(h,m) = z(m,h) = 1$ for some (h,m); it could contain longer cycles; or it could contain words that modify more or less than one head. Nevertheless, with suitably powerful functions g_h, z^* may be a good approximation to y^*. Furthermore we can use this relaxation to find exact solutions for decoding in sibling models.

6.3 Sibling Models

We now give the main assumption underlying sibling models:

Assumption 6.3.1 (Sibling Decompositions). *A scoring function $g(z)$ satisfies the sibling-decomposition assumption if:*

1. $g(z) = \sum_{h=0}^{n} g_h(z_{|h})$ *for some set of functions* $g_0 \dots g_n$.
2. *For any* $h \in \{0 \dots n\}$, *for any value of the variables* $\lambda(h, m) \in \mathbb{R}$ *for* $m \in \{1 \dots n\}$, *it is possible to compute*

$$\arg\max_{z_{|h} \in \mathcal{Z}_h} \left(g_h(z_{|h}) - \sum_{m=1}^{n} \lambda(h, m) z_{|h}(m) \right)$$

in polynomial time.

The first condition is the decomposition from the previous section. The second condition states that we can efficiently decode the sibling models independently, even with additional Lagrangian multipliers. As noted above, in general the scoring function g does not satisfy Property 2; however several important models do meet this condition. These include:

Bigram Sibling Models Recall that $z_{|h}$ is a binary vector specifying which words are modifiers to the head word h. Define $h > l_1 > \dots > l_p$ to be the sequence of left modifiers to word h under $z_{|h}$ and $h < r_1 < \dots < r_q$ to be the sequence of right modifiers, i.e. positions where $z_{|h}(l_1) = 1 \dots z_{|h}(l_p) = 1$ and $z_{|h}(l_1) = 1 \dots z_{|h}(l_p) = 1$ and all other $z_{|h}$ indices are zero. Additionally define boundary positions such that $l_0 = r_0 =$ `<d>` indicate an initial state, and $l_{p+1} = r_{q+1} =$ `</d>` indicate an end state.

In *bigram sibling models* we define the parameters $\omega^{(L)}$ and $\omega^{(R)}$ for the left and right sides of the head word. The decomposed scoring function is

$$g_h(z_{|h}; \omega) = \sum_{k=1}^{p+1} \omega^{(L)}(h, l_{k-1}, l_k) + \sum_{k=1}^{q+1} \omega^{(R)}(h, r_{k-1}, r_k)$$

The decoding problem for this decomposition is to find

$$\arg\max_{z_{|h} \in \mathcal{Z}_h} g_h(z_{|h}) - \sum_{m=1}^{n} \lambda(h,m) z_{|h}(m)$$

By definition the scoring function $g(y) = \sum_{h=0}^{n} g_h(y_{|h})$ satisfies Property 1. To satisfy Property 2 we must show that this optimization problem is efficiently solvable. Fortunately this problem can be solved using dynamic programming over a lattice using the Viterbi algorithm as introduced in Section 2.2.

The following lattice $(\mathcal{V}, \mathcal{E})$ defines the bigram decoding problem, which for simplicity we describe for right-facing modifiers for head word index h

- For each possible modifier or boundary $m \in \{(h+1) \dots n\} \cup \{\texttt{<d>}, \texttt{</d>}\}$, define a vertex $m \in \mathcal{V}$.
- For each possible modifier or end boundary $m \in \{(h+1) \dots n\} \cup \{\texttt{</d>}\}$ and each possible sibling (previous modifier) or start boundary $s \in \{(h+1) \dots (j-1)\} \cup \{\texttt{<d>}\}$, define an edge (s, m).

  Traversing this edge implies that s is the sibling of m. Define the weight of this edge as $\omega^{(R)}(h, s, m)$.

Each possible path through this lattice corresponds to a setting of right modifiers for head word h and has weight $\sum_{k=1}^{q+1} \omega^{(R)}(h, r_{k-1}, r_k)$. We can find the highest-scoring path by simply using the Viterbi decoding algorithm. Since the size of this lattice is $|\mathcal{E}| = O(n^2)$, this algorithm can be used to efficiently find the best path in quadratic-time. Hence the model satisfies Property 2.

As an example, consider the sentence `John saw a movie today that he liked` with length $n = 8$. And consider possible right modifiers of the word $\texttt{saw}_2$ indicated by vector $z_{|2}$. One possible (incorrect) setting of modifiers is to have $\texttt{saw}_2 \rightarrow \texttt{a}_3$ and $\texttt{saw}_2 \rightarrow \texttt{today}_5$. For this setting we have $z_{|2}(3) = 1$ and $z_{|2}(5) = 1$ and the rest equal to zero.

Under the notation given for this setting of right modifiers, we have $q = 2$, with $r_0, r_1, r_2, r_3 = \texttt{<d>}, 3, 5, \texttt{</d>}$. Figure 6-4 shows the full lattice described for this problem.

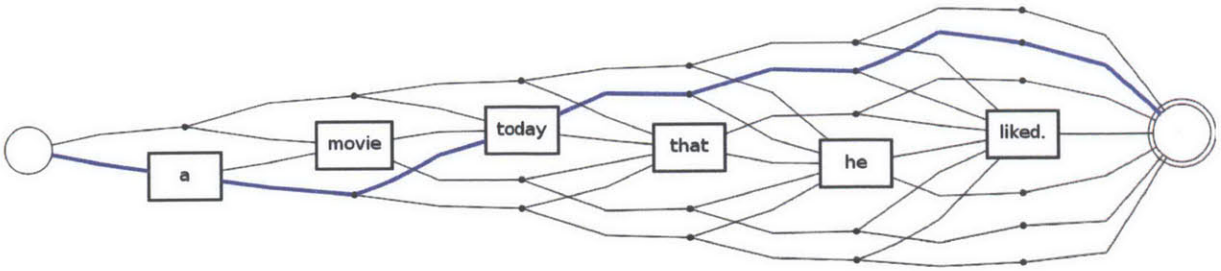


Figure 6-4: Lattice for the sibling head automaton for the right modifiers of the word `saw`. *Path with modifiers* `a` *and* `movie` *is highlighted.*

Each path in the lattice is a valid setting of right modifiers, and the path corresponding to this setting is highlighted.

Sibling Head Automata Sibling head automata are a generalization of bigram sibling models that still satisfy the sibling-decomposition assumption. In bigram sibling models, we scored an arc based on the head word, the modifier word and the previous sibling. In head automata models, we score the arc based on the head-modifier pair and an automata state computed from the previous history of siblings. This model generalizes bigram and n-gram sibling models, as well as models where history is dependent on context.

Head automata models assume there is a weighted finite-state automaton specified for each head word h and direction $d \in \{L, R\}$ specified by the 6-tuple $(\Sigma, \mathcal{Q}, q_0, \mathcal{F}, \tau, w)^{(h,d)}$. Recall the Σ is the symbol alphabet, $\mathcal{Q}$ is the state set, q_0 and $\mathcal{F}$ are the starting and ending states respectively, and τ and w are the transition and weight function. (For more details on FSAs see background Section 2.2.5.) We make use of the notation $w(\sigma_1 \dots \sigma_n)$ to indicate the score under a (deterministic) FSA of the symbol sequence $\sigma_1 \dots \sigma_n$ where for $i \in \{1 \dots n\}$, $\sigma_i \in \Sigma$.

For this problem we define the symbol alphabet $\Sigma = \{1 \dots n\} \cup \{\texttt{<d>}, \texttt{</d>}\}$ as possible modifiers or boundaries, and specify the decomposed scoring function as

$$g_h(z_{|h}) = w^{(h,L)}(l_0 \dots l_{p+1}) + w^{(h,R)}(r_0 \dots r_{q+1})$$

where recall l and r are the sequence of modifiers to the left and right of h respectively as

defined above.

In this case finding the highest-scoring member

$$\arg\max_{z_{|h} \in \mathcal{Z}_h} g_h(z_{|h})$$

can be computed in $O(n|\mathcal{Q}|)$ time again using a variant of the Viterbi algorithm. Hence the model satisfies the sibling-decomposition assumption.

As an example consider a head automata model that captures trigram sibling information. For simplicity we will consider just the right-facing modifiers for head word h. Let the set of states be

$$\mathcal{Q}^{(h,R)} = \{(i,j) : i \in \{h+1 \ldots n+1\} \cup \{\texttt{<d>}\}, j \in \{i+1 \ldots n\} \cup \{\texttt{<d>}, \texttt{</d>}\}\}$$

where state (i,j) indicates that i and j were the previous sibling modifiers chosen. The start state is defined as $q_0 = (\texttt{<d>}, \texttt{<d>})$ and the set of final states is defined as $\mathcal{F} = \{(i, \texttt{</d>}) : i \in \{h+1 \ldots n\} \cup \{\texttt{<d>}\}\}$. The transition function τ is defined as

$$\tau(\sigma, (i,j)) = (j, \sigma)$$

The weight function may be defined as any score on triples of modifiers and the head index.

Consider the set of right-facing arcs from Figure 6-1. The right-facing arcs of $h = 2$ are `movie` at position 4 and `today` at position 5, and so the sequence $r_0 \ldots r_3 = \texttt{<d>}, 4, 5, \texttt{</d>}$. The state sequence of the head automata is therefore $(\texttt{<d>}, \texttt{<d>}), (\texttt{<d>}, 4), (4,5), (5, \texttt{</d>})$.

6.4 A Dual Decomposition Parsing Algorithm

We now describe the dual decomposition parsing algorithm for models that satisfy Assumption 6.3. Consider the following generalization of the decoding problem from Eq. 6.1, where

$g(z) = \sum_{h=0}^{n} g_h(z_{|h})$ and $f(y) = \theta^\top y$ is a first-order scoring function:[1]

$$\begin{aligned} \underset{y \in \mathcal{Y}, z \in \mathcal{Z}}{\arg\max} \quad & f(y) + g(z) \text{ such that} \\ & y(h,m) = z(h,m) \text{ for all } (h,m) \in \mathcal{I} \end{aligned} \tag{6.3}$$

Although the maximization w.r.t. z is taken over the set $\mathcal{Z}$, the constraints in Eq. 6.3 ensure that $y = z$ for some $y \in \mathcal{Y}$, and hence that $z \in \mathcal{Y}$.

As we have seen in the previous dual decomposition examples, without the $y(h,m) = z(h,m)$ constraints, the objective would decompose into separate maximizations $y^* = \arg\max_{y \in \mathcal{Y}} f(y)$ and $z^* = \arg\max_{z \in \mathcal{Z}} g(z)$, which can be easily solved using maximum-directed spanning tree and Property 2, respectively. Thus, it is these constraints that complicate the optimization. These variables $\lambda(h,m)$ correspond to Lagrange multipliers for the $z(h,m) = y(h,m)$ constraints.

Set $\lambda^{(1)}(h,m) \leftarrow 0$ for all $(h,m) \in \mathcal{I}$
for $k = 1$ to K **do**
  $y^{(k)} \leftarrow \arg\max_{y \in \mathcal{Y}} \left(\sum_{(h,m) \in \mathcal{I}} \big(\theta(h,m) + \lambda^{(k)}(h,m)\big) y(h,m) \right)$
  for $h \in \{0 \ldots n\}$ **do**
    $z_{|h}^{(k)} \leftarrow \arg\max_{z_{|h} \in \mathcal{Z}_h} \left(g_h(z_{|h}) - \sum_{m=1}^{n} \lambda^{(k)}(h,m) z_{|h}(m) \right)$
  if $y^{(k)}(h,m) = z^{(k)}(h,m)$ for all $h, m \in \mathcal{I}$ **then**
    return $y^{(k)}$
  for $(h,m) \in \mathcal{I}$ **do**
    $\lambda^{(k+1)}(h,m) \leftarrow \lambda^{(k)}(h,m) + \alpha_k \left(z^{(k)}(h,m) - y^{(k)}(h,m) \right)$
return $y^{(K)}$

Algorithm 8: The dual decomposition parsing algorithm for sibling decomposable models.

The full parsing algorithm is shown in Figure 8. At each iteration k, the algorithm finds $y^{(k)} \in \mathcal{Y}$ using the maximum directed spanning tree algorithm, and $z^{(k)} \in \mathcal{Z}$ through indi-

[1] This is equivalent to Eq. 6.1 when $\theta(h,m) = 0$ for all (h,m). In some cases, however, it is convenient to have a model with non-zero values for the θ variables.

vidual decoding of the $(n+1)$ sibling models. The $\lambda^{(k)}$ variables are updated if $y^{(k)}(h,m) \neq z^{(k)}(h,m)$ for some (h,m); these updates modify the objective functions for the two decoding steps, and intuitively encourage the $y^{(k)}$ and $z^{(k)}$ variables to be equal.

The updates in Figure 8 correspond to the standard subgradient updates:

$$\lambda^{(k+1)} = \lambda^{(k)} - \alpha_k(y^{(k)} - z^{(k)}),$$

where α_k is a step size.

6.5 Grandparent Dependency Models

Empirically, another useful source of features beyond sibling relationships is the chain of head words leading to the current modifier. In particular incorporating the grandparent word (head of the head) has been shown to increase performance for many languages. In this section we extend the decoding algorithm to consider grandparent relations.

Define an extended index set $\mathcal{I}_\uparrow$ for dependency parsing that mirrors the original indices as

$$\mathcal{I}_\uparrow = \{(h,m,\uparrow) : (h,m) \in \mathcal{I}\}$$

We then redefine the set $\mathcal{Y}$ to use these redundant indices, $\mathcal{Y} \subset \{0,1\}^{\mathcal{I} \cup \mathcal{I}_\uparrow}$, and constrain the two sets of variables to take identical values.

$$\begin{aligned} \mathcal{Y} = \{y \ : \ & y(h,m) \text{ variables form a directed tree,} \\ & y_\uparrow(h,m) = y(h,m) \ \text{ for all } (h,m) \in \mathcal{I}\} \end{aligned}$$

where the notation $y_\uparrow(h,m)$ indicates $y(h,m,\uparrow)$.

We again partition the variables into $n+1$ subsets, $y_{|0} \ldots y_{|n}$, by (re)defining the subvector $y_{|h}$ as $y_{|h} \triangleq y(h,m)$ and also $y_{|h\uparrow}(t) = y_\uparrow(t,h)$. So as before $y_{|h}$ contains n variables, $y_{|h}(m)$ which indicate which words modify the h'th word. In addition, $y_{|h}$ includes $n+1$ "grandparent" variables $y_{|h\uparrow}(t)$ that indicate the index t that head word h itself modifies.

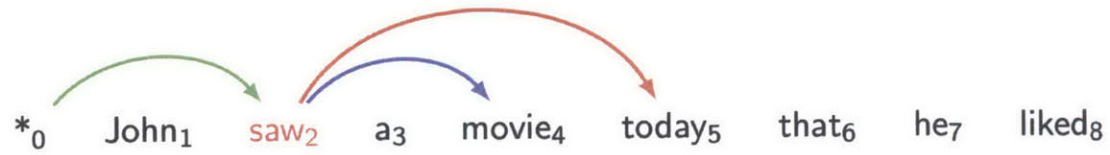


Figure 6-5: Example of a scored part in a grandparent/sibling model. The function $\omega(h, t, s, m)$ looks at the head, grandparent, sibling, and modifier to produce a score. For this example the grandparent is $*_0$, the head is $\texttt{saw}_2$, the sibling is $\texttt{movie}_4$ and the modifier is $\texttt{today}_5$.

As before we also consider the set of structures over these dependency arcs, $\mathcal{Z} \subset \{0,1\}^{\mathcal{I} \cup \mathcal{I}_\uparrow}$, that are not constrained to be valid directed trees. For any structure $z \in \mathcal{Z}$, the set of possible values of $z_{|h\uparrow}$ is

$$\begin{aligned}\mathcal{Z}_h = \{z_{|h\uparrow} \quad : \quad & z(h,m) \in \{0,1\} \text{ for } j \in \{1 \ldots n\}, m \neq h, \\ & z_\uparrow(t,h) \in \{0,1\} \text{ for } k = \{0 \ldots n\}, t \neq h, \\ & \sum_t z_\uparrow(t,h) = 1\}\end{aligned}$$

Hence the $z(h,m)$ variables can take any values, but only one of the $n+1$ variables $z|h\uparrow$ can be equal to 1 (as only one word can be a parent of word h). We fully define the set $\mathcal{Z} = \left\{z : z_{|h} \in \mathcal{Z}_h \text{ for } h \in \{0 \ldots n\}\right\}$.

As with sibling models we make the following grand-sibling assumptions:

Assumption 6.5.1 (GS Decompositions)**.**

A scoring function g satisfies the grandparent/sibling-decomposition (GSD) assumption if:

1. $g(z) = \sum_{h=0}^{n} g_h(z_{|h})$ *for some set of functions* $g_0 \ldots g_n$.
2. *For any* $h \in \{0 \ldots n\}$, *for any value of the variables* $\lambda(h,m) \in \mathbb{R}$ *for* $m \in \{1 \ldots n\}$, *and* $\lambda_\uparrow(t,h) \in \mathbb{R}$ *for* $t \in \{0 \ldots n\}$, *it is possible to compute*

$$\arg\max_{z_{|h} \in \mathcal{Z}_h} g_h(z_{|h}) - \sum_{m=1}^{n} \lambda(h,m) z(h,m) - \sum_{t=0}^{n} \lambda_\uparrow(t,h) z_\uparrow(t,h)$$

in polynomial time.

Again, it follows that we can approximate $y^* = \arg\max_{y \in \mathcal{Y}} \sum_{h=0}^{n} g_h(y_{|h})$ by

$$z^* = \arg\max_{z \in \mathcal{Z}} \sum_{h=0}^{n} g_h(z_{|h})$$

by defining $z^*_{|h} = \arg\max_{z_{|h} \in \mathcal{Z}_h} g_h(z_{|h})$ for $h \in \{0 \ldots n\}$. The resulting vector z^* may be deficient in two respects. First, the variables $z^*(h, m)$ may not form a well-formed directed spanning tree. Second, we may have $z^*_{\uparrow}(h, m) \neq z^*(h, m)$ for some values of (h, m).

Grandparent/Sibling Models An important class of models that satisfy Assumption 2 are defined as follows. Again, for a vector $z_{|h}$ define $l_1 \ldots l_p$ to be the sequence of left modifiers to word i under $z_{|h}$, and $r_1 \ldots r_q$ to be the set of right modifiers. Define t^* to the value for t such that $z_{\uparrow}(t, h) = 1$. Then the model is defined as follows:

$$g_h(z_{|h}) = \sum_{j=1}^{p+1} \omega^{(L)}(h, t^*, l_{j-1}, l_j) + \sum_{j=1}^{q+1} \omega^{(R)}(h, t^*, r_{j-1}, r_j)$$

This is very similar to the bigram-sibling model, but with the modification that the $\omega^{(L)}$ and $\omega^{(R)}$ functions depend in addition on the value for t^*. This allows these functions to model grandparent dependencies such as (t^*, i, l_j) and sibling dependencies such as (h, l_{j-1}, l_j). Finding $z^*_{|h}$ under the definition can be accomplished in $O(n^3)$ time, by decoding the model using dynamic programming separately for each of the $O(n)$ possible values of t^*, and picking the value for t^* that gives the maximum value under these decodings. $\square$

A dual-decomposition algorithm for models that satisfy the GSD assumption is shown in Figure 9. The algorithm can be justified as an instance of dual decomposition applied to the problem

$$\arg\max_{y \in \mathcal{Y}, z \in \mathcal{Z}} \quad f(y) + g(z) \tag{6.4}$$

with constraints

$$y(h, m) = z(h, m) \text{ for all } (h, m) \in \mathcal{I} \tag{6.5}$$

$$y(h, m) = z_{\uparrow}(h, m) \text{ for all } (h, m) \in \mathcal{I} \tag{6.6}$$

The algorithm employs two sets of Lagrange multipliers, $\lambda(h, m)$ and $\lambda_{\uparrow}(t, h)$, corresponding to constraints in Eqs. 6.5 and 6.6. If at any point in the algorithm $y^{(k)} = z^{(k)}$, then $y^{(k)}$ is an exact solution to the problem in Eq. 6.4.

6.6 Training Relaxations

In our experiments we make use of discriminative linear models, where for an input sentence x, the score for a parse y is $g(y) = w \cdot \phi(x, y)$ where $w \in \mathbb{R}^d$ is a feature parameter vector, and $\phi(x, y) \in \mathbb{R}^d$ is a feature vector representing parse tree y in conjunction with sentence x. We will assume that the features decompose in the same way as the sibling-decomposable or grandparent/sibling-decomposable models, that is $\phi(x, y) = \sum_{i=0}^{n} \phi(x, y_{|h})$ for some feature vector definition $\phi(x, y_{|h})$. In the *bigram sibling* models in our experiments, we assume that

$$\phi(x, y_{|h}) = \sum_{k=1}^{p+1} \phi^{(L)}(x, h, l_{k-1}, l_k) + \sum_{k=1}^{q+1} \phi^{(R)}(x, h, r_{k-1}, r_k)$$

where as before $l_1 \ldots l_p$ and $r_1 \ldots r_q$ are left and right modifiers under $y_{|h}$, and where $\phi^{(L)}$ and $\phi^{(R)}$ are feature vector definitions. In the *grandparent models* in our experiments, we use a similar definition with feature vectors $\phi^{(L)}(x, h, t^*, l_{k-1}, l_k)$ and $\phi^{(R)}(x, h, t^*, r_{k-1}, r_k)$, where t^* is the parent for word h under $y_{|h}$.

We train the model using the averaged perceptron for structured problems (Collins, 2002). Given the i'th example in the training set, $(x^{(i)}, y^{(i)})$, the perceptron updates are as follows:

- $z^* = \arg\max_{z \in \mathcal{Z}} w \cdot \phi(x^{(i)}, z)$
- If $z^* \neq y^{(i)}$, $w = w + \phi(x^{(i)}, y^{(i)}) - \phi(x^{(i)}, z^*)$

The first step involves inference over the set $\mathcal{Z}$, rather than $\mathcal{Y}$ as would be standard in the perceptron. Thus, inference during training can be achieved by dynamic programming over head automata alone, which is very efficient.

Our training approach is closely related to *local training methods* (Punyakanok et al., 2005). We have found this method to be effective, very likely because $\mathcal{Z}$ is a superset of $\mathcal{Y}$. Our training algorithm is also related to recent work on training using *outer bounds* (see, e.g., (Taskar et al., 2003a; Finley & Joachims, 2008; Kulesza & Pereira, 2008; Martins et al., 2009a)). Note, however, that the LP relaxation optimized by dual decomposition is significantly tighter than $\mathcal{Z}$. Thus, an alternative approach would be to use the dual decomposition algorithm for inference during training.

