STATISTICAL LTAG PARSING

Libin Shen

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2006

Aravind K. Joshi
Supervisor of Dissertation

Rajeev Alur
Graduate Group Chairperson

Acknowledgments

First and foremost, I would like to thank my advisor Aravind Joshi for his continuous support and guidance in both academic and daily life ever since my first day at Penn.

Many thanks to my dissertation committee members, Mark Johnson, Mitch Marcus, Martha Palmer and Fernando Pereira. Their valuable advice and suggestions help me to sharpen the focus of this research.

I am grateful to Anoop Sarkar and Fei Xia. During my first two years at Penn, I learned a lot of NLP from them. I appreciate enlightening discussions with NLP people at Penn. They are Dan Bikel, John Blitzer, Jinying Chen, David Chiang, Susan Converse, Nikhil Dinesh, Yuan Ding, Julia Hockenmaier, Liang Huang, Alexandra Kinyon, Seth Kulick, Edward Loper, Ryan McDonald, Rashmi Prasad, Fei Sha, Nianwen Xue and Szu-ting Yi. Thanks to Srinivas Bangalore, Michael Collins, Dan Gildea, Giorgio Satta, Franz Och and other members of the SMT team at JHU Summer Workshop 2003, who helped my research work in various ways. Thanks also to my past and current office-mates, Eva Banik, Alan Lee and Carlos Prolo.

Last but not the lease, I am indebted to my family; to my parents for their understanding and endless patience, to my wife Ye for her unconditional support and encouragement, and to my daughter Sophie for sharing our pleasure just at the right moment.

ABSTRACT

STATISTICAL LTAG PARSING

Libin Shen

Aravind K. Joshi

In this work, we apply statistical learning algorithms to Lexicalized Tree Adjoining Grammar (LTAG) parsing, as an effort toward statistical analysis over deep structures. LTAG parsing is a well known hard problem. Statistical methods successfully applied to LTAG parsing could also be used in many other structure prediction problems in NLP.

For the purpose of achieving accurate and efficient LTAG parsing, we will investigate two aspects of the problem, the data structure and the algorithm.

1. We introduce LTAG-spinal, a variant of LTAG with very desirable linguistic, computational and statistical properties. It can be shown that LTAG-spinal with adjunction constraints is weakly equivalent to the traditional LTAG. For the purpose of statistical processing, we extract an LTAG-spinal treebank from the Penn Treebank with Propbank annotation.

2. We not only explore various parsing strategies, but also investigate the reranking approach.

    - We first propose a left-to-right incremental parser for LTAG-spinal, as an attempt to dynamically incorporate supertagging and dependency analysis. A perceptron like discriminative learning algorithm is used for training.

      We further investigate a bidirectional dependency parser for LTAG-spinal, in order to overcome the limitation of left-to-right processing. We propose a novel algorithm for graph-based incremental construction, and apply this algorithm to LTAG style dependency parsing.

    - We also explore learning algorithms for parse reranking, as well as other NLP problems, e.g. Machine Translation. We propose a novel reranking strategy,

Ordinal Regression with Uneven Margins (ORUM), which achieves state-of-the-art performance on parse reranking for CFG parsing and MT reranking.

To sum up, we have accomplished the following achievements.

- A new formalism, LTAG-spinal, which is weakly equivalent to LTAG.
- An LTAG-spinal Treebank extracted from the PTB with the Propbank annotation.
- A left-to-right incremental parser for LTAG-spinal.
- A bidirectional LTAG-spinal dependency parser.
- A novel graph-based incremental construction algorithm, which could be applied to many structure prediction problem in NLP, e.g. semantic role labeling.
- A novel discriminative reranking algorithm, ORUM, which has been successfully applied to parse reranking as well as other tasks, e.g. MT reranking.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Lexicalized Tree Adjoining Grammar (LTAG) (Joshi and Schabes, 1997) is a grammar which has attractive properties from the point of view of Natural Language Processing (NLP). LTAG has appropriate generative capacity and a strong linguistic foundation. Processing over deep structures in the LTAG representation leads to better understanding of natural language. Analysis with a well studied grammar like LTAG provides an integrated method of natural language processing.

Statistical methods, especially discriminative learning algorithms, have been successfully used in NLP. However, most of the current research on statistical NLP is focused on shallow syntactic analysis, due to the difficulty of modeling deep analysis with basic statistical learning algorithms.

In this work, we apply statistical learning algorithms to LTAG-based analysis as an effort toward statistical analysis over deep structures. We are especially interested in LTAG parsing, which is a well known hard problem due to its computational complexity. Statistical methods successfully applied to LTAG parsing can be applied to many other structure prediction problems in NLP.

For the purpose of achieving accurate and efficient statistical LTAG parsing, we will investigate two aspects of the problem, the structure and the algorithm.

Structure

TAG parsing is a well known difficult problem. The time complexity of TAG parsing is $O(n^6)$, where n is the length of the sentence. Variants of TAG have been proposed to reduce the complexity. For example, Schabes and Waters (1995) introduced the Tree Insertion Grammar (TIG) which is $O(n^3)$ parsable. However, TIG is weakly equivalent to Context-Free Grammar since it does not allow *wrapping adjunction*.

In Chapter 2, we will propose LTAG-spinal, a variant of LTAG with very desirable linguistic, computational and statistical properties. It can be shown that LTAG-spinal with adjunction constraints is weakly equivalent to LTAG. We extract an LTAG-spinal treebank from the Penn Treebank (Marcus et al., 1994) with Propbank annotation (Palmer et al., 2005), as described in Chapter 5.

Algorithm

We apply discriminative learning to LTAG based analysis in two ways, with parsing and with reranking.

In Chapter 3, we will propose a left-to-right incremental parser for LTAG-spinal. In this approach, we solve supertagging and dependency analysis problems in parallel. A perceptron like learning algorithm (Collins and Roark, 2004) is employed in training.

In Chapter 4, we will propose a bidirectional incremental parser to obtain LTAG-spinal dependency. We model the bidirectional parsing as a search problem, in which hypotheses of outside partial parses are utilized as features to compute the score of a partial parse. Instead of giving an algorithm specially designed for parsing, we propose a novel perceptron like learning algorithm generalized for graphs.

We also explore the discriminative reranking approach which could be used to improve the performance of parsing and other NLP problems, e.g. machine translation. In Chapter 6, we will propose a novel reranking algorithm based on ordinal regression with uneven margins.

In the rest of this chapter, we will first provide a short introduction to the LTAG formalism in Section 1.2. We will introduce the LTAG terminologies to be used later in this dissertation. More details of LTAG are available in a relatively recent paper by Joshi and Schabes (1997).

In Section 1.3, we will briefly describe the learning methods used in NLP learning tasks, labeling and structure prediction, which could be applied to LTAG based analysis. We will provide a short introduction to the Perceptron algorithm. Most of the learning methods proposed in dissertation are based on the Perceptron algorithm.

1.2 Lexicalized Tree Adjoining Grammar

Tree Adjoining Grammar (TAG) was first introduced in (Joshi et al., 1975). A recent review of TAG is given in (Joshi and Schabes, 1997), which provides a detailed description of TAG with respect to linguistic, formal, and computational properties. In this section, we will briefly describe the TAG formalism and the relation to linguistics for the sake of completeness.

1.2.1 Formalism

In LTAG, each word is associated with a set of *elementary trees*. Each elementary tree represents a possible tree structure for the word [1].

There are two kinds of elementary trees, *initial trees* and *auxiliary trees*. Elementary trees can be combined through two operations, *substitution* and *adjunction*.

Substitution is used to attach an initial tree, and adjunction is used to attach an auxiliary tree. In addition to standard adjunction, we also use *sister adjunction* as defined in the statistical LTAG parser described in (Chiang, 2000) [2].

[1]An elementary tree may have more than one lexical items.

[2]Adjunction is used in the case where both the root node and the foot node appear in the Treebank tree. Sister adjunction is used in generating modifi er sub-trees as sisters to the head, e.g in base NPs.

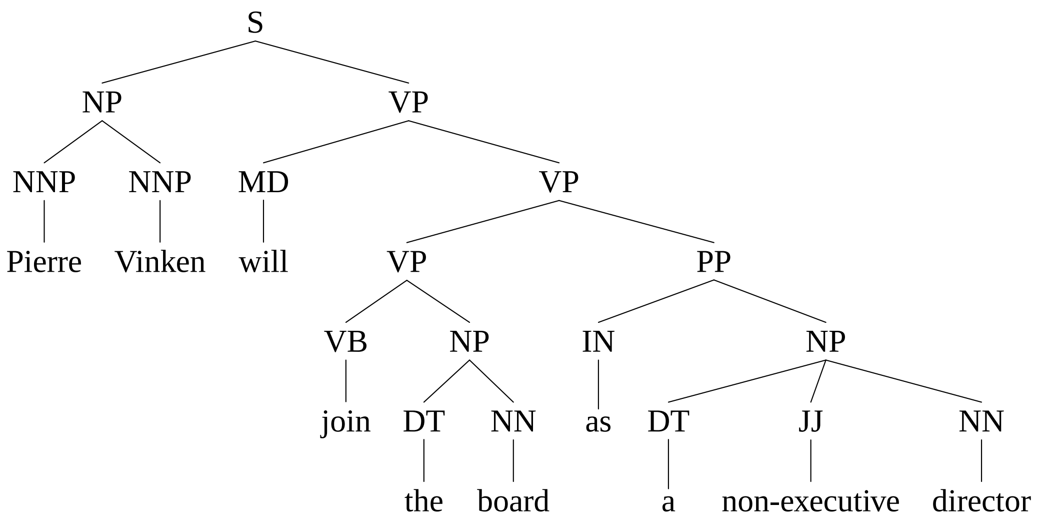


Figure 1.1: Derived tree (parse tree)

The tree resulting from the combination of elementary trees is is called a *derived tree*. The tree that records the history of how a derived tree is built from the elementary trees is called a *derivation tree* [3].

1.2.2 An Example

We illustrate the LTAG formalism with an example.

- Pierre Vinken will join the board as a non-executive director.

The derived tree for the example is shown in Fig. 1.1. Fig. 1.2 shows the elementary trees for each word in the sentence. Fig. 1.3 is the derivation tree. α stands for an initial trees, and β stands for an auxiliary tree.

One of the properties of LTAG is that it factors recursion in clause structure from the statement of linguistic constraints, thus making these constraints strictly local. For example, in the derivation tree, $\alpha_1(join)$ and $\alpha_2(Vinken)$ are directly connected no matter if there is an auxiliary tree $\beta_2(will)$ or not.

[3]Each node $\eta\langle n\rangle$ in the derivation tree is an elementary tree name η along with the location n in the parent elementary tree where η is inserted. The location n is the Gorn tree address (see Fig. 1.4).

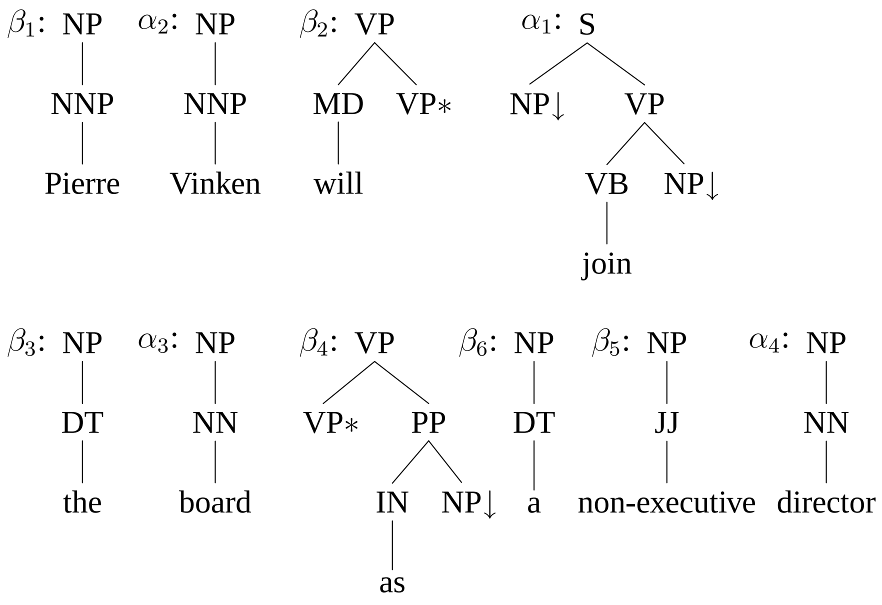


Figure 1.2: Elementary trees.

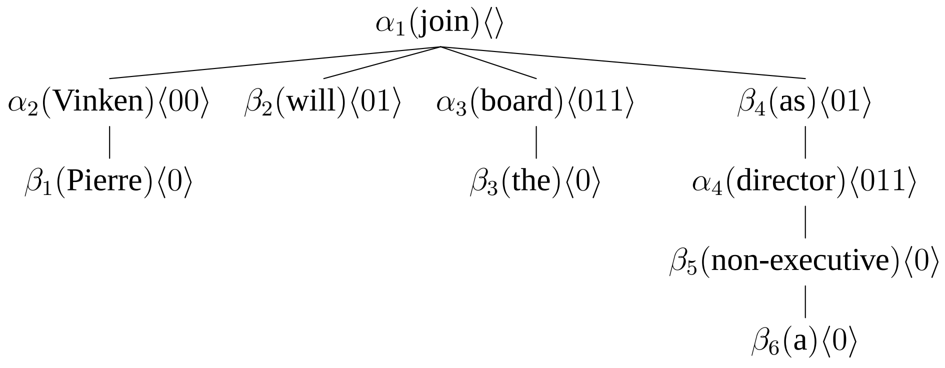

α_1(join)$\langle\rangle$
α_2(Vinken)$\langle 00\rangle$
β_2(will)$\langle 01\rangle$
α_3(board)$\langle 011\rangle$
β_4(as)$\langle 01\rangle$
β_1(Pierre)$\langle 0\rangle$
β_3(the)$\langle 0\rangle$
α_4(director)$\langle 011\rangle$
β_5(non-executive)$\langle 0\rangle$
β_6(a)$\langle 0\rangle$

Figure 1.3: Derivation tree: shows how the elementary trees shown in Fig. 1.2 can be combined to provide an analysis for the sentence.

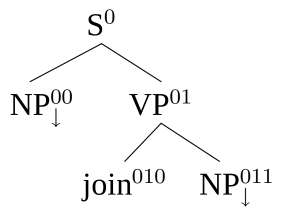


Figure 1.4: Example of how each node in an elementary tree has a unique node address using the Gorn notation. 0 is the root with daughters 00, 01, and so on recursively, e.g. first daughter 01 is 010.

1.2.3 Properties of LTAG

Compared with Context Free Grammar (CFG), TAG is mildly context-sensitive (Joshi, 1985), which means

- TAG can be parsed in polynomial time ($O(n^6)$).
- TAG has linear growth property.
- TAG captures nested dependencies and limited kinds of crossing dependencies.

There exist non-context-free languages that can be generated by a TAG. For example, it can be shown that $L_4 = \{a^n b^n c^n d^n\}$ can be generated by a TAG, but it is not a context-free language. On the other hand, it can be shown that $L_5 = \{a^n b^n c^n d^n e^n\}$ is not a tree adjoining language (Vijay-Shanker, 1987), but it is a context-sensitive language. It follows that $\mathcal{L}(CFG) \subset \mathcal{L}(TAG) \subset \mathcal{L}(CSG)$.

Because TAG has a stronger generative capacity than CFG, we can use TAG to represent many structures that cannot be represented with CFG, mainly due to the adjunction operation. There follow the two key properties of LTAG:

- Extended Domain of Locality (EDL), which allows
- Factoring Recursion from the domain of Dependencies (FRD), thus making all dependencies local (Joshi and Schabes, 1997).

It is claimed in (Frank, 2002) that these two properties reflect *the fundamental TAG hypothesis: Every syntactic dependency is expressed locally within a single elementary tree*.

It is shown in (Kroch and Joshi, 1985) and (Frank, 2002) that a variety of constraints on transformational derivations can be nicely represented with TAG derivation without any stipulation on the TAG operations. This property is related to the fact that TAGs are not as strong as context sensitive grammars, which allow too much flexibility for natural language description.

1.3 Discriminative Learning with Perceptron

1.3.1 Discriminative Learning in NLP

Learning tasks in NLP are more complicated than the basic machine learning problems like classification and regression. In NLP research, we are usually required to solve a set of basic problems together, and the results of the individual problems are related to other problems with respect to some underlying structure. This is usually called *structured learning*.

Two classes of structured learning problems are well studied in NLP literature. One is the labeling problem, such as POS tagging. The other is structure prediction, such as various parsing problems.

In recent years, discriminative learning algorithms have been successfully applied to these two classes of problems. Collins (2004) provided a detailed tutorial on using discriminative learning in NLP, especially in the parsing problems.

Three different types of strategies have been employed in structured learning[4].

[4]This view of classifi cation was pointed out to me by Prof. Fernando Pereira.

- **Type 1**: Parameters are learned in the gold-standard context only. Algorithms included in this class include Maximum Entropy Markov Model (MEMM) and Projection based Markov Model (PMM) (Punyakanok and Roth, 2001). For example, in the Ratnaparkhi's MEMM parser, the history-related features are always extracted from the gold standard parse. A potential problem with this approach is the so called label bias problem (Bottou, 1991; Lafferty et al., 2001).

  Punyakanok et al. (2005) showed that an extra step of inference over the output of those locally trained classifiers could greatly improve the overall performance in some applications.

- **Type 2**: Parameters are learned in all possible contexts. Algorithms included in this class are Conditional Random Fields (CRF) (Lafferty et al., 2001), Max-Margin Markov Models (MMMN) (Taskar et al., 2003), as well as Collins' Perceptron learning algorithm (2002) and its variant on parsing (McDonald et al., 2005). For example, in CRF, we compute all the possible pairs of labels for nearby nodes.

  In this way, inference is incorporated in the training. So it is a more direct model compared to Type 1. However, context in Type 2 is limited to a rather simple form due to the constraint of complexity.

- **Type 3**: Parameters are learned in partial contexts. This type can be viewed as a balance between the previous two types. Algorithms included in this class are the Perceptron like learning algorithms proposed in (Collins and Roark, 2004) and (Daumé III and Marcu, 2005). In these algorithms, beam search is employed to maintain a set of possible context settings, while inference is still incorporated in training.

  In this way, we can use rich structures as context. However, we can no longer run a precise inference over rich structures, but the hope is that the partial context used in training is closely related to the competing incorrect hypotheses that we will face in test.

Considering the rich structure that we have to use with the LTAG formalism, we will take the third approach in our research.

1.3.2 Perceptron Algorithms

Our learning algorithms to be proposed in this dissertation are mainly based on the previous research on Perceptron learning, as reported in (Collins, 2002; Collins and Roark, 2004; Daumé III and Marcu, 2005). Therefore, we will provide a short introduction to the Perceptron algorithm. The reader may refer to a machine learning textbook, e.g. (Bishop, 1996), for more details of perceptron learning. Here, we will only cover the content that will be used later in this dissertation.