Set $\lambda^{(1)}(h,m) \leftarrow 0$, $\lambda_{\uparrow}^{(1)}(h,m) \leftarrow 0$ for all $(h,m) \in \mathcal{I}$
for $k = 1$ to K **do**

$$y^{(k)} \leftarrow \arg\max_{y \in \mathcal{Y}} \left(\sum_{(h,m) \in \mathcal{I}} y(h,m)(\theta(h,m) + \lambda^{(k)}(h,m) + \lambda_{\uparrow}^{(k)}(h,m)) \right)$$

for $h \in \{0 \ldots n\}$ **do**

$$z_{|h}^{(k)} \leftarrow \arg\max_{z_{|h} \in \mathcal{Z}_h} \; g_h(z_{|h}) - \sum_{m=1}^{n} \lambda^{(k)}(h,m) z(h,m) - \sum_{t=0}^{n} \lambda_{\uparrow}^{(k)}(t,h) z_{\uparrow}(t,h)$$

if $y^{(k)}(h,m) = z^{(k)}(h,m) = z_{\uparrow}^{(k)}(h,m)$ for all $(h,m) \in \mathcal{I}$ **then**
return $y^{(k)}$
for $(h,m) \in \mathcal{I}$ **do**
$\lambda^{(k+1)}(h,m) \leftarrow \lambda^{(k)}(h,m) + \alpha_k(z^{(k)}(h,m) - y^{(k)}(h,m))$
$\lambda_{\uparrow}^{(k+1)}(h,m) \leftarrow \lambda_{\uparrow}^{(k)}(h,m) + \alpha_k(z_{\uparrow}^{(k)}(h,m) - y^{(k)}(h,m))$
return $y^{(K)}$

Algorithm 9: The parsing algorithm for grandparent/sibling-decomposable models.

6.7 Related Work

McDonald et al. (2005) describe MST-based parsing for non-projective dependency parsing models with arc-factored decompositions; McDonald and Pereira (2006) make use of an approximate (hill-climbing) algorithm for parsing with more complex models. McDonald and Pereira (2006) and McDonald and Satta (2007) describe complexity results for non-projective parsing, showing that parsing for a variety of models is NP-hard. Riedel and Clarke (2006c) describe ILP methods for the problem; Martins et al. (2009a) recently introduced alternative LP and ILP formulations.

The algorithm presented differs in that we do not use general-purpose LP or ILP solvers, instead using an MST solver in combination with dynamic programming; thus we leverage the underlying structure of the problem, thereby deriving more efficient decoding algorithms.

6.8 Experiments

We report results on a number of data sets. For comparison to Martins et al. (2009a), we perform experiments for Danish, Dutch, Portuguese, Slovene, Swedish and Turkish data from the CoNLL-X shared task (Buchholz & Marsi, 2006), and English data from the CoNLL-2008 shared task (Surdeanu et al., 2008). We use the official training/test splits for these data sets, and the same evaluation methodology as Martins et al. (2009a). For comparison to Smith and Eisner (2008a), we also report results on Danish and Dutch using their alternate training/test split. Finally, we report results on the English WSJ treebank, and the Prague treebank. We use feature sets that are very similar to those described in Carreras (2007). We use marginal-based pruning, using marginals calculated from an arc-factored spanning tree model using the matrix-tree theorem (McDonald & Satta, 2007; Smith & Smith, 2007; Koo et al., 2007).

In all of our experiments we set the value K, the maximum number of iterations of dual decomposition in Figures 8 and 9, to be 5,000. If the algorithm does not terminate—i.e., it does not return $y^{(k)}$ within 5,000 iterations—we simply take the parse $y^{(k)}$ with the maximum

value of $g(y^{(k)})$ as the output from the algorithm. At first sight 5,000 might appear to be a large number, but decoding is still fast—see Sections 6.8 and 6.8 for discussion.[2]

We first discuss performance in terms of *accuracy*, *success in recovering an exact solution*, and *parsing speed*. We then describe additional experiments examining various aspects of the algorithm.

Accuracy Table 6.1 shows results for previous work on the various data sets, and results for an arc-factored model with pure MST decoding with our features. (We use the acronym UAS (unlabeled attachment score) for dependency accuracy.) We also show results for the bigram-sibling and grandparent/sibling (G+S) models under dual decomposition. Both the bigram-sibling and G+S models show large improvements over the arc-factored approach; they also compare favorably to previous work—for example the G+S model gives better results than all results reported in the CoNLL-X shared task, on all languages. Note that we use different feature sets from both Martins et al. (2009a) and Smith and Eisner (2008a).

Success in Recovering Exact Solutions Next, we consider how often our algorithms return an exact solution to the original optimization problem, with a certificate—i.e., how often the algorithms in Figures 8 and 9 terminate with $y^{(k)} = z^{(k)}$ for some value of $k < 5000$ (and are thus optimal, by the theorem from Chapter 3). The CertS and CertG columns in Table 6.1 give the results for the sibling and G+S models respectively. For all but one setting[3] over 95% of the test sentences are decoded exactly, with 99% exactness in many cases.

For comparison, we also ran both the single-commodity flow and multiple-commodity flow LP relaxations of Martins et al. (2009a) with our models and features. We measure how often these relaxations terminate with an exact solution. The results in Table 6.2 show that our method gives exact solutions more often than both of these relaxations.[4] In

[2]Note also that the feature vectors ϕ and inner products $w \cdot \phi$ only need to be computed once, thus saving computation.

[3]The exception is Slovene, which has the smallest training set at only 1534 sentences.

[4]Note, however, that it is possible that the Martins et al. relaxations would have given a higher proportion of integral solutions if their relaxation was used during training.

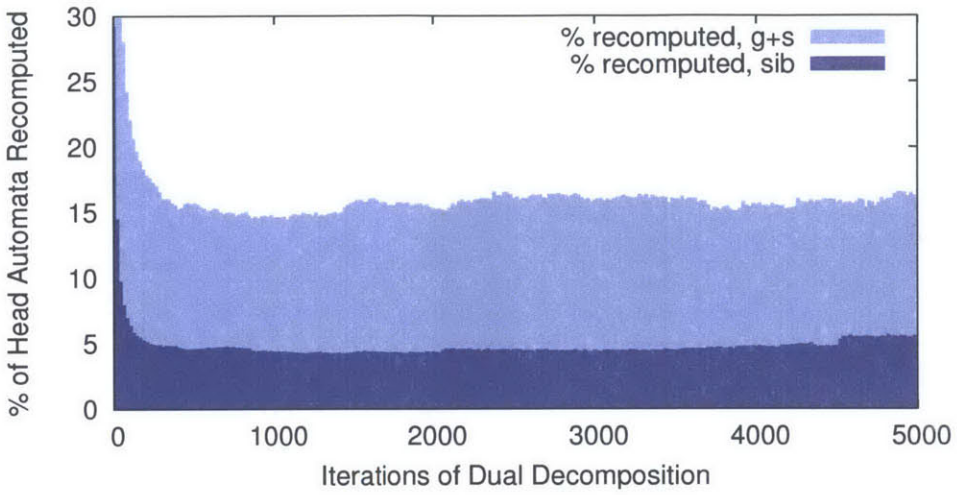


Figure 6-6: The average percentage of head automata that must be recomputed on each iteration of dual decomposition on the PTB validation set.

computing the accuracy figures for Martins et al. (2009a), we project fractional solutions to a well-formed spanning tree, as described in that paper.

Finally, to better compare the tightness of our LP relaxation to that of earlier work, we consider randomly-generated instances. Table 6.2 gives results for our model and the LP relaxations of Martins et al. (2009a) with randomly generated scores on automata transitions. We again recover exact solutions more often than the Martins et al. relaxations. Note that with random parameters the percentage of exact solutions is significantly lower, suggesting that the exactness of decoding of the trained models is a special case. We speculate that this is due to the high performance of approximate decoding with $\mathcal{Z}$ in place of $\mathcal{Y}$ under the trained models for g_h; the training algorithm described in Section 6.6 may have the tendency to make the LP relaxation tight.

Speed Table 6.1, columns TimeS and TimeG, shows decoding times for the dual decomposition algorithms. Table 6.2 gives speed comparisons to Martins et al. (2009a). Our method gives significant speed-ups over the Martins et al. (2009a) method, presumably because it leverages the underlying structure of the problem, rather than using a generic solver.

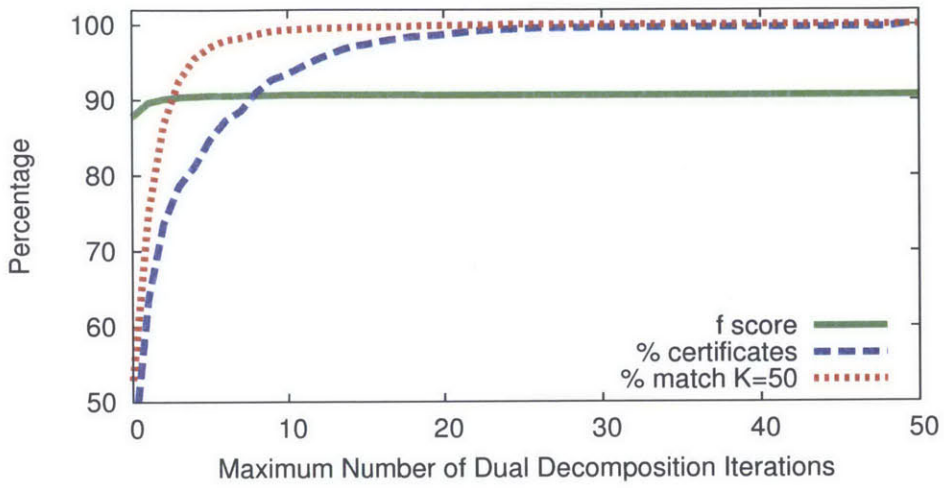


Figure 6-7: The behavior of the dual-decomposition parser with sibling automata as the value of K is varied.

Lazy Decoding Here we describe an important optimization in the dual decomposition algorithms. Consider the algorithm in Figure 8. At each iteration we must find

$$z_{|h}^{(k)} = \arg\max_{z_{|h} \in \mathcal{Z}_i} (g_h(z_{|h}) - \sum_{m=1}^{n} \lambda^{(k)}(h,m) z_{|h}(m))$$

for $h \in \{1 \ldots 0\}n$. However, if for some i, $\lambda^{(k)}(h,m) = \lambda^{(k-1)}(h,m)$ for all j, then $z_{|h}^{(k)} = z_{|h}^{(k-1)}$. In *lazy decoding* we immediately set $z_{|h}^{(k)} = z_{|h}^{(k-1)}$ if $\lambda^{(k)}(h,m) = \lambda^{(k-1)}(h,m)$ for all j; this check takes $O(n)$ time, and saves us from decoding with the i'th automaton. In practice, the updates to u are very sparse, and this condition occurs very often in practice. Figure 6-6 demonstrates the utility of this method for both sibling automata and G+S automata.

Early Stopping We also ran experiments varying the value of K—the maximum number of iterations—in the dual decomposition algorithms. As before, if we do not find $y^{(k)} = z^{(k)}$ for some value of $k \leq K$, we choose the $y^{(k)}$ with optimal value for $g(y^{(k)})$ as the final solution. Figure 6-7 shows three graphs: 1) the accuracy of the parser on PTB validation data versus the value for K; 2) the percentage of examples where $y^{(k)} = z^{(k)}$ at some point during the algorithm, hence the algorithm returns a certificate of optimality; 3) the percentage of

examples where the solution returned is the same as the solution for the algorithm with $K = 5000$ (our original setting). It can be seen for K as small as 250 we get very similar accuracy to $K = 5000$ (see Table 6.2). In fact, for this setting the algorithm returns the same solution as for $K = 5000$ on 99.59% of the examples. However only 89.29% of these solutions are produced with a certificate of optimality ($y^{(k)} = z^{(k)}$).

How Good is the Approximation z^*? We ran experiments measuring the quality of $z^* = \arg\max_{z \in \mathcal{Z}} g(z)$, where $g(z)$ is given by the perceptron-trained bigram-sibling model. Because z^* may not be a well-formed tree with n dependencies, we report precision and recall rather than conventional dependency accuracy. Results on the PTB validation set were 91.11%/88.95% precision/recall, which is accurate considering the unconstrained nature of the predictions. Thus the z^* approximation is clearly a good one; we suspect that this is one reason for the good convergence results for the method.

Importance of Non-Projective Decoding It is simple to adapt the dual-decomposition algorithms in figures 8 and 9 to give *projective* dependency structures: the set $\mathcal{Y}$ is redefined to be the set of all projective structures, with the arg max over $\mathcal{Y}$ being calculated using a projective first-order parser (Eisner & Satta, 1999). Table 6.3 shows results for projective and non-projective parsing using the dual decomposition approach. For Czech data, where non-projective structures are common, non-projective decoding has clear benefits. In contrast, there is little difference in accuracy between projective and non-projective decoding on English.

6.9 Conclusion

Non-projective dependency parsing is an important task in natural language processing, but the difficulty of decoding has often led researchers to either use simple scoring functions (McDonald et al., 2005), approximate algorithms (McDonald & Pereira, 2006) or off-the-shelf solvers (Riedel & Clarke, 2006c; Martins et al., 2009a). In this chapter we described a

method for this problem that both used efficient combinatorial algorithms but also is able to provide formal guarantees. We show empirically that this method finds provably optimal solutions for the vast majority of examples and gives performance gains on the underlying parsing tasks.

From a high-level, the work shows that while many decoding problems may be quite difficult in the worst-case, the actual instances of the problems that we are interested in – in real language and with a trained (non-random) scoring function – may be much easier to solve. This will not be true of all the problems we discuss, for instance for translation it was necessary to add additional constraints to find optimal solutions; however, when this case does occur these methods can be quite effective.

	Ma09	MST	Sib	G+S	Best	CertS	CertG	TimeS	TimeG	TrainS	TrainG
Dan	91.18	89.74	91.08	**91.78**	91.54	99.07	98.45	0.053	0.169	0.051	0.109
Dut	85.57	82.33	84.81	**85.81**	85.57	98.19	97.93	0.035	0.120	0.046	0.048
Por	92.11	90.68	92.57	**93.03**	92.11	99.65	99.31	0.047	0.257	0.077	0.103
Slo	85.61	82.39	84.89	**86.21**	85.61	90.55	95.27	0.158	0.295	0.054	0.130
Swe	90.60	88.79	90.10	**91.36**	90.60	98.71	98.97	0.035	0.141	0.036	0.055
Tur	76.34	75.66	77.14	**77.55**	76.36	98.72	99.04	0.021	0.047	0.016	0.036
Eng[1]	91.16	89.20	91.18	**91.59**	—	98.65	99.18	0.082	0.200	0.032	0.076
Eng[2]	—	90.29	92.03	**92.57**	—	98.96	99.12	0.081	0.168	0.032	0.076
	Sm08	MST	Sib	G+S	—	CertS	CertG	TimeS	TimeG	TrainS	TrainG
Dan	86.5	87.89	89.58	**91.00**	—	98.50	98.50	0.043	0.120	0.053	0.065
Dut	88.5	88.86	90.87	**91.76**	—	98.00	99.50	0.036	0.046	0.050	0.054
	Mc06	MST	Sib	G+S	—	CertS	CertG	TimeS	TimeG	TrainS	TrainG
PTB	91.5	90.10	91.96	**92.46**	—	98.89	98.63	0.062	0.210	0.028	0.078
PDT	85.2	84.36	86.44	**87.32**	—	96.67	96.43	0.063	0.221	0.019	0.051

Table 6.1: A comparison of non-projective automaton-based parsers with results from previous work. MST: Our first-order baseline. Sib/G+S: Non-projective head automata with sibling or grandparent/sibling interactions, decoded via dual decomposition. Ma09: The best UAS of the LP/ILP-based parsers introduced in Martins et al. (2009a). Sm08: The best UAS of any LBP-based parser in Smith and Eisner (2008a). Mc06: The best UAS reported by McDonald and Pereira (2006). Best: For the CoNLL-X languages only, the best UAS for any parser in the original shared task (Buchholz & Marsi, 2006) or in any column of ? (?)Table 1]martins-smith-xing:2009:ACLIJCNLP; note that the latter includes McDonald and Pereira (2006), Nivre and McDonald (2008), and Martins et al. (2008). CertS/CertG: Percent of test examples for which dual decomposition produced a certificate of optimality, for Sib/G+S. TimeS/TimeG: Seconds/sentence for test decoding, for Sib/G+S. TrainS/TrainG: Seconds/sentence during training, for Sib/G+S. For consistency of timing, test decoding was carried out on identical machines with zero additional load; however, training was conducted on machines with varying hardware and load. We ran two tests on the CoNLL-08 corpus. Eng[1]: UAS when testing on the CoNLL-08 validation set, following Martins et al. (2009a). Eng[2]: UAS when testing on the CoNLL-08 test set.

Sib	Acc	Int	Time	Rand
LP(S)	92.14	88.29	0.14	11.7
LP(M)	92.17	93.18	0.58	30.6
ILP	92.19	100.0	1.44	100.0
DD-5000	92.19	98.82	0.08	35.6
DD-250	92.23	89.29	0.03	10.2
G+S	Acc	Int	Time	Rand
LP(S)	92.60	91.64	0.23	0.0
LP(M)	92.58	94.41	0.75	0.0
ILP	92.70	100.0	1.79	100.0
DD-5000	92.71	98.76	0.23	6.8
DD-250	92.66	85.47	0.12	0.0

Table 6.2: A comparison of dual decomposition with linear programs described by Martins et al. (2009a). LP(S): Linear Program relaxation based on single-commodity flow. LP(M): Linear Program relaxation based on multi-commodity flow. ILP: Exact Integer Linear Program. DD-5000/DD-250: Dual decomposition with non-projective head automata, with $K = 5000/250$. Upper results are for the sibling model, lower results are G+S. Columns give scores for UAS accuracy, percentage of solutions which are integral, and solution speed in seconds per sentence. These results are for Section 22 of the PTB. The last column is the percentage of integral solutions on a random problem of length 10 words. The (I)LP experiments were carried out using Gurobi, a high-performance commercial-grade solver.

	Sib	P-Sib	G+S	P-G+S
PTB	92.19	92.34	92.71	92.70
PDT	86.41	85.67	87.40	86.43

Table 6.3: UAS of projective and non-projective decoding for the English (PTB) and Czech (PDT) validation sets. Sib/G+S: as in Table 6.1. P-Sib/P-G+S: Projective versions of Sib/G+S, where the MST component has been replaced with the Eisner and Satta (1999) first-order projective parser.

Part III

Relaxation Variants

Chapter 7

Translation Decoding with Optimal Beam Search

Machine translation presents one of the more difficult decoding challenges in natural language processing. For a given sentence we aim to find the highest-scoring translation out of a large combinatorial set of possibilities. The difficulty of this problem has led to interest in heuristic algorithms for efficiently finding feasible translations.

In recent years, beam search (Koehn et al., 2003) and cube pruning (Chiang, 2007) have become the *de facto* decoding algorithms for phrase- and syntax-based translation. The algorithms are central to large-scale machine translation systems due to their efficiency and tendency to produce high-quality translations (Koehn, 2004; Koehn et al., 2007; Dyer et al., 2010). However despite practical effectiveness, neither algorithm provides any bound on possible decoding error.

In Chapter 5 we introduced the task of statistical machine translation and developed a Lagrangian relaxation algorithm for a variant of translation known as syntax-based translation. We employed a constrained hypergraph to derive a relaxation-based decoding algorithm. Additionally we saw that, unlike our previous experiments on dual decomposition for parsing, the relaxation sometimes failed to find an exact solution. To deal with this problem we used a heuristic for incrementally introducing constraints. This approach was able to

produce optimal solutions for many instances of translation decoding.

This chapter extends that work in two ways. We begin by introducing a different method of statistical machine translation known as phrase-based translation, and show an analogous constrained hypergraph representation of the decoding problem. The second part of the chapter develops a general approach for constrained hypergraph problems utilizing a variant of the popular beam search heuristic. However, unlike the standard beam search heuristic our method is able to produce exact solutions with a certificate of optimality. We apply this method to phrase-based translation, as well as revisiting syntax-based translation. Our results show that this method provides significant speed-ups for decoding with both phrase-based and syntax-based systems.

7.1 Background: Constrained Hypergraphs

This chapter will make heavy use of hypergraph decoding and Lagrangian relaxation. For an overview of these topics see the background chapter.

In particular we will consider a variant of the hypergraph decoding problem: *constrained* hypergraph search. Constraints will be necessary for both phrase- and syntax-based decoding. In phrase-based models, the constraints will ensure that each source word is translated exactly once. In syntax-based models, the constraints will be used to intersect a translation forest with a language model (as in Chapter 5).

In the constrained hypergraph problem, hyperpaths must fulfill additional edge constraints. To align this notation with that of Lagrangian relaxation, we will define the set of valid hyperpaths in a given hypergraph $(\mathcal{V}, \mathcal{E})$ as $\mathcal{Y}'$ and define a set of weights $\theta \in \mathbb{R}^{\mathcal{E}}$ (for this chapter vertex weights will not used). Additionally we will find it useful to include a constant additive term τ in the objective, i.e. the score of a hyperpath will be $\theta^\top y + \tau$. For now this term can be set as $\tau = 0$, but we will see that it will be useful after defining our relaxation.

Define the set of *constrained* hyperpaths as

$$\mathcal{Y} = \{y \in \mathcal{Y}' : Ay = b\}$$

where we have a constraint matrix $A \in \mathbb{R}^{p \times |\mathcal{E}|}$ and vector $b \in \mathbb{R}^p$ encoding p constraints. The optimal constrained hyperpath is $y^* = \arg\max_{y \in \mathcal{Y}} \theta^\top y + \tau$.

Generally the constrained hypergraph search problem may be NP-Hard. Crucially this is true even when the corresponding unconstrained search problem is solvable in polynomial time. For instance, phrase-based decoding is known to be NP-Hard (Knight, 1999), but we will see that it can be expressed as a polynomial-sized hypergraph with constraints.

7.2 Decoding Problem for Phrase-Based Translation

Recall that the translation problem is to find the best scoring derivation of a source sentence to a target-language sentence. The derivation consists of the target-language output as well as some internal structure, for instance a synchronous parse tree. We score the derivation based on the combination of a model for this structure and a language model score.

In this section, we consider translating a source sentence $s_1 \dots s_{|s|}$ to a target sentence in a language with vocabulary Σ. A simple phrase-based translation model consists of a tuple $(\mathcal{P}, w, \Omega)$ with

- $\mathcal{P}$; a set of pairs (q, r) where $q_1 \dots q_{|q|}$ is a sequence of source-language words and $r_1 \dots r_{|r|}$ is a sequence of target-language words drawn from the target vocabulary Σ.
- $w : \mathbb{R}^{|\mathcal{P}|}$; parameters for the translation model mapping each pair in $\mathcal{P}$ to a real-valued score.
- $\Omega : \mathbb{R}^{\Sigma \times \Sigma}$; parameters of the language model mapping a bigram of target-language words to a real-valued score.

The translation decoding problem is to find the best derivation for a given source sentence. A derivation consists of a sequence of *phrases* $p = p_1 \dots p_n$. Define a phrase as a tuple

(q, r, j, k) consisting of a span in the source sentence $q = s_j \dots s_k$ and a sequence of target words $r_1 \dots r_{|r|}$, with $(q, r) \in \mathcal{P}$. We say the source words $s_j \dots s_k$ are *translated* to r.

The score of a derivation, $f(p)$, is the sum of the translation score of each phrase plus the language model score of the target sentence

$$f(p) = \sum_{i=1}^{n} w(q(p_i), r(p_i)) + \sum_{i=0}^{|u|+1} \Omega(u_{i-1}, u_i)$$

where u is the sequence of words in Σ formed by concatenating the phrases $r(p_1) \dots r(p_n)$, with boundary cases $u_0 =$ `<s>` and $u_{|u|+1} =$ `</s>`.

Crucially for a derivation to be valid it must satisfy an additional condition: it must translate every source word *exactly* once. The decoding problem for phrase-based translation is to find the highest-scoring derivation satisfying this property.

7.3 Constrained Hypergraph for Translation

We can represent this decoding problem as a constrained hypergraph using the construction of Chang and Collins (2011). The hypergraph weights encode the translation and language model scores, and its structure ensures that the count of source words translated is $|s|$, i.e. the length of the source sentence. Each vertex will remember the preceding target-language word and the count of source words translated so far.

The hypergraph, which for this problem is just a lattice/directed graph, takes the following form

- Vertices $v \in \mathcal{V}$ are labeled (c, u) where $c \in \{1 \dots |s|\}$ is the count of source words translated and $u \in \Sigma$ is the last target-language word produced by a partial hypothesis at this vertex. Additionally there is an initial terminal vertex labeled $(0,$ `<s>`$)$.

- There is a hyperedge $e \in \mathcal{E}$ with head (c', u') and tail $\langle (c, u) \rangle$ if there is a valid corresponding phrase (q, r, j, k) such that $c' = c + |q|$ and $u' = r_{|r|}$, i.e. c' is the count of words translated and u' is the last word of target phrase r. We call this phrase $p(e)$.

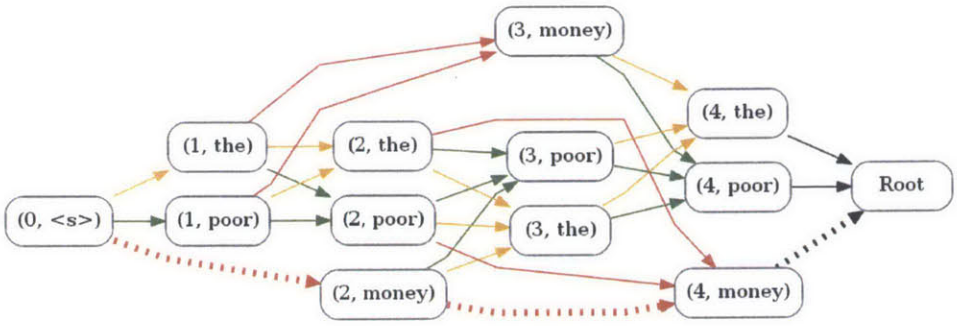


Figure 7-1: Hypergraph for translating the sentence $w =$ `les`$_1$ `pauvres`$_2$ `sont`$_3$ `demunis`$_4$ *with set of pairs* $\mathcal{P} = \{$(`les`, `the`), (`pauvres`, `poor`), (`sont demunis`, `don't have any money`)$\}$. *Hyperedges are color-coded by source words translated: orange for* `les`$_1$, *green for* `pauvres`$_2$, *and red for* `sont`$_3$ `demunis`$_4$. *The dotted lines show an invalid hyperpath* x *that has signature* $Ax = \langle 0, 0, 2, 2 \rangle \neq \langle 1, 1, 1, 1 \rangle$.

The weight of this hyperedge, $\theta(e)$, is the translation model score of the pair plus its language model score

$$\theta(e) = w(q, r) + \left(\sum_{i=2}^{|r|} \Omega(r_{i-1}, r_i) \right) + \Omega(u, r_1)$$

- To handle the end boundary, there are hyperedges with head v_0 and tail $\langle (|s|, u) \rangle$ for all $u \in \Sigma$. The weight of these edges is the cost of the stop bigram following u, i.e. $\Omega(u, \texttt{</s>})$.

While any valid derivation corresponds to a hyperpath in this graph, a hyperpath may not correspond to a valid derivation. For instance, a hyperpath may translate some source words more than once or not at all.

We handle this problem by adding additional constraints. For all source words $i \in \{1 \dots |s|\}$, define ρ as the set of hyperedges that translate s_i

$$\rho(i) = \left\{ e \in \mathcal{E} : j(p(e)) \leq i \leq k(p(e)) \right\}$$

Next define $|s|$ constraints enforcing that each word in the source sentence is translated exactly once

$$\sum_{e \in \rho(i)} y(e) = 1 \quad \forall\, i \in \{1 \ldots |s|\}$$

These linear constraints can be represented with a matrix $A \in \{0,1\}^{|s| \times |\mathcal{E}|}$ where the rows correspond to source indices and the columns correspond to edges. We call the product Ay the *signature*, where in this case $(Ay)_i$ is the number of times word i has been translated. The full set of constrained hyperpaths is $\mathcal{Y} = \{ y \in \mathcal{Y}' : Ay = \mathbf{1} \}$, and the best derivation under this phrase-based translation model has score $\max_{y \in \mathcal{Y}} \theta^\top y + \tau$ where τ can be set to 0.

Figure 7-1 shows an example hypergraph with constraints for translating the sentence `les pauvres sont demunis` into English using a simple set of phrases. Even in this small example, many of the possible hyperpaths violate the constraints and correspond to invalid derivations.

Example 7.3.1 (Phrase-based Translation). Consider now an example of phrase-based translation decoding. To define the problem we need to specify a phrase-based translation model, a language model, and finally the constrained hypergraph. For this example we will consider translating from German into a toy version of English. Define our English-language vocabulary as

$$\Sigma = \{\texttt{criticism}, \texttt{must}, \texttt{seriously}, \texttt{take}, \texttt{this}, \texttt{we}\}$$

And our German input sentence as $s = \texttt{wir}_1\ \texttt{mussen}_2\ \texttt{diese}_3\ \texttt{kritik}_4\ \texttt{ernst}_5\ \texttt{nehman}_6$ with length $m = 6$.

Let the phrase-based translation model $(\mathcal{P}, w)$ be defined as

Phrase $(q, r) \in \mathcal{P}$	$w(q, r)$
(`wir mussen`, `we must`)	1
(`nehmen`, `take`)	2
(`diese kritik`, `this criticism`)	10
(`ernst`, `seriously`,)	1

Let the bigram language model Ω have the entries $\Omega(u, u') = 2.0$ for all $u, u' \in \Sigma$.

Once we have these two models it is deterministic to construct the hypergraph for this model. The hypergraph is shown with labeled vertices in Figure 7-2(a). Note that the hypergraph is relatively small, and so we can find the best (unconstrained) hyperpath efficiently using dynamic programming. Figure 7-2(b) shows a possible hyperpath from this set $\mathcal{Y}'$; however, we have no guarantee that the underlying translation will be valid. For instance this hyperpath translates `wir mussen` twice. Its score is

$$2 \times w(\texttt{diese kritik}, \texttt{this criticism}) + w(\texttt{wir mussen}, \texttt{we must}) +$$
$$g(\texttt{<s> this criticism we must this criticism </s>}) = 28$$

To fix this issue, we add constraints to the problem. The constraint matrix has the form

$$A = \begin{array}{l} \text{source\textbackslash edge} \\ \texttt{wir} \\ \texttt{mussen} \\ \texttt{diese} \\ \texttt{kritik} \\ \texttt{nehmen} \\ \texttt{ernst} \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 & \ldots \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 0 & \ldots \\ 1 & 0 & 1 & 0 & \ldots \\ 0 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 0 & \ldots \\ 0 & 1 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 1 & \ldots \end{pmatrix} \end{array}$$

We can visualize these constraints on the hypergraph itself. Figure 7-3(a) shows the full hypergraph with the source words indicated. In the invalid derivation in Figure 7-2(b), there were two edges indicating translations of `diese kritik`. This implies that that hyperpath $y \notin \mathcal{Y}$. Figure 7-3(b) shows optimal valid translation. It has score

$$w(\texttt{diese kritik}, \texttt{this criticism}) + w(\texttt{wir mussen}, \texttt{we must}) + w(\texttt{nehmen}, \texttt{take}) +$$
$$w(\texttt{ernst}, \texttt{seriously}) + g(\texttt{<s> seriously take we must this criticism</s>}) = 21$$

Enforcing the constraints is necessary to produce a valid translation of this form.

7.4 A Variant of the Beam Search Algorithm

The focus of this chapter is on finding the best translation from a constrained hypergraph like the one described in the previous section. The main tool will be a variant of the beam search algorithm for finding the highest-scoring constrained hyperpath. The algorithm uses three main techniques: (1) dynamic programming with additional signature information to satisfy the constraints, (2) beam pruning where some, possibly optimal, hypotheses are discarded, and (3) branch-and-bound-style application of upper and lower bounds to discard provably non-optimal hypotheses.

Any solution returned by the algorithm will be a valid constrained hyperpath and a member of $\mathcal{Y}$. Additionally the algorithm returns a certificate flag opt that, if true, indicates that no beam pruning was used, implying the solution returned is optimal. Generally it will be hard to produce a certificate even by reducing the amount of beam pruning; however in the next section we will introduce a method based on Lagrangian relaxation to tighten the upper bounds. These bounds will help eliminate most solutions before they trigger pruning.

Algorithm Algorithm 10 shows the complete beam search algorithm. At its core it is a dynamic programming algorithm filling in the chart π. The beam search chart indexes hypotheses by vertices $v \in \mathcal{V}$ as well as a signature $sig \in \mathbb{R}^p$ where p is the number of constraints. A new hypothesis is constructed from each hyperedge and all possible signatures of tail nodes. We define the function SIGS to take the tail of an edge and return the set of

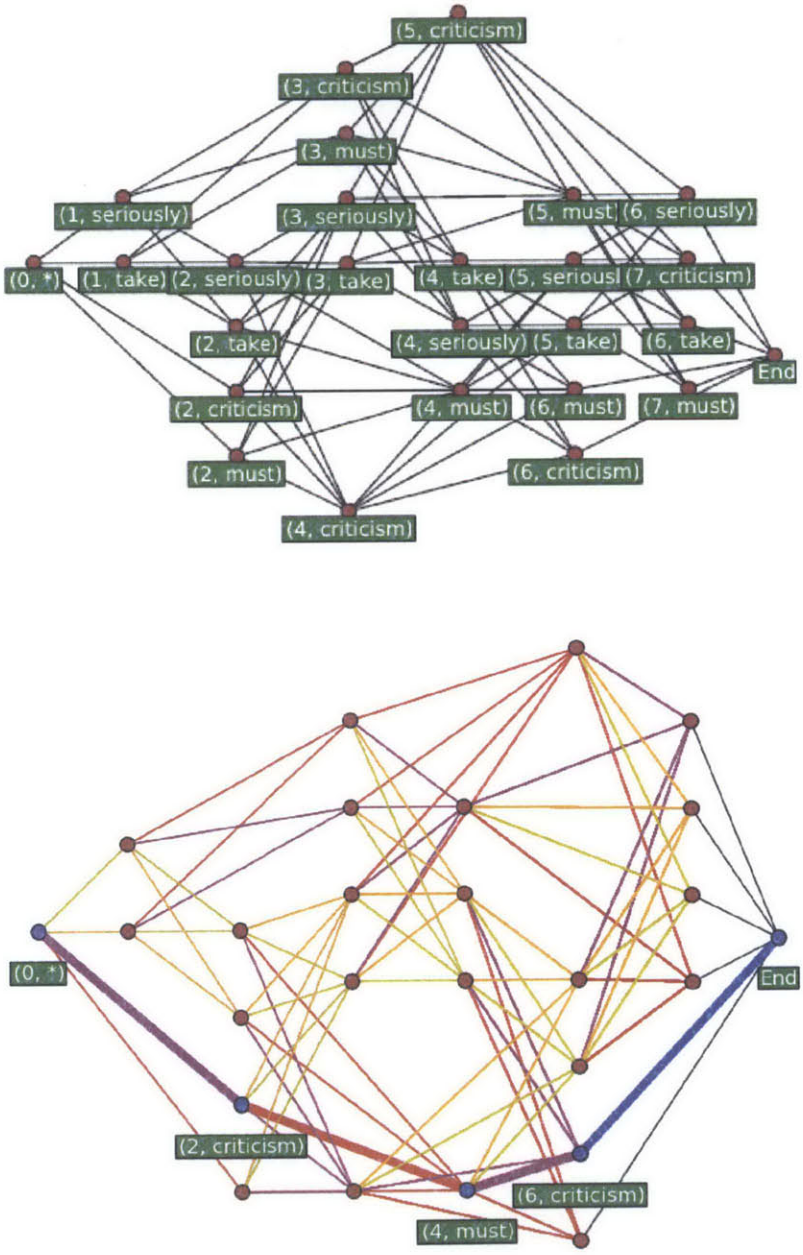


Figure 7-2: Two views of the phrase-based translation hypergraph. (a) Full unconstrained hypergraph for translation decoding with vertices labeled. (b) Best unconstrained path with edges colored by source words. Note that the purple edge is used more than once, and yellow is not used at all.

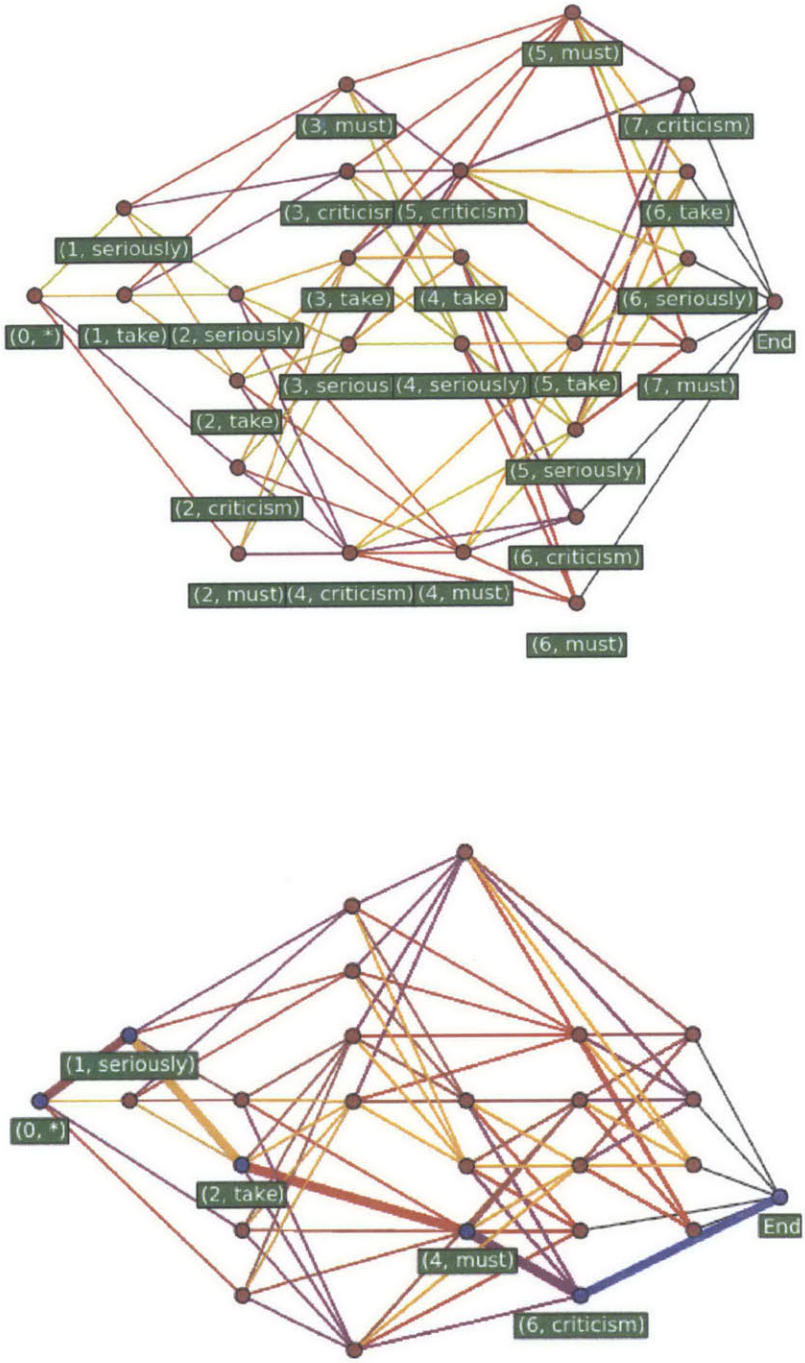


Figure 7-3: Two views of the constrained hypergraph. (a) Original hypergraph with labels and edges colored by source words. (b) The best path in the constrained hypergraph. Note all source words are used exactly once.

possible signature combinations

$$\textsc{Sigs}(v_2, \dots v_{|v|}) = \prod_{i=2}^{|v|} \{sig : \pi[v_i, sig] \neq -\infty\}$$

where the product is the Cartesian product over sets. Line 7 loops over this entire set.[1] For hypothesis y, the algorithm ensures that its signature sig is equal to Ay. This property is updated on line 8.

The signature provides proof that a hypothesis is still valid. Let the function $\textsc{Check}(sig)$ return true if the hypothesis can still fulfill the constraints. For example, in phrase-based decoding, we will define $\textsc{Check}(sig) = (sig \leq \mathbf{1})$; this ensures that each word has been translated 0 or 1 times. This check is applied on line 10.

Unfortunately maintaining all signatures is inefficient. For example we will see that in phrase-based decoding the signature is a bit-string recording which source words have been translated; the number of possible bit-strings is exponential in the length of the sentence. The algorithm includes two methods for removing hypotheses, bounding and pruning.

Bounding allows us to discard provably non-optimal solutions. The algorithm takes as arguments a lower bound on the optimal score $\text{lb} \leq \theta^\top y^* + \tau$, and computes upper bounds on the outside score for all vertices v: $\text{ubs}[v]$, i.e. an overestimate of the score for completing the hyperpath from v. If a hypothesis has score s, it can only be optimal if $s + \text{ubs}[v] \geq \text{lb}$. This bound check is performed on line 10.

Pruning removes weak partial solutions based on problem-specific checks. The algorithm invokes the black-box function, $\textsc{Prune}$, on line 12, passing it a pruning parameter β and a vertex-signature pair. The parameter β controls a threshold for pruning. For instance for phrase-based translation, it specifies a hard-limit on the number of hypotheses to retain. The function returns true if it prunes from the chart. Note that pruning may remove optimal hypotheses, so we set the certificate flag opt to false if the chart is modified.

[1] For simplicity we write this loop over the entire set. In practice it is important to use data structures to optimize look-up. See Tillmann (2006) and Huang and Chiang (2005).

1: **procedure** BEAMSEARCH(lb, ubs, β)
2: opt $\leftarrow$ true
3: $\pi[v, sig] \leftarrow -\infty$ for all $v \in \mathcal{V}, sig \in \mathcal{S}$
4: $\pi[v, 0] \leftarrow 0$ for all $v \in \mathcal{V}^{(t)}$
5: **for** $e \in \mathcal{E}$ in bottom-up order **do**
6: $\langle\langle v_2, \ldots, v_{|v|}\rangle, v_1\rangle \leftarrow e$
7: **for** $sig^{(2)} \ldots sig^{(|v|)} \in \text{SIGS}(\langle v_2, \ldots, v_{|v|}\rangle)$ **do**
8: $sig \leftarrow A\delta(e) + \sum_{i=2}^{|v|} sig^{(i)}$
9: $s \leftarrow \theta(e) + \sum_{i=2}^{|v|} \pi[v_i, sig^{(i)}]$
10: **if** $\begin{pmatrix} s > \pi[v_1, sig] \wedge \\ \text{CHECK}(sig) \wedge \\ s + \text{ubs}[v_1] \geq \text{lb} \end{pmatrix}$ **then**
11: $\pi[v_1, sig] \leftarrow s$
12: **if** PRUNE(π, v_1, sig, β) **then** opt $\leftarrow$ false
13: lb$' \leftarrow \pi[1, c]$
14: **return** lb$'$, opt

Input:

$(\mathcal{V}, \mathcal{E}, \theta)$	hypergraph with weights
(A, b)	matrix and vector for constraints
lb $\in \mathbb{R}$	lower bound
ubs $\in \mathbb{R}^{\mathcal{V}}$	upper bounds on outside scores
β	a pruning parameter

Output:

lb$'$	resulting lower bound score
opt	certificate of optimality

Algorithm 10: A variant of the beam search algorithm. Uses dynamic programming to produce a lower bound on the optimal constrained solution and, possibly, a certificate of optimality. Function OUTSIDE *computes upper bounds on outside scores. Function* SIGS *enumerates all possible tail signatures. Function* CHECK *identifies signatures that do not violate constraints. Bounds* lb *and* ubs *are used to remove provably non-optimal solutions. Function* PRUNE*, taking parameter β, returns true if it prunes hypotheses from π that could be optimal.*

This variant on beam search satisfies the following two properties (recall y^* is the optimal constrained solution):

Property 7.4.1 (Primal Feasibility). *The returned score* lb' *lower bounds the optimal constrained score, that is* $\text{lb}' \leq \theta^\top y^* + \tau$.

Property 7.4.2 (Dual Certificate). *If beam search returns with* $\text{opt} = \text{true}$*, then the returned score is optimal, i.e.* $\text{lb}' = \theta^\top y^* + \tau$.

An immediate consequence of Property 7.4.1 is that the output of beam search, lb', can be used as the input lb for future runs of the algorithm. Furthermore, if we loosen the amount of beam pruning by adjusting the pruning parameter β we can produce tighter lower bounds and discard more hypotheses. We can then iteratively apply this idea with a sequence of parameters $\beta_1 \ldots \beta_K$ producing lower bounds $\text{lb}^{(1)}$ through $\text{lb}^{(K)}$. We return to this idea in Section 7.6.

Example 7.4.1 (Phrase-based Beam Search). Recall that the constraints for phrase-based translation consist of a binary matrix $A \in \{0,1\}^{|s| \times |\mathcal{E}|}$ and vector $b = \mathbf{1}$. The value sig_i is therefore the number of times source word i has been translated in the hypothesis. We define the predicate CHECK as $\text{CHECK}(sig) = (sig \leq \mathbf{1})$ in order to remove hypotheses that already translate a source word more than once, and are therefore invalid. For this reason, phrase-based signatures are called *bit-strings*.

A common beam pruning strategy is to group together items into a set $\mathcal{C}$ and retain a (possibly complete) subset. An example phrase-based beam pruner is given in Figure 11. It groups together hypotheses based on $||sig_i||_1$, i.e. the number of source words translated, and applies a hard pruning filter that retains only the β highest-scoring items $(v, sig) \in \mathcal{C}$ based on $\pi[v, sig]$.

7.4.1 Computing Upper Bounds

Define the set $\mathcal{O}(v, y)$ to contain all outside edges of vertex v in hyperpath y (informally, hyperedges that do not have v as an ancestor). For all $v \in \mathcal{V}$, we set the upper bounds, ubs,

procedure PRUNE(π, v, sig, β)
  $\mathcal{B} \leftarrow \{(u, sig') : ||sig'||_1 = ||sig||_1,$
    $\pi[u, sig'] \neq -\infty\}$
  $\mathcal{P} \leftarrow \mathcal{B} \setminus \text{mBEST}(\beta, \mathcal{B}, \pi)$
  $\pi[u, sig'] \leftarrow -\infty$ for all $u, sig' \in \mathcal{P}$
  if $\mathcal{P} = \emptyset$ **then return** true
  else return false

Input: (v, sig) the last hypothesis added to the chart
$m \in Z^+$ # of hypotheses to retain

Output: true, if π is modified

Algorithm 11: Pruning function for phrase-based translation. Set $\mathcal{C}$ contains all hypotheses with $||sig||_1$ source words translated. The function prunes all but the top-β scoring hypotheses in this set.

to be the best unconstrained outside score

$$\text{ubs}[v] = \max_{y \in \mathcal{Y}' : v \in y} \sum_{e \in \mathcal{O}(v,y)} \theta(e) + \tau$$

This upper bound can be efficiently computed for all vertices using the standard outside dynamic programming algorithm. We will refer to this algorithm as OUTSIDE(θ, τ).