Rosenblatt's Perceptron

Perceptron is a kind of binary classifier proposed by Rosenblatt (1958). It first maps a real valued vector input $\mathbf{x}$ to a real number via linear function

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b,$$

where $\mathbf{w}$ is the weight vector, and b is the bias. The sign of $f(\mathbf{x})$ stands for a positive or a negative sample.

Given a set of independently and identically distributed (*iid*) training samples $\{\mathbf{x}_i, y_i\}$, $y_i \in \{+1, -1\}$, according to a given distribution $\mathcal{D}$, the perceptron algorithm learns the weight vector and the bias with Algorithm 1.

Suppose $\mathbf{x}_i =< x_1, x_2, ..., x_k >$ is a k-dimension vector. Let $\mathbf{u}_i =< x_1, x_2, ..., x_k, 1 >$, $\mathbf{v} =< w_1, w_2, ..., w_k, b >$. Then we have

$$\mathbf{w} \cdot \mathbf{x}_i + b = \mathbf{v} \cdot \mathbf{u}_i$$

By redefining samples in this way, we can incorporate the bias into the weight vector, which means we can always assume the bias as zero by using the augmented vector as samples.

Algorithm 1 Perceptron learning algorithm

```
Require: ${(\mathbf{x}_i, y_i), i = 1..n}$
 1: $\mathbf{w}^0 \leftarrow \mathbf{0}; b^0 \leftarrow 0; t \leftarrow 0$
 2: repeat
 3:   for $(i = 1, \ldots, n)$ do
 4:     if $(sign(\mathbf{w}^t \cdot \mathbf{x}_i + b^t) \neq y_i)$ then
 5:       $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t + y_i \mathbf{x}_i$
 6:       $b^{t+1} \leftarrow b^t + y_i$
 7:       $t \leftarrow t + 1$
 8:     end if
 9:   end for
10: until no updates made in the for loop
11: return $< \mathbf{w}^t, b^t >$
```

Novikoff's Bound for Separable Data

Novikoff (1962) shows that the Perceptron learning stops in finite steps if the training data is separable.

Theorem 1.1 *Suppose we have the training data* $S = \{(\mathbf{x}_i, y_i), i{=}1,,n\}$. *Let* $R = \max_i ||\mathbf{x}_i||$. *If there exists a weight vector* $\mathbf{w}^*$, $||\mathbf{w}^*|| = 1$, *such that* $\forall i, \lambda_i = y_i(\mathbf{w}^* \cdot \mathbf{x}_i) \geq \lambda^*$, *Then the Perceptron learning algorithm makes at most* $(\frac{R}{\lambda^*})^2$ *updates.*

Perceptron with Margin

There are variants of the Perceptron learning algorithm which could output a weight vector that separates the training data with a large margin proportional to the optimal margin. Krauth and Mezard (1987) proposed a large margin Perceptron that updates the weight vector if the current weight vector cannot separate the samples in the training set by a margin c. Let λ^* be the optimal margin. It can be shown that the output weight vector separates the samples in training set with a margin λ_c, such that

$$\lambda_c \geq \frac{1}{2 + \frac{1}{c}} \lambda^* \tag{1.1}$$

Therefore, if c is large enough, the output margin could be close to half of the optimal margin.

Crammer and Singer (2003) proposed a variant of Perceptron called the Margin Infused Relaxed Algorithm (MIRA), in which a margin related hinge loss is used in the condition check of weight update, in line 4 of Algorithm 1. A *hinge loss* with margin parameter λ is defined as

$$L_\lambda(\mathbf{w};(\mathbf{x},y)) = \max\{0, \lambda - y(\mathbf{w}\cdot\mathbf{x})\} \tag{1.2}$$

The wight vector is updated if $L_\lambda(\mathbf{w};(\mathbf{x},y)) > 0$. The update rule is as follows.

$$\begin{aligned} \mathbf{w}^{t+1} &= \arg\min_{\mathbf{w}} \frac{1}{2}||\mathbf{w}-\mathbf{w}^t||^2 \\ s.t. &\quad L_\lambda(\mathbf{w};(\mathbf{x},y)) = 0 \end{aligned} \tag{1.3}$$

In this way, we separate the current sample with a large margin. This updating strategy is call *aggressive*, because the updated weight vector can always assign the correct label to the current sample.

Inseparable Data

Theorem 1.1 only provides an upper bound of updates if the training data is separable. The following theorem in (Freund and Schapire, 1999) provides an upper bound even when the training data is inseparable.

Theorem 1.2 *Suppose we have the training data* $S = \{(\mathbf{x}_i,y_i), i{=}1,,n\}$. *Let* $R = \max_i ||\mathbf{x}_i||$. *Let* $\mathbf{w}$ *be any vector with* $||\mathbf{w}|| = 1$ *and let* $\lambda > 0$. *Define* $D = (\sum_1^n L_\lambda(\mathbf{w};(\mathbf{x}_i,y_i)))^{\frac{1}{2}}$. *Then the Perceptron learning algorithm makes at most* $(\frac{R+D}{\lambda})^2$ *updates in one iteration over the training data.*

A variant of the Perceptron learning algorithm proposed in (Freund and Schapire, 1999) under this framework is called **voted Perceptron**. The difference with the original Perceptron algorithm is that, for prediction, we use the averaged weight vector instead of the final result.

Another method to handle inseparable data is to make the samples artificially separable. This method is called **λ-trick** in (Herbrich, 2002). For each sample, we insert a

Algorithm 2 Collins' Perceptron learning algorithm

```
Require: $\{(x_i, y_i), i = 1..n\}$
1: $\mathbf{w}^0 \leftarrow \mathbf{0}; t \leftarrow 0$
2: for $(r = 1, .., T; i = 1, .., n)$ do
3:     Calculate $z_i = \arg\max_{z \in \mathbf{GEN}(x_i)} \mathbf{w}^t \cdot \Phi(x_i, z)$
4:     if $(z_i \neq y_i)$ then
5:         $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t + \Phi(x_i, y_i) - \Phi(x_i, z_i)$
6:         $t \leftarrow t + 1$
7:     end if
8: end for
9: return $\mathbf{w}^t$
```

unique dimension for this sample, and assign a small number $\sqrt{\lambda}$ to this dimension. If λ is large enough, the augmented samples are always separable.

Collins' Perceptron

Collins (2002) proposed a perceptron like learning algorithm combined with Viterbi decoding. This algorithm and its variants have been widely used in labeling and structure prediction in NLP. Many of our contributions in this dissertation are based of this work. For the sake of completeness, we list the pseudo code of Collins' Perceptron learning in Algorithm 2.

In this algorithm, function **GEN** enumerates a set of candidates $\mathbf{GEN}(x)$ for an input x by Viterbi decoding. This algorithm was justified in a way similar to Theorem 1.2 in (Collins, 2002). (Collins and Roark, 2004) and (Daumé III and Marcu, 2005) extended this algorithm by employing non-exhaustive search strategy, as described in Section 1.3.1, which can be useful in rich-context applications.

Chapter 2

LTAG-spinal

In this chapter, we introduce LTAG-spinal, a variant of LTAG with very desirable linguistic, computational and statistical properties. It can be shown that LTAG-spinal with adjunction constraints is weakly equivalent to the traditional LTAG. In Chapter 5, we will present an LTAG-spinal treebank, and this treebank will be used to train and evaluate two LTAG-spinal style parsers in Chapters 3 and 4.

In Section 2.1, we will describe the motivation for the LTAG-spinal formalism. We will provide the definition in Section 2.2, and illustrate it with an example in Section 2.3.

2.1 Motivation

2.1.1 Linguistics

Argument-adjunct ambiguity is both a theoretical as well as a computational problem. In the XTAG English grammar (XTAG-Group, 2001), arguments are treated as substitution while adjuncts are handled as adjunction. Substitution is obligatory, while adjunction is optional.

However, in building an LTAG treebank automatically extracted from other resources like the Penn Treebank and Propbank, it becomes a big problem to distinguish arguments from adjuncts. In LTAG, arguments refer to obligatory constituents only, but there is no way to obtain this information directly for other resources In previous works, heuristic rules were used to distinguish arguments from adjuncts. However, it turns out to be a non-trivial task to map those automatically generated template to the XTAG elementary trees.

Therefore, we are searching for a framework, under which the representations for arguments and adjuncts are similar. In this way, we can encode the ambiguity with a single structure, and leave the disambiguation for post-processing.

Our solution is the *sister adjunction* like operation, which was previously proposed to represent adjuncts in (Chiang, 2000) for Tree Insertion Grammar (Schabes and Waters, 1995). We propose to use sister adjunction for both arguments and non-predicate adjuncts[1], as a method of encoding argument-adjunct ambiguity. The *domain of locality* of LTAG is still maintained in a way that syntactically dependent arguments are directly attached to the predicate.

As a result, elementary trees are in the so-called spinal form since arguments do not appear in the elementary tree of the predicate. This turns out to be a great advantage in handling coordination. In the traditional LTAG, one needs to transform the templates of predicate conjuncts in order to represent the shared arguments. However, representation of predicate coordination is rather easy with the spinal form.

2.1.2 Generative Capability

When Tree Adjoining(Adjunct) Grammar was first introduced in (Joshi et al., 1975), the only composition operation was adjunction, and there was no substitution. The fact that the addition of substitution does not change the generative capability leads us to combining substitution and non-wrapping adjunction.

[1]By *non-predicate* adjuncts, we mean the auxiliary trees whose anchor is not a predicate, i.e., whose foot node does not subcategorize for the anchor. In the XTAG English Grammar, non-predicate adjuncts are always non-wrapping.

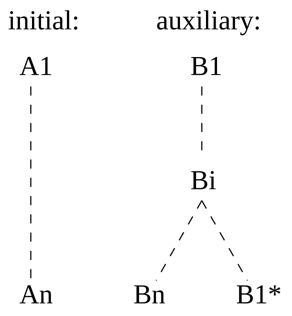


Figure 2.1: Spinal elementary trees

2.1.3 Statistical Processing

As we have noted, it is difficult to obtain the corresponding XTAG template form an automatically extracted template. On the other hand, the complexity of using those automatically extracted templates in parsing is greatly increased. According to the *coarse to fine* spirit, it is attractive to use some structure to encode these templates, so as to decrease of the degree of perplexity at each step of parsing. We have proposed LTAG-spinal for this purpose also.

2.2 Formalism

In LTAG-spinal, we have two kinds of elementary trees, initial trees and auxiliary trees, as shown in Figure 2.1. What makes LTAG-spinal different is that elementary trees are in the spinal form. A spinal initial tree is composed of a **lexical spine** from the root to the anchor, and nothing else. A spinal auxiliary tree is composed of a lexical spine and a **recursive spine** from the root to the foot node. The common part of a lexical spine and a recursive spine is called the **shared spine** of an auxiliary tree. For example, in Figure 2.1, the lexical spine for the auxiliary tree is $B_1,..,B_i,..,B_n$, the recursive spine is $B_1,..,B_i,..,B_1^*$, and the shared spine is $B_1,..,B_i$.

There are two operations in LTAG-spinal, which are **adjunction** and **attachment**. Adjunction in LTAG-spinal is the same as adjunction in the traditional LTAG. Attachment stems from sister adjunction in Tree Insertion Grammar. By attachment, we take the root of an initial tree as a child of a node of another spinal elementary tree.

Attachment can be viewed as a special adjunction. We can add artificial root and foot nodes to an initial tree to build an auxiliary tree, and simulate the attachment of an initial tree by a non-wrapping adjunction of the artificial auxiliary tree as in TIG. On the other hand, attachment is close to substitution since it will not generate a non-projective dependency. Further, attachment does not require the node for substitution on the parent tree. However, the flexibility of attachment can be restrained by attachment constraints which is similar to adjunction constraints in the traditional LTAG (Joshi and Schabes, 1997).

Since attachment can be viewed as a special adjunction, we still call these constraints as **adjunction constraints**. There could be three types of constraints, which are null adjunction (NA), obligatory adjunction(OA) and selective adjunction(SA). It can be shown that LTAG-spinal with adjunction constraints is weakly equivalent to LTAG, which means that they generate the same set of string languages.

Theorem 2.1 *LTAG-spinal and LTAG are weakly equivalent.*

The proof of Theorem 2.1 is given in the Appendix A.

It should be noted that, in practice, the foot node is usually the only node which is not on the lexical spine of an auxiliary tree. In the LTAG-spinal treebank described in Chapter 5, the foot node of an auxiliary tree is always a child of the lowest node in the shared spine. In this case, an auxiliary tree can only adjoin onto the lexical spine of another elementary tree. We call this the **spinal adjunction** property, and we will employ this attribute in the parsers proposed in the next two chapters.

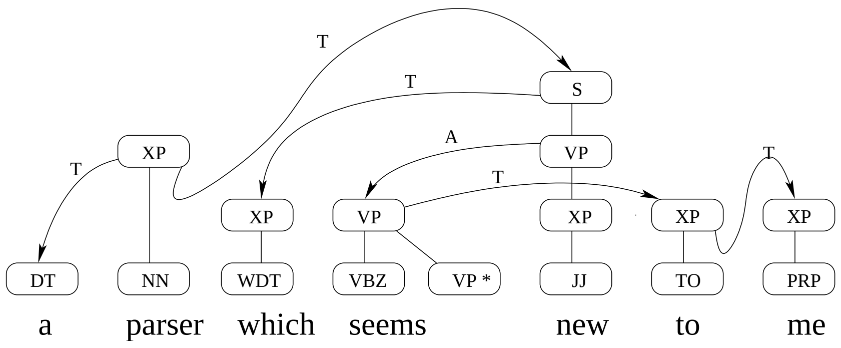


Figure 2.2: An example in LTAG-spinal.

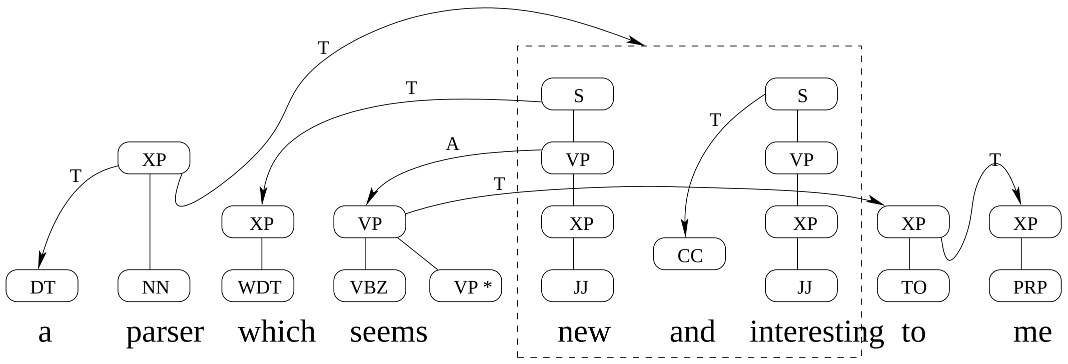


Figure 2.3: An example of predicate coordination.

2.3 Examples

An example of the LTAG-spinal derivation tree is shown in Figure 2.2.

Two types of operations are used to connect the elementary trees into a derivation tree, which are *attachment* and *adjunction*. In Figure 2.2, each arc is associated with a label which represents the type of operation. We use **T** for aTtach and **A** for Adjoin.

The *adjunction* operation can effectively do wrapping, which distinguishes it from *sister adjunction*. In Fig. 2.2, *seems* adjoins to *new* as a *wrapping adjunction*, which means that the leaf nodes of the adjunct subtree appear on both sides of the anchor of the main elementary tree in the resulting derived tree. Here, *seems* is to the left of *new* and *to me* is to the right of *new*.

In addition, it is also a *spinal adjunction*, because *seems* adjoins onto the VP node, which in on the lexical spine of the spinal elementary tree for *new*.

In the extracted LTAG-spinal treebank reported in Chapter 5, we will represent predication coordination explicitly, as shown in Figure 2.3. This representation is similar to the *conjoin* operation proposed in (Sarkar and Joshi, 1996). However, here predicate coordination applies to spinal elementary trees, so we do not need to specify shared argument as in (Sarkar and Joshi, 1996).

Predicate coordination over spinal template provides a concise way to encode the ambiguity of shared arguments. For example, in Fig. 2.3, *a parser*, *which* and *seems* are shared by both predicates, although *which* and *seems* attach to the left conjunct only. The explicit representation of predicate coordination indicates possible sharing on the deep level.

Chapter 3

Incremental LTAG-spinal Parsing

The idea of incremental parsing with LTAG is closely related to the work on Supertagging (Joshi and Srinivas, 1994). A supertagger first assigns the correct LTAG elementary tree to each word. Then a Lightweight Dependency Analyzer (LDA) (Srinivas, 1997) builds the whole derivation tree with these elementary trees.

Here, we will explore left-to-right incremental parsing, as a method of incorporating the supertagger and the LDA dynamically.

There is a strong connection between incremental parsing and psycholinguistics, and this connection is observed with respect to the LTAG formalism, as shown in (Sturt and Lombardo, 2005).

In Section 3.1, we will describe the previous works on incremental parsing. In Section 3.2, we will propose an incremental parser for LTAG-spinal. We will illustrate the parsing algorithm with an example in Section 3.3. We will formalize LTAG-spinal parsing in Section 3.4. In Section 3.5, we will describe the learning algorithm and the features used in this parser. We will describe the experiments on the LTAG-spinal treebank in Section 3.6.

3.1 Previous Works

In recent years, there have been many interesting works on incremental parsing or semi-incremental parsing. By semi-incremental we mean the parsers that allow several rounds of left to right scans instead of one.

In Ratnaparkhi's MEMM parser (1997), three rounds of scans were used, one each for POS tagging, NP chunking and shift-reduce parsing respectively. Roark (2001) introduced a probabilistic top-down parsing model which maintained a partial parse tree as the left context. Prolo (2003) proposed a shift-reduce parser for LTAG. Recently, Henderson (2003) and Collins and Roark (2004) compared generative versus discriminative approaches for statistical left-corner parsing. All these incremental parsers described above employed the left-corner strategy.

Some other incremental parsers are more like head-corner parsers such as those proposed in (Henderson, 2000) and (Yamada and Matsumoto, 2003). The SSN parser presented in (Henderson, 2000) is closely related to Dependency Grammar; Each constituent is associated with a head word, and the tree structure is rather flat. In particular, it does not distinguish between an *S* node and its child *VP* node, the two constituents having been collapsed. The parser proposed in (Yamada and Matsumoto, 2003) used a multi-scan bottom-up strategy to build a dependency tree. In each left to right scan, some of the treelets were combined. The iteration stopped when there was only one tree.

The head-corner approach is more natural to the LTAG formalism. We use a stack of derivation treelets to represent the partial parsing results. Furthermore, the LTAG formalism allow us to handle non-projective dependencies, which cannot be generated by a CFG or a CFG-based dependency parser.

The model of incremental LTAG parsing is similar to Structured Language Modeling (SLM) in (Chelba and Jelinek, 2000; Xu et al., 2002). In SLM, the left context of the history is represented with a stack of binary trees. At each step, one computes the conditional probability of the current word, its tag and potential operations over the new context trees. As far as LTAG parsing is concerned, the difficulty is that we need to handle the adjoin operation which gives rise to non-projective structures. Furthermore, the training criteria are different, since we are interested in the quality of the top candidate parse instead of perplexity.