Unfortunately, as we will see, these upper bounds are often quite loose. The issue is that unconstrained outside paths are able to violate the constraints without being penalized, and therefore greatly overestimate the score.

7.5 Finding Tighter Bounds with Lagrangian Relaxation

Beam search produces a certificate only if beam pruning is never used. In the case of phrase-based translation, the certificate is dependent on all groups $\mathcal{C}$ having β or less hypotheses. The only way to ensure this is to bound out enough hypotheses to avoid pruning. The effectiveness of the bounding inequality, $s + \text{ubs}[v] < \text{lb}$, in removing hypotheses is directly dependent on the tightness of the bounds.

In this section we propose using Lagrangian relaxation to improve these bounds. We first give a brief overview of the method and then apply it to computing bounds. Our experiments show that this approach is very effective at finding certificates.

7.5.1 Algorithm

In Lagrangian relaxation, instead of solving the constrained search problem, we relax the constraints and solve an unconstrained hypergraph problem with modified weights. Recall the constrained hypergraph problem: $\max_{y \in \mathcal{Y}' : Ay = b} \theta^\top y + \tau$. The Lagrangian dual of this optimization problem is

$$\begin{aligned} L(\lambda) &= \max_{y \in \mathcal{Y}'} \theta^\top y + \tau - \lambda^\top (Ay - b) \\ &= \left(\max_{y \in \mathcal{Y}'} (\theta - A^\top \lambda)^\top y \right) + \tau + \lambda^\top b \\ &= \max_{y \in \mathcal{Y}'} \theta'^\top y + \tau' \end{aligned}$$

where $\lambda \in R^{|b|}$ is a vector of dual variables and define $\theta' = \theta - A^\top \lambda$ and $\tau' = \tau + \lambda^\top b$. This maximization is over $\mathcal{Y}'$, so for any value of λ, $L(\lambda)$ can be calculated as `BestPathScore`(θ', τ').

Note that for all valid constrained hyperpaths $y \in \mathcal{Y}$ the term $Ax - b$ equals 0, which implies that these hyperpaths have the same score under the modified weights as under the original weights, $\theta^\top y + \tau = \theta'^\top y + \tau'$. This leads to the following two properties, where $y \in \mathcal{Y}'$ is the hyperpath computed within the max:

Property 7.5.1 (Dual Feasibility). *The value $L(\lambda)$ upper bounds the optimal solution, that is $L(\lambda) \geq \theta^\top y^* + \tau$*

Property 7.5.2 (Primal Certificate). *If the hyperpath y is a member of $\mathcal{Y}$, i.e. $Ay = b$, then $L(\lambda) = \theta^\top y^* + \tau$.*

Property 7.5.1 states that $L(\lambda)$ always produces some upper bound; however, to help beam search, we want as tight a bound as possible: $\min_\lambda L(\lambda)$.

The Lagrangian relaxation algorithm, shown in Algorithm 12, uses subgradient descent to find this minimum. The subgradient of $L(\lambda)$ is $Ay - b$ where y is the argmax of the modified objective $y = \arg\max_{y \in \mathcal{Y}'} \theta'^\top y + \tau'$. Subgradient descent iteratively solves unconstrained hypergraph search problems to compute these subgradients and updates λ. See the background chapter on Lagrangian relaxation for an extensive discussion of this style of optimization in natural language processing.

procedure LRROUND(α_k, λ)
  $y \leftarrow \arg\max_{y \in \mathcal{Y}'} \theta^\top y + \tau - \lambda^\top (Ay - b)$
  $\lambda' \leftarrow \lambda - \alpha_k (Ay - b)$
  opt $\leftarrow Ay = b$
  ub $\leftarrow \theta^\top y + \tau$
  return λ', ub, opt
procedure LAGRANGIANRELAXATION(α)
  $\lambda^{(0)} \leftarrow 0$
  ub* $\leftarrow \infty$
  for k in $1 \ldots K$ **do**
    $\lambda^{(k)}$, ub, opt $\leftarrow$ LRROUND($\alpha_k, \lambda^{(k-1)}$)
    if opt **then return** $\lambda^{(k)}$, ub, opt
    ub* $\leftarrow \min\{$ub, ub$^*\}$
  return $\lambda^{(K)}$, ub*, opt

Input: $\alpha_1 \ldots \alpha_K$ sequence of subgradient rates

Output:
- λ final dual vector
- ub upper bound on optimal constrained solution
- opt certificate of optimality

Algorithm 12: Lagrangian relaxation algorithm. The algorithm repeatedly calls LRROUND *to compute the subgradient, update the dual vector, and check for a certificate.*

Example: Phrase-based Relaxation. For phrase-based translation, we expand out the Lagrangian to

$$
\begin{aligned}
L(\lambda) &= \max_{y \in \mathcal{Y}'} \theta^\top y + \tau - \lambda^\top (Ay - b) = \\
&\max_{y \in \mathcal{Y}'} \sum_{e \in \mathcal{E}} \left(\theta(e) - \sum_{i=j(p(e))}^{k(p(e))} \lambda_i \right) y(e) + \tau + \sum_{i=1}^{|s|} \lambda_i
\end{aligned}
$$

The weight of each edge $\theta(e)$ is modified by the dual variables λ_i for each source word translated by the edge, i.e. if $(q, r, j, k) = p(e)$, then the score is modified by $\sum_{i=j}^{k} \lambda_i$. A solution under these weights may use source words multiple times or not at all. However if the solution uses each source word exactly once ($Ay = \mathbf{1}$), then we have a certificate and the solution is optimal.

7.5.2 Utilizing Upper Bounds in Beam Search

For many problems, it may not be possible to satisfy Property 4.2 by running the subgradient algorithm alone. Yet even for these problems, applying subgradient descent will produce an improved estimate of the upper bound, $\min_\lambda L(\lambda)$.

To utilize these improved bounds, we simply replace the weights in beam search and the outside algorithm with the modified weights from Lagrangian relaxation, θ' and τ'. Since the result of beam search must be a valid constrained hyperpath $y \in \mathcal{Y}$, and for all $y \in \mathcal{Y}$, $\theta^\top y + \tau = \theta'^\top y + \tau'$, this substitution does not alter the necessary properties of the algorithm; i.e. if the algorithm returns with opt equal to true, then the solution is optimal.

Additionally the computation of upper bounds now becomes

$$
\text{ubs}[v] = \max_{y \in \mathcal{Y}' : v \in y} \sum_{e \in \mathcal{O}(v,y)} \theta'(e) + \tau'
$$

These outside paths may still violate constraints, but the modified weights now include penalty terms to discourage common violations.

7.6 Optimal Beam Search

The optimality of the beam search algorithm is dependent on the tightness of the upper and lower bounds. We can produce better lower bounds by varying the pruning parameter β; we can produce better upper bounds by running Lagrangian relaxation. In this section we combine these two ideas and present a complete optimal beam search algorithm.

Our general strategy will be to use Lagrangian relaxation to compute modified weights and to use beam search over these modified weights to attempt to find an optimal solution. One simple method for doing this, shown at the top of Algorithm 13, is to run in stages. The algorithm first runs Lagrangian relaxation to compute a λ vector that minimizes $L(\lambda)$. The algorithm then iteratively runs beam search using the parameter sequence β_k. These parameters allow the algorithm to loosen the amount of beam pruning. For example in phrase based pruning, we would raise the number of hypotheses stored per group until no beam pruning occurs.

A clear disadvantage of the staged approach is that it needs to wait until Lagrangian relaxation is completed before even running beam search. Often beam search will be able to quickly find an optimal solution even with good but non-optimal λ. In other cases, beam search may still improve the lower bound lb.

This motivates the alternating algorithm OptBeam shown in Algorithm 13. In each round, the algorithm alternates between computing subgradients to tighten ubs and running beam search to maximize lb. In early rounds we set β for aggressive beam pruning, and as the upper bounds get tighter, we loosen pruning to try to get a certificate. If at any point either a primal or dual certificate is found, the following theorem holds:

Theorem 7.6.1 (Certificate of Optimality). *At any round k, if Lagrangian relaxation produces $y^{(k)}$ such that $Ay^{(k)} = b$, then $\text{ub}^{(k)}$ is optimal*

$$\max_{y \in \mathcal{Y}} \theta^\top y + \tau = \text{ub}^{(k)}$$

Alternatively if at round k, beam search returns with opt = true, *then* $\text{lb}^{(k)}$ *is optimal*

$$\max_{y \in \mathcal{Y}} \theta^\top y + \tau = \text{lb}^{(k)}$$

Proof: This theorem is a combination of Property 7.4.2 and Property 7.5.2.

procedure OPTBEAMSTAGED(α, β)
  $\lambda, \text{ub}, \text{opt} \leftarrow$LAGRANGIANRELAXATION($\alpha$)
  if opt **then return** ub
  $\text{ubs} \leftarrow$ DUALOUTSIDE(λ)
  $\text{lb}^{(0)} \leftarrow -\infty$
  for k in $1 \ldots K$ **do**
    $\text{lb}^{(k)}, \text{opt} \leftarrow$ BEAMSEARCH($\text{ubs}, \text{lb}^{(k-1)}, \beta_k$)
    if opt **then return** $\text{lb}^{(k)}$
  return $\max_{k \in \{1 \ldots K\}} \text{lb}^{(k)}$

procedure OPTBEAM(α, β)
  $\lambda^{(0)} \leftarrow 0$
  $\text{lb}^{(0)} \leftarrow -\infty$
  for k in $1 \ldots K$ **do**
    $\lambda^{(k)}, \text{ub}^{(k)}, \text{opt} \leftarrow$ LRROUND($\alpha_k, \lambda^{(k-1)}$)
    if opt **then return** $\text{ub}^{(k)}$
    $\text{ubs}^{(k)} \leftarrow$ DUALOUTSIDE($\lambda^{(k)}$)
    $\text{lb}^{(k)}, \text{opt} \leftarrow$ BEAMSEARCH($\text{ubs}^{(k)}, \text{lb}^{(k-1)}, \beta_k$)
    if opt **then return** $\text{lb}^{(k)}$
  return $\max_{k \in \{1 \ldots K\}} \text{lb}^{(k)}$

Input: $\alpha_1 \ldots \alpha_K$ sequence of subgradient rates
$\beta_1 \ldots \beta_K$ sequence of pruning parameters
Output: optimal constrained score or lower bound

Algorithm 13: Two versions of optimal beam search: staged and alternating. Staged runs Lagrangian relaxation to find the optimal λ, uses λ to compute upper bounds, and then repeatedly runs beam search with pruning sequence $\beta_1 \ldots \beta_k$. Alternating switches between running a round of Lagrangian relaxation and a round of beam search with the updated λ. If either produces a certificate it returns the result.

7.7 Experiments

To evaluate the effectiveness of optimal beam search for translation decoding, we implemented decoders for phrase- and syntax-based models. In this section we compare the speed and optimality of these decoders to several baseline methods.

7.7.1 Setup and Implementation

For phrase-based translation we used a German-to-English data set taken from Europarl (Koehn, 2005). We tested on 1,824 sentences of length at most 50 words. For experiments the phrase-based systems uses a trigram language model and includes standard distortion penalties. Additionally the unconstrained hypergraph includes further derivation information similar to the graph described in Chang and Collins (2011).

For syntax-based translation we used a Chinese-to-English data set. The model and hypergraphs come from the work of Huang and Mi (2010). We tested on 691 sentences from the newswire portion of the 2008 NIST MT evaluation test set. For experiments, the syntax-based model uses a trigram language model. The translation model is tree-to-string syntax-based model with a standard context-free translation forest. The constraint matrix A is based on the constraints described by Rush and Collins (2011).

Our decoders use a two-pass architecture. The first pass sets up the hypergraph in memory, and the second pass runs search. When possible the baselines share optimized construction and search code.

The performance of optimal beam search is dependent on the sequences α and β. For the step-size α we used a variant of Polyak's rule (Polyak, 1987; Boyd & Mutapcic, 2007a), substituting the unknown optimal score for the last computed lower bound: $\alpha_{k+1} \leftarrow \frac{\text{ub}^{(k)} - \text{lb}^{(k)}}{||Ay^{(k)} - b||_2^2}$. We adjust the order of the pruning parameter β based on a function μ of the current gap: $\beta_{k+1} \leftarrow 10^{\mu(\text{ub}^{(k)} - \text{lb}^{(k)})}$ where β is the maximum number of hypotheses kept in each set during beam pruning.

Previous work on these data sets has shown that exact algorithms do not result in a significant increase in translation accuracy. We focus on the efficiency and model score of

	11-20 (558)			21-30 (566)			31-40 (347)			41-50 (168)			all (1824)		
Phrase-Based	time	cert	exact	time	cert	exact	time	cert	exact	time	cert	exact	time	cert	exact
BEAM (100)	2.33	19.5	38.0	8.37	1.6	7.2	24.12	0.3	1.4	71.35	0.0	0.0	14.50	15.3	23.2
BEAM (1000)	2.33	37.8	66.3	8.42	3.4	18.9	21.60	0.6	3.2	53.99	0.6	1.2	12.44	22.6	36.9
BEAM (100000)	3.34	83.9	96.2	18.53	22.4	60.4	46.65	2.0	18.1	83.53	1.2	6.5	23.39	43.2	62.4
MOSES (100)	0.18	0.0	81.0	0.36	0.0	45.6	0.53	0.0	14.1	0.74	0.0	6.0	0.34	0.0	52.3
MOSES (1000)	2.29	0.0	97.8	4.39	0.0	78.8	6.52	0.0	43.5	9.00	0.0	19.6	4.20	0.0	74.6
ASTAR (cap)	11.11	99.3	99.3	91.39	53.9	53.9	122.67	7.8	7.8	139.61	1.2	1.2	67.99	58.8	58.8
LR-TIGHT	4.20	100.0	100.0	23.25	100.0	100.0	88.16	99.7	99.7	377.9	97.0	97.0	60.11	99.7	99.7
OPTBEAM	2.85	100.0	100.0	10.33	100.0	100.0	28.29	100.0	100.0	84.34	97.0	97.0	17.27	99.7	99.7
ChangCollins	10.90	100.0	100.0	57.20	100.0	100.0	203.4	99.7	99.7	679.9	97.0	97.0	120.9	99.7	99.7
MOSES-GC (100)	0.14	0.0	89.4	0.27	0.0	84.1	0.41	0.0	75.8	0.58	0.0	78.6	0.26	0.0	84.9
MOSES-GC (1000)	1.33	0.0	89.4	2.62	0.0	84.3	4.15	0.0	75.8	6.19	0.0	79.2	2.61	0.0	85.0

	11-20 (192)			21-30 (159)			31-40 (136)			41-100 (123)			all (691)		
Syntax-Based	time	cert	exact	time	cert	exact	time	cert	exact	time	cert	exact	time	cert	exact
BEAM (100)	0.40	4.7	75.9	0.40	0.0	66.0	0.75	0.0	43.4	1.66	0.0	25.8	0.68	5.72	58.7
BEAM (1000)	0.78	16.9	79.4	2.65	0.6	67.1	6.20	0.0	47.5	15.5	0.0	36.4	4.16	12.5	65.5
CUBE (100)	0.08	0.0	77.6	0.16	0.0	66.7	0.23	0.0	43.9	0.41	0.0	26.3	0.19	0.0	59.0
CUBE (1000)	1.76	0.0	91.7	4.06	0.0	95.0	5.71	0.0	82.9	10.69	0.0	60.9	4.66	0.0	85.0
LR-TIGHT	0.37	100.0	100.0	1.76	100.0	100.0	4.79	100.0	100.0	30.85	94.5	94.5	7.25	99.0	99.0
OPTBEAM	0.23	100.0	100.0	0.50	100.0	100.0	1.42	100.0	100.0	7.14	93.6	93.6	1.75	98.8	98.8
ILP	9.15	100.0	100.0	32.35	100.0	100.0	49.6	100.0	100.0	108.6	100.0	100.0	40.1	100.0	100.0

Table 7.1: Experimental results for translation experiments. Column time is the mean time per sentence in seconds, cert is the percentage of sentences solved with a certificate of optimality, exact is the percentage of sentences solved exactly, i.e. $\theta^\top y + \tau = \theta^\top y^* + \tau$. *Results are grouped by sentence length (group 1-10 is omitted).*

the algorithms.

7.7.2 Baseline Methods

The experiments compare optimal beam search (OPTBEAM) to several different decoding methods. For both systems we compare to: BEAM, the beam search decoder from Algorithm 10 using the original weights θ and τ, and $\beta \in \{100, 1000\}$; LR-TIGHT, Lagrangian relaxation followed by incremental tightening constraints, which is a reimplementation of Chang and Collins (2011) and Rush and Collins (2011).

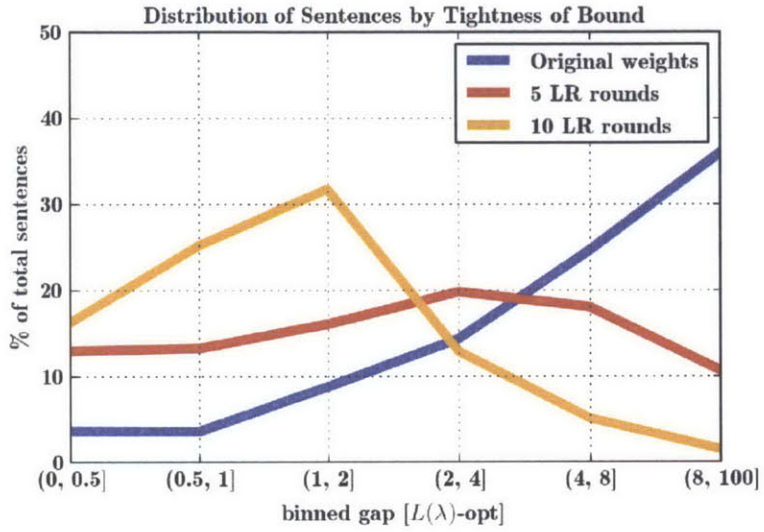
For phrase-based translation we compare with: MOSES-GC, the standard Moses beam search decoder with $\beta \in \{100, 1000\}$ (Koehn et al., 2007); MOSES, a version of Moses without gap constraints more similar to BEAM (see Chang and Collins (2011)); ASTAR, an implementation of A* search using original outside scores, i.e. OUTSIDE(θ, τ), and capped at 20,000,000 queue pops.

For syntax-based translation we compare with: ILP, a general-purpose integer linear programming solver (Gurobi Optimization, 2013) and CUBEPRUNING, an approximate decoding method similar to beam search (Chiang, 2007), tested with $\beta \in \{100, 1000\}$.

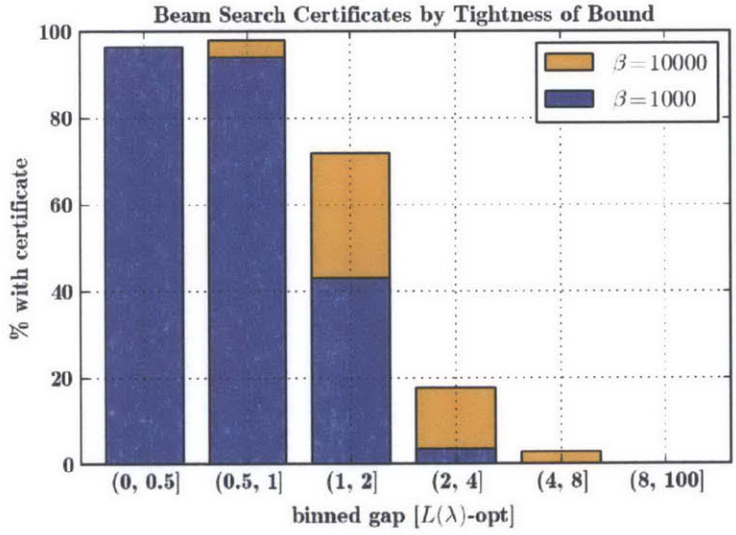
7.7.3 Experiments

Table 7.1 shows the main results. For phrase-based translation, OPTBEAM decodes the optimal translation with certificates in 99% of sentences with an average time of 17.27 seconds per sentence. This is seven times faster than the decoder of Chang and Collins (2011) and 3.5 times faster then our reimplementation, LR-TIGHT. ASTAR performs poorly, taking lots of time on difficult sentences. BEAM runs quickly, but rarely finds an exact solution. MOSES without gap constraints is also fast, but less exact than OPTBEAM and unable to produce certificates.

For syntax-based translation. OPTBEAM finds a certificate on 98.8% of solutions with an average time of 1.75 seconds per sentence, and is four times faster than LR-TIGHT. CUBE (100) is an order of magnitude faster, but is rarely exact on longer sentences. CUBE



(a)



(b)

Figure 7-4: Two graphs from phrase-based decoding. Graph (a) shows the duality gap distribution for 1,824 sentences after 0, 5, and 10 rounds of LR. Graph (b) shows the % of certificates found for sentences with differing gap sizes and beam search parameters β. Duality gap is defined as, ub - $(\theta^\top y^* + \tau)$.

		≥ 30		all	
		mean	median	mean	median
PB	Hypergraph	56.6%	69.8%	59.6%	69.6%
	Lag. Relaxation	10.0%	5.5%	9.4%	7.6%
	Beam Search	33.4%	24.6%	30.9%	22.8%
SB	Hypergraph	0.5%	1.6%	0.8%	2.4%
	Lag. Relaxation	15.0%	35.2%	17.3%	41.4%
	Beam Search	84.4%	63.1%	81.9 %	56.1%

Table 7.2: Distribution of time within optimal beam search, including: hypergraph construction, Lagrangian relaxation, and beam search. Mean is the percentage of total time. Median is the distribution over the median values for each row.

(1000) finds more exact solutions, but is comparable in speed to optimal beam search. BEAM performs better than in the phrase-based model, but is not much faster than OPTBEAM.

Figure 7-4 shows the relationship between beam search optimality and duality gap. Graph (a) shows how a handful of LR rounds can significantly tighten the upper bound score of many sentences. Graph (b) shows how beam search is more likely to find optimal solutions with tighter bounds. BEAM effectively uses 0 rounds of LR, which may explain why it finds so few optimal solutions compared to OPTBEAM.

Table 7.2 breaks down the time spent in each part of the algorithm. For both methods, beam search has the most time variance and uses more time on longer sentences. For phrase-based sentences, Lagrangian relaxation is fast, and hypergraph construction dominates. If not for this cost, OPTBEAM might be comparable in speed to MOSES (1000).

7.8 Conclusion

This chapter focuses on a standard heuristic algorithm used for translation known as beam search. We use Lagrangian relaxation to develop an optimal variant of beam search and apply it to machine translation decoding. The algorithm uses beam search to produce constrained solutions and bounds from Lagrangian relaxation to eliminate non-optimal solutions. Results show that this method can efficiently find exact solutions for two important styles of machine translation.

Unlike the previous chapters in this thesis which used Lagrangian relaxation alone or

in combination with constraint generation to search for a certificates, this chapter instead directly uses the Lagrange multipliers produced by the method. The multipliers are used to provide upper bounds that help to bound out bad hypotheses within a combinatorial framework. This style of search algorithm shows the general applicability of relaxation methods.

Chapter 8

Approximate Dependency Parsing

[This chapter is adapted from joint work with Slav Petrov entitled "Vine pruning for Efficient Multi-Pass Dependency Parsing" (Rush & Petrov, 2012)]

To this point the decoding problems we have considered have been NP-Hard or have had very poor worse-case running times. In practice though, efficiency is an issue even for relatively fast algorithms, particularly when they need to run on large sets of textual data. How can similar relaxation methods help boost performance for this class of problem?

To look at this question, we return to the decoding problem for dependency parsing. In Chapter 6 we looked at the problem of higher-order, non-projective dependency parsing. This is known to be NP-Hard, and so it was challenging to produce optimal solutions at all. The algorithm produced was relatively fast, and was able to parse dozens of sentences per second.

In this chapter, we consider higher-order *projective* parsing. Unlike non-projective parsing, this decoding problem is known to be solvable in polynomial-time. The challenge is that in practice it is often desirable to solve it even faster than an exact dynamic programming approach; some algorithms are able to process hundreds of sentences per second. Unfortunately, even just computing scores under a simple objective function f, e.g. a first-order scoring function, often is often too slow in practice, which hinders the speed of exact methods.

Instead we consider an approximate method known as coarse-to-fine parsing. At the

core of this algorithm is a relaxation of dependency parsing known as vine parsing. This technique will enables us to both use high-accuracy parsing models and approximately solve the decoding problem efficiently.

8.1 Coarse-to-Fine Parsing

Coarse-to-fine decoding has been extensively used to speed up structured prediction models. The general idea is simple: use a coarse model where decoding is cheap to prune the search space for more complex models. In this chapter, we present a multi-pass coarse-to-fine architecture for projective dependency parsing. We start with a linear-time vine pruning pass and build up to higher-order models, achieving speed-ups of two orders of magnitude while maintaining state-of-the-art accuracies.

In constituency parsing, exhaustive decoding for all but the simplest grammars tends to be prohibitively slow. Consequently, most high-accuracy constituency parsers routinely employ a coarse grammar to prune dynamic programming chart cells of the final grammar of interest (Charniak et al., 2006; Carreras et al., 2008; Petrov, 2009). While there are no strong theoretical guarantees for these approaches,[1] in practice one can obtain significant speed improvements with minimal loss in accuracy. This benefit comes primarily from reducing the large grammar constant $|\mathcal{G}|$ that can dominate the runtime of the cubic-time CKY decoding algorithm. Dependency parsers on the other hand do not have a multiplicative grammar factor $|\mathcal{G}|$, and until recently were considered efficient enough for exhaustive decoding. However, the increased model complexity of a third-order parser forced Koo and Collins (2010) to prune with a first-order model in order to make decoding practical. While fairly effective, all these approaches are limited by the fact that decoding in the coarse model remains cubic in the sentence length. The desire to parse vast amounts of text necessitates more efficient dependency parsing algorithms.

[1] This is in contrast to optimality preserving methods such as A* search, which typically do not provide sufficient speed-ups (Pauls & Klein, 2009).

8.2 Motivation & Overview

The goal of this chapter is fast, projective dependency parsing. Previous work on constituency parsing demonstrates that performing several passes with increasingly more complex models results in faster decoding (Charniak et al., 2006; Petrov & Klein, 2007). The same technique applies to dependency parsing with a cascade of models of increasing order; however, this strategy is limited by the speed of the simplest model. The commonly-used algorithm for first-order dependency parsing (Eisner, 2000) already requires $O(n^3)$ time. Lee (2002) shows that binary matrix multiplication gives a lower bound for the speed of exhaustively parsing context-free grammars, which limits the potential of these approaches.

We thus need to leverage domain knowledge to obtain faster parsing algorithms. It is well-known that natural language is fairly linear, and most head-modifier dependencies tend to be short. This property is exploited by transition-based dependency parsers (Yamada & Matsumoto, 2003; Nivre et al., 2004) and empirically demonstrated in Figure 8-1. The heat map on the left shows that most of the probability mass of modifiers is concentrated among nearby words, corresponding to a diagonal band in the matrix representation. On the right we show the frequency of arc lengths for different modifier part-of-speech tags. As one can expect, almost all arcs involving adjectives (ADJ) are very short (length 3 or less), but even arcs involving verbs and nouns are often short. This structure suggests that it may be possible to disambiguate most dependencies by considering only the "banded" portion of the sentence.

We exploit this linear structure by employing a variant of vine parsing (Eisner & Smith, 2005). Vine parsing is a dependency parsing algorithm that considers only close words as modifiers. Because of this assumption it runs in linear time.[2] Of course, any parse tree with hard limits on dependency lengths will contain major parse errors. We therefore use the vine parser only for pruning and augment it to allow arcs to remain unspecified (by including so called *outer* arcs). The vine parser can thereby eliminate a possibly quadratic number

[2]Vine parsers are as expressive as finite-state automata. This allows them to circumvent the cubic-time bound.

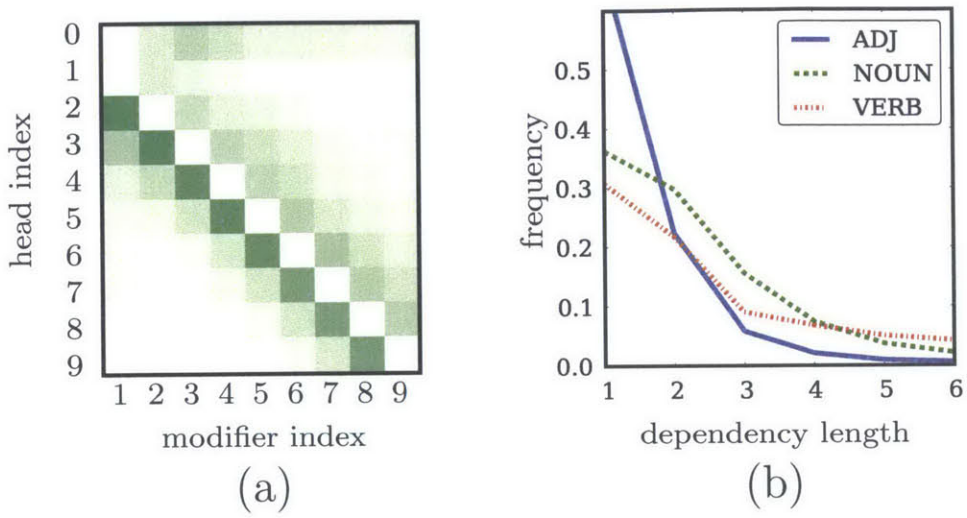


Figure 8-1: (a) Heat map indicating how likely a particular head position is for each modifier position. Greener/darker is higher probability. (b) Arc length frequency for three common modifier tags. Both charts are computed from all sentences in Section 22 of the PTB.

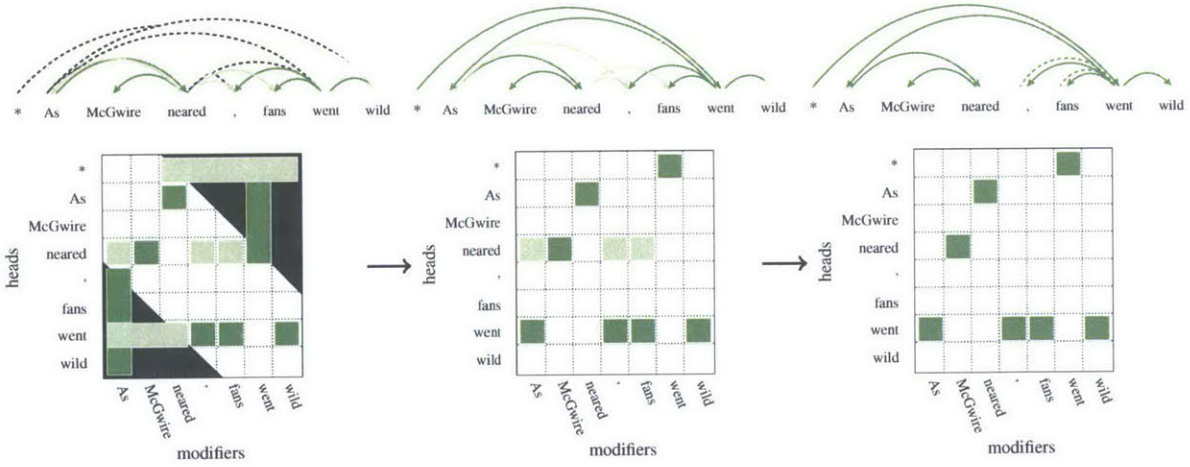


Figure 8-2: Multi-pass pruning with a vine, first-order, and second-order model shown as dependencies and filtered index sets after each pass. Darker cells have higher max-marginal values, while empty cells represent pruned arcs.

of arcs, while having the flexibility to defer some decisions and preserve ambiguity to be resolved by later passes. In Figure 8-2 for example, the vine pass correctly determined the head-word of *McGwire* as *neared*, limited the head-word candidates for *fans* to *neared* and *went*, and decided that the head-word for *went* falls outside the band by proposing an outer arc. A subsequent first-order pass needs to score only a small fraction of all possible arcs and can be used to further restrict the search space for the following higher-order passes.

8.3 Projective Dependency Parsing

We now briefly review projective dependency parsing and Eisner's algorithm. For a more complete overview of projectivity and the underlying dynamic programming algorithm see the background chapter.

Dependency parsing models factor all valid parse trees for a given sentence into smaller units, which can be scored independently. For instance, in a first-order factorization, the units are just dependency arcs. We represent these units by an index set $\mathcal{I}$ and use binary vectors $\mathcal{Y} \subset \{0,1\}^{\mathcal{I}}$ to specify a parse tree $y \in \mathcal{Y}$ such that $y(i) = 1$ iff the index i exists in the tree. The index sets of higher-order models can be constructed out of the index sets of lower-order models, thus forming a hierarchy that we will exploit in our coarse-to-fine cascade.

The decoding problem is to find the 1-best parse tree

$$\arg\max_{y \in \mathcal{Y}} \theta^\top y$$

where $\theta \in \mathbb{R}^{\mathcal{I}}$ is a weight vector that assigns a score to each index i (we discuss how θ is learned in Section 8.4). A generalization of the 1-best decoding problem is to find the max-marginal score for each index i. Max-marginals are given by the function $\mu : \mathcal{I} \mapsto \mathcal{Y}$ defined as

$$\mu(i) = \arg\max_{y \in \mathcal{Y}: y(i)=1} \theta^\top y$$

For first-order parsing, this corresponds to the best parse utilizing a given dependency arc. Clearly there are exponentially many possible parse tree structures, but fortunately there exist well-known dynamic programming algorithms for searching over all possible structures. We review these below, starting with the first-order factorization for ease of exposition.

8.3.1 First-Order Parsing

The simplest way to index a dependency parse structure is by the individual arcs of the parse tree. This model is known as first-order or arc-factored. For a sentence of length n the index set is:

$$\mathcal{I}^{(1)} = \{(h, m) : h \in \{0 \dots n\}, m \in \{1 \dots n\}\}$$

Each dependency tree has $y(h, m) = 1$ iff it includes an arc from head h to modifier m. We follow common practice and use position 0 as the root ($*$) of the sentence. The full set $\mathcal{I}^{(1)}$ has size $|\mathcal{I}^{(1)}| = O(n^2)$.

As we have seen, the first-order bilexical parsing algorithm of Eisner (2000) can be used to find the best parse tree and max-marginals. The algorithm defines a dynamic program over two types of items: *incomplete* items $I(h, m)$ that denote the span between a modifier m and its head h, and *complete* items $C(h, e)$ that contain a full subtree spanning from the head h and to the word e on one side. The algorithm builds larger items by applying the composition rules shown in Figure 8-3. Rule 8-3(a) builds an incomplete item $I(h, m)$ by attaching m as a modifier to h. This rule has the effect that $y(h, m) = 1$ in the final parse. Rule 8-3(b) completes item $I(h, m)$ by attaching item $C(m, e)$. The existence of $I(h, m)$ implies that m modifies h, so this rule enforces that the constituents of m are also constituents of h.

We can find the best derivation for each item by adapting the standard CKY parsing algorithm to these rules. Since both rule types contain three variables that can range over the entire sentence ($h, m, e \in \{0 \dots n\}$), the bottom-up, inside dynamic programming algorithm requires $O(n^3)$ time. Furthermore, we can find max-marginals with an additional top-down outside pass also requiring cubic time. (The full derivation for these algorithms is given in

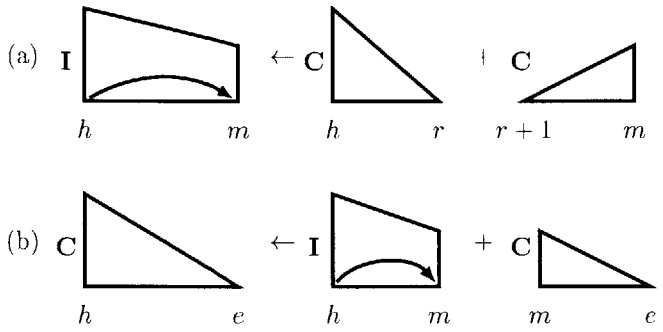


Figure 8-3: Parsing rules for first-order dependency parsing. The complete items C are represented by triangles and the incomplete items I are represented by trapezoids. Symmetric left-facing versions are also included.

background Section 2.2.4.)

To speed up search, we will consider filtering indices from $\mathcal{I}^{(1)}$ and reducing possible applications of Rule 8-3(a).

8.3.2 Higher-Order Parsing

Higher-order models generalize the index set by using siblings s (modifiers that previously attached to a head word) and grandparents t (head words above the current head word). For compactness, we use t_1 for the head word and s_{k+1} for the modifier and parameterize the index set to capture arbitrary higher-order decisions in both directions:

$$\mathcal{I}^{(k,l)} = \left\{(t,s) : t \in (\texttt{NULL} \cup \{0 \dots n\})^{l+1}, s \in (\texttt{NULL} \cup \{1 \dots n\})^{k+1}\right\}$$

where $k+1$ is the sibling order, $l+1$ is the parent order, and $k+l+1$ is the model order. The canonical second-order model uses $\mathcal{I}^{(1,0)}$, which has a cardinality of $O(n^3)$. Although there are several possibilities for higher-order models, we use $\mathcal{I}^{(1,1)}$ as our third-order model. Generally, the parsing index set has cardinality $|\mathcal{I}^{(k,l)}| = O(n^{2+k+l})$.

Example 8.3.1 (Index Set). Consider a parsing the example sentence $\texttt{As}_1$ $\texttt{McGwire}_2$ $\texttt{neared}_3$ $\texttt{,}_4$ $\texttt{fans}_5$ $\texttt{went}_6$ $\texttt{wild}_7$. One parse structure is shown in Figure 8-5. We will call this sentence y,

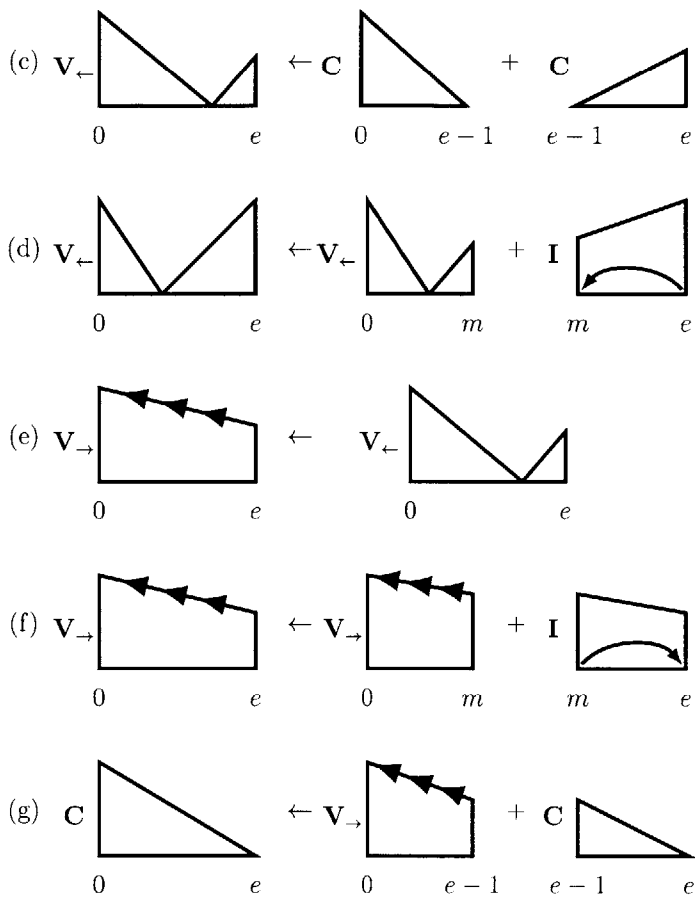


Figure 8-4: Additional rules for vine parsing. Vine left ($V_{\leftarrow}$) items are pictured as right-facing triangles and vine right ($V_{\rightarrow}$) items are marked trapezoids. Each new item is anchored at the root and grows to the right.

If we are using a first-order index set $\mathcal{I}^{(0,0)}$, the following indices are set:

$$y(\langle 0\rangle, 6\rangle) = 1, \quad y(\langle 6\rangle, \langle 5\rangle) = 1, \quad y(\langle 6\rangle, \langle 4\rangle) = 1,$$
$$y(\langle 6\rangle, \langle 1\rangle) = 1, \quad y(\langle 6\rangle, \langle 7\rangle) = 1, \quad y(\langle 1\rangle, \langle 3\rangle) = 1,$$
$$y(\langle 3\rangle, \langle 2\rangle) = 1$$

If we are using a third-order index set $\mathcal{I}^{(1,1)}$, the following indices are set:

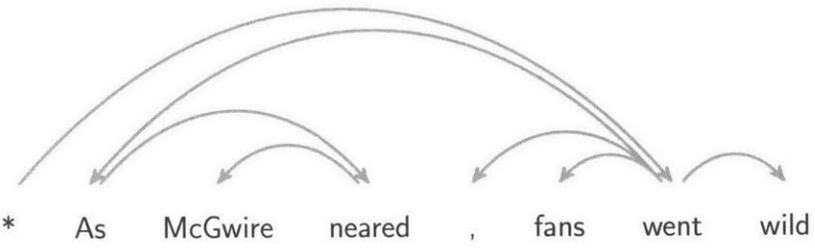


Figure 8-5: An example parse structure over the sentence $\texttt{As}_1$ $\texttt{McGwire}_2$ $\texttt{neared}_3$ $,_4$ $\texttt{fans}_5$ $\texttt{went}_6$ $\texttt{wild}_7$.

$$y(\langle \texttt{NULL}, 0\rangle, \langle \texttt{NULL}, 6\rangle) = 1, \quad y(\langle 0, 6\rangle, \langle \texttt{NULL}, 5\rangle) = 1, \quad y(\langle 0, 6\rangle, \langle 5, 4\rangle) = 1,$$
$$y(\langle 0, 6\rangle, \langle 4, 1\rangle) = 1, \quad y(\langle 0, 6\rangle, \langle \texttt{NULL}, 7\rangle) = 1, \quad y(\langle 6, 1\rangle, \langle \texttt{NULL}, 3\rangle) = 1,$$
$$y(\langle 1, 3\rangle, \langle \texttt{NULL}, 2\rangle) = 1,$$

Decoding in higher-order (projective) models uses variants of the dynamic program for first-order parsing, and we refer to previous work for the full set of rules. For second-order models with index set $\mathcal{I}^{(1,0)}$, parsing can be done in $O(n^3)$ time (McDonald & Pereira, 2006) and for third-order models in $O(n^4)$ time (Koo & Collins, 2010). Even though second-order parsing has the same asymptotic time complexity as first-order parsing, practical decoding is significantly slower. This speed difference is because in practice there is some cost for scoring each of the indices.

More specifically, assume that it takes c-time for each decoding operation and d-time to compute the score of each element in the index set (for the model we use, this time cost is from a feature vector dot product). First-order dependency parsing has time-complexity $O(n^3c + n^2d)$, whereas second-order parsing requires $O(n^3c + n^3d)$ time and third-order parsing requires $O(n^4c + n^4d)$ time. The trade-off between these terms is dependent on size of d, which for our models is roughly proportional to the number of features used.

We aim to prune the index set, by mapping each higher-order index down to a set of

small set indices that can be pruned using a coarse pruning model. For example, to use a first-order model for pruning, we would map the higher-order index to the individual indices for its arc, grandparents, and siblings:

$$\begin{aligned} p_{(k,l)\to 1}(t,s) &= \big\{(t_1, s_j) : j \in \{1 \dots k+1\}\big\} \\ &\cup \big\{(t_{j+1}, t_j) : j \in \{1 \dots l\}\big\} \end{aligned}$$

The first-order pruning model can then be used to score these indices, and to produce a filtered index set $F(\mathcal{I}^{(1)})$ by removing low-scoring indices (see Section 8.4). We retain only the higher-order indices that are supported by the filtered index set:

$$\big\{(t,s) \in \mathcal{I}^{(k,l)} : p_{(k,l)\to 1}(t,s) \subset F(\mathcal{I}^{(1)})\big\}$$

8.3.3 Vine Parsing

To further reduce the cost of parsing and produce faster pruning models, we need a model with less structure than the first-order model. A natural choice, following Section 8.2, is to only consider "short" arcs:

$$\mathcal{S} = \big\{(h,m) \in \mathcal{I}^{(1)} \quad : \quad |h-m| \leq b\big\}$$

where b is a small constant. This constraint reduces the size of the set to $|\mathcal{S}| = O(nb)$.

Clearly, this index set is severely limited; it is necessary to have some long arcs for even short sentences. We therefore augment the index set to include *outer* arcs:

$$\begin{aligned} \mathcal{I}^{(0)} = \mathcal{S} \;\cup\; & \big\{(d,m) : d \in \{\leftarrow, \rightarrow\}, m \in \{1 \dots n\}\big\} \\ \cup\; & \big\{(h,d) : h \in \{0 \dots n\}, d \in \{\leftarrow, \rightarrow\}\big\} \end{aligned}$$

The first set lets modifiers choose an outer head word and the second set lets head words accept outer modifiers, and both sets distinguish the direction of the arc. Figure 8-6 shows

a right outer arc. The size of $\mathcal{I}^{(0)}$ is linear in the sentence length. To parse the index set $\mathcal{I}^{(0)}$, we can modify the parse rules in Figure 8-3 to enforce additional length constraints ($|h - e| \leq b$ for $I(h, e)$ and $|h - m| \leq b$ for $C(h, m)$). This way, only indices in $\mathcal{S}$ are explored. Unfortunately, this is not sufficient since the constraints also prevent the algorithm from producing a full derivation, since no item can expand beyond length b.

Eisner and Smith (2005) therefore introduce vine parsing, which includes two new items, *vine left*, $V_{\leftarrow}(e)$, and *vine right*, $V_{\rightarrow}(e)$. Unlike the previous items, these new items are left-anchored at the root and grow only towards the right. The items $V_{\leftarrow}(e)$ and $V_{\rightarrow}(e)$ encode the fact that a word e has not taken a close (within b) head word to its left or right. We incorporate these items by adding the five new parsing rules shown in Figure 8-4.

The major addition is Rule 8-4(e) which converts a vine left item $V_{\leftarrow}(e)$ to a vine right item $V_{\rightarrow}(e)$. This implies that word e has no close head to either side, and the parse has outer head arcs, $y(\leftarrow, e) = 1$ or $y(\rightarrow, e) = 1$. The other rules are structural and dictate creation and extension of vine items. Rules 8-4(c) and 8-4(d) create vine left items from items that cannot find a head word to their left. Rules 8-4(f) and 8-4(g) extend and finish vine right items. Rules 8-4(d) and 8-4(f) each leave a head word incomplete, so they may set $y(e, \leftarrow) = 1$ or $y(m, \rightarrow) = 1$ respectively. Note that for all the new parse rules, $e \in \{0 \ldots n\}$ and $m \in \{e - b \ldots n\}$, so parsing time of this so called vine parsing algorithm is linear in the sentence length $O(nb^2)$.