3.2 Incremental LTAG-spinal Parsing Algorithm

There are four different types of operations in our parser. Two of them, attachment and adjunction, are described in the previous section. The third operation is **conjunction**, which is a special adjunction operation designed to implement incremental construction of predicate coordination. The last one is **prediction**, which is used to predict a possible spine for a given word according to the context and lexicon.

Our left to right parsing algorithm is a variant of the shift-reduce algorithm with beam-search. We use a stack of disconnected derivation treelets to represent the left context. When the parser reads a word, it first *predicts* a possible spinal elementary trees for this word, For each elementary tree, we first push it on the stack. Then we recursively pop the top two treelets from the stack and push the combined tree into the stack until we choose not to combine the top two treelets with one of the three combination operations (we can also choose not to pop anything at the beginning). Then we shift to the next word. We call this model the **Flex Model**.

A potential problem with the Flex Model is that a single *LTAG derivation* tree can be generated by several *shift-reduce derivation* steps, which only differ in the order of operations. For example, let *A*, *B* and *C* be three trees. In an LTAG derivation, *A* adjoins to *B*, and *B* adjoins to *C*. Then we have two different shift-reduce derivations, which are $(A \rightarrow (B \rightarrow C))$ and $((A \rightarrow B) \rightarrow C)$. This is similar to spurious ambiguity in CCG parsing.

Now we introduce the **Eager Model**, an eager evaluation strategy; For any two elementary trees which are directly connected in the LTAG derivation tree, they are combined immediately when they can be combined in the first context under which they can be combined. Furthermore, they cannot be combined afterward, if they miss the first chance.

This can be implemented by memorizing all the spine pairs that have been checked. In the previous example, the parser will generate $((A \to B) \to C)$, while $(A \to (B \to C))$ is ruled out. Then for each LTAG derivation tree, there exists a unique left-to-right derivation.

The Eager Model is motived by the treatment of coordination in (Sturt and Lombardo, 2005). For example, we have the following two sentences.

1. He knows Ben likes philly steak.

2. He knows Ben likes philly steak and Jane likes pizza.

Suppose we are parsing these two sentences, and for each case the current word is *likes*. Now we have just the same local contexts for both cases. According to the Eager Model, the parser takes the same action according to the context, which is to combine the *knows* tree and the *likes* tree. For sentence 2, the second *likes* tree will be conjoined with the first *likes* tree later.

In Algorithm 3, we present the left-to-right parsing algorithm.

In the following section, we will explain the parsing mechanism for the Eager Model with an example. The Flex Model is similar except that the order of operations is flexible to some extent.

3.3 An Example

Figure 3.1 shows the left to right parsing of the phrase *a parser which seems new and interesting to me* with the Eager Model.

In Figure 3.1, each arc is associated with a number and a label. The number represents the order of operation, and the label stands for the type of operation as in Figure 2.3. Furthermore we use **P** to represent predict, and **C** for conjoin.

In steps 1 and 2, two disconnected spines are predicted for *a* and *parser*. The spine for *a* is attached to the spine for *parser* on the XP node in step 3. Here we use XP instead of NP to handle the noise in PTB. This applies to all the second level projections except VP.

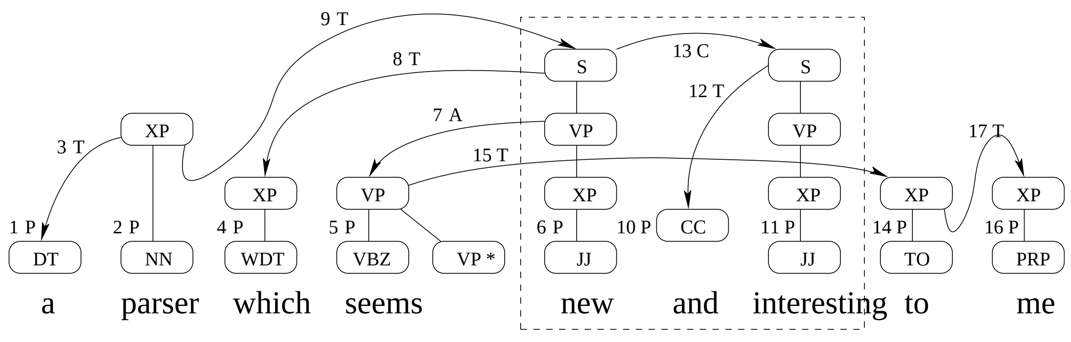
Algorithm 3 Left-to-right Parsing

```
 1: context set $C \leftarrow \{\varepsilon\}$;
 2: new context set $C' \leftarrow \phi$;
 3: for (word $w \leftarrow w_1, \ldots, w_n$) do
 4:     for (context $c \in C$ and candidate spine $s$ for $w$) do
 5:         new context $c' \leftarrow$ shift($c$, $s$);
 6:         $C' \leftarrow C' \cup \{c'\}$;
 7:     end for
 8:     $C \leftarrow C'$;
 9:     $C' \leftarrow \phi$;
10:     repeat
11:         remove context $c$ from $C$;
12:         $C' \leftarrow C' \cup \{c\}$;
13:         for (candidate operation $p$) do
14:             new context $c' \leftarrow$ reduce($c$, $p$);
15:             $C \leftarrow C \cup \{c'\}$;
16:         end for
17:     until ($C = \phi$)
18:     $C \leftarrow top_k(C', k)$;
19:     $C' \leftarrow \phi$;
20: end for
```

In step 6, the spine for *new*, the first conjunct of the predicate coordination, is predicted. Then the auxiliary tree for *seems* adjoins to the spine for *new* on node *VP*. the latter is further combined with *which*, and is attached to the tree for *parser*.

In step 13, the conjoin operation is used to combine the treelet anchored on *new* and the treelet anchored on *interesting*. Alignments between the two spines are built, through which argument sharing is implemented in an implicit and underspecified way.

In step 15, for the spine for *to*, the *visible* nodes of the conjoined treelet include nodes on some auxiliary trees adjoined on the left of the spines, like the root *VP* node for *seems*. In this way, wrapping adjunction is implemented. Obviously, it results in a non-projective dependency.



P: predict T: attach A: adjoin C: conjoin

Figure 3.1: Incremental parsing with the Eager Model.

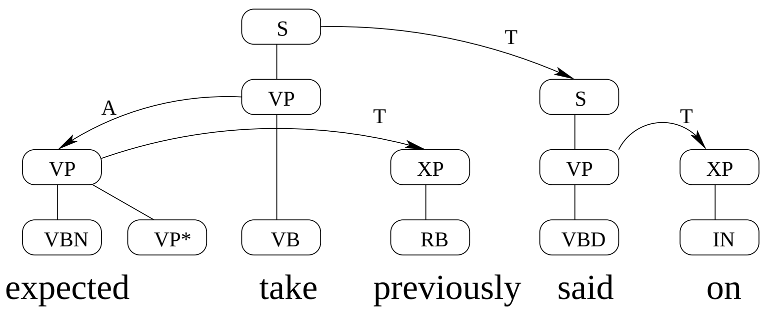


Figure 3.2: Partial derivation tree for the example of visibility

3.4 Formalism for Incremental Parsing

As shown in the example in the previous section, the left-to-right parsing is similar to the traditional shift-reduce parsing, while the definition of *visible* nodes is crucial to the left-to-right mechanism.

Before giving the formal definition of visibility, we illustrate the idea with a simple example. Suppose the input sentence is as follows.

- Graduate students were not expected to take ten courses previously, said the professor on the group meeting.

Figure 3.2 shows part of the derivation tree for the example. Here, *expected* adjoins to *take*. Suppose the top two trees in the stack anchor on *take* and *previously* respectively. In this case, we need to make *expected* visible to *previously*, due to the nature of the adjoin operation. In the corresponding derived tree, the root and foot nodes of the tree for *expected* insert into the spine for *take*, so they become visible to *previously*.

Then *previously* attaches to *expected*, and *said* attaches to *take*. Suppose the top two trees in the stack anchor on *take* and *on* respectively. In this case, *expected* becomes invisible to *on*, because *said* attaches to *take*, and it blocks *expected* from the right side.

Now, we present the formal definition of *visible* nodes.

Definition A **spinal elementary tree** e is a tuple $< nd, tp, lv, dr, lx, ps >$.

- nd is the vector of nodes on the spine, enumerated from the anchor to the root, $nd(1), ..., nd(nd.length)$, where $nd.length$ is the height of e. We use *root* to represent $nd(nd.length)$ for simplicity.

- tp is type of e, which can be α or β.

- lv and dr are only meaningful for the β tree. lv represents the index of the parent of the foot node in nd. $dr = left$ or $right$, which indicate whether the foot node is on the left or the right of the spine.

- lx is the lexical item of e.

- ps is the position of the lexical item in the flat sentence. ∎

For example, in Figure 3.1, the elementary tree for *new* is represented as

$$<< JJ, XP, VP, S >, \alpha, -, -, new, 5 >,$$

and the elementary tree for *seems* is represented as

$$<< VBZ, VP >, \beta, 2, right, seems, 4 > .$$

An LTAG-spinal derivation can be represented as a tree. However, it is more than a tree, since we need to also represent the type of operation that connects two elementary trees, e.g. attach, adjoin or conjoin, as well as the landing site on the main tree. So we use a *labeled tree* to represent a derivation tree.

Definition An **LTAG-spinal derivation tree** T is a tuple $< E, L, D, re >$.

- E is a set of elementary trees.
- L is a tuple $< op, st >$ which represents the type of the operation and the landing site.
- D is relation defined on $E \times L \times E$ which meets the following condition. Let D_U be the projection of D on $E \times E$, then we have $|D| = |D_U|$ and D_U is a *tree* relation.
- *re* is the root elementary tree in D_U. $\blacksquare$

Now we define some functions to be used in the definition of *visible* nodes.

Definition Let e be a subtree, we use $LF(e)$ to represent the position of the leftmost descendant of e. Formally, LF can be defined as follows

$$LF(e) = \min(e.ps, \min_{(e,e_x) \in D_U} LF(e_x))$$

Similarly, the position of the right most descendant, RT, is

$$RT(e) = \max(e.ps, \max_{(e,e_x) \in D_U} RT(e_x)) \quad \blacksquare$$

Now we are only one step away from defining *visible* nodes. Suppose we have two derivation treelets T_L and T_R on the top of the stack, T_L on the left and T_R on the right. We are searching for the nodes on T_L that are visible to T_R, which should be on the right side of T_L. On the other hand the nodes on T_R that are visible to T_L are on the left side of T_R. So we need to distinguish these two cases, although they are similar to each other.

We use VL, a binary feature, to represent whether a node on the right tree is visible to the left tree. and we use VR to represent whether a node on the left tree is visible to the right tree. The formal definition of these two are very similar. So here we only give the definition of VR. VL can defined easily according to VR.

Definition Let T_L and T_R be the top two partial derivation trees. For each node n in T_L, $VR(n) = true$ means n is **visible** to the right tree T_R. Now we give the rules to recursively search for all the nodes whose VR values are $true$. For all the rest nodes, their VR values are all $false$.

- **Initial Rule**

  $VR(T_L.re.root) = true$, which means the root node of the root elementary tree is visible from the right side.

- **Recursive Rule**

  if node $e.nd(i)$ is visible from right, $VR(e.nd(i)) = true$, we define e_L, the leftmost adjoined tree, and e_R the right most sub tree as follows

$$\begin{aligned} e_L &= argmin_{(e,<adjoin,i>,e_x)\in R \text{ and } e_x.ps<e.ps} e_x.ps \\ e_R &= argmax_{(e,<-,i>,e_x)\in R \text{ and } e_x.ps>e.ps} e_x.ps \end{aligned}$$

  Let $B_L = RT(e_L)$, the position of the right most descendant of e_L, if e_L exists, otherwise $B_L = 0$. Similarly, $B_R = RT(e_R)$, the position of the right most descendant of e_R, if e_R exists, otherwise $B_R = 0$.

  - if $B_L = B_R = 0$ and $i > 2$
    - if $e.tp = \alpha$, then $VR(e.nd(i-1)) = true$.
    - if $e.tp = \beta$ and ($e.dr = left$ or $e.lv < i$) then $VR(e.nd(i-1)) = true$.
  - if $B_L < B_R$, then $VR(e_R) = true$.
  - if $B_L > B_R$ or $0 < B_L < e.ps$, then $VR(e_L) = true$. ∎

The definition given above does not only define which nodes are *visible* nodes, but also provides an algorithm to find all the *visible* nodes in a partial tree.

The idea behind the recursive rule is that we recursively look for the next right-visible nodes in a top-down style. If there is no adjunction from the left side, where $B_L = 0$, then it is easy to handle. Otherwise, we need to decide whether this adjunct is still visible or not. For example, if there is a right attachment to this left conjunct and there is another attachment to the main tree, which is the right most, then any node on this left conjunct becomes invisible from the right side. Conditions like this are handled by B_L and B_R as described above.

3.5 Training

3.5.1 Training Algorithm

Many machine learning algorithms have been successfully applied to parsing, incremental parsing, or shallow parsing, which can be applied to our incremental parsing algorithm.

Ratnaparkhi (1997) used log-linear models in his incremental parser trained on PTB. Punyakanok and Roth (2001) proposed several models of using classifiers in sequential inference. Yamada and Matsumoto (2003) employed chained SVMs in their dependency parser. Lafferty et al. (2001) proposed the Conditional Random Fields that does not suffer from the label bias problem. The Max-Margin Markov Networks proposed in (Taskar et al., 2003) attacked the same problem with the max margin approach, and this method had been adapted to parsing successfully (Taskar et al., 2004). But due to limitation of computing resources, this method cannot be used to train on the whole PTB.

We use the perceptron-like algorithm proposed in (Collins and Roark, 2004). We also employ the voted perceptron algorithm (Freund and Schapire, 1999) and the *early update* technique as in (Collins and Roark, 2004).

3.5.2 Features

Features are defined in the format of (*operation, main spine, sub spine, spine node, context*), where spine node is the node on the main spine onto which the sub spine is *attached* or *adjoined*. For *predict*, sub spine and spine node are undefined, and for *conjoin* spine node is undefined. *context* is the field to describe the constituent label or lexical item of a certain node in the context. The context of an operation includes the top two treelets involved in the operation as well as the two closest words on both sides of the current word.

- Context for *predict* :
  - The (-2, 2) window in the flat sentence.
  - The *visible* [1] spines on the topmost treelet.

- Context for *attach* and *adjoin* :
  - The (0, 2) window in the flat sentence.
  - The most recent spine previously attached or adjoined to the same location on the main spine.
  - The leftmost child spine attached to the sub spines.
  - The spines that are *visible* before the operation and become *invisible* after the operation.

- Context for *conjoin* :
  - The (0, 2) window in the flat sentence.
  - The leftmost child spine attached to the main spine, which is the first adjunct.
  - The two leftmost children spines attached to the sub spine, which is the current adjunct.

We have about 1.4M features extracted from the gold-standard parses, and about 600K features dynamically extracted from the generated parses in 10 rounds of training with the Eager Model.

[1]By visible spines we mean the spines onto which the current spine can be attached or adjoined.

3.6 Experiments and Analysis

We use an LTAG-spinal treebank for experiments. The details of the treebank will be described in Section 5. The LTAG-spinal parses for the 39434 sentences extracted from WSJ section 2-21 are used as the training data. Section 24 is used as the development data. The 2401 sentences in section 23 are used for test [2].

We use syntactic dependency for evaluation. It is worth mentioning that, for predicate coordination, dependency is defined on the parent of the coordination structure and each of the conjunct predicate. For example, in Figure 2.3, we have dependency relation on (parser, new) and (parser, interesting). Compared to other dependency parsers on PTB, the dependency defined on LTAG-spinal reveals deeper relations because of the treatment of traditional adjoining and predicate coordination described above.

In the community of parsing, *labeled recall* and *labeled precision* on phrase structures are often used for evaluation. However, in our experiments we cannot evaluate our parser with respect to the phrase structures in PTB. As shown in Chapter 5, various tree irrecoverable transformations were used to extract the LTAG-spinal treebank according the Propbank annotation on PTB. Therefore, we use syntactic dependency for evaluation.

3.6.1 Eager vs. Flex

We first train our incremental parser with Eager Model and Flex Model respectively. In the training, beam width is set to 10. Lexical features are limited to words appearing for at least 5 times in the training data. Figure 3.3 and Figure 3.4 show the learning curve on the training and the development data. The X axis represents the number of iterations over the training data, and the Y axis represents the f-score of dependency with respected to the LTAG derivation tree.

In both cases, the voted weights provide an f-score which is more than 3% higher. The voted results converge faster and are more stable. The result with Flex Model is 0.6%

[2]The complete section 23 has 2416 sentences.

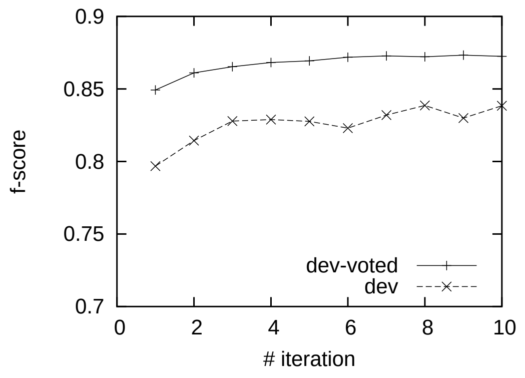


Figure 3.3: f-score of syntactic dependency on the development data with the Eager Model

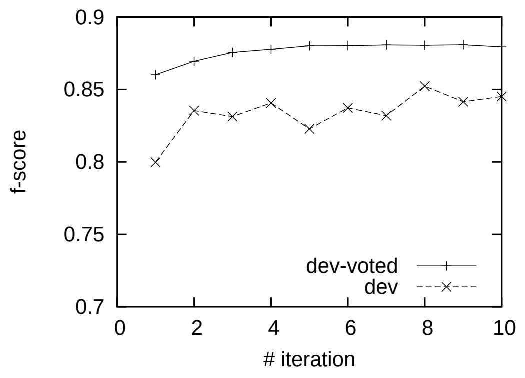


Figure 3.4: f-score of syntactic dependency on the development data with the Flex Model

higher than the one with Eager Model, but the parsing time is much longer with Flex Model as we will show later.

We use the voted weights obtained after 10 rounds of iteration for the evaluation on the test data. We achieve an f-score of **88.7%** on dependency with the Eager Model, and **89.3%** with the Flex Model.