Alone, vine parsing is a poor model of syntax - it does not even score most dependency pairs. However, it can act as a pruning model for other parsers. We prune a first-order model by mapping first-order indices to indices in $\mathcal{I}^{(0)}$.

$$p_{1\rightarrow 0}(h, m) = \begin{cases} \{(h, m)\} & \text{if } |h - m| \leq b \\ \{(\rightarrow, m), (h, \rightarrow)\} & \text{if } h < m \\ \{(\leftarrow, m), (h, \leftarrow)\} & \text{if } h > m \end{cases}$$

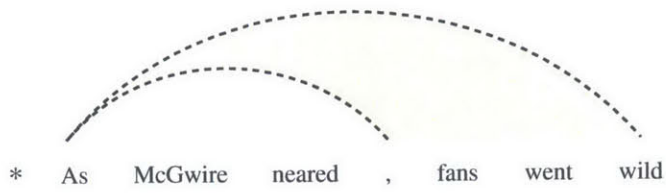


Figure 8-6: An outer arc $(1, \rightarrow)$ from the word "As" to possible right modifiers.

The remaining first-order indices are then given by:

$$\{(h, m) \in \mathcal{I}^{(1)} : p_{1 \rightarrow 0}(h, m) \subset F(\mathcal{I}^{(0)})\}$$

Figure 8-2 depicts a coarse-to-fine cascade, incorporating vine and first-order pruning passes and finishing with a higher-order parse model.

8.4 Training Methods

Our coarse-to-fine parsing architecture consists of multiple pruning passes followed by a final pass of 1-best parsing. The training objective for the pruning models comes from the prediction cascade framework of Weiss and Taskar (2010), which explicitly trades off pruning efficiency versus accuracy. The models used in the final pass on the other hand are trained for 1-best prediction.

8.4.1 Max-Marginal Filtering

At each pass of coarse-to-fine pruning, we apply an index filter function F to trim the index set:

$$F(\mathcal{I}) = \{i \in \mathcal{I} : f(i) = 1\}$$

Several types of filters have been proposed in the literature, with most work in coarse-to-fine parsing focusing on predicates that threshold the posterior probabilities. In structured prediction cascades, we use a non-probabilistic filter, based on the max-marginal value of

the index:

$$f(i) = \mathbf{1}[\ \theta^\top \mu(i) < \tau(\eta)\ ]$$

where $\tau(\eta)$ is a sentence-specific threshold value. To counteract the fact that the max-marginals are not normalized, the threshold $\tau(\eta)$ is set as a convex combination of the 1-best parse score and the average max-marginal value:

$$\tau(\eta) \quad = \quad \eta \max_{y \in \mathcal{Y}} \theta^\top y + (1-\eta) \frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} \theta^\top \mu(i)$$

where the model-specific parameter $0 \leq \eta \leq 1$ is the tradeoff between $\eta = 1$, pruning all indices i not in the best parse, and $\eta = 0$, pruning all indices with max-marginal value below the mean. (These pruning decisions will be made based on decoding with a lower-order model. We discuss this in the next section.)

The threshold function has the important property that for any parse y, if $\theta^\top y \geq \tau(\eta)$ then $y(i) = 1$ implies $f(i) = 0$, i.e. if the parse score is above the threshold, then none of its indices will be pruned.

8.4.2 Filter Loss Training

The aim of our pruning models is to filter as many indices as possible without losing the gold parse. In structured prediction cascades, we incorporate this pruning goal into our training objective.

Let y be the gold structure for a sentence. We define *filter loss* to be an indicator of whether any i with $y(i) = 1$ is filtered:

$$\Delta(y) = \mathbf{1}[\exists i \in y, \theta^\top \mu(i) < \tau(\eta)]$$

During training we minimize the expected filter loss using a standard structured SVM setup (Tsochantaridis et al., 2006). First we form a convex, continuous upper-bound of our loss

function:

$$\begin{aligned}\Delta(y) &\leq \mathbf{1}[\theta^\top y < \tau(\eta)] \\ &\leq \max\{0, 1 - \theta^\top y + \tau(\eta)\}\end{aligned}$$

where the first inequality comes from the properties of max-marginals and the second is the standard hinge-loss upper-bound on an indicator.

Now assume that we have a corpus of P training sentences. Let the sequence $(y_1^{(g)}, \ldots, y_P^{(g)})$ be the gold parses for each sentences and the sequence $(\mathcal{Y}_1, \ldots, \mathcal{Y}_P)$ be the set of possible output structures. We can form the regularized risk minimization for this upper bound of filter loss:

$$\min_w \beta\|w\|^2 + \frac{1}{P}\sum_{p=1}^{P} \max\{0, 1 - \theta_w^\top y_p^{(g)} + \tau(\eta; \mathcal{Y}_p, w)\}$$

This objective is convex and non-differentiable, due to the max inside t. We optimize using stochastic subgradient descent (Shalev-Shwartz et al., 2007). The stochastic subgradient at example p, $H(y^{(g)})$ is 0 if $y^{(g)} - 1 \geq \tau(\eta)$ otherwise,

$$\begin{aligned}H(y^{(g)}; w, \mathcal{Y}, \mathcal{I}) &= \frac{2\beta w}{P} - y^{(g)} + \eta \arg\max_{y \in \mathcal{Y}} \theta_w^\top y \\ &+ (1-\eta)\frac{1}{|\mathcal{I}|}\sum_{i \in \mathcal{I}} \mu(i)\end{aligned}$$

Each step of the algorithm has an update of the form:

$$w^{(k+1)} \leftarrow w^{(k)} - \alpha_k H(y_p^{(g)}; w^{(k)}, \mathcal{Y}_p, \mathcal{I}_p)$$

where α is an appropriate update rate for subgradient convergence. If $\eta = 1$ the objective is identical to structured SVM with 0/1 hinge loss. For other values of η, the subgradient includes a term from the features of all max-marginal structures at each index. These feature counts can be computed using dynamic programming.

Setup	First-order				Second-order				Third-order			
	Speed	PE	Oracle	UAS	Speed	PE	Oracle	UAS	Speed	PE	Oracle	UAS
NoPrune	1.00	0.00	100	91.4	0.32	0.00	100	92.7	0.01	0.00	100	93.3
LengthDictionary	1.94	43.9	99.9	91.5	0.76	43.9	99.9	92.8	0.05	43.9	99.9	93.3
LocalShort	3.08	76.6	99.1	91.4	1.71	76.4	99.1	92.6	0.31	77.5	99.0	93.1
Local	4.59	89.9	98.8	91.5	2.88	83.2	99.5	92.6	1.41	89.5	98.8	93.1
FirstOnly	3.10	95.5	95.9	91.5	2.83	92.5	98.4	92.6	1.61	92.2	98.5	93.1
FirstAndSecond			-				-		1.80	97.6	97.7	93.1
VinePosterior	3.92	94.6	96.5	91.5	3.66	93.2	97.7	92.6	1.67	96.5	97.9	93.1
VineCascade	5.24	95.0	95.7	91.5	3.99	91.8	98.7	92.6	2.22	97.8	97.4	93.1
	k=8				k=16				k=64			
ZhangNivre	4.32	-	-	92.4	2.39	-	-	92.5	0.64	-	-	92.7

Table 8.1: Results comparing pruning methods on PTB Section 22. Oracle is the max achievable UAS after pruning. Pruning efficiency (PE) is the percentage of non-gold first-order dependency arcs pruned. Speed is parsing time relative to the unpruned first-order model (around 2000 tokens/sec), i.e. speed of 5 is 5× faster than first-order or around 10,000 tokens/sec. UAS is the unlabeled attachment score of the final parses.

8.4.3 1-Best Training

For the final pass, we want to train the model for 1-best output. Several different learning methods are available for structured prediction models including structured perceptron (Collins, 2002), max-margin models (Taskar et al., 2003b), and log-linear models (Lafferty et al., 2001). In this work, we use the margin infused relaxed algorithm (MIRA) (Crammer & Singer, 2003; Crammer et al., 2006) with a hamming-loss margin. MIRA is an online algorithm with similar benefits as structured perceptron in terms of simplicity and fast training time. In practice, we found that MIRA with hamming-loss margin gives a performance improvement over structured perceptron and structured SVM.

8.5 Parsing Experiments

To empirically demonstrate the effectiveness of our approach, we compare our vine pruning cascade with a wide range of common pruning methods on the Penn WSJ Treebank (PTB) (Marcus et al., 1993). We then also show that vine pruning is effective across a variety of different languages.

For English, we convert the PTB constituency trees to dependencies using the Stanford dependency framework (De Marneffe et al., 2006). We then train on the standard PTB split with sections 2-21 as training, section 22 as validation, and section 23 as test. Results are similar using the Yamada and Matsumoto (2003) conversion. We additionally selected six languages from the CoNLL-X shared task (Buchholz & Marsi, 2006) that cover a number of different language families: Bulgarian, Chinese, Japanese, German, Portuguese, and Swedish. We use the standard CoNLL-X training/test split and tune parameters with cross-validation.

All experiments use unlabeled dependencies for training and test. Accuracy is reported as unlabeled attachment score (UAS), the percentage of tokens with the correct head word. For English, UAS ignores punctuation tokens and the test set uses predicted POS tags. For the other languages we follow the CoNLL-X setup and include punctuation in UAS and use gold POS tags on the set set. Speed-ups are given in terms of time relative to a highly optimized C++ implementation. Our unpruned first-order baseline can process roughly two thousand tokens a second and is comparable in speed to the greedy shift-reduce parser of Nivre et al. (2004).

8.5.1 Models

Our parsers perform multiple passes over each sentence. In each pass we first construct a (pruned) hypergraph (Klein & Manning, 2005) and then perform feature computation and decoding. We choose the highest η that produces a pruning error of no more than 0.2 on the validation set (typically $\eta \approx 0.6$) to filter indices for subsequent rounds (similar to Weiss and Taskar (2010)). We compare a variety of pruning models:

LengthDictionary a deterministic pruning method that eliminates all arcs longer than the maximum length observed for each head-modifier POS pair.

Local an unstructured arc classifier that chooses indices from $\mathcal{I}^{(1)}$ directly without enforcing parse constraints. Similar to the quadratic-time filter from Bergsma and Cherry (2010).

LocalShort an unstructured arc classifier that chooses indices from $\mathcal{I}^{(0)}$ directly without

enforcing parse constraints. Similar to the linear-time filter from Bergsma and Cherry (2010).

FirstOnly a structured first-order model trained with filter loss for pruning.

FirstAndSecond a structured cascade with first- and second-order pruning models.

VineCascade the full cascade with vine, first- and second-order pruning models.

VinePosterior the vine parsing cascade trained as a CRF with L-BFGS (Nocedal & Wright, 1999) and using posterior probabilities for filtering instead of max-marginals.

ZhangNivre an unlabeled reimplementation of the linear-time, k-best, transition-based parser of Zhang and Nivre (2011). This parser uses composite features up to third-order with a greedy decoding algorithm. The reimplementation is about twice as fast as their reported speed, but scores slightly lower.

We found LENGTHDICTIONARY pruning to give significant speed-ups in all settings and therefore always use it as an initial pass. The maximum number of passes in a cascade is five: dictionary, vine, first-, and second-order pruning, and a final third-order 1-best pass.[3] We tune the pruning thresholds for each round and each cascade separately. This is because we might be willing to do a more aggressive vine pruning pass if the final model is a first-order model, since these two models tend to often agree.

8.5.2 Features

For the non-pruning models, we use a standard set of features proposed in the discriminative dependency parsing literature (McDonald et al., 2005; Carreras, 2007; Koo & Collins, 2010). Included are lexical features, part-of-speech features, features on in-between tokens, as well as feature conjunctions, surrounding part-of-speech tags, and back-off features. In addition, we replicate each part-of-speech (POS) feature with an additional feature using coarse POS

[3]For the first-order parser, we found it beneficial to employ a reduced feature first-order pruner before the final model, i.e. the cascade has four rounds: dictionary, vine, first-order pruning, and first-order 1-best.

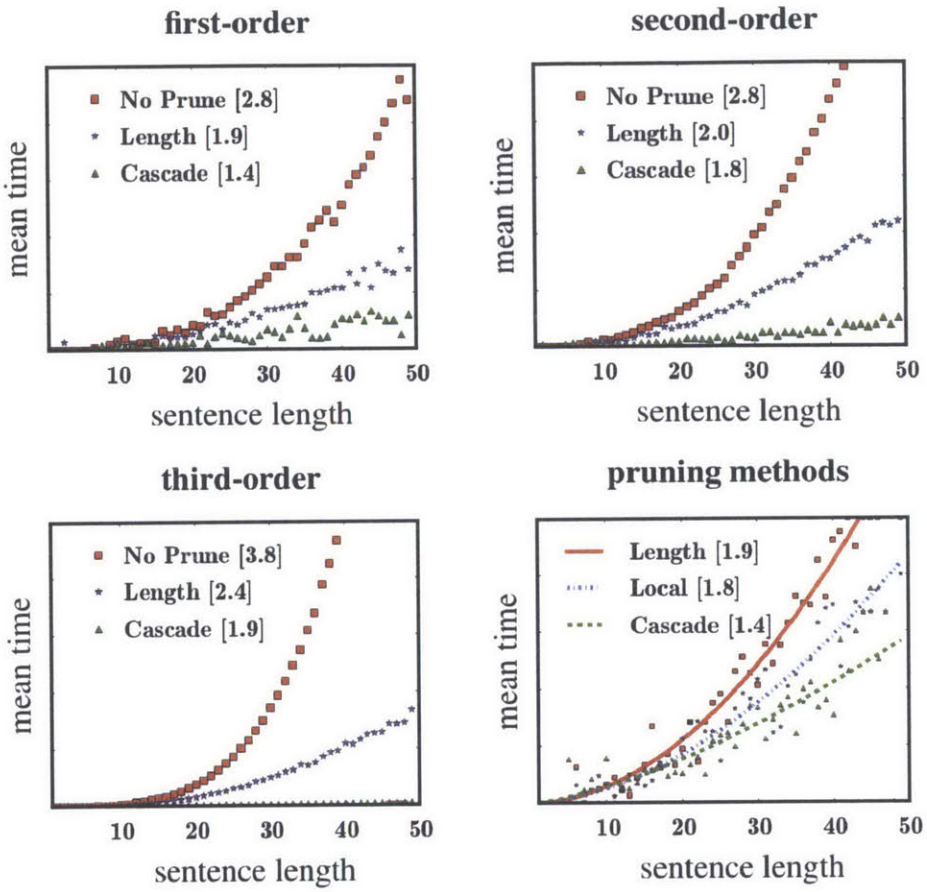


Figure 8-7: Mean parsing speed by sentence length for first-, second-, and third-order parsers as well as different pruning methods for first-order parsing. [b] indicates the empirical complexity obtained from fitting ax^b.

representations (Petrov et al., 2012). Our baseline parsing models replicate and, for some experiments, surpass previous best results.

The first- and second-order pruning models have the same structure, but for efficiency use only the basic features from McDonald et al. (2005). As feature computation is quite costly, future work may investigate whether this set can be reduced further. VINEPRUNE and LOCALSHORT use the same feature sets for short arcs. Outer arcs have features of the unary head or modifier token, as well as features for the POS tag bordering the cutoff and the direction of the arc.

	1-Best Model		
Round	First	Second	Third
Vine	37%	27%	16%
First	63%	30%	17%
Second	-	43%	18%
Third	-	-	49%

Table 8.2: Relative speed of pruning models in a multi-pass cascade. Note that the 1-best models use richer features than the corresponding pruning models.

8.5.3 Results

A comparison between the pruning methods is shown in Table 8.1. The table gives relative speed-ups, compared to the unpruned first-order baseline, as well as accuracy, pruning efficiency, and oracle scores. Note particularly that the third-order cascade is twice as fast as an unpruned first-order model and >200 times faster than the unpruned third-order baseline. The comparison with posterior pruning is less pronounced. Filter loss training is faster than VINEPOSTERIOR for first- and third-order parsing, but the two models have similar second-order speeds. It is also noteworthy that oracle scores, the best possible obtainable UAS after pruning, are consistently high even after multiple pruning rounds: the oracle score of our third-order model for example is 97.4%.

Vine pruning is particularly effective. The vine pass is faster than both LOCAL and FIRSTONLY and prunes more effectively than LOCALSHORT. Vine pruning benefits from having a fast, linear-time model, but still maintaining enough structure for pruning. While our pruning approach does not provide any asymptotic guarantees, Figure 8-7 shows that in practice our multi-pass parser scales well even for long sentences: Our first-order cascade scales almost linearly with the sentence length, while the third-order cascade scales better than quadratic. Table 8.2 shows that the final pass dominates the computational cost, while each of the pruning passes takes up roughly the same amount of time.

Our second- and third-order cascades also significantly outperform ZHANGNIVRE. The transition-based model with $k = 8$ is very efficient and effective, but increasing the k-best list size scales much worse than employing multi-pass pruning. We also note that while direct speed comparison are difficult, our parser is significantly faster than the published results

Setup		First-order Speed	First-order UAS	Second-order Speed	Second-order UAS	Third-order Speed	Third-order UAS
BG	B	1.90	90.7	0.67	92.0	0.05	92.1
	V	6.17	90.5	5.30	91.6	1.99	91.9
DE	B	1.40	89.2	0.48	90.3	0.02	90.8
	V	4.72	89.0	3.54	90.1	1.44	90.8
JA	B	1.77	92.0	0.58	92.1	0.04	92.4
	V	8.14	91.7	8.64	92.0	4.30	92.3
PT	B	0.89	90.1	0.28	91.2	0.01	91.7
	V	3.98	90.0	3.45	90.9	1.45	91.5
SW	B	1.37	88.5	0.45	89.7	0.01	90.4
	V	6.35	88.3	6.25	89.4	2.66	90.1
ZH	B	7.32	89.5	3.30	90.5	0.67	90.8
	V	7.45	89.3	6.71	90.3	3.90	90.9
EN	B	1.0	91.2	0.33	92.4	0.01	93.0
	V	5.24	91.0	3.92	92.2	2.23	92.7

Table 8.3: Speed and accuracy results for the vine pruning cascade across various languages. B *is the unpruned baseline model, and* V *is the vine pruning cascade. The first section of the table gives results for the CoNLL-X test datasets for Bulgarian* (BG)*, German* (DE)*, Japanese* (JA)*, Portuguese* (PT)*, Swedish* (SW)*, and Chinese* (ZH)*. The second section gives the result for the English* (EN) *test set, PTB Section 23.*

for other high accuracy parsers, e.g. Huang and Sagae (2010) and Koo et al. (2010).

Table 8.3 shows our results across a subset of the CoNLL-X datasets, focusing on languages that differ greatly in structure. The unpruned models perform well across datasets, scoring comparably to the top results from the CoNLL-X competition. We see speed increases for our cascades with almost no loss in accuracy across all languages, even for languages with fairly free word order like German. This is encouraging and suggests that the outer arcs of the vine-pruning model are able to cope with languages that are not as linear as English.

8.6 Conclusion

We presented a multi-pass architecture for dependency parsing that leverages vine parsing and structured prediction cascades. The resulting 200-fold speed-up leads to a third-order model that is twice as fast as an unpruned first-order model for a variety of languages, and

that also compares favorably to a state-of-the-art transition-based parser.

Unlike the other methods presented in this thesis, this chapter focused on an approximate decoding algorithm with a training method that aims to trade-off efficiency and optimality. While this method cannot provide any formal guarantees about model accuracy, in practice it is able to maintain state-of-the-art results, while running significantly faster than other approaches. Depending on the underlying application, it may be beneficial to utilize approximate algorithms of this sort, which opens up a wide-range of efficient methods.

Chapter 9

Conclusion

Modern statistical natural language systems can be seen as having three main facets: a statistical model of an underlying language phenomenon, a method for parameter estimation from data, and algorithms for making predictions based on these parameters. The focus of this work has primarily been on the final stage, prediction, particularly on the problem of decoding the highest-scoring structure from a statistical model. We have analyzed decoding problems for several core natural language tasks including:

- part-of-speech tagging and sequence decoding;
- projective and non-projective dependency parsing;
- lexicalized and unlexicalized context-free parsing; and
- syntax- and phrase-based machine translation

For some variants of these problems, decoding can be performed efficiently, but for others the decoding problem is challenging or even NP-hard. For the more difficult variants we turned to methods based on relaxations of the decoding problem.

The main tool we used for these challenging decoding problems was Lagrangian relaxation, a classical method from combinatorial optimization. From a high-level, Lagrangian relaxation gives a methodology for using relaxed versions of a problem in order to minimize

an upper-bound and, perhaps, produce optimal solutions to the original problem. It also gives us a set of formal guarantees that are not available for standard heuristic techniques.

Using Lagrangian relaxation, we developed relaxation algorithms for several core language decoding problems. Sometimes the relaxations involved combining multiple subproblems, which gives a dual decomposition algorithm, as for joint syntactic decoding. Other times the relaxation required exploiting the specific structure of the underlying decoding problem, as for syntax-based machine translation.

For these problems, the relaxation method yielded algorithms with the following important properties:

1. the algorithms used only basic combinatorial methods, e.g. dynamic programming, directed spanning tree algorithms, or all-pairs shortest path;

2. empirically, the relaxations were significantly faster than off-the-shelf solvers, and for some problems the approach used was comparable in speed to heuristic algorithms;

3. these methods have strong theoretical guarantees, potentially giving a certificate of optimality, and empirically are often able to produce this certificate on real examples.

These properties make Lagrangian relaxation useful for many NLP tasks.

Further Applications In recent years, Lagrangian relaxation and dual decomposition have become widely-used tools in natural language processing even beyond the work presented in this thesis.

One popular application area has been for decoding problems for various forms of parsing. This includes: combinatorial categorical grammar (CCG) super-tagging (Auli & Lopez, 2011); specialized algorithms of modeling syntactic coordination (Hanamoto et al., 2012); further state-of-the-art work on dependency parsing (Martins et al., 2011); a joint model combining latent-variable constituency parsers (Roux et al., 2013); and a shallow semantic parser with additional linguistic constraints (Das et al., 2012).

Another application area of dual decomposition has been for information extraction from text, a rich area of NLP. Work in this domain includes: a joint decoding algorithm for extracting biomedical events from text (Riedel & McCallum, 2011); a relaxation algorithm for decoding with Markov logic programs (Niu et al., 2012); a global constraint framework for event extraction (Reichart & Barzilay, 2012); and a joint bilingual word alignment and entity recognition systems (Wang et al., 2013).

Additionally, Lagrangian relaxation has been employed in a variety of other language systems including algorithms to produce symmetric word alignments (DeNero & Macherey, 2011), to find compressive summarizations of text (Almeida & Martins, 2013), and to decode intersections of weighted FSAs (Paul & Eisner, 2012).

Future Work While this dissertation has focused mainly on predictive problems in natural language processing and decoding in particular, there are many interesting future directions across other elements of statistical NLP, including model development and parameter estimation.

In terms of model development, in most of this work we looked at classical models that are widely used in natural language processing, e.g. context-free grammars, part-of-speech taggers, etc. However, part of the reasons these models are used is because of ease of decoding – for instance using a first-order dependency parser with maximum directed spanning tree algorithms. The empirical results in this work show that it is often possible to decode exactly even with challenging natural language models. In future work, we hope to experiment with developing much richer models that incorporate a range of wider structure – for instance much higher-order parsing models – or combine many more underlying models – for instance a single decoding problem for parsing, tagging, and translation. Generally these methods can allow for more experimentation in development of model structure.

In terms of estimation, much of the work in this thesis uses standard parameter estimation from supervised data sets. However, with the ever growing amount of available data, it is increasingly important to make use of the large amount of text without supervised annotations. NLP researchers have traditionally relied heavily on methods like Expectation-

Maximization (EM) for unsupervised learning, which can only guarantee convergence to a local minimum of its objective. An interesting future challenge is to develop and utilize alternative methods for learning from unsupervised textual data, with the goal of providing efficient, accurate algorithms that can also give improved theoretical guarantees. There are forms of unsupervised estimation can be posed withing a combinatorial optimization framework. An interesting future direction is using ideas from combinatorial optimization to try to directly solve these challenging estimation problems. While it is likely more difficult to find optimal solutions to these problems, combinatorial methods may provide an interesting alternative for unsupervised estimation.

Appendix A

Practical Issues

This appendix reviews various practical issues that arise with dual decomposition algorithms. We describe diagnostics that can be used to track progress of the algorithm in minimizing the dual, and in providing a primal solution; we describe methods for choosing the step sizes, α_k, in the algorithm; and we describe heuristics that can be used in cases where the algorithm does not provide an exact solution. We will use the algorithm from Chapter 4 as a running example, although our observations are easily generalized to other Lagrangian relaxation algorithms.

The first thing to note is that each iteration of the algorithm produces a number of useful terms, in particular:

- The solutions $y^{(k)}$ and $z^{(k)}$.
- The current dual value $L(\lambda^{(k)})$ (which is equal to $L(\lambda^{(k)}, y^{(k)}, z^{(k)})$.

In addition, in cases where we have a function $l : \mathcal{Y} \rightarrow \mathcal{Z}$ that maps each structure $y \in \mathcal{Y}$ to a structure $l(y) \in \mathcal{Z}$, we also have

- A primal solution $y^{(k)}, l(y^{(k)})$.
- A primal value $f(y^{(k)}) + g(l(y^{(k)}))$.

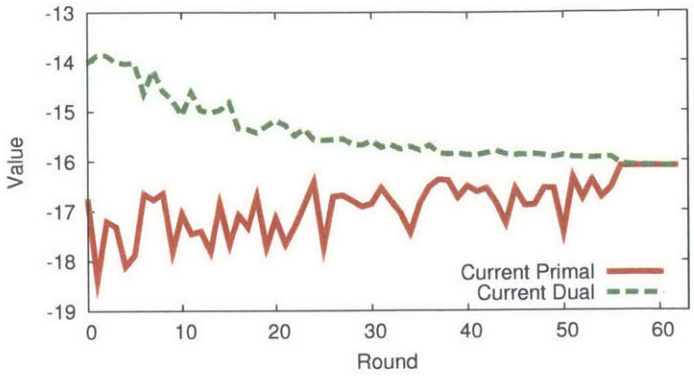


Figure A-1: Graph showing the dual value $L(\lambda^{(k)})$ and primal value $f(y^{(k)})+g(l(y^{(k)}))$, versus iteration number k, for the subgradient algorithm on a translation example from the work of Rush and Collins (2011).

By a "primal solution" we mean a pair (y, z) that satisfies all constraints in the optimization problem. For example, in optimization problem 3.5 (the combined HMM and PCFG problem from Section 4) a primal solution has the properties that $y \in \mathcal{Y}$, $z \in \mathcal{Z}$, and $y(i,t) = z(i,t)$ for all (i,t).

As one example, in the main dual-decomposition algorithm, at each iteration we produce a parse tree $y^{(k)}$. It is simple to recover the POS sequence $l(y^{(k)})$ from the parse tree, and to calculate the score $f(y^{(k)}) + g(l(y^{(k)}))$ under the combined model. Thus even if $y^{(k)}$ and $z^{(k)}$ disagree, we can still use $y^{(k)}, l(y^{(k)})$ as a potential primal solution. This ability to recover a primal solution from the value $y^{(k)}$ does not always hold—but in cases where it does hold, it is very useful. It will allow us, for example, to recover an approximate solution in cases where the algorithm hasn't e-converged to an exact solution.

We now describe how the various items described above can be used in practical applications of the algorithm.

A.1 An Example Run of the Algorithm

Figure A-1 shows a run of the subgradient algorithm for the inference approach for machine translation described in the work of Rush and Collins (2011). The behavior is typical of

cases where the algorithm e-converges to an exact solution. We show the dual value $L(\lambda^{(k)})$ at each iteration, and the value for $f(y^{(k)}) + g(l(y^{(k)}))$. A few important points are as follows:

- Because $L(\lambda)$ provides an upper bound on $f(y^*) + g(z^*)$ for any value of u, we have

$$L(\lambda^{(k)}) \geq f(y^{(k)}) + g(l(y^{(k)}))$$

  at every iteration.

- On this example we have e-convergence to an exact solution, at which point we have

$$L(\lambda^{(k)}) = f(y^{(k)}) + g(z^{(k)}))$$

  with $(y^{(k)}, z^{(k)})$ guaranteed to be optimal (and in addition, with $z^{(k)} = l(y^{(k)})$).

- The dual values $L(\lambda^{(k)})$ are not monotonically decreasing—that is, for some iterations we have

$$L(\lambda^{(k+1)}) > L(\lambda^{(k)})$$

  even though our goal is to minimize $L(\lambda)$. This is typical: subgradient algorithms are not in general guaranteed to give monotonically decreasing dual values. However, we do see that for most iterations the dual decreases—this is again typical.

- Similarly, the primal value $f(y^{(k)}) + g(z^{(k)})$ fluctuates (goes up and down) during the course of the algorithm.

The following quantities can be useful in tracking progress of the algorithm at the k'th iteration:

- $L(\lambda^{(k)}) - L(\lambda^{(k-1)})$ is the change in the dual value from one iteration to the next. We will soon see that this can be useful when choosing the step size for the algorithm (if this value is positive, it may be an indication that the step size should decrease).

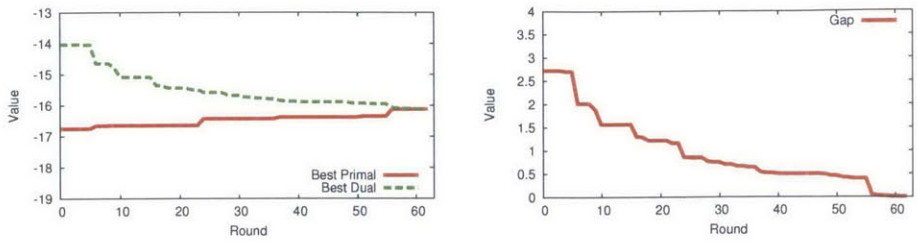


Figure A-2: The graph on the left shows the best dual value L_k^* and the best primal value p_k^*, versus iteration number k, for the subgradient algorithm on a translation example from the work of Rush and Collins (2011). The graph on the right shows $L_k^* - p_k^*$ plotted against k.

- $L_k^* = \min_{k' \leq k} L(\lambda^{(k')})$ is the best dual value found so far. It gives us the tightest upper bound on $f(y^*) + g(z^*)$ that we have after k iterations of the algorithm.
- $p_k^* = \max_{k' \leq k} f(y^{(k')}) + g(l(y^{(k')}))$ is the best primal value found so far.
- $L_k^* - p_k^*$ is the gap between the best dual and best primal solution found so far by the algorithm. Because $L_k^* \geq f(y^*) + g(z^*) \geq p_k^*$, we have

$$L_k^* - p_k^* \geq f(y^*) + g(z^*) - p_k^*$$

  hence the value for $L_k^* - p_k^*$ gives us an upper bound on the difference between $f(y^*) + g(z^*)$ and p_k^*. If $L_k^* - p_k^*$ is small, we have a guarantee that we have a primal solution that is close to being optimal.

Figure A-2 shows a plot of L_k^* and p_k^* versus the number of iterations k for our previous example, and in addition shows a plot of the gap $L_k^* - p_k^*$. These graphs are, not surprisingly, much smoother than the graph in Figure A-1. In particular we are guaranteed that the values for L_k^* and p_k^* are monotonically decreasing and increasing respectively.

A.2 Choice of the Step Sizes α_k

Figure A-3 shows convergence of the algorithm for various choices of step size, where we have chosen to keep the stepsize constant at each iteration. We immediately see a potential

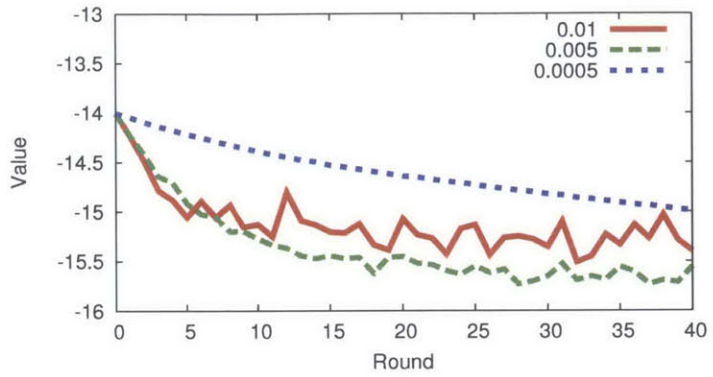


Figure A-3: Graph showing the dual value $L(\lambda^{(k)})$ versus the number of iterations k, for different fixed step sizes.

dilemma arising. With too small a step size ($\alpha = 0.0005$), convergence is smooth—the dual value is monotonically decreasing—but convergence is slow. With too large a step size ($\alpha = 0.01$), convergence is much faster in the initial phases of the algorithm, but the dual then fluctuates quite erratically. In practice it is often very difficult to choose a constant step size that gives good convergence properties in both early and late iterations of the algorithm.

Instead, we have found that we often find improved convergence properties with a choice of step size α_k that decreases with increasing k. One possibility is to use a definition such as $\alpha_k = c/k$ or $\alpha_k = c/\sqrt{k}$ where $c > 0$ is a constant. However these definitions can decrease the step size too rapidly—in particular, they decrease the step size at all iterations, even in cases where progress is being made in decreasing the dual value. In many cases we have found that a more effective definition is

$$\alpha_k = \frac{c}{t+1}$$

where $c > 0$ is again a constant, and $t < k$ is the number of iterations prior to k where the dual value increases rather than decreases (i.e., the number of cases for $k' \leq k$ where $L(\lambda^{(k')}) > L(\lambda^{(k'-1)})$). Under this definition the step size decreases only when the dual value moves in the wrong direction.

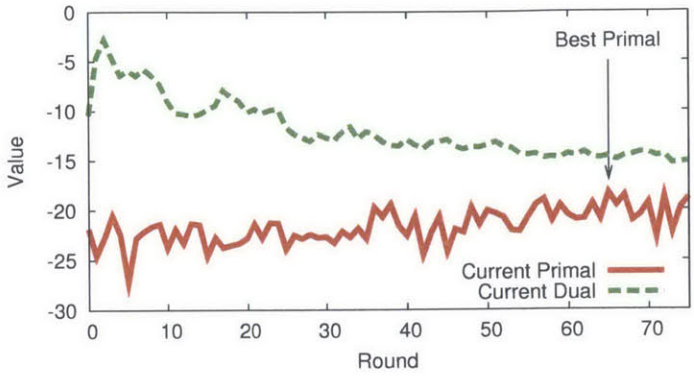


Figure A-4: Graph showing the dual value $L(\lambda^{(k)})$ and primal value $f(y^{(k)})+g(l(y^{(k)}))$, versus iteration number k, for the subgradient algorithm on a translation example from the work of Rush and Collins (2011), where the method does not e-converge to an exact solution.

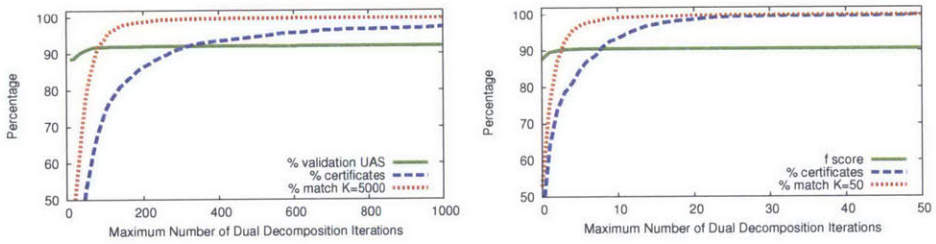


Figure A-5: Figures showing effects of early stopping for the non-projective parsing algorithm of ? (?) (left graph) and combined constituency and dependency parsing (right graph). In each case we plot three quantities versus the number of iterations, k: 1) the accuracy (UAS or f-score); 2) the percentage of cases where the algorithm e-converges giving an exact solution, with a certificate of optimality; 3) the percentage of cases where the best primal solution up to the k'th iteration is the same as running the algorithm to e-convergence.

A.3 Recovering Approximate Solutions

Figure A-4 shows a run of the algorithm where we fail to get e-convergence to an exact solution. One strategy, which is approximate, is to simply choose the best primal solution generated after k iterations of the algorithm, for some fixed k: i.e., to choose $y^{(k')}, l(y^{(k')})$ where

$$k' = \arg\max_{k' \leq k} f(y^{(k')}) + g(l(y^{(k')}))$$

As described before, we can use $L_k^* - p_k^*$ as an upper bound on the difference between this approximate solution and the optimal solution.

A.4 Early Stopping

It is interesting to also consider the strategy of returning the best primal solution early in the algorithm in cases where the algorithm *does* eventually e-converge to an exact solution. In practice, this strategy can sometimes produce a high quality solution, albeit without a certificate of optimality, faster than running the algorithm to e-convergence. Figure A-5 shows graphs for two problems: non-projective dependency parsing (?), and combined constituency and dependency parsing (Rush et al., 2010). In each case we show how three quantities vary with the number of iterations of the algorithm. The first quantity is the percentage of cases where the algorithm e-converges, giving an exact solution, with a certificate of optimality. For combined constituency and dependency parsing it takes roughly 50 iterations for most (over 95%) of cases to e-converge; the second algorithm takes closer to 1000 iterations.

In addition, we show graphs indicating the quality of the best primal solution generated up to iteration k of the algorithm, versus the number of iterations, k. An "early stopping" strategy would be to pick some fixed value for k, and to simply return the best primal solution generated in the first k iterations of the algorithm. We first plot the accuracy (f-score, or dependency accuracy respectively) for the two models under early stopping: we can see that accuracy very quickly asymptotes to its optimal value, suggesting that returning a primal solution before e-convergence can often yield high quality solutions. We also plot the percentage of cases where the primal solution returned is in fact identical to the primal solution returned when the algorithm is run to e-convergence. We again see that this curve asymptotes quickly, showing that in many cases the early stopping strategy does in fact produce the optimal solution, albeit without a certificate of optimality.

Appendix B

Proofs

In this appendix we derive various results for dual decomposition. Recall that in this case the Lagrangian is defined as

$$L(\lambda, y, z) = \theta^\top y + \omega^\top z + \lambda^\top (Ay + Cz - b)$$

and that the dual objective is $L(\lambda) = \max_{y \in \mathcal{Y}, z \in \mathcal{Z}} L(\lambda, y, z)$.

We first prove that $L(\lambda)$ is a convex function; we then derive the expression for subgradients of $L(\lambda)$; and finally give a convergence theorem for the subgradient algorithm minimizing $L(\lambda)$.

B.1 Proof of Convexity of $L(\lambda)$

The theorem is as follows:

Theorem B.1.1. *$L(\lambda)$ is convex. That is, for any $\lambda^{(1)} \in \mathbb{R}^d$, $\lambda^{(2)} \in \mathbb{R}^d$, $u \in [0, 1]$,*

$$L(u\lambda^{(1)} + (1-u)\lambda^{(2)}) \leq uL(\lambda^{(1)}) + (1-u)L(\lambda^{(2)})$$

Proof: Define

$$(y^*, z^*) = \arg\max_{y \in \mathcal{Y}, z \in \mathcal{Z}} L(\lambda^*, y, z)$$

where $\lambda^* = u\lambda^{(1)} + (1-u)\lambda^{(2)}$. It follows that

$$L(\lambda^*) = L(\lambda^*, y^*, z^*)$$

In addition, note that

$$L(\lambda^{(1)}, y^*, z^*) \leq \max_{y \in \mathcal{Y}, z \in \mathcal{Z}} L(\lambda^{(1)}, y, z) = L(\lambda^{(1)})$$

and similarly

$$L(\lambda^{(2)}, y^*, z^*) \leq L(\lambda^{(2)})$$

from which it follows that

$$uL(\lambda^{(1)}, y^*, z^*) + (1-u)L(\lambda^{(2)}, y^*, z^*) \leq uL(\lambda^{(1)}) + (1-u)L(\lambda^{(2)})$$

Finally, it is easy to show that

$$uL(\lambda^{(1)}, y^*, z^*) + (1-u)L(\lambda^{(2)}, y^*, z^*) = L(\lambda^*, y^*, z^*) = L(\lambda^*)$$

hence

$$L(\lambda^*) \leq uL(\lambda^{(1)}) + (1-u)L(\lambda^{(2)})$$

which is the desired result. □

B.2 Subgradients of $L(\lambda)$

For any value of $\lambda \in \mathbb{R}^d$, as before define

$$(y^{(\lambda)}, z^{(\lambda)}) = \arg\max_{y \in \mathcal{Y}, z \in \mathcal{Z}} L(\lambda, y, z)$$

or equivalently,

$$y^{(\lambda)} = \arg\max_{y \in \mathcal{Y}} (\theta + A^\top \lambda)^\top y$$

and

$$z^{(\lambda)} = \arg\max_{z \in \mathcal{Z}} (\omega + C^\top \lambda)^\top z$$

Then if we define $\gamma^{(\lambda)}$ as the vector with components

$$\gamma^{(\lambda)} = Ay^{(\lambda)} + Cz^{(\lambda)} - b$$

, then $\gamma^{(\lambda)}$ is a subgradient of $L(\lambda)$ at λ.

This result is a special case of the following theorem:[1]

Theorem B.2.1. *Define the function* $L : \mathbb{R}^d \to \mathbb{R}$ *as*

$$L(\lambda) = \max_{i \in \{1 \ldots m\}} (a_i \cdot \lambda + b_i)$$

where $a_i \in \mathbb{R}^d$ *and* $b_i \in \mathbb{R}$ *for* $i \in \{1 \ldots m\}$. *Then for any value of* λ, *if*

$$j = \arg\max_{i \in \{1 \ldots m\}} (a_i \cdot \lambda + b_i)$$

then a_j *is a subgradient of* $L(\lambda)$ *at* λ.

Proof: For a_j to be a subgradient at the point λ, we need to show that for all $v \in \mathbb{R}^d$,

$$L(v) \geq L(\lambda) + a_j \cdot (v - \lambda)$$

Equivalently, we need to show that for all $v \in \mathbb{R}^d$,

$$\max_{i \in \{1 \ldots m\}} (a_i \cdot v + b_i) \geq \max_{i \in \{1 \ldots m\}} (a_i \cdot \lambda + b_i) + a_j \cdot (v - \lambda) \tag{B.1}$$

[1] To be specific, our definition of $L(\lambda)$ can be written in the form $\max_{i \in \{1 \ldots m\}} (a_i \cdot \lambda + b_i)$ as follows. Define the integer m to be $|\mathcal{Y}| \times |\mathcal{Z}|$. Define $(y^{(i)}, z^{(i)})$ for $i \in \{1 \ldots m\}$ to be a list of all possible pairs (y, z) such that $y \in \mathcal{Y}$ and $z \in \mathcal{Z}$. Define $b_i = f(y^{(i)}) + g(z^{(i)})$, and a_i to be the vector with components $a_i(l,t) = y^{(i)}(l,t) - z^{(i)}(l,t)$ for $l \in \{1 \ldots n\}, t \in \mathcal{T}$. Then it can be verifed that $L(\lambda) = \max_{i \in \{1 \ldots m\}} (a_i \cdot \lambda + b_i)$.

To show this, first note that

$$a_j \cdot \lambda + b_j = \max_{i \in \{1 \ldots m\}} (a_i \cdot \lambda + b_i)$$

hence

$$\max_{i \in \{1 \ldots m\}} (a_i \cdot \lambda + b_i) + a_j \cdot (v - \lambda) = b_j + a_j \cdot v \leq \max_{i \in \{1 \ldots m\}} (a_i \cdot v + b_i)$$

thus proving the theorem. □

B.3 Convergence Proof for the Subgradient Method

Consider a convex function $L : \mathbb{R}^d \to \mathbb{R}$, which has a minimizer λ^* (i.e., $\lambda^* = \arg\min_{\lambda \in \mathbb{R}^d} L(\lambda)$). The subgradient method is an iterative method which initializes u to some valuc $\lambda^{(0)} \in \mathbb{R}^d$, then sets

$$\lambda^{(k+1)} = \lambda^{(k)} - \alpha_k g_k$$

for $k = 0, 1, 2, \ldots$, where $\alpha_k > 0$ is the stepsize at the k'th iteration, and g_k is a subgradient at $\lambda^{(k)}$: that is, for all $v \in \mathbb{R}^d$,

$$L(v) \geq L(\lambda^{(k)}) + g_k \cdot (v - \lambda^{(k)})$$

The following theorem will then be very useful in proving convergence of the method (the theorem and proof is taken from Boyd & Mutapcic, 2007b):

Theorem B.3.1. *Assume that for all k, $||g_k||^2 \leq G^2$ where G is some constant. Then for any $k \geq 0$,*

$$\min_{i \in \{0 \ldots k\}} L(\lambda^{(i)}) \leq L(\lambda^*) + \frac{||\lambda^{(0)} - \lambda^*||^2 + G^2 \sum_{i=0}^{k} \alpha_i^2}{2 \sum_{i=0}^{k} \alpha_i}$$

Proof: First, given the updates $\lambda^{(k+1)} = \lambda^{(k)} - \alpha_k g_k$, we have for all $k \geq 0$,

$$\begin{aligned} ||\lambda^{(k+1)} - \lambda^*||^2 &= ||\lambda^{(k)} - \alpha_k g_k - \lambda^*||^2 \\ &= ||\lambda^{(k)} - \lambda^*||^2 - 2\alpha_k g_k \cdot (\lambda^{(k)} - \lambda^*) + \alpha_k^2 ||g_k||^2 \end{aligned}$$

By the subgradient property,

$$L(\lambda^*) \geq L(\lambda^{(k)}) + g_k \cdot (\lambda^* - \lambda^{(k)})$$

hence

$$-g_k \cdot (\lambda^{(k)} - \lambda^*) \leq L(\lambda^*) - L(\lambda^{(k)})$$

Using this inequality, together with $||g_k||^2 \leq G^2$, gives

$$||\lambda^{(k+1)} - \lambda^*||^2 \leq ||\lambda^{(k)} - \lambda^*||^2 + 2\alpha_k \left(L(\lambda^*) - L(\lambda^{(k)})\right) + \alpha_k^2 G^2$$

Taking a sum over both sides of $i = 0 \ldots k$ gives

$$\sum_{i=0}^{k} ||\lambda^{(i+1)} - \lambda^*||^2 \leq \sum_{i=0}^{k} ||\lambda^{(i)} - \lambda^*||^2 + 2\sum_{i=0}^{k} \alpha_i \left(L(\lambda^*) - L(\lambda^{(i)})\right) + \sum_{i=0}^{k} \alpha_i^2 G^2$$

and hence

$$||\lambda^{(k+1)} - \lambda^*||^2 \leq ||\lambda^{(0)} - \lambda^*||^2 + 2\sum_{i=0}^{k} \alpha_i \left(L(\lambda^*) - L(\lambda^{(i)})\right) + \sum_{i=0}^{k} \alpha_i^2 G^2$$

Finally, using $||\lambda^{(k+1)} - \lambda^*||^2 \geq 0$ and

$$\sum_{i=0}^{k} \alpha_i \left(L(\lambda^*) - L(\lambda^{(i)})\right) \leq \left(\sum_{i=0}^{k} \alpha_i\right) \left(L(\lambda^*) - \min_{i \in \{0 \ldots k\}} L(\lambda^{(i)})\right)$$

gives

$$0 \leq ||\lambda^{(0)} - \lambda^*||^2 + 2\left(\sum_{i=0}^{k} \alpha_i\right) \left(L(\lambda^*) - \min_{i \in \{0 \ldots k\}} L(\lambda^{(i)})\right) + \sum_{i=0}^{k} \alpha_i^2 G^2$$

Rearranging terms gives the result in the theorem. □

This theorem has a number of consequences. As one example, for a constant step-size,

$\alpha_k = h$ for some $h > 0$,

$$\lim_{k \to \infty} \left(\frac{||\lambda^{(0)} - \lambda^*||^2 + G^2 \sum_{i=1}^{k} \alpha_i^2}{2 \sum_{i=1}^{k} \alpha_i} \right) = \frac{Gh}{2}$$

hence in the limit the value for

$$\min_{i \in \{1 \ldots k\}} L(\lambda^{(i)})$$

is within $Gh/2$ of the optimal solution. A slightly more involved argument shows that under the assumptions that $\alpha_k > 0$, $\lim_{k \to \infty} \alpha_k = 0$, and $\sum_{k=0}^{\infty} \alpha_k = \infty$,

$$\lim_{k \to \infty} \left(\frac{||\lambda^{(0)} - \lambda^*||^2 + G^2 \sum_{i=1}^{k} \alpha_i^2}{2 \sum_{i=1}^{k} \alpha_i} \right) = 0$$

See Boyd and Mutapcic for the full derivation.