3.6.2 K-Best Parsing

The next experiment is on K-best parsing. As a first attempt, we just use the same algorithm as in the previous section, except that we study the oracle parse, or the best parse, among the top 10 parses. The f-score on the oracle in top 10 in the development data is

Table 3.1: F-score of the oracle parse in the 10-best parses on the development data with the Eager Model

algorithm	f-score%
top (eager)	87.3
oracle (eager)	88.5
top (eager+combined parses)	87.4
oracle (eager+combined parses)	91.0

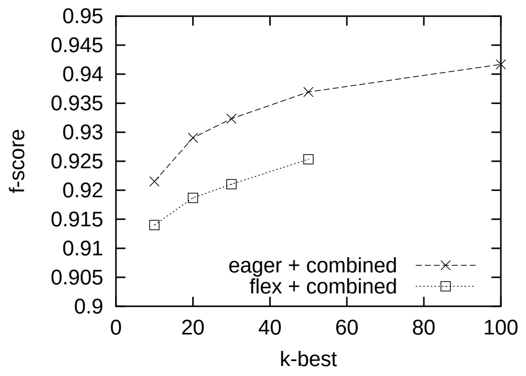


Figure 3.5: f-score of the K-Best oracle on the test data

88.5%, compared with 87.3% of top candidate, as shown in Table 3.1. However, we are not satisfied with the number on oracle, which is not good enough for post-processing, i.e. parse reranking.

We notice that from a single partial derivation we can generate a large set of different partial derivations, just by combining the elementary tree of the next word. It is easy to see that these similar derivations may use up the search beam quickly, which is not good for parse search. Many of the new derivations share the same dependency structure. So we revised our learning procedure by combining derivations with the same dependency structure before each shift operation. We repeated the K-best parsing experiments by using **Combined Parses** as described above on both the Eager and the Flex Models, and achieve significant improvement on the oracle, as shown in Table 3.1 and Figure 3.5.

Figure 3.5 shows the f-score of the oracle on K-best parsing using combined parses on the test data. For each K-best oracle test, we set the beam width to K in parsing. The f-score of oracle in 100-best parsing is **94.2%** with the Eager Model + Combined Parses.

Table 3.2: Speed of parsing on the test data set.

model	cp?	beam	sen/sec	f-score%
single best				top
flex	no	10	0.37	89.3
eager	no	10	0.79	88.7
K-best				oracle
eager	yes	10	0.62	92.2
eager	yes	20	0.31	92.9
eager	yes	30	0.22	93.2
eager	yes	50	0.13	93.7
eager	yes	100	0.07	94.2

3.6.3 Speed of Parsing

Efficiency is important to the application of incremental parsing. This set of experiments is related to the speed of our parser on single best and K-best parsing with both the Eager Model and the Flex Model. All the experiments are performed on a Linux node with two 1.13GHz PIII CPUs and 2GB RAM. The parser is coded in Java.

Table 3.2 shows that the Eager Model is more than two times faster than the Flex Model, as we expected. The time spent on K-best parsing is proportional to the beam width. In the table, cp? = whether the method of Combined Parses is used; sen/sec = sentence per second; top = top candidate given by the parser; oracle = oracle of the K-best parses where K equals the width of the beam.

In these experiments, we evaluated the parsing results with the LTAG treebank instead of the phrase structures in PTB, due to the fact that some transformations used to extract the LTAG treebank from PTB is non-reversible. Therefore, it is unfeasible for us to restore the original phrase structure as in PTB.

3.7 Summary

In this chapter, we have presented an efficient incremental parser for LTAG-spinal. As far as we know, it is the first comprehensive attempt of efficient statistical parsing with a formal grammar with provably stronger generative power than CFG, supporting the *adjoining* operation, dynamic predicate coordination, as well as non-projective dependencies.

The parser is trained with a perceptron like learning algorithm proposed in (Collins and Roark, 2004). We have evaluated our parser on the LTAG-spinal treebank, extracted from the Penn Treebank with Propbank annotation. Using gold standard POS tags as part of the input, the parser achieves an f-score of 89.3% for syntactic dependency on section 23 of PTB. Because of the treatment of adjunction and predicate coordination in the LTAG-spinal treebank, as shown in Chapter 5, these dependencies, which are defined on LTAG-spinal derivation trees, are deeper than the dependencies extracted from PTB alone with head rules.

Chapter 4

Bidirectional LTAG-spinal Dependency Parsing

In this chapter, we will propose a parser which searches for the LTAG-spinal dependency tree in both directions. By LTAG-spinal dependency, we mean the dependency relation of the words encoded in an LTAG-spinal derivation tree. It should be noted that LTAG-spinal dependency is *non-projective*, because we can represent *wrapping adjunction* with LTAG-spinal. Furthermore, due to the introduction of predicate coordination as shown in Fig. 2.3, it results in more ambiguity in recognition.

We will first describe the idea of our approach in Section 4.1. In Section 4.2, we will use a detailed example to illustrate our algorithm. The formal definition of the algorithms are described in Section 4.3. In Section 4.4, we will illustrate the details of the bidirectional LTAG-spinal dependency parsing. In Section 4.5, we will compare the novel algorithm to other related works, and in Section 4.6, we will report the experiments with dependency parsing on the LTAG-spinal treebank.

4.1 Idea

4.1.1 Traditional Parsing Strategies

In the incremental parsing algorithm described in the previous chapter, we build partial structures in a left-to-right scan. During this procedure, we can only use context information on the left side. In addition, it is hard to employ shared structure in this framework. Suppose we have two parses which only differ at the very beginning. The incremental parser has to duplicate the construction for the remainder of the parses.

Chart parsing (Kay, 1980) is a well-known strategy designed to employ shared structure. However, TAG chart parsing has a time complexity of $O(n^6)$, where n is the length of an input sentence. This computational problem also applies to LTAG-spinal, since wrapping adjunction is allowed with LTAG-spinal. Hence, for the purpose of building an efficient parser, we cannot directly use the chart parsing strategy.

Island parsing (Woods, 1976) was proposed for speech recognition with Augmented Transition Network (ATN) grammars. With island parsing, we can start from any positions of the input which we are the most confident of, and expand the partial analyses along the transition network in both directions.

Satta and Stock (1994) described a bidirectional chart parser, in which we can start from any position of an input sentence and expand partial analyses in both directions. The *subsumption block* technique was used to avoid artificial redundancy in bidirectional analysis. This bidirectional parsing strategy stems from the so called island driven parsing.

A similar search strategy is employed in the chunking-attachment framework of parsing. A chunker first recognizes chunks in an input sentence, and an attacher builds an parse tree with chunks. In this way, chunks can be viewed as the segments that we are the most confident of in an input sentence. Chunks serve as the shared structure in this two-step parser.

Here, we are going to extend the idea of island parsing to LTAG-spinal.

4.1.2 Parsing as Search

Parsing can be modeled as a search problem. Klein and Manning (2003) proposed an A* search algorithm for PCFG parsing. The score of a partial parse is the sum of two part, *a* and *b*, where *a* is the estimated score for the *outside* parse and *b* is the actual score for the *inside* parse. The outside estimate is required to be *admissible* for A* search, which means it should be higher than the actual score. Two methods were proposed to estimate the outside score, One is based on the input information in a local context, e.g. POS tags of nearby words. The other is to utilize the score (log-likelihood) given by a simplified grammar.

In fact, incremental parsing can also be viewed as search problem, in which path selection is to some extent fixed. The Perceptron like algorithm proposed in (Collins and Roark, 2004) is a learning algorithm specially designed for the problems of this class. They also improved this learning algorithm by introducing *early update*, which makes the search in training stop when the gold standard parse is not in the beam. This algorithm has been successfully applied to incremental parsing for CFG.

Daumé III and Marcu (2005) generalized this Perceptron like learning algorithm to general search problems. In their framework, the score of a path is also composed of two parts, *g* and *h*. Here *g* is the score of the path component, computed as a linear function of features as in Perceptron learning, and *h* is the estimate of the heuristic component, which is similar to A* search [1]. What makes it different from Perceptron with *early update* is that a set of gold-standard compatible paths are introduced into the queue of candidate paths when there is no gold-standard compatible path in the queue, instead of making the search stop. This algorithm has been successfully employed in applications like chunking and POS tagging, but it has not been used in parsing yet.

Here, we are going to explore the greedy search mechanism and the Perceptron learning algorithm for the island parsing strategy.

[1] *h* is not required to be *admissible* here.

4.1.3 Our approach

In the search algorithms described above, the estimation of the outside score is limited to the information given in the input string, for example, like POS tags of nearby words. Each single path (hypothesis) is unaware of all other outside paths (hypotheses).

Intuitively, if we can take advantage of nearby hypotheses for the estimation of the outside score, the overall score for each hypothesis will be more accurate. In this way, we can find the global hypothesis more efficiently. This method will be especially useful for greedy search. However, this approach results in the difficulty of maintaining a hypothesis with its context hypotheses, for which we will provide a solution later in this chapter.

This approach can be viewed as a statistical extension of the chunking-attachment framework. By chunking, we employ the heuristic rule that, for example, noun phrases should be recognized first, and they will be used as context information for further processing. Now, our approach is equivalent to saying that the chunks to be processed first are determined statistically by features.

This mechanism is consistent with island parsing; one can start from any positions in a given sentence and search in both directions. We do not specify seeds or boundaries beforehand. All the words compete for the positions of seeds. In general, all the candidate operations compete for the next operation.

As far as LTAG-spinal dependency parsing is concerned, we need to design some data structures to represent the partial result of an LTAG-spinal dependency tree, so that we can build partial dependency trees step by step. As we have noted, LTAG-spinal dependency is a non-projective relation. So we will use the mechanism of *visibility* as in our incremental parser described in Chapter 3.

It should be noted that the parser to be proposed in the chapter is only a dependency parser, which does not generate spinal templates as in the incremental parser. Therefore, we do not have the spine prediction operation any more. POS tags are used as the initial hypotheses. We still maintain distinction among attachment, adjunction, and predicate coordination. However, the operations are now with respect to the dependencies only.

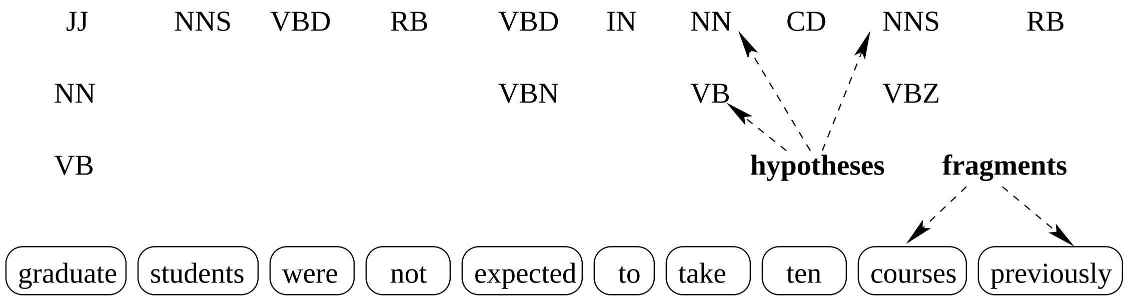


Figure 4.1: initialization

We will illustrate the LTAG-spinal dependency parsing mechanism with an example in the next section.

4.2 An Example

In this section, we show the data structures and the bidirectional dependency parsing algorithm with an example, and leave the formalization to the next section.

Initialization

Suppose the input sentence is as follows.

- *graduate students were not expected to take ten courses previously*

Each word is associated a set of hypothesis POS tags in the input, as shown in Figure 4.1. For initialization, each word comprises a **fragment**. A POS tag with the lexical item is called a **node** in dependency parsing. For initialization, each node comprises a **fragment hypothesis**. Due to the limitation of space, we ignore the lexical item in a node in Figure 4.1.

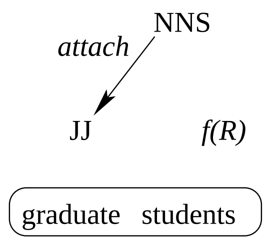


Figure 4.2: attach *JJ(graduate)* to *NNS(students)*

Step 1

We can combine the hypotheses for two nearby fragments with various operations like attachment and adjunction. For example, we can attach *JJ(graduate)* to *NNS(students)*, which is compatible with the gold standard, or adjoin *VB(take)* to *CD(ten)*, which is incompatible with the gold standard. We can represent an **operation** with a 6-tuple

$$R = (type, main, f_l, f_r, n_l, n_r), \tag{4.1}$$

where *type* is the type of the operation, and *main* = *left* or *right*, representing whether the left or the right tree is the main tree. f_l and f_r stand for the left and right fragment hypotheses involved in the operation. n_l and n_r stand for the left and right nodes involved in the operation.

If we apply R to fragment hypotheses $R.f_l$ and $R.f_r$, we generate a new hypotheses $f(R)$ for the new fragment which contains the fragments of both $R.f_l$ and $R.f_r$. For example, Figure 4.2 shows the result of attaching *JJ(graduate)* to *NNS(students)*. The new fragment hypothesis $f(R)$ is for the new fragment *graduate students*.

We use a queue Q to store all the candidate operations that could be applied to the current partial results. Operations in Q are ordered with the score of an operation. We have

$$s(R) = \mathbf{w} \cdot \Phi(R) \tag{4.2}$$

$$score(f(R)) = s(R) + score(R.f_l) + score(R.f_r), \tag{4.3}$$

where $s(R)$ is the score of the operation R, which is calculated as the dot product of a weight vector $\mathbf{w}$ and $\Phi(R)$, the feature vector of R. $s(R)$ is used to order the operations in Q. $score(f)$ is initialized to 0, if f contains only one node.

The **feature** vector $\Phi(R)$ is defined on $R.f_l$ and $R.f_r$, as well as the context. If $\Phi(R)$ only contains information in $R.f_l$ and $R.f_r$, we call this **level-0 feature dependency** [2].

However, we may want to use the information in nearby fragment hypotheses in some cases. For example, for the operation of attaching *JJ(graduate)* to *NN(student)*, we can check whether the root node of the hypothesis for the fragment containing *were* is a verb. We can define a feature for this, and this feature will be a strong indicator of the attachment operation. If features contain information of nearby fragment hypotheses, we call this **level-1 feature dependency**. By introducing level-1 feature dependencies, we actually calculate the score of a hypothesis by exploiting the information of outside hypotheses, as we have proposed earlier. Throughout this example, we will use level-1 feature dependency.

Suppose the operation of attaching *JJ(graduate)* to *NNS(students)* has the highest score, which is conditioned on the context that the POS tag for *were* is VBD. *were* is a nearby fragment. Therefore we need a data structure to maintain this relation. As shown in Figure 4.3, we introduce a **chain** which consists of two fragments, *graduate students* and *were*. There exist feature dependency relations between the hypotheses for different fragments in the same chain. Furthermore, each stand-alone fragment also comprises as a chain, as shown in Figure 4.3.

Suppose we use beam search and set beam width to two for each chain, which means that we keep the top two chain hypotheses for each chain. Figure 4.3 shows two **chain hypotheses** for the chain of *graduate students - were*. For the chain *graduate student - were*, each chain hypothesis consists of two fragment hypotheses respectively.

The score of a chain hypothesis is the sum of the scores of the fragment hypotheses in

[2]The dependency in *feature dependency* is different from the dependency in *dependency parsing*.

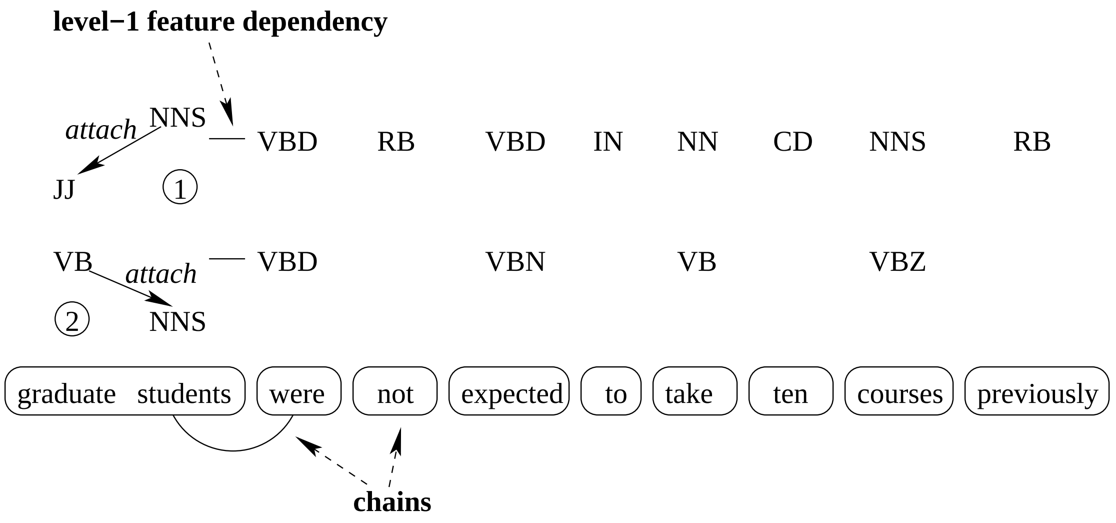


Figure 4.3: step 1 : to combine *graduate* and *students*

this chain hypothesis. For chain hypothesis *c*, we have

$$score(c) = \sum_{\text{fragment hypothesis } f \text{ of } c} score(f) \tag{4.4}$$

It is easy to see that

- *hypotheses for the same chain are mutually exclusive*, and
- *hypotheses for different chains are compatible with each other*

By saying two partial hypotheses compatible, we mean that there exists a global hypothesis which contains these two hypotheses.

For ease of the description later in this section, we assign unique IDs, ① and ②, to the two fragment hypotheses for *graduate students*, as shown in Figure 4.3.

After building the new chain and its hypotheses, we update the queue of candidate operations *Q*.

Step 2

Suppose the next operation with the highest score is to attach *CD(ten)* to *NNS(courses)* under the context of *VB(take)* and *RB(previously)*, generating hypothesis ③, as shown in

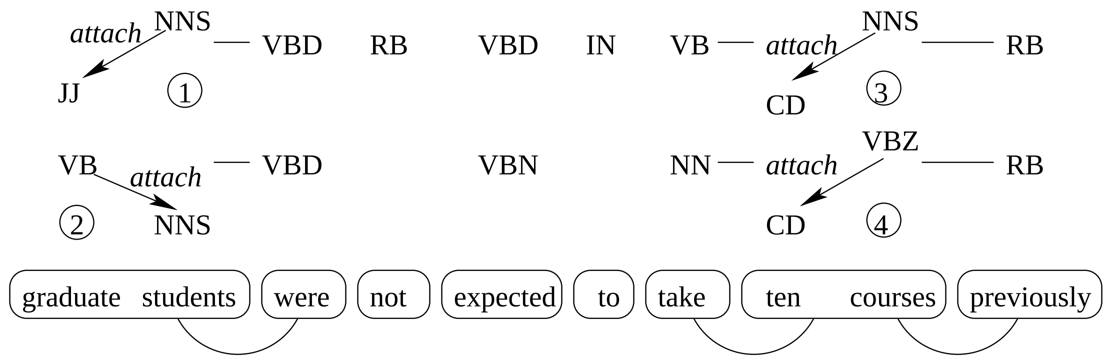


Figure 4.4: step 2 : to combine *ten* and *courses*

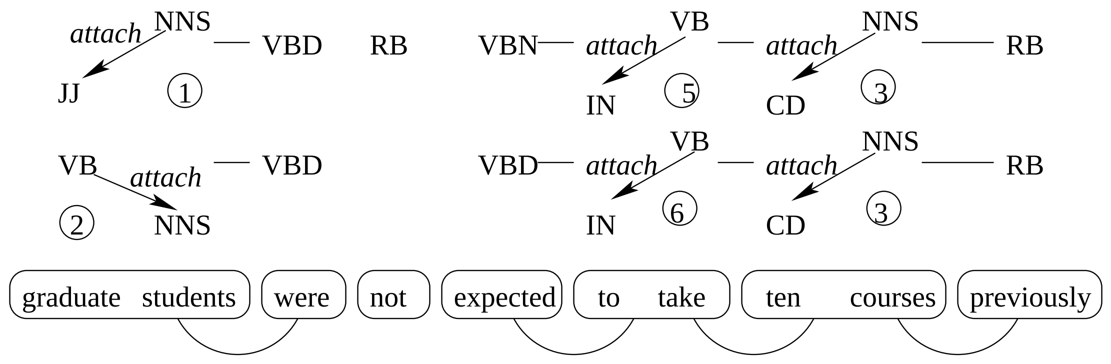


Figure 4.5: step 3 : to combine *to* and *take*

Figure 4.4. So we generate a new fragment, *ten courses*, and build a new chain containing *take* - *ten courses* - *previously*. Now, both NP chunks have been recognized.

Step 3

Suppose the next operation with the highest score is to attach *to* to *take* under the context of *VBN(expected)* and hypothesis ③, generating hypothesis ⑤, as shown in Figure 4.5. The chain *take* - *ten courses* - *previously* grows leftwards. We still keep top two chain hypotheses. The second best operation on the two fragments is to attach *to* to *take* under the context of *VBD(expected)* and hypothesis ③, generating hypothesis ⑥.

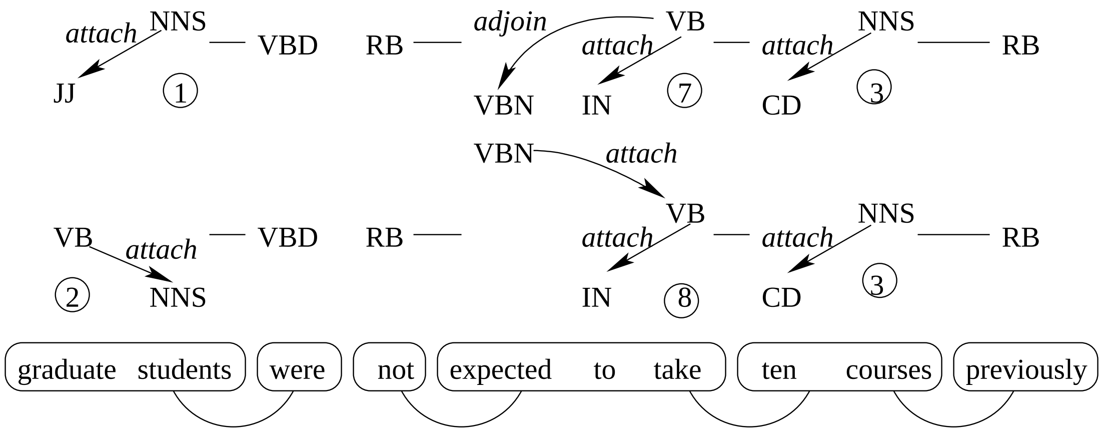


Figure 4.6: step 4 : to combine *expected* and *to take*

Step 4

Suppose the next operation is to adjoin *VBN(expected)* to *VB(take)* in hypothesis ⑤, generating hypothesis ⑦, as shown in Figure 4.6.

Step 5

Suppose the next operation with the highest score is to attach *NNS(courses)* in hypothesis ③ to *VB(take)* in hypothesis ⑦, generating hypothesis ⑨, as shown in Figure 4.7.

Step 6

Suppose the next operation with the highest score is to attach hypothesis *previously(RB)* to *VBN(expected)* in hypothesis ⑨, generating hypothesis ⑪. Here, the node *VBN(expected)* is *visible* to *ADV(previously)*.

We use a *visibility* check similar to what was previous used in the incremental parser. We will describe visibility in detail in Section 4.4.1, in which the *spinal adjunction* property of the LTAG-spinal treebank will be utilized.

It should be noted that, this operation results in a non-projective relation, because *take* is between *expected* and *previously*.

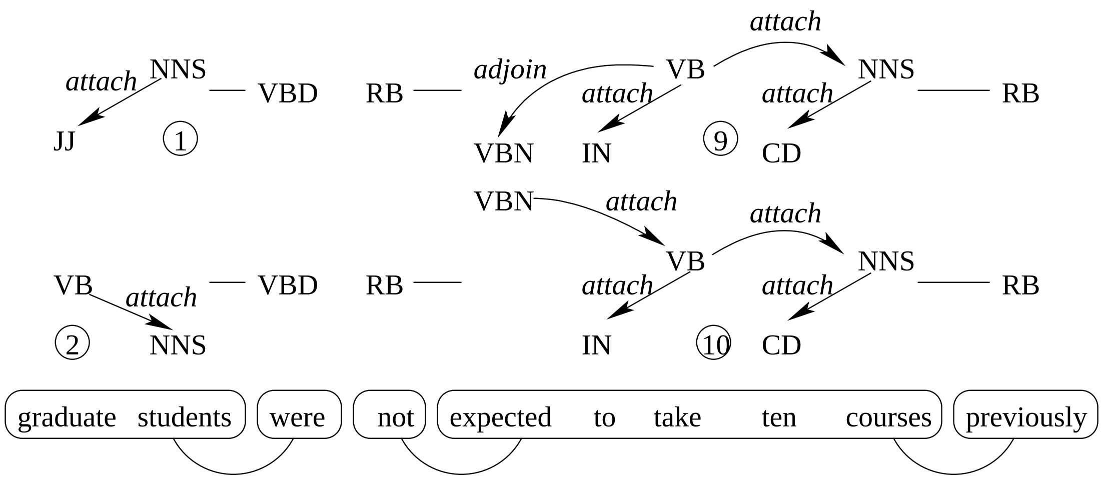


Figure 4.7: step 5 : to combine *expected to take* and *ten courses*

Final Output

We repeat combining hypotheses until there is only one fragment which ranges over the whole input sentence. Then we output the parse with the highest score, as shown in Figure 4.9.

4.3 Data Structures and Algorithms

Now we define the algorithm formally. Instead of giving an algorithm specially designed for parsing, we generalize the problem for graphs. A sentence can be viewed as a linear graph. We define the data structures in Section 4.3.1. In Sections 4.3.2 and 4.3.3, we present perceptron like search and training algorithms respectively.

4.3.1 Data Structures

We are given a connected graph $G(V,E)$ whose hidden structure is U, where vertices $V = \{v_i\}$, edges $E \subseteq V \times V$ is a symmetric relation, and $U = \{u_k\}$ is composed of a set of elements that vary with applications. As far as dependency parsing is concerned,

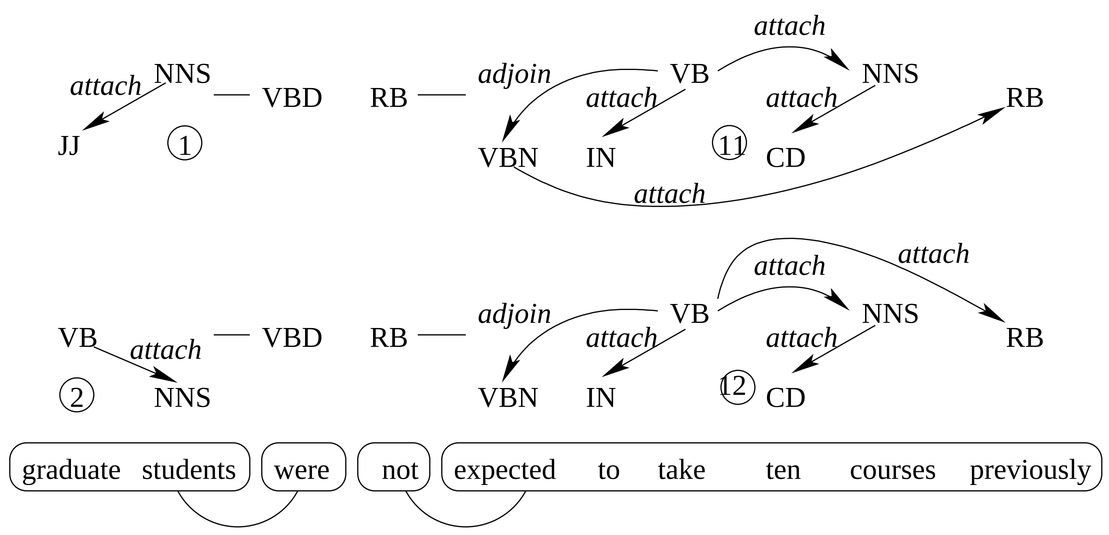


Figure 4.8: step 6 : to combine *expected to take ten courses* and *previously*

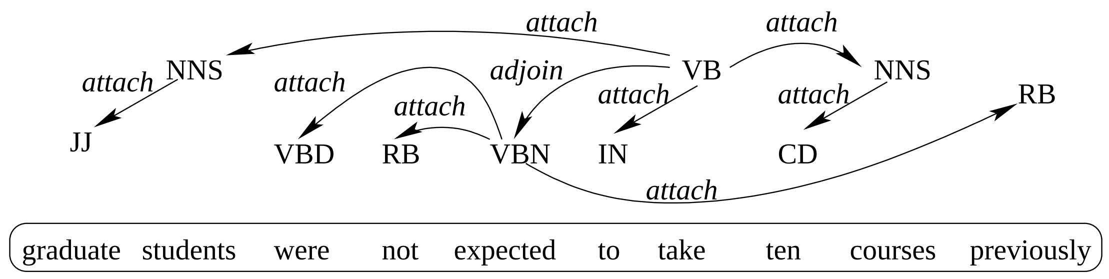

attach
adjoin
NNS
VB
NNS
RB
JJ
VBD
RB
VBN
IN
CD
graduate students were not expected to take ten courses previously

Figure 4.9: final output

the input graph is simply a linear graph, where $E(v_{i-1}, v_i)$. As to the hidden structure, $u_k = (v_{s_k}, v_{e_k}, b_k)$, where vertex v_{e_k} depends on vertex v_{s_k} with label b_k.

A graph-based incremental construction algorithm looks for the hidden structure with greedy search in a bottom-up style.

Let x_i and x_j be two sets of connected vertexes in V, where $x_i \cap x_j = \phi$ and they are directly connected via an edge in E. Let y^{xi} be a hypothesized hidden structure of x_i, and y^{xj} a hypothesized hidden structure of x_j.

Suppose we choose to combine y^{xi} and y^{xj} with an operation R to build a hypothesized hidden structure for $x_k = x_i \cup x_j$. We say the process of construction is **incremental** if the output of the operation, $y^{xk} = R(x_i, x_j, y^{xi}, y^{xj}) \supseteq y^{xi} \cup y^{xj}$ for all the possible x_i, x_j, y^{xi}, y^{xj} and operation R. As far as dependency parsing is concerned, incrementality means that we cannot remove any links coming from the substructures.

Once y^{xk} is built, we can no longer use y^{xi} or y^{xj} as a building block in the framework of greedy search. It is easy to see that left to right incremental construction is a special case of our approach. So the question is how to decide the order of construction as well as the type of operation R. For example, in the very first step of dependency parsing, we need to decide which two words are to be combined as well as the dependency label to be used.

This problem is solved statistically, based on the features defined on the substructures involved in the operation and their context. Given the weights of these features, we will show in the next section how these weights guide us to build a set of hypothesized hidden structures with beam search. In Section 4.3.3, we will present a perceptron like algorithm to obtain the weights.

Now we formally define the data structure to be used in our algorithms. Most of them were previously introduced in an informal way.

A **fragment** is a *connected* sub-graph of $G(V,E)$. Each fragment x is associated with a set of hypothesized hidden structures, or **fragment hypotheses** for short: $Y^x = \{y_1^x, \ldots, y_k^x\}$. Each y^x is a possible fragment hypothesis of x.

It is easy to see that an operation to combine two fragments may depend on the fragment hypotheses in the context, i.e. hypotheses for fragments directly connected to one of the operands. So we introduce the **dependency** relation over fragments [3]. Suppose there is a dependency relation $D \subseteq F \times F$, where $F \subseteq 2^V$ is the set of all fragments in graph G. $D(x_i, x_j)$ means that any operation on a fragment hypothesis of x_i depends on the features in the fragment hypothesis of x_j, and vice versa.

We are especially interested in the following two dependency relations.

[3]*Dependency* relation over fragments is different from the *dependency* in dependency parsing

- *level-0 dependency*: $D_0(x_i, x_j) \iff i = j$.
- *level-1 dependency*: $D_1(x_i, x_j) \iff x_i$ and x_j are directly connected in G.

So, in the incremental construction, we need to introduce a data structure to maintain the hypotheses with dependency relations among them.

A set of fragments, $c = \{x_1, x_2, ..., x_n\}$, is called a chain of fragments depending on x, or **chain** for x for short, for a given dependency relation D, if

- $x \in c$.
- $x_i \cap x_j = \phi, \forall i, j$.
- For any x_i, there exists $x_{i,0}, ..., x_{i,m}$, such that $x_{i,0} = x, x_{i,m} = x_i$ and $D(x_{i,j}, x_{i,j-1})$.

It is easy to show that, for any $x_i \in c$, c is a also a well defined chain for x_i.

For a given chain $c = \{x_1, x_2, ..., x_n\}$, we use $h^c = \{y^{x1}, ..., y^{xn}\}$ to represent a set of fragment hypotheses for the fragments in c, where y^{xi} is a fragment hypothesis for x_i in c. h^c is called a **chain hypothesis** for chain c. We use $H^c = \{h^c_1, ..., h^c_m\}$ to represent a set of chain hypotheses for chain c.

Now we can divide a given graph $G(V, E)$ with chains. A **cut** T of a given G, $T = \{c_1, c_2, ..., c_m\}$, is a set of chains satisfy

- exclusiveness: $\cup c_i \cap \cup c_j = \phi, \forall i, j$, and
- completeness: $\cup(\cup T) = V$.

Furthermore, we use $H^T = \{H^c | c \in T\}$ to represent of sets of chain hypotheses for all the chains in cut T. During the greedy search, we always maintain one cut over the whole graph.

As noted in the previous section, the idea behind the chain structure is that

- *hypotheses for the same chain are mutually exclusive*, and
- *hypotheses for different chains are compatible with each other*

In this way, we can generate hypotheses for different chains in parallel from different starting points in a graph. Two chains merge when certain operation happens, which depends on both of the two chains.

4.3.2 Search Algorithm

Algorithm 4 describes the procedure of building hypotheses incrementally on a given graph $G(V,E)$. Parameter k is used to set the beam width of search. Weight vector $\mathbf{w}$ is used to compute score of an operation.

We first initiate the cut T by treating each vertex in V as a fragment and a chain. Then we set the initial hypotheses for each vertex/fragment/chain. For example, in dependency parsing, the initial value is a set of possible POS tags for each single word. Then we use a queue Q to collect all the possible operations over the initial cut T and hypotheses H^T.

Whenever Q is not empty, we search for the chain hypothesis with highest score on operation according to a given weight vector $\mathbf{w}$. Suppose we find a new (fragment, hypothesis) pair (x,y) which is generated by the operation with the highest score. We first update the cut and the hypotheses according to (x,y). Let c^x be the chain for x. We remove from the cut T all the chains that overlap with c^x, and add c^x to T. Furthermore, we remove the chain hypotheses for those removed chains, and add the top k chain hypotheses for c^x to H^T. Then, we update the candidate queue Q by removing operations depending on the chain hypotheses that has been removed from H^T, and adding new operations depending on the chain hypotheses of c^x.

Now we explain the functions in Algorithm 4 one by one.

- *initCut*(V) initiates a cut T with vertexes V by setting $T = \{c_i\}$, where $c_i = \{x_i\}$, and $x_i = \{v_i\}$ for each $v_i \in V$. This means that we take each vertex as a fragment, and each fragment constitutes a chain.

- *initHypo*(T) initiates hypothesis H^T with the cut T described above. Here we set the initial fragment hypotheses, $Y^{xi} = \{y_1^{xi}, \ldots, y_k^{xi}\}$, where $x_i = \{v_i\}$ contains only

Algorithm 4 Incremental Construction

```
Require: graph $G(V,E)$;
Require: beam width $k$;
Require: weight vector $\mathbf{w}$;
1: cut $T \leftarrow initCut(V)$;
2: hypotheses $H^T \leftarrow initHypo(T)$;
3: candidate queue $Q \leftarrow initQueue(H^T)$;
4: repeat
5:     $(x',y') \leftarrow arg\max_{(x,y)\in Q} score(y)$;
6:     $T \leftarrow updCut(T,x',y')$;
7:     $H^T \leftarrow updHypo(H^T,x',y',k)$;
8:     $Q \leftarrow updQueue(Q,x',y',H^T)$;
9: until $(Q = \phi)$
```

one vertex.

- $initQueue(H^T)$ initiates the queue of candidate operations over the current cut T and H^T. Supposed there exist v_i and v_j which are directly connected in G. Let

  $$C = \{c^{xi}, c^{xj}\} \cup N(D,x_i) \cup N(D,x_j),$$

  where $N(D,x_i) = \{c^x | D(x_i,x), c^x \in T\}$ is the set of chains one of whose fragments depends on x_i.

  We apply all possible operations to all compatible fragment hypotheses of x_i and x_j with respect to all possible chain hypotheses combinations for C, and put them in Q. Suppose we generate (x_p, y^{xp}) with some operation, where x_p is equivalent to $x_i \cup x_j$ and $c^{xp} = \cup C$.

  All the candidate operations are organized with respect to the chain each operation generates. For each chain c, we maintain the top k candidates according to the score of the chain hypotheses. Scores of operations, fragments and chains are calculated with formula (4.2), (4.3) and (4.4) respectively.

- $updCut(T,x,y)$ is used to update cut T with respect to the candidate operation that generates $y^x = R(x_i,x_j,y^{xi},y^{xj})$. Let $C = \{c^{xi}, c^{xj}\} \cup N(D,x_i) \cup N(D,x_j)$ as described above. We remove all the chains in C from T, and add $\cup C$ to T.

Algorithm 5 Training Algorithm

```
 1: $\mathbf{w} \leftarrow \mathbf{0}$;
 2: for (round $r = 0$; $r < R$; $r$++) do
 3:     load graph $G_r(V,E)$, hidden structure $H_r$;
 4:     initiate $T, H^T$ and $Q$;
 5:     repeat
 6:         $(x',y') \leftarrow arg\max_{(x,y)\in Q} score(y)$;
 7:         if ($y'$ is compatible with $H_r$) then
 8:             update $T, H^T$ and $Q$ with $(x',y')$;
 9:         else
10:             $\tilde{y} \leftarrow schPosi(Q,x')$;
11:             $promote(\mathbf{w},\tilde{y})$;
12:             $demote(\mathbf{w},y')$;
13:             update $Q$ with $\mathbf{w}$;
14:         end if
15:     until ($Q = \phi$)
16: end for
```

- $updHypo(H^T,x,y,k)$ is used to update hypothesis H^T. We remove from H^T all the chain hypotheses whose corresponding chain has been removed from T in function $updCut(T,x,y)$. Furthermore, we add the top k chain hypotheses for chain $\cup C$ to H^T.