Bibliography

Almeida, M. B., & Martins, A. F. (2013). Fast and robust compressive summarization with dual decomposition and multi-task learning. In *Proceedings of ACL*, pp. 196–206.

Alshawi, H. (1996). Head Automata and Bilingual Tiling: Translation with Minimal Representations. In *Proc. ACL*, pp. 167–176.

Auli, M., & Lopez, A. (2011). A comparison of loopy belief propagation and dual decomposition for integrated ccg supertagging and parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pp. 470–480, Portland, Oregon, USA. Association for Computational Linguistics.

Bar-Hillel, Y., Perles, M., & Shamir, E. (1964). On formal properties of simple phrase structure grammars. In *Language and Information: Selected Essays on their Theory and Application*, pp. 116–150.

Bergsma, S., & Cherry, C. (2010). Fast and accurate arc filtering for dependency parsing. In *Proc. of COLING*, pp. 53–61.

Boyd, S., & Vandenberghe, L. (2004). *Convex optimization*. Cambridge Univ Pr.

Boyd, S., & Mutapcic, A. (2007a). Subgradient methods..

Boyd, S., & Mutapcic, A. (2007b). *Subgradient Methods*. Course Notes for EE364b, Stanford University, Winter 2006-07.

Buchholz, S., & Marsi, E. (2006). CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, pp. 149–164. Association for Computational Linguistics.

Carreras, X. (2007). Experiments with a Higher-Order Projective Dependency Parser. In *Proc. EMNLP-CoNLL*, pp. 957–961.

Carreras, X., Collins, M., & Koo, T. (2008). TAG, dynamic programming, and the perceptron for efficient, feature-rich parsing. In *Proc CONLL*, pp. 9–16.

Chang, Y., & Collins, M. (2011). Exact Decoding of Phrase-based Translation Models through Lagrangian Relaxation. In *To appear proc. of EMNLP*.

Charniak, E., Johnson, M., Elsner, M., Austerweil, J., Ellis, D., Haxton, I., Hill, C., Shrivaths, R., Moore, J., Pozar, M., et al. (2006). Multilevel coarse-to-fine PCFG parsing. In *Proc. of NAACL/HLT*, pp. 168–175.

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

Chiang, D. (2007). Hierarchical phrase-based translation. *computational linguistics*, *33*(2), 201–228.

Chu, Y., & Liu, T. (1965). On the Shortest Arborescence of a Directed Graph. *Science Sinica*, *14*, 1396–1400.

Collins, M. (1997). Three Generative, Lexicalised Models for Statistical Parsing. In *Proc. ACL*, pp. 16–23.

Collins, M. (2002). Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *Proc. EMNLP*, pp. 1–8.

Collins, M. (2003). Head-driven statistical models for natural language parsing. In *Computational linguistics*, Vol. 29, pp. 589–637.

Crammer, K., Dekel, O., Keshet, J., Shalev-Shwartz, S., & Singer, Y. (2006). Online passive-aggressive algorithms. *The Journal of Machine Learning Research*, *7*, 551–585.

Crammer, K., & Singer, Y. (2003). Ultraconservative online algorithms for multiclass problems. *The Journal of Machine Learning Research*, *3*, 951–991.

Dantzig, G., & Wolfe, P. (1960). Decomposition principle for linear programs. In *Operations research*, Vol. 8, pp. 101–111.

Das, D., Martins, A., & Smith, N. (2012). An exact dual decomposition algorithm for shallow semantic parsing with constraints. In *Proceedings of SEM.[ii, 10, 50].*

de Gispert, A., Iglesias, G., Blackwood, G., Banga, E. R., & Byrne, W. (2010). Hierarchical Phrase-Based Translation with Weighted Finite-State Transducers and Shallow-n Grammars. In *Computational linguistics*, Vol. 36, pp. 505–533.

De Marneffe, M., MacCartney, B., & Manning, C. (2006). Generating typed dependency parses from phrase structure parses. In *Proc. of LREC*, Vol. 6, pp. 449–454.

DeNero, J., & Macherey, K. (2011). Model-Based Aligner Combination Using Dual Decomposition. In *Proc. ACL*.

Duchi, J., Tarlow, D., Elidan, G., & Koller, D. (2007a). Using combinatorial optimization within max-product belief propagation. In *NIPS*, Vol. 19.

Duchi, J., Tarlow, D., Elidan, G., & Koller, D. (2007b). Using combinatorial optimization within max-product belief propagation. In *Advances in Neural Information Processing Systems (NIPS)*.

Dyer, C., Lopez, A., Ganitkevitch, J., Weese, J., Ture, F., Blunsom, P., Setiawan, H., Eidelman, V., & Resnik, P. (2010). cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models..

Edmonds, J. (1967). Optimum Branchings. *Journal of Research of the National Bureau of Standards*, *71B*, 233–240.

Eisner, J. (2000). Bilexical grammars and their cubic-time parsing algorithms. In *Advances in Probabilistic and Other Parsing Technologies*, pp. 29–62.

Eisner, J., & Satta, G. (1999). Efficient Parsing for Bilexical Context-Free Grammars and Head-Automaton Grammars. In *Proc. ACL*, pp. 457–464.

Eisner, J., & Smith, N. (2005). Parsing with soft and hard constraints on dependency length. In *Proc. of IWPT*, pp. 30–41.

Everett III, H. (1963). Generalized lagrange multiplier method for solving problems of optimum allocation of resources.. pp. 399–417. JSTOR.

Felzenszwalb, P., & Huttenlocher, D. (2006). Efficient belief propagation for early vision. *International journal of computer vision*, *70*(1), 41–54.

Finley, T., & Joachims, T. (2008). Training structural svms when exact inference is intractable. In *ICML*, pp. 304–311.

Fisher, M. L. (1981). The lagrangian relaxation method for solving integer programming problems. *Management Science*, *27*(1), pp. 1–18.

Floyd, R. W. (1962). Algorithm 97: Shortest path. *Commun. ACM*, *5*, 345.

Germann, U., Jahr, M., Knight, K., Marcu, D., & Yamada, K. (2001). Fast decoding and optimal decoding for machine translation. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pp. 228–235. Association for Computational Linguistics.

Globerson, A., & Jaakkola, T. (2007). Fixing max-product: Convergent message passing algorithms for MAP LP-relaxations. In *NIPS*, Vol. 21.

Gurobi Optimization, I. (2013). Gurobi optimizer reference manual..

Hanamoto, A., Matsuzaki, T., & Tsujii, J. (2012). Coordination structure analysis using dual decomposition. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 430–438, Avignon, France. Association for Computational Linguistics.

Held, M., & Karp, R. M. (1971). The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, *1*, 6–25. 10.1007/BF01584070.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2001). Introduction to automata theory, languages, and computation. *ACM SIGACT News*, *32*(1), 60–65.

Huang, L., & Sagae, K. (2010). Dynamic programming for linear-time incremental parsing. In *Proc. of ACL*, pp. 1077–1086.

Huang, L., & Chiang, D. (2005). Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pp. 53–64. Association for Computational Linguistics.

Huang, L., & Chiang, D. (2007). Forest rescoring: Faster decoding with integrated language models. In *Proceedings of the 45th Annual Meeting of the Association of Computational*

Linguistics, pp. 144–151, Prague, Czech Republic. Association for Computational Linguistics.

Huang, L., & Mi, H. (2010). Efficient incremental decoding for tree-to-string translation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pp. 273–283, Cambridge, MA. Association for Computational Linguistics.

Iglesias, G., de Gispert, A., Banga, E. R., & Byrne, W. (2009). Rule filtering by pattern for efficient hierarchical translation. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, pp. 380–388, Athens, Greece. Association for Computational Linguistics.

Johnson, J., Malioutov, D., & Willsky, A. (2007). Lagrangian relaxation for map estimation in graphical models. In *45th Annual Allerton Conference on Communication, Control and Computing*.

Klein, D., & Manning, C. (2002). Fast exact inference with a factored model for natural language parsing. *Advances in neural information processing systems*, *15*(2002).

Klein, D., & Manning, C. D. (2005). Parsing and hypergraphs. In *New developments in parsing technology*, pp. 351–372. Springer.

Knight, K. (1999). Decoding complexity in word-replacement translation models. *Computational Linguistics*, *25*(4), 607–615.

Knuth, D. E. (1977). A generalization of dijkstra's algorithm. *Information Processing Letters*, *6*(1), 1–5.

Koehn, P., Och, F., Marcu, D., et al. (2003). Statistical phrase-based translation. In *Proc. NAACL*, Vol. 1, pp. 48–54.

Koehn, P. (2004). Pharaoh: a beam search decoder for phrase-based statistical machine translation models. *Machine translation: From real users to research*, *1*, 115–124.

Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., Cowan, B., Shen, W., Moran, C., Zens, R., Dyer, C., Bojar, O., Constantin, A., & Herbst, E. (2007). Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, ACL '07, pp. 177–180.

Komodakis, N., Paragios, N., & Tziritas, G. (2007). MRF Optimization via Dual Decomposition: Message-Passing Revisited. In *Proc. ICCV*.

Komodakis, N., Paragios, N., & Tziritas, G. (2011). Mrf energy minimization and beyond via dual decomposition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, pp. 1–1.

Koo, T., Carreras, X., & Collins, M. (2008). Simple semi-supervised dependency parsing. In *Proc. ACL/HLT*.

Koo, T., & Collins, M. (2010). Efficient third-order dependency parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pp. 1–11. Association for Computational Linguistics.

Koo, T., Globerson, A., Carreras, X., & Collins, M. (2007). Structured Prediction Models via the Matrix-Tree Theorem. In *Proc. EMNLP-CoNLL*, pp. 141–150.

Koo, T., Rush, A. M., Collins, M., Jaakkola, T., & Sontag, D. (2010). Dual decomposition for parsing with non-projective head automata. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pp. 1288–1298, Cambridge, MA. Association for Computational Linguistics.

Korte, B., & Vygen, J. (2008). *Combinatorial Optimization: Theory and Algorithms*. Springer Verlag.

Kuhlmann, M., Gómez-Rodríguez, C., & Satta, G. (2011). Dynamic programming algorithms for transition-based dependency parsers. In *Proc. of ACL/HLT*, pp. 673–682.

Kulesza, A., & Pereira, F. (2008). Structured learning with approximate inference. In *NIPS*.

Kumar, S., & Byrne, W. (2005). Local phrase reordering models for statistical machine translation. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pp. 161–168, Vancouver, British Columbia, Canada. Association for Computational Linguistics.

Lacoste-Julien, S., Taskar, B., Klein, D., & Jordan, M. (2006). Word alignment via quadratic assignment. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pp. 112–119. Association for Computational Linguistics.

Lafferty, J., McCallum, A., & Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE-*, pp. 282–289.

Lee, L. (2002). Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM*, *49*(1), 1–15.

Lemaréchal, C. (2001). Lagrangian Relaxation. In *Computational Combinatorial Optimization, Optimal or Provably Near-Optimal Solutions [based on a Spring School]*, pp. 112–156, London, UK. Springer-Verlag.

Marcu, D., Wang, W., Echihabi, A., & Knight, K. (2006). Spmt: Statistical machine translation with syntactified target language phrases. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pp. 44–52, Sydney, Australia. Association for Computational Linguistics.

Marcus, M., Santorini, B., & Marcinkiewicz, M. (1993). Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, *19*(2), 313–330.

Martin, R., Rardin, R., & Campbell, B. (1990). Polyhedral characterization of discrete dynamic programming. *Operations research*, *38*(1), 127–138.

Martins, A., Das, D., Smith, N., & Xing, E. (2008). Stacking Dependency Parsers. In *Proc. EMNLP*, pp. 157–166.

Martins, A., Smith, N., & Xing, E. (2009a). Concise Integer Linear Programming Formulations for Dependency Parsing. In *Proc. ACL*, pp. 342–350.

Martins, A., Smith, N., & Xing, E. (2009b). Concise integer linear programming formulations for dependency parsing. In *Proc. ACL*.

Martins, A., Smith, N., Figueiredo, M., & Aguiar, P. (2011). Dual decomposition with many overlapping components. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pp. 238–249, Edinburgh, Scotland, UK. Association for Computational Linguistics.

McDonald, R. (2006). *Discriminative Training and Spanning Tree Algorithms for Dependency Parsing*. Ph.D. thesis, University of Pennsylvania, Philadelphia, PA, USA.

McDonald, R., Crammer, K., & Pereira, F. (2005). Online large-margin training of dependency parsers. In *Proc. of ACL*, pp. 91–98.

McDonald, R., & Pereira, F. (2006). Online Learning of Approximate Dependency Parsing Algorithms. In *Proc. EACL*, pp. 81–88.

McDonald, R., Pereira, F., Ribarov, K., & Hajič, J. (2005). Non-Projective Dependency Parsing using Spanning Tree Algorithms. In *Proc. HLT-EMNLP*, pp. 523–530.

McDonald, R., & Satta, G. (2007). On the Complexity of Non-Projective Data-Driven Dependency Parsing. In *Proc. IWPT*.

Mohri, M. (1997). Finite-state transducers in language and speech processing. *Comput. Linguist.*, *23*(2), 269–311.

Niu, F., Zhang, C., Ré, C., & Shavlik, J. W. (2012). Scaling inference for markov logic via dual decomposition.. In *ICDM*, pp. 1032–1037.

Nivre, J., Hall, J., & Nilsson, J. (2004). Memory-Based Dependency Parsing. In *Proc. CoNLL*, pp. 49–56.

Nivre, J., & McDonald, R. (2008). Integrating Graph-Based and Transition-Based Dependency Parsers. In *Proc. ACL*, pp. 950–958.

Nocedal, J., & Wright, S. J. (1999). *Numerical Optimization*. Springer.

Paul, M. J., & Eisner, J. (2012). Implicitly intersecting weighted automata using dual decomposition. In *Proc. NAACL*.

Pauls, A., & Klein, D. (2009). Hierarchical search for parsing. In *Proc. of NAACL/HLT*, pp. 557–565.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference (2nd edition)*. Morgan Kaufmann Publishers.

Petrov, S. (2009). *Coarse-to-Fine Natural Language Processing*. Ph.D. thesis, University of California at Bekeley, Berkeley, CA, USA.

Petrov, S., Das, D., & McDonald, R. (2012). A universal part-of-speech tagset. In *LREC*.

Petrov, S., & Klein, D. (2007). Improved inference for unlexicalized parsing. In *Proc. of NAACL/HLT*, pp. 404–411.

Petrov, S., Haghighi, A., & Klein, D. (2008). Coarse-to-fine syntactic machine translation using language projections. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pp. 108–116, Honolulu, Hawaii. Association for Computational Linguistics.

Polyak, B. (1987). *Introduction to Optimization*. Optimization Software, Inc.

Punyakanok, V., Roth, D., Yih, W., & Zimak, D. (2005). Learning and Inference over Constrained Output. In *Proc. IJCAI*, pp. 1124–1129.

Reichart, R., & Barzilay, R. (2012). Multi event extraction guided by global constraints. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 70–79. Association for Computational Linguistics.

Riedel, S., & Clarke, J. (2006a). Incremental integer linear programming for non-projective dependency parsing. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pp. 129–137. Association for Computational Linguistics.

Riedel, S., & Clarke, J. (2006b). Incremental integer linear programming for non-projective dependency parsing. In *Proc. EMNLP*, pp. 129–137.

Riedel, S., & Clarke, J. (2006c). Incremental Integer Linear Programming for Non-projective Dependency Parsing. In *Proc. EMNLP*, pp. 129–137.

Riedel, S., & Clarke, J. (2006d). Incremental integer linear programming for non-projective dependency parsing. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, EMNLP '06, pp. 129–137, Stroudsburg, PA, USA. Association for Computational Linguistics.

Riedel, S., & McCallum, A. (2011). Fast and robust joint models for biomedical event extraction. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pp. 1–12, Edinburgh, Scotland, UK. Association for Computational Linguistics.

Roark, B., & Hollingshead, K. (2008). Classifying chart cells for quadratic complexity context-free inference. In *Proc. of COLING*, pp. 745–751.

Roth, D., & Yih, W. (2005a). Integer linear programming inference for conditional random fields. In *Proceedings of the 22nd international conference on Machine learning*, pp. 736–743. ACM.

Roth, D., & Yih, W. (2005b). Integer linear programming inference for conditional random fields. In *Proc. ICML*, pp. 737–744.

Roux, J. L., Rozenknop, A., & Foster, J. (2013). Combining pcfg-la models with dual decomposition: A case study with function labels and binarization.. In *EMNLP*, pp. 1158–1169. ACL.

Rush, A., & Petrov, S. (2012). Vine pruning for efficient multi-pass dependency parsing. In *Joint Human Language Technology Conference/Annual Meeting of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL12)*.

Rush, A. M., & Collins, M. (2012). A tutorial on dual decomposition and lagrangian relaxation for inference in natural language processing. *J. Artif. Intell. Res. (JAIR), 45*, 305–362.

Rush, A., & Collins, M. (2011). Exact Decoding of Syntactic Translation Models through Lagrangian Relaxation. In *Proc. ACL*.

Rush, A., Sontag, D., Collins, M., & Jaakkola, T. (2010). On Dual Decomposition and Linear Programming Relaxations for Natural Language Processing. In *Proc. EMNLP*.

Shalev-Shwartz, S., Singer, Y., & Srebro, N. (2007). Pegasos: Primal estimated sub-gradient solver for svm. In *Proc. of ICML*, pp. 807–814.

Shen, L., Xu, J., & Weischedel, R. (2008). A new string-to-dependency machine translation algorithm with a target dependency language model. In *Proceedings of ACL-08: HLT*, pp. 577–585, Columbus, Ohio. Association for Computational Linguistics.

Shor, N. Z. (1985). *Minimization Methods for Non-differentiable Functions*. Springer Series in Computational Mathematics. Springer.

Smith, D., & Eisner, J. (2008a). Dependency Parsing by Belief Propagation. In *Proc. EMNLP*, pp. 145–156.

Smith, D., & Eisner, J. (2008b). Dependency parsing by belief propagation. In *Proc. EMNLP*, pp. 145–156.

Smith, D., & Smith, N. (2007). Probabilistic Models of Nonprojective Dependency Trees. In *Proc. EMNLP-CoNLL*, pp. 132–140.

Sontag, D., Globerson, A., & Jaakkola, T. (2010). Introduction to dual decomposition for inference. In Sra, S., Nowozin, S., & Wright, S. J. (Eds.), *Optimization for Machine Learning*. MIT Press.

Sontag, D., Meltzer, T., Globerson, A., Jaakkola, T., & Weiss, Y. (2008). Tightening LP relaxations for MAP using message passing. In *Proc. UAI*.

Surdeanu, M., Johansson, R., Meyers, A., Màrquez, L., & Nivre, J. (2008). The CoNLL-2008 Shared Task on Joint Parsing of Syntactic and Semantic Dependencies. In *Proc. CoNLL*.

Taskar, B., Guestrin, C., & Koller, D. (2003a). Max-margin Markov networks. In *NIPS*.

Taskar, B., Guestrin, C., & Koller, D. (2003b). Max-margin markov networks. *Advances in neural information processing systems, 16*, 25–32.

Taskar, B., Klein, D., Collins, M., Koller, D., & Manning, C. (2004). Max-margin parsing. In *Proc. EMNLP*, pp. 1–8.

Tillmann, C. (2006). Efficient dynamic programming search algorithms for phrase-based SMT. In *Proceedings of the Workshop on Computationally Hard Problems and Joint Inference in Speech and Language Processing*, CHSLP '06, pp. 9–16.

Toutanova, K., & Manning, C. (2000). Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proc. EMNLP*, pp. 63–70.

Tromble, R. W., & Eisner, J. (2006). A fast finite-state relaxation method for enforcing global constraints on sequence decoding. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, HLT-NAACL '06, pp. 423–430, Stroudsburg, PA, USA. Association for Computational Linguistics.

Tsochantaridis, I., Joachims, T., Hofmann, T., & Altun, Y. (2006). Large margin methods for structured and interdependent output variables. *Journal of Machine Learning Research*, *6*(2), 1453.

Wainwright, M., & Jordan, M. I. (2008). *Graphical Models, Exponential Families, and Variational Inference*. Now Publishers Inc., Hanover, MA, USA.

Wang, M., Che, W., & Manning, C. D. (2013). Joint word alignment and bilingual named entity recognition using dual decomposition. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, Vol. 1.

Watanabe, T., Tsukada, H., & Isozaki, H. (2006). Left-to-right target generation for hierarchical phrase-based translation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, ACL-44, pp. 777–784, Morristown, NJ, USA. Association for Computational Linguistics.

Weaver, W. (1955). Translation. *Machine translation of languages*, *14*, 15–23.

Weiss, D., & Taskar, B. (2010). Structured prediction cascades. In *Proc. of AISTATS*, Vol. 1284, pp. 916–923.

Yamada, H., & Matsumoto, Y. (2003). Statistical dependency analysis with support vector machines. In *Proc. of IWPT*, Vol. 3, pp. 195–206.

Yanover, C., Meltzer, T., & Weiss, Y. (2006). Linear Programming Relaxations and Belief Propagation–An Empirical Study. In *The Journal of Machine Learning Research*, Vol. 7, p. 1907. MIT Press.

Zhang, Y., & Nivre, J. (2011). Transition-based dependency parsing with rich non-local features. In *Proc. of ACL*, pp. 188–193.