- $updQueue(Q,x,y,H^T)$ is designed to complete two tasks. First it removes from Q all the chain hypotheses which depend on one of the chains in C. Then it adds new candidate chain hypotheses depending on chain hypotheses of $\cup C$ in a way similar to the $initQueue(H^T)$ function. In Q, candidate chain hypotheses are organized with respect to the chains. For each chain c, we maintain the top k candidates according to the score of the chain hypotheses.

4.3.3 Training Algorithm

In the previous section, we described a search algorithm for graph-based incremental construction for a given weight vector $\mathbf{w}$. In Algorithm 5, we present a perceptron like algorithm to obtain the weight vector from the training data.

For each given training sample (G_r, H_r), where H_r is the gold standard hidden structure of graph G_r, we first initiate cut T, hypotheses H^T and candidate queue Q by calling *initCut*, *initHypo* and *initQueue* as in Algorithm 4.

Then we use the gold standard H_r to guide the search. We select candidate (x', y') which has the highest operation score in Q. If y' is compatible with H_r, we update T, H^T and Q by calling *updCut*, *updHypo* and *updQueue* as in Algorithm 4. If y' is incompatible with H_r, we treat y' as a negative sample, and look for a positive sample $\tilde{y}$ in Q with $schPosi(Q, x')$.

If there exists a hypothesis $\tilde{y}^{x'}$ for fragment x' which is compatible with H_r, then *schPosi* returns $\tilde{y}^{x'}$. Otherwise *schPosi* returns the candidate hypothesis which is compatible with H_r and has the highest score of operation in Q.

Then we update the weight vector $\mathbf{w}$ with $\tilde{y}$ and y'. At the end, we update the candidate Q by using the new weights $\mathbf{w}$ to compute the score of operation.

We can use various methods to improve the performance of the Perceptron algorithm. In our implementation, we use Perceptron with margin in the training (Krauth and Mezard, 1987). The margins are proportional to the loss of the hypotheses. Furthermore, we use voted Perceptron (Freund and Schapire, 1999).

4.4 LTAG-spinal Dependency Parsing

In this section, we will illustrate the details for the LTAG-spinal dependency parser which is not covered in the example in Section 4.2. In Section 4.4.1, we will describe how the hidden structure U in the algorithms is implemented for LTAG-spinal dependency parsing. In Section 4.4.2, we will illustrate the features used in our parser.

4.4.1 Incremental Construction

With the incremental construction algorithm described in the previous sections, we build the LTAG-spinal dependency tree incrementally. A hypothesis of a fragment is represented

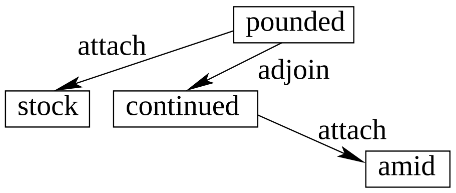


Figure 4.10: Wrapping adjunction with raising verbs

with a sub dependency tree. When the fragment hypotheses of two nearby fragments combine, the two sub dependency trees are combined into one.

Adjunction

It seems trivial to combine two partial dependency (derivation) trees with attachment. We can simply attach the root of tree *A* to some node on tree *B* which is *visible* to tree *A*. However, if adjunction is involved, the operation becomes a little bit complicated. An adjoined subtree may be *visible* from the other side of the dependency (derivation) tree. This is usually called *wrapping adjunction*.

Wrapping adjunction may occur with passive ECM verbs as shown in Figure 4.9, or raising verbs as in the following example, shown in Figure 4.10.

- *The stock of UAL Corp. continued to be pounded amid signs that British Airways ...*

Here *continued* adjoins onto *pounded*, and *amid* attaches to *continued* from the other side of the dependency (derivation) tree (*pounded* is between *continued* and *amid*).

In order to implement wrapping adjunction, we employ a similar definition of visibility as in the left to right parser described in Chapter 3. Now, visibility is defined on the nodes of a dependency (derivation) tree, but not the nodes of a derived tree as in the left to right parser. So the situation is simplified in this case. Instead of giving the formal definition, here we use plain description and figures to define visibility.

We still define visibility with respect to the direction. Without losing generality, we only illustrate *visibility from right* here.

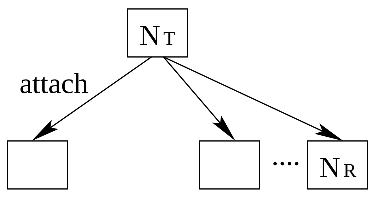


Figure 4.11: Case 1: no adjunction from left

We define visibility recursively, starting from the root of a partial dependency tree. The root node is always visible. Suppose a node in a partial dependency tree is visible from right, or visible for short here. We search for the child nodes which are visible also. Let the parent node be N_T.

- If there is no adjunction from the left side, then the rightmost child, N_R, is visible, as in Figure 4.11.
- Otherwise, let N_L adjoin to N_T from the left side [4]. Let N_{LX} be the rightmost descendant in the sub tree rooted on N_L, and let N_{RX} be the rightmost descendant in the sub tree rooted on N_R.
  - If N_T is to the right of N_{LX}, then both N_L and N_R are visible, as in Figure 4.12.
  - If N_{LX} is to the right of N_T and N_{RX} is to the right of N_{LX}, then N_R is visible, as in Figure 4.13.
  - If N_{LX} is to the right of N_{RX}, then N_L is visible, as in Figure 4.14.

In this way, we obtain all the visible nodes recursively, and all the rest are invisible from right. If there are multiple adjunction from the left side, we need to compare among those adjoined nodes in a similar way, although this rarely happens in the real data.

[4]For the sake of ease in description, we assume there is only one adjunction from the left side. However, the reader can easily extend this to the case of multiple adjunctions.

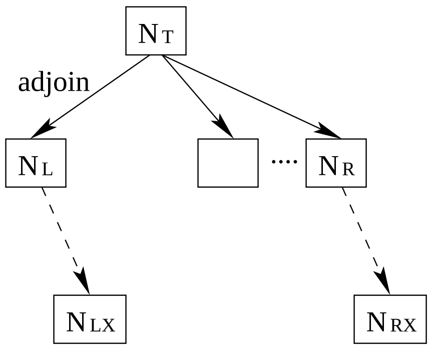


Figure 4.12: Case 2: both N_L and N_R is visible

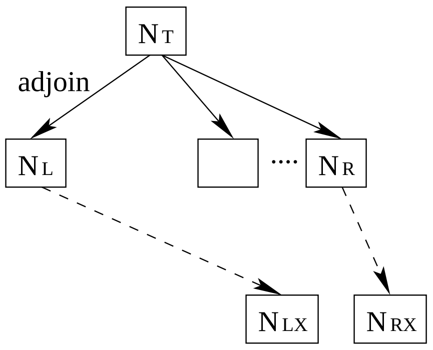


Figure 4.13: Case 3: N_R is visible

Predicate Coordination

As we have noted in Chapter 2, predicate coordination is represented explicitly in the LTAG-spinal Treebank. In order to build predicate coordination incrementally, we need to decompose coordination into a set of **conjoin** operations. Suppose a coordinated structure attaches to the parent node on the left side. We build this structure incrementally by attaching the first conjunct to the parent and conjoining other conjuncts to first one. In this way, we do not need to force the coordination to be built before the attachment. Either operation could be executed first.

Figure 4.15 shows the incremental construction of predicate coordination of the following sentence.

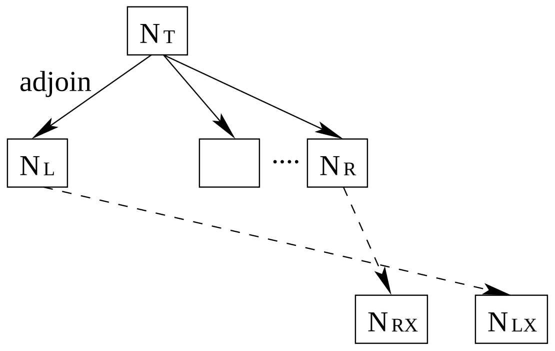


Figure 4.14: Case 4: N_L is visible

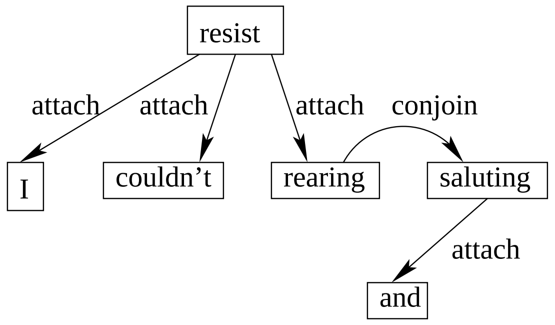


Figure 4.15: Partial dependency tree for the example of conjunction

- *I couldn't resist rearing up on my soggy loafers and saluting.*

4.4.2 Features

In this section, we will describe the features used in LTAG dependency parsing. As to feature definition, an operation is represented by a 4-tuple

- $op = (type, dir, pos_{left}, pos_{right})$,

where $type \in \{attach, adjoin, conjoin\}$ and dir is used to represent the direction of the operation. pos_{left} and pos_{right} are the POS tags of the two operands.

Features are defined on POS tags and lexical items of the nodes. In order to represent the features, we use m for the main-node of the operation, s for the sub-node, m_r for the

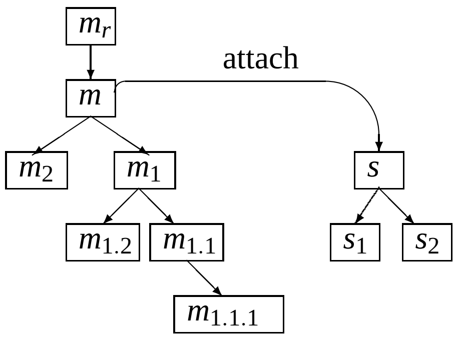


Figure 4.16: Representation of nodes

parent of the main-node, $m_1..m_i$ for the children of m, and $s_1..s_j$ for the children of s. The index always starts from the side where the operation takes place. We use the Gorn address to represent the nodes in the subtrees rooted on m and s.

Furthermore, we use l_k and r_k to represent the nodes in the left and right context of the flat sentence. We use h_l and h_r to represent the head of the outside hypothesis trees on the left and right context respectively.

Let x be a node. We use $x.p$ to represent the POS tag of node x, and $x.w$ to represent the lexical item of node x.

Table 4.1 show the features used in LTAG dependency parsing. There are seven classes of features. The first three classes of features are those defined on only one operand, on both operands, and on the siblings respectively. If the gold standard POS tags are used as input, we define features on the POS tags in the context. If level-1 dependency is used, we define features on the root node of the hypothesis partial dependency trees in the neighborhood.

Half check features and full check features in Table 4.1 are designed for grammatical checks. For example, in Figure 4.16, node s attaches onto node m. Then nothing can attach onto s from the left side. The children of the left side of s are fixed, so we use the half check features to check the completeness of the children of the left half for s. Furthermore, we notice that all the left-edge descendants of s and the right-edge descendants of m become

Table 4.1: Features defined on the context of operation

category	description	templates
one operand	Features defined on only one operand. For each template tp, $[type, dir, tp]$ is used as a feature.	$(m.p)$, $(m.w)$, $(m.p, m.w)$, $(s.p)$, $(s.w)$, $(s.p, s.w)$
two operands	Features defined on both operands. For each template tp, $[op, tp]$ is used as a feature. In addition, $[op]$ is also used as a feature.	$(m.w)$, $(c.w)$, $(m.w, c.w)$
siblings	Features defined on the children of the main nodes. For each template tp, $[op, tp]$, $[op, m.w, tp]$, $[op, m_r.p, tp]$ and $[op, m_r.p, m.w, tp]$ are used as features.	$(m_1.p)$, $(m_1.p, m_2.p)$, .., $(m_1.p, m_2.p, .., m_i.p)$
POS context	In the case that gold standard POS tags are used as input, features are defined on the POS tags of the context. For each template tp, $[op, tp]$ is used as a feature.	$(l_2.p)$, $(l_1.p)$, $(r_1.p)$, $(r_2.p)$, $(l_2.p, l_1.p)$, $(l_1.p, r_1.p)$, $(r_1.p, r_2.p)$
tree context	In the case that level-1 dependency is employed, features are defined on the trees in the context. For each template tp, $[op, tp]$ is used as a feature.	$(h_l.p)$, $(h_r.p)$
half check	Suppose $s_1, ..., s_k$ are all the children of s which are between s and m in the flat sentence. For each template tp, $[tp]$ is used as a feature.	$(s.p, s_1.p, s_2.p, .., s_k.p)$, $(m.p, s.p, s_1.p, s_2.p, .., s_k.p)$ and $(s.w, s.p, s_1.p, s_2.p, .., s_k.p)$, $(s.w, m.p, s.p, s_1.p, s_2.p, .., s_k.p)$ if $s.w$ is a verb
full check	Let x_1, x_2, .., x_k be the children of x, and x_r the parent of x. For any $x = m_{1.1...1}$ or $s_{1.1...1}$, template tp, $[tp(x)]$ is used as a feature.	$(x.p, x_1.p, x_2.p, .., x_k.p)$, $(x_r.p, x.p, x_1.p, x_2.p, .., x_k.p)$ and $(x.w, x.p, x_1.p, x_2.p, .., x_k.p)$, $(x.w, x_r.p, x.p, x_1.p, x_2.p, .., x_k.p)$ if $x.w$ is a verb

unavailable for any further operation. So their children are fixed after this operation. All these nodes are in the form of $m_{1.1...1}$ or $s_{1.1...1}$. We use full check features to check the children from both sides for these nodes.

4.5 Discussion

4.5.1 On Weight Update

Let us first have a close look at the function *schPosi*.

1. This function first tries to find a local correction to the wrong operation.

2. If it fails, which means that, in the gold standard, there is no direct operation over the two fragments involved, the function returns a correct operation which the current weight vector is the most confident of.

This learning strategy is designed to modify the current weight vector, or path preference, as little as possible, so that the context information learned previously will still be useful.

In case 1, only the weights of the features directly related to the mistake are updated, and unrelated features are kept unchanged.

In case 2, the feature vector of the operation with a higher score is closer to the direction of the weight vector. Therefore, to use the one with highest score helps to keep the weight vector in the previous direction.

4.5.2 On Path Selection

After the weight vector is updated, we re-compute the score of the candidate operations. The new operation with the highest score could still be over the same two fragments as the last round. However, it could be over other fragments as well. Intuitively, in this case, it means that the operation over the previous fragments is hard to decide with the current context, and we'd better work on other fragments first. This strategy is designed to learning a desirable order of operations.

Actually, the training algorithm makes the score of all operations comparable, since they always compete head to head in the queue of candidates. In this way, we can use the score to select the path.

4.5.3 Related Works

Yamada and Matsumoto (2003) has proposed a deterministic dependency parser which builds the parse tree incrementally via rounds of left-to-right scans. As each step, they check whether an attachment should be built or not. Each local decision is determined by a local classifier. This framework allows certain level bidirectional mechanism.

Compared to their work, our parser has real bidirectional capability. At each step, we always compare all the possible operations. In addition, in our model, inference is incorporated in the training, as described in Section 1.3.

In the search algorithms proposed in (Klein and Manning, 2003; Daumé III and Marcu, 2005), heuristics is used for the outside score. In our algorithm, outside hypotheses are used to compute the score. In this way, a greedy search algorithm can take advantage of more information.

Similar to (Collins and Roark, 2004) and (Daumé III and Marcu, 2005), our training algorithm learns the inference in a subset of all possible contexts. However, our algorithm is more *aggressive*. In (Collins and Roark, 2004), a search stops if there is no hypothesis compatible with the gold standard in the queue of candidates. In (Daumé III and Marcu, 2005), the search is resumed after some gold-standard compatible hypotheses are inserted into a queue for future expansion, and the weights are updated correspondingly. However, there is no guarantee that the updated weights assign a higher score to those inserted gold-standard compatible hypotheses.

In our algorithm, the gold-standard compatible hypotheses are used for weight update only. As a result, after each sentence is processed, the weight vector can usually successfully predict the gold standard parse. As far as this aspect is concerned, our algorithm is similar to the MIRA algorithm in (Crammer and Singer, 2003).

In MIRA, one always knows the correct hypothesis. However, in our case, we do not know the correct order of operations. So we have to use our form of weight update to implement aggressive learning.

In general, the learning model described in Algorithm 5 is more difficult than supervised learning, because we do not know the correct order of operations. On the other hand, our algorithm is not as difficult as reinforcement learning, due to the fact that we can check the compatibility with the gold-standard for each candidate hypothesis.

4.6 Experiments and Analysis

We use the same data set as in Chapter 3. We train our LTAG dependency parser on section 2-21 of the LTAG Treebank. Section 22 is used as the development set for feature hunting. Section 23 is used for test.

Table 4.2 shows the comparison of different models. Beam size is set to five in our experiments. With level-0 dependency, our system achieves an f-score of 90.3% at the speed of 4.25 sentences a second on a Xeon 3G Hz processor with JDK 1.5. With level-1 dependency, the parser achieves 90.5% at 3.59 sentences a second. Level-1 dependency does not provide much improvement due to the fact that level-0 features provide most of the useful information for this specific application. But this is not always true in other applications, e.g. POS tagging, in which level-1 features provide much more useful information.

If we decrease the beam width to one for both training and test, which means that we use this model for deterministic parsing, the level-1 system achieves an f-score of 90.0%, only 0.5 points lower than our best result.

It is interesting to compare our system with other dependency parsers. The f-score on

Table 4.2: Results of bidirectional dependency parsing on Section 23 of the LTAG Treebank

model	f-score%
left-to-right, flex	89.3
level-0 dependency	90.3
level-1 dependency	90.5

LTAG dependency is comparable to the numbers of the previous best systems on dependency extracted from PTB with Magerman's rules, for example, 90.3% in (Yamada and Matsumoto, 2003) and 90.9% in (McDonald et al., 2005). However, their experiments are on the PTB, which is different from ours. To learn the LTAG dependency is more difficult for the following reasons.

Theoretically, the LTAG dependencies reveal deeper relations. Adjunction can lead to non-projective dependencies, and the dependencies defined on predicate adjunction are linguistically more motivated, as shown in the example in Figure 4.10. The explicit representation of predicate coordination also provides deeper relations. For example, in Figure 4.15, the LTAG dependency contains *resist* → *rearing* and *resist* → *saluting*, while the Magerman's dependency only contains *resist* → *rearing*. The explicit representation of predicate coordination helps to solve for the dependencies for shared arguments.

This claim is also verified by our experiments in Section 5.5.2. We will show that one can recognize unlabeled Propbank arguments from the LTAG Treebank at the f-score of 91.3% with simple rules, while one of the state-of-the-art systems (Pradhan et al., 2004) working on the same task directly on Penn Treebank can only achieve an f-score of 90.4%. So the LTAG dependencies reveal deeper relations, and is more difficult to learn at the syntax level. Our graph-based incremental construction clearly shows very desirable performance.

4.7 Summary

In this chapter, we first introduced bidirectional incremental parsing, a new architecture of parsing for LTAG. We have proposed a novel algorithm for graph-based incremental construction, and applied this algorithm to LTAG-spinal dependency parsing, revealing deep relations, which are unavailable in other approaches and difficult to learn. We have evaluated the parser on the LTAG-spinal Treebank. Experimental results show a significant improvement over the incremental parser described in Chapter 3. Graph-based incremental construction could be applied to other structure prediction problems in NLP.

Chapter 5

Resource Construction

In Chapters 3 and 4, we have proposed two parsers based on the LTAG-spinal formalism. In this chapter, we will present the treebank used to train and evaluate these two parsers.

We will briefly describe previous works on LTAG treebank extraction in Section 5.1. We will present our approach in Section 5.2, and illustrate the procedure of treebank induction in Section 5.3. In Section 5.4, we will describe the format of the treebank, as well as the representations for some interesting linguistic structures. In Section 5.5, we will present the statistics for the extracted LTAG-spinal treebank and evaluation of its compatibility with the Propbank.

5.1 Previous Works

For the purpose of statistical processing, many attempts have been made for automatic construction of LTAG treebanks. Joshi and Srinivas (1994) presented a supertag corpus extracted from the Penn Treebank with heuristic rules. However, due to the limit of supertag extraction algorithm, the extracted supertags of the words in a sentence cannot always be successfully put together. Xia (2001) and Chen (2001) described deterministic systems that extract LTAG-style grammars from PTB. In their systems, a *head table* in Magerman's style (1995) and the PTB functional tags were used to resolve ambiguities in extraction. Chiang (2000) reported a similar method of extracting an LTAG treebank from PTB, and used it in a statistical parser for Tree Insertion Grammar.

5.2 Our Approach

We will extract an LTAG-spinal treebank from the Penn Treebank with Propbank annotation. The following two properties make our treebank different from the previous works.

- **Incorporating Propbank**

  Propbank provides annotation of predicate-argument structures and semantic roles on the Penn Treebank, which is unavailable to the previous LTAG treebank extraction systems. There is an obvious connection between Propbank argument sets and elementary trees in LTAG. Therefore, one of the purposes of our work is to incorporate Propbank annotation into the extracted LTAG treebank. In this way, the extracted elementary trees for each lexical anchor (predicate) will become semantically meaningful. On the other hand, Propbank provides more information that helps us to successfully extract various structures of interest. For example, Propbank annotation on discontinuous arguments helps us to recognize auxiliary trees in LTAG.

- **Treatment for Predicate Operation**

  In the previous LTAG treebank extraction work, the second conjunct is always represented as a partial template. In our work, we use a special operation to represent predicate coordination explicitly. We use this method to encode shared arguments.

5.3 Extracting an LTAG-spinal Treebank

In this section, we describe the algorithm that we have used to extract the LTAG-spinal treebank from PTB with Propbank annotation. We use a rule-based method for treebank extraction. We take a PTB tree as an LTAG *derived* tree.

Since the latest release of Propbank does not provide annotation for the verb “be”, we first automatically generate the annotation for “be”. Then we incorporate PTB and Propbank by tree transformations based on Propbank annotation. LTAG predicate coordination and full adjunction are recognized by Propbank annotation. Then we extract LTAG elementary trees from the transformed PTB subtrees recursively, according to Propbank annotation and a head table. At the end, we map all the elementary trees into a small set of normalized elementary trees.

The extraction algorithm is implemented in several rounds of tree traversal. Each round is implemented with a recursive function over trees. Therefore, whenever possible, we always try to divide different operations into different rounds so as to simplify the implementation.

Now we will explain these steps one by one.

5.3.1 Pseudo Annotations for *be*

We generate annotations for “be” by using PRD and SBJ labels in PTB. We first look for VP nodes that directly dominate a “be” node and a node with a PRD label. Then we look for the lowest node with a SBJ label that c-commands the VP node. According to Propbank style, the “be” node is labeled as the *relation*, and the SBJ node is labeled as ARG0 and the node with PRD tag is labeled as ARG1.

This module will be skipped if the next version of the Propbank provides the accurate annotation of the verb “be”.

5.3.2 PTB and Propbank Integration

An argument or modifier in Propbank may correspond to several nodes in the PTB parse tree. This is mainly due to the syntax-semantic discrepancy, e.g. extrapositions (Xia and Bleam, 2000). Some of these are due to the different analyses or different granularities of analyses of PTB and Propbank. Some are just typos or mistakes.

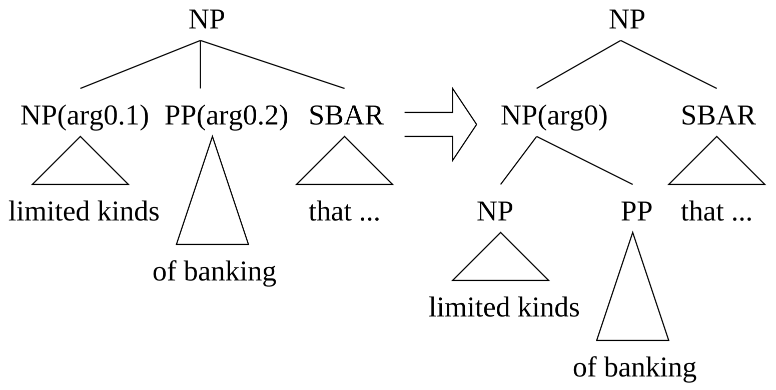


Figure 5.1: Continuous flat segments.

As for as LTAG is concerned, an argument corresponds to one subtree in the extracted treebank, so we use tree transformations to combine several nodes of PTB into one constituent according to the Propbank annotation for some of these cases, since Propbank is a second round annotation over PTB.

An argument in Propbank may correspond to continuous constituents or discontinuous constituents in PTB. We first describe the tree transformations applied to continuous constituents. There are two types of transformations, depending on the location of the constituent segments.

- *Continuous Flat Segments*. All the segments are directly dominated by the same node. We move all the constituents that are not part of the argument upwards by one level as shown in Figure 5.1.

- *Continuous Uneven Segments*. The segments are on different levels of the parse tree. We move all the constituents that are not part of the argument upwards by several levels as shown in Figure 5.2. However, the result tree is not totally correct. (IN of) and (NP workers) should form a PP node first, which then modifies (NP a group). But we do not have enough context information the obtain this structure automatically.

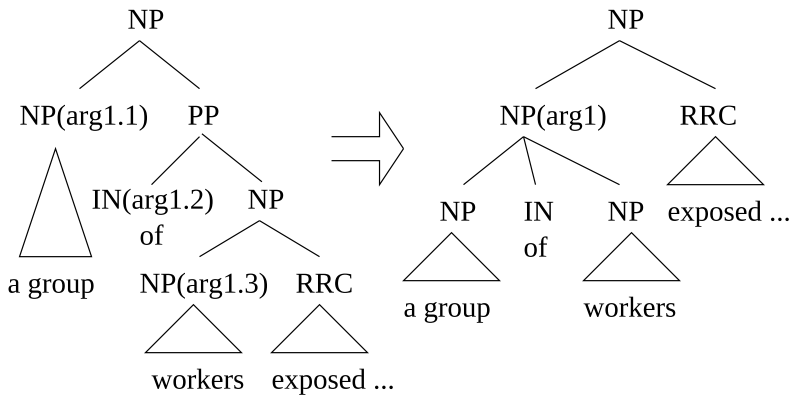


Figure 5.2: Continuous uneven segments.

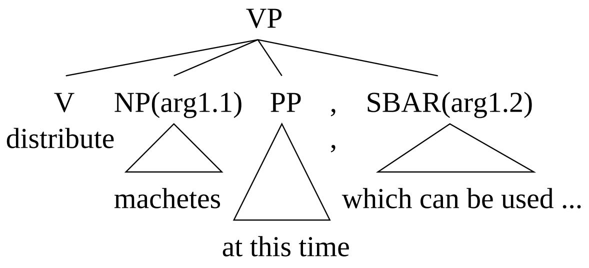


Figure 5.3: Extraposition.

5.3.3 Full Adjunction

As we have mentioned before there are arguments which are composed of discontinuous constituents. One class of these discontinuous constituents is extraposition, which can only be represented by multi-component LTAG trees (Joshi et al., 2002) in the LTAG formalism, as in Figure 5.3. Currently, we leave them as they are.

There is a class of discontinuous segments which appear frequently in Propbank that can be exactly represented with full auxiliary trees under LTAG. For example, raising verbs and passive ECM verbs are associated with traditional auxiliary trees. Furthermore, we notice that in many cases the word “say” can only be represented with full auxiliary trees, as shown in Figure 5.4.

We cut the *PRN* rooted sub-tree from the *S* node, and an extra root *S* and an extra foot node are added to this tree as shown in Figure 5.4.

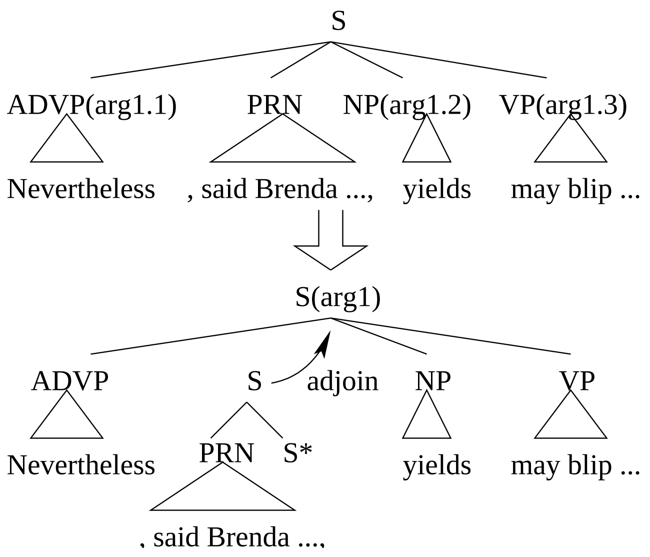


Figure 5.4: Full auxiliary.

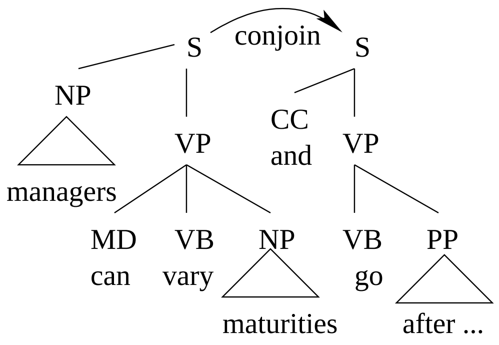


Figure 5.5: Predicate Coordination.

5.3.4 Coordination

Propbank annotation is used to determine the head of “S” type nodes and “VP” nodes. The head information is further used to detect predicate coordination. Predicate coordination structures are transformed into subtrees connected with the conjoin operation, as shown in Figure 5.5.

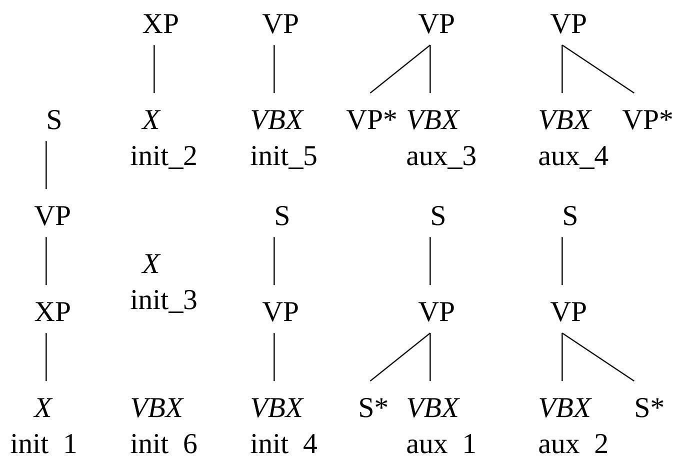


Figure 5.6: Types of normalized spinal elementary trees.

5.3.5 Elementary Tree Extraction

After coordination and adjunction recognition, the original tree is cut into treelets. Then we recursively extract LTAG elementary trees from the treelets, which is similar to the work reported in Xia's (2001) and Chen's (2001) papers. What is special here is the following: in order to generate elementary trees in the LTAG-spinal format, we need to move some arguments upwards to make them as direct children of a node on spine.

5.3.6 Normalization

Finally, we normalize the elementary trees by mapping them to a fixed set of elementary trees, which are shown in Figure 5.6. In order to decrease redundancy, a sister auxiliary tree is mapped to an initial tree by removing the root node. Therefore, we have only 6 different kinds of initial trees (3 for verbs and 3 for non-verbs) and 4 different kinds of full auxiliary trees. In Figure 5.6, *VBX* represents a verbal POS tag, and *X* represents a non-verbal POS tag.

5.4 The LTAG-spinal Treebank

5.4.1 Format

We describe the format of the LTAG-spinal Treebank with an example sentence shown in Figure 5.7.

Each sentence in the LTAG Treebank begins with a line of index information. For example, “2 31 7” means that this sentence is the 7th sentence in file 31 of WSJ section 2 in the PTB . The second line shows the root of the sentence. In this example, the root is the 20th elementary tree, or **e-tree** for short, which is a structure for predicate coordination which we will explain later. Then follows the information for each e-tree in the derivation tree in the order of the position in the flat sentence.

The information of an e-tree is composed of three parts. The first part is the ID of the e-tree as well as the lexical information if there exists. The second part is the spinal template for this e-tree. The third part is optional. It consists of the information of the child e-trees.

Now we first describe the format of spinal templates, then explain the information for the children. A spinal template is represented with a string which starts with a, b, or c, which mean initial tree, auxiliary tree, and predicate coordination respectively. The rest of the substring represents the spine in the List format as in the PTB. For example, (S (VP VBˆ S*)) is used to represent a template rooted on S, where S dominates VP, and VP dominates the anchor VB and the foot S. Here ˆ represents anchor, and * represents foot.

Each child is represented with a line which starts with adj, *att*, and *crd*. adj means adjunction. *att* means attachment, and it also means connective in coordination as shown in #20 of Figure 5.7. *crd* means predicate conjunct. Then follows the ID of the child e-trees.

The rest of the line is the landing information, which is composed of three parts. The first part is the node where the child attaches or adjoins. It is represented with Gorn address. For example, in e-tree #3 of the sentence, e-tree #6 attaches to the VP node, whose address is 0.0 in the spine. The second part is the ID of the slot, which represents the position relative to the children of the landing node in the spine. For example, in e-tree #3, the VP node has two slots. Slot 0 represents the left side of the V node and slot 1 represents the right side of the V node. The IDs of the slots are ordered from left to right. Since e-tree #6 attaches onto the VP node from right, the slot number is 1. The third part represents the order of child e-trees in the same slot. For e-tree #3, both #0 and #2 e-trees attach onto VP in the same slot, and #0 is to the left of #2 e-tree. So the order number is 0 for #0, and 1 for #2, which is also ordered from left to right.

As to the example in Figure 5.7, the root e-tree is a structure of coordination between *fail* and *remained*. *But* and *sectors* attach to the S node of *fail* from left, and *attract* attaches to the VP node of *fail* from right. *sluggish* and *making* attach to the VP node of *remained* from right. Furthermore, the e-tree for *appear* adjoins to the VP node of the e-tree for *mixed*.

```
2 31 7
root 20
#0 but
 spine: a_CCˆ
#1 other
 spine: a_JJˆ
#2 sectors
 spine: a_( XP NNSˆ )
 att #1, on 0, slot 0, order 0
#3 failed
 spine: a_( S ( VP VBDˆ ) )
 att #0, on 0, slot 0, order 0
 att #2, on 0, slot 0, order 1
 att #6, on 0.0, slot 1, order 0
#4 *-1
 spine: a_( XP NONEˆ )
#5 to
 spine: a_TOˆ
#6 attract
 spine: a_( S ( VP VBˆ ) )
 att #4, on 0, slot 0, order 0
 att #5, on 0.0, slot 0, order 0
 att #8, on 0.0, slot 1, order 0
#7 investor
 spine: a_NNˆ
#8 interest
 spine: a_( XP NNˆ )
 att #7, on 0, slot 0, order 0
#9 and
 spine: a_CCˆ
#10 remained
 spine: a_( S ( VP VBDˆ ) )
 att #11, on 0.0, slot 1, order 0
 att #12, on 0.0, slot 1, order 1
 att #14, on 0.0, slot 1, order 2
 att #19, on 0, slot 1, order 0
#11 sluggish
 spine: a_( XP JJˆ )
#12 ,
 spine: a_,ˆ
#13 *-1
 spine: a_( XP NONEˆ )
#14 making
 spine: a_( S ( VP VBGˆ ) )
 att #13, on 0, slot 0, order 0
 att #16, on 0.0, slot 1, order 0
 att #18, on 0.0, slot 1, order 1
#15 overall
 spine: a_JJˆ
#16 trading
 spine: a_( XP NNˆ )
 att #15, on 0, slot 0, order 0
#17 appear
 spine: b_( VP VBˆ VP* )
#18 mixed
 spine: a_( S ( VP ( XP JJˆ ) ) )
 adj #17, on 0.0, slot 0, order 0
#19 .
 spine: a_.ˆ
&20
 spine: c_( S S S )
 crd #3, on 0.0
 att #9, on 0, slot 1, order 0
 crd #10, on 0.1
```

Figure 5.7: An Example in the treebank

5.4.2 Case Studies

Raising Verbs and Passive ECM Verbs

In the LTAG Treebank, raising verbs and passive ECM verbs are associated with an auxiliary tree. For example, in Figure 5.8, the e-tree for *continue* adjoins onto the S node of the e-tree for *soften*. Furthermore, *in* attaches to *soften*. Since, *soften* is between *continue* and *in* in the flat sentence, this leads to a case of non-projective dependency.

Predicate Coordination

Predicate coordination is represented with a special structure in the LTAG Treebank. E-tree #20 in Figure 5.7 shows an example of coordination, in which conjuncts are explicitly annotated. In this approach, we encode the ambiguity of argument sharing. In this example, *sectors* attaches to *failed*, the first conjunct. But it is also the subject of *maintained*, the second conjunct. The explicit representation of coordination used in the Treebank makes it easier to solve the ambiguity of shared arguments.

Relative Clauses

In the LTAG Treebank, a relative clause is represented by attaching the predicate of the clause to the head of the phrase that it modifies. For example, in Figure 5.9, the predicate of the relative clause is *shown*. *which* attaches onto *shown*, and *shown* attaches onto *earnings*.

```
#19 the
 spine: a_DTˆ
#20 market
 spine: a_( XP NNˆ )
 att #19, on 0, slot 0, order 0
#21 could
 spine: a_MDˆ
#22 continue
 spine: b_( S ( VP VBˆ S* ) )
 att #21, on 0.0, slot 0, order 0
 att #26, on 0.0, slot 2, order 0
#23 *-1
 spine: a_( XP NONEˆ )
#24 to
 spine: a_TOˆ
#25 soften
 spine: a_( S ( VP VBˆ ) )
 att #18, on 0, slot 0, order 0
 att #20, on 0, slot 0, order 1
 adj #22, on 0, slot 0, order 2
 att #23, on 0, slot 0, order 3
 att #24, on 0.0, slot 0, order 0
#26 in
 spine: a_( XP INˆ )
 att #28, on 0, slot 1, order 0
#27 the
 spine: a_DTˆ
#28 months
 spine: a_( XP NNSˆ )
 att #27, on 0, slot 0, order 0
 att #29, on 0, slot 1, order 0
#29 ahead
 spine: a_RBˆ
```

Figure 5.8: Raising Verb

```
#4 earnings
 spine: a_( XP NNSˆ )
 att #3, on 0, slot 0, order 0
 att #10, on 0, slot 1, order 0
#5 ,
 spine: a_,ˆ
#6 which
 spine: a_( XP WDTˆ )
#7 *t*-48
 spine: a_( XP NONEˆ )
#8 were
 spine: a_VBDˆ
#9 mistakenly
 spine: a_( XP RBˆ )
#10 shown
 spine: a_( S ( VP VBNˆ ) )
 att #5, on 0, slot 0, order 0
 att #6, on 0, slot 0, order 1
 att #7, on 0, slot 0, order 2
 att #8, on 0.0, slot 0, order 0
 att #9, on 0.0, slot 0, order 1
 att #11, on 0.0, slot 1, order 0
```

Figure 5.9: Relative Clauses

```
#0 Ms.
 spine: a_NNPˆ
#1 Waleson
 spine: a_( XP NNPˆ )
 att #0, on 0, slot 0, order 0
#2 is
 spine: a_( S ( VP VBZˆ ) )
 att #1, on 0, slot 0, order 0
 att #5, on 0.0, slot 1, order 0
 att #11, on 0, slot 1, order 0
#3 a
 spine: a_DTˆ
#4 free-lance
 spine: a_JJˆ
#5 writer
 spine: a_( XP NNˆ )
 att #3, on 0, slot 0, order 0
 att #4, on 0, slot 0, order 1
 att #6, on 0, slot 1, order 0
```

Figure 5.10: Predicative Trees

Predicative Trees

In the current version of the LTAG Treebank, most of the predicate nominals and adjectives are not annotated as the head predicate. Instead, the copula is treated as the head of the sentence. For example, in Figure 5.10, *writer* attaches to *is*. We are aware that, in the XTAG English grammar, predicate nominals and adjectives are regarded as the head.

The treatment here is due to the difficulty in finding the head of a noun phrase. In the PTB, NP representation is rather flat, so that it is non-trivial to recognize coordination under the NP level automatically. For example, NP(*those workers and managers*) and NP(*the US sales and marketing arm*) are both represented as flat NP. Furthermore, appositives and NP lists are represented in the same way. The problem of recognizing NP coordination and coordination within an NP results in the difficulty of choosing the head of NPs. Therefore, in this version of the LTAG Treebank, copulas are annotated as the predicate.

Parentheticals

Many parentheticals with predicate structures in the PTB are analyzed as adjunction in the LTAG Treebank, especially for the use of *say*. In Figure 5.11, *testified*, the head of the parenthetical, adjoins to *began* from left.

Extrapositions

Extraposition is a class of dependencies that cannot be represented with traditional LTAG. It is also a problem for the LTAG-spinal formalism. In Figure 5.12, *more than three times the expected number* modifies *28*. However, in the LTAG Treebank, *numbers*, the head of the NP, attaches to the predicate *died* instead.

```
#0 Eventually
 spine: a_( XP RBˆ )
#1 ,
 spine: a_,ˆ
#2 Mr.
 spine: a_NNPˆ
#3 Green
 spine: a_( XP NNPˆ )
 att #2, on 0, slot 0, order 0
#4 testified
 spine: b_( S ( VP VBDˆ S* ) )
 att #1, on 0, slot 0, order 0
 att #3, on 0, slot 0, order 1
 att #7, on 0.0, slot 1, order 0
#5 0
 spine: a_NONEˆ
#6 *t*-1
 spine: a_( S ( VP ( XP NONEˆ ) ) )
 att #5, on 0.0.0, slot 0, order 0
#7 ,
 spine: a_,ˆ
#8 he
 spine: a_( XP PRPˆ )
#9 began
 spine: a_( S ( VP VBDˆ ) )
 att #0, on 0, slot 0, order 0
 adj #4, on 0, slot 0, order 1
 att #6, on 0, slot 0, order 2
 att #8, on 0, slot 0, order 3
 att #11, on 0.0, slot 1, order 0
 att #25, on 0, slot 1, order 0
```

Figure 5.11: Parentheticals

```
#11 28
 spine: a_( XP CDˆ )
 att #12, on 0, slot 1, order 0
#12 *ich*-1
 spine: a_( XP NONEˆ )
#13 have
 spine: a_VBPˆ
#14 died
 spine: a_( S ( VP VBNˆ ) )
 att #0, on 0, slot 0, order 0
 att #10, on 0, slot 0, order 1
 att #11, on 0, slot 0, order 2
 att #13, on 0.0, slot 0, order 0
 att #15, on 0.0, slot 1, order 0
 att #22, on 0.0, slot 1, order 1
 att #23, on 0, slot 1, order 0
#15 --
 spine: a_:ˆ
#16 more
 spine: a_JJˆ
#17 than
 spine: a_INˆ
#18 three
 spine: a_( XP CDˆ )
 att #16, on 0, slot 0, order 0
 att #17, on 0, slot 0, order 1
 att #19, on 0, slot 1, order 0
#19 times
 spine: a_NNSˆ
#20 the
 spine: a_DTˆ
#21 expected
 spine: a_VBNˆ
#22 number
 spine: a_( XP NNˆ )
 att #18, on 0, slot 0, order 0
 att #20, on 0, slot 0, order 1
 att #21, on 0, slot 0, order 2
```

Figure 5.12: Extrapositions

5.5 Properties of the LTAG-spinal Treebank

We ran the extraction algorithm on 49,208 sentences in PTB. However, 454 sentences, or less than 1% of the total, are skipped. 314 of these 454 sentences have gapping structures. Since PTB does not annotate the trace of deleted predicates, additional manual annotation is required to handle these sentences. For the rest of the 146 sentences, abnormal structures are generated due to tagging errors.

5.5.1 Statistics

In the grammar extracted from the remaining 48,754 sentences in PTB, there are 1,224 different types of spinal elementary trees, and 507 of them appear only once in the LTAG treebank. However, there are 135 different *normalized* spinal elementary trees, and only 7 of them appear only once in the treebank. There are 1,159,198 tokens in the treebank. 2,365 of them are associated with full auxiliary trees, Roughly full adjunction appears in 2,365/48,754 = 5% of the stentences. Predicate coordination appears 8,467 times, which roughly accounts for 17% of the sentences.

5.5.2 Compatibility with Propbank

Since we have used Propbank in LTAG extraction, we are interested in the compatibility between this treebank and the Propbank annotation.

Propbank arguments are represented in phrases, while an LTAG derivation tree is similar to a dependency tree. So the first question is how to represent Propbank arguments with LTAG derivation trees.

We say that an argument is **well-formed** in the LTAG-spinal treebank if it can be generated by a subtree some of whose direct children trees may be cut away. For example, *and the stocks* is generated by a sub-derivation tree anchored on *stocks*, while *and*, and *the* sister adjoin to the tree for *stocks*. Then we say that the argument *the stocks* is well-formed because we can get it by cutting the *and* tree, a direct child of the *stocks* tree.

Table 5.1: Distribution of pred-arg pairs with respect to the distance between the predicate and the argument.

Distance	Number	Percent
0	261554	88.4
1	12287	4.2
2	10789	3.6
≥3	3426	1.2
ill-formed	1661	0.6
complex arg	6135	2.1
total	295852	100.0

As shown in Table 5.1, we have 295,852 pairs [1] of predicate-argument structures. Only 1661 arguments, 0.6% of all of the arguments, are not well-formed. Most of these cases are extraposition structures.

Now the question is whether it is easy to recover the scope of the argument from the derivation tree for the rest of the 294,191 arguments. By using less than 10 simple rules, for example, removing the subtrees for the punctuation marks at the beginning and at the end, we can easily recover 288,056, or 97.4% of all the arguments. For the remaining 6,135 arguments, more contextual information is required to recover the argument phrase. For example, we have a phrase *NP PP SBAR*, where both *PP* and *SBAR* adjoin to the *NP* as modifiers. Here *NP*, instead of *NP PP*, is an argument of the main verb of *SBAR*. In order to handle cases like these, some learning methods should be used. However, we have a baseline of 97.4% for this task, which is obtained by just ignoring these difficult cases.

The next question is how to find the subtree of an argument if we are given a predicate. The LTAG formalism has the desirable locality of the predicate-argument structures. We evaluate the LTAG-spinal treebank by studying the pattern of the path from the predicate to the argument for all the well-formed arguments. Table 5.1 shows the distribution of the distances between the predicate and the argument in the derivation trees. In the table, Number = number of pairs in the LTAG-spinal treebank; Percent = Number / 295,852 * 100. Distance = 0 means the predicate and the argument are directly connected.

[1] Particles are represented as arguments for the sake of convenience.

Table 5.2: Distribution of pred-arg pairs with respect to the path from the predicate to the argument.

	Path Pattern	Number	Percent
1	P → A	243796	82.4
2	P ← A	14658	5.0
3	P ← Px → A	10990	3.7
4	P ← Coord → Px → A	5613	1.9
5	V ← A	3100	1.0
6	P ← Ax ← Py → A	3028	1.0
7	P ← Coord ← Px → A	839	0.3
8	P ← Px ← Py → A	704	0.2
other patterns		5328	1.8
ill-formed		1661	0.6
complex arg		6135	2.1
total		295852	100.0

The following is a list of the most frequent patterns of the path from the predicate to the argument. *P* represents a predicate, *A* represents an argument, *V* represents a modifying verb, and *Coord* represents predicate coordination. Arrows point to the child from the parent. The number of the arrows minus 1 is the distance between the predicate and argument.

We also use A_x, P_x and P_y to represent other arguments or predicates appeared in the sentence.

1. **P → A**

   ex: (What)$_{arg1}$ (will)$_{argM}$ *happen* (to dividend growth)$_{arg2}$?

2. **P ← A** (relative clause, predicate adjunction)

   ex: (the amendment)$_{arg0}$ which *passed* today

   ex: (the price)$_{arg1.1}$ *appears* (to go up)$_{arg1.2}$

3. **P ← Px → A** (subject and object controls, Figure 5.13)

   ex: (It)$_{arg0}$ plans to *seek* approval.

4. **P ← Coord → Px → A** (shared arguments)

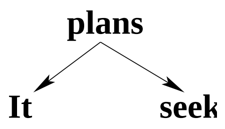


Figure 5.13: Pattern: **P** ← **Px** → **A**

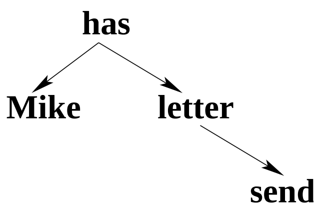


Figure 5.14: Pattern: **P** ← **Ax** ← **Py** → **A**

ex: (Chrysotile fibers)$_{arg1}$ are curly and are more easily *rejected* by the body.

5. **V** ← **A**

   ex: the Dutch *publishing* (group)$_{arg0}$

6. **P** ← **Ax** ← **Py** → **A** (Figure 5.14)

   ex: (Mike)$_{arg0}$ has a letter to *send*.

7. **P** ← **Coord** ← **Px** → **A** (control+coordination)

   ex: (It)$_{arg0}$ expects to *obtain* regulatory approval and *complete* the transaction.

8. **P** ← **Px** ← **Py** → **A** (chained controls, Figure 5.15)

   ex: (Officials)$_{arg0}$ began visiting about 26,000 cigarette stalls to *remove* illegal posters.

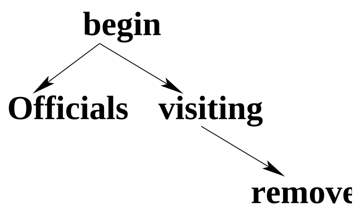


Figure 5.15: Pattern: **P** ← **Px** ← **Py** → **A**

These 8 patterns account for 95.5% of the total 295,852 pred-arg pairs in the treebank. Table 5.2 shows the frequency of these patterns.

Model	Section	Rec	Prc	F
rules on LTAG	all	91.0	92.3	91.6
rules on LTAG	23	90.8	91.7	91.3
SVMs on PTB	23	89.8	90.9	90.4

Table 5.3: Unlabeled non-trace argument identification.

5.5.3 Unlabeled Argument Identification

For the purpose of showing the compatibility of the LTAG-spinal treebank with the Propbank, here we present a preliminary experiment on unlabeled argument identification, which is used to generate all the argument candidates for an argument classification system.

Therefore we have implemented a very simple system for unlabeled argument identification. For each verbal predicate, we first collect all the sub-derivation trees in the local context based on path patterns 1, 2 and 5 in the previous section. If there is no argument candidate at the subject position, we look for the subject by collecting sub-derivation trees according to patterns 3 and 4. Then we transform these sub-derivation trees into phrases with a few simple rules as we have described in the previous section. We achieved an f-score = 91.3% for unlabeled non-trace argument identification on section 23 of this treebank [2].

To illustrate how good this result is, we compare it with the state-of-the-art system reported in (Pradhan et al., 2004), as shown in Table 5.3. In the table, Rec = Recall%, Prc = Precision%, F = f-score%.

In that paper, multi-class SVMs were used for argument identification and classification with rich features. The system using the baseline features for argument identification is the closest to our algorithm. The baseline feature set includes predicate lemma, gold standard pred-arg path in PTB, phrase label, position, voice, argument head word, and local phrase structure. Our rule-based algorithm only employs pred-arg path in the LTAG

[2]Section 23 of the LTAG-spinal treebank contains 2401 out of the 2416 sentences in PTB section 23.

derivation tree and POS tags.

Chapter 6

Reranking Approach

In recent years, reranking has been successfully applied to some NLP problems, especially to the problem of parse reranking (Collins, 2000). The reranking methods can also be applied to LTAG based analysis.

In (Shen et al., 2003), we proposed a system which extracted LTAG derivation trees from the N-best CFG parses generated by a state-of-the-art statistical CFG parser. We employed rich features as well as *tree kernels* to rerank the extracted LTAG derivation trees. In this way, we obtained an LTAG derivation tree based on a CFG parser.

The reranking method could also be used to rerank the N-best LTAG derivation trees generated by the parsers described in the previous two chapters. Therefore, it is interesting to us to design a better reranker.

However, in this chapter, we will report the experiments on two other reranking tasks instead, which are more useful to evaluate a reranker. One of these two tasks is CFG parse reranking. Many reranking methods have been used on this task, which makes it easy to compare the new reranking algorithm with previous works. The other task is Machine Translation reranking, which is more challenging than parse reranking.

In this chapter, we will introduce variants of the perceptron algorithm for the reranking task. In Section 6.1, we summarize the previous works on *reranking* in NLP research and *ranking* in Machine Learning. Then we investigate these works in the context of ranks and margins in Section 6.2, and propose general models for ranking and reranking in Section 6.3. In Section 6.4 we propose two new perceptron like algorithms in the new framework. We will justify these two algorithms in Section 6.5. The new algorithms are applied to parse reranking and machine translation reranking in Section 6.6.

6.1 Previous Works

6.1.1 Reranking in NLP

Global features are useful in many NLP systems. However, the introduction of global features results in difficulties in dynamic programming of the generative models. In recent years, the so-called reranking techniques (Collins, 2000) have been successfully used in many applications, which were previously modeled as generative models. The basic idea of reranking is as follows. A baseline generative model generates N-best candidates, and then these candidates are reranked by using a rich set of local and global features. Usually, only the top candidate of the reranked results is used, so reranking is also called *selection* or *voting* in some cases.

Now we present a brief survey of the previous works on reranking.

Ratnaparkhi (1997) noticed that by ranking the 20-best parsing results generated by his maximal entropy parser, the F-measure could be improved to 93% from 87%, if the oracle parse could be successfully detected.

Collins (2000) first used machine learning algorithms in parse reranking. Two approaches were proposed in that paper; one used Boost Loss and the other used Log-Likelihood Loss. The approach of Boost Loss achieved better results. The Boost Loss model is as follows. Let $\mathbf{x}_{i,j}$ be the feature vector of the j^{th} parse of the i^{th} sentence. Let $\tilde{\mathbf{x}}_i$ be the feature vector of best parse for the i^{th} sentence [1]. Let F_w be a score function

$$F_w(\mathbf{x}_{i,j}) \equiv \mathbf{w} \cdot \mathbf{x}_{i,j},$$

[1]By best we mean the parse that is the closest to the gold standard.

where $\mathbf{w}$ is a weight vector. The *margin* $M_{w,i,j}$ on example $\mathbf{x}_{i,j}$ is defined as

$$M_{w,i,j} \equiv F_w(\tilde{\mathbf{x}}_i) - F_w(\mathbf{x}_{i,j})$$

Finally the Boost Loss function is defined as

$$BoostLoss(w) \equiv \sum_i \sum_j e^{-(F_w(\tilde{\mathbf{x}}_i) - F_w(\mathbf{x}_{i,j}))} = \sum_i \sum_j e^{-M_{w,i,j}}$$

The Boosting algorithm was used to search the weight vector $\mathbf{w}$ to minimize the Boost Loss.

We may rewrite the definition of the margin $M_{w,i,j}$ by using pairwise samples as follows.

$$\mathbf{s}_{i,j} \equiv \tilde{\mathbf{x}}_i - \mathbf{x}_{i,j}$$

then

$$M_{w,i,j} = F_w(\tilde{\mathbf{x}}_i) - F_w(\mathbf{x}_{i,j}) = F_w(\tilde{\mathbf{x}}_i - \mathbf{x}_{i,j}) = F_w(\mathbf{s}_{i,j})$$

So the Boost Loss approach in (Collins, 2000) is equivalent to maximizing the margin distribution (Schapire et al., 1997) between 0 and $F_w(\mathbf{s}_{i,j})$, where $\mathbf{s}_{i,j}$ are pairwise samples as we have described above.

In (Collins and Duffy, 2002), a perceptron like reranking algorithm was applied to parse reranking. Similar to (Collins, 2000), pairwise samples were used as training samples, although implicitly. The perceptron updating step was defined as

$$\mathbf{w}^{t+1} = \mathbf{w}^t + \tilde{\mathbf{x}}_i - \mathbf{x}_{i,j},$$

where $\mathbf{w}^t$ was the weight vector at the t^{th} updating. This is equivalent to using pairwise sample $\mathbf{s}_{i,j}$ which we have defined above.

$$\mathbf{w}^{t+1} = \mathbf{w}^t + \mathbf{s}_{i,j}$$

In (Shen and Joshi, 2003), we applied Support Vector Machines to parse reranking. In that paper, pairwise samples were used explicitly through the *Preference* kernel. $\mathbf{u}^+_{i,j}$ and $\mathbf{u}^-_{i,j}$, defined below, were used as positive samples and negative samples respectively.

$$\mathbf{u}^+_{i,j} \equiv (\tilde{\mathbf{x}}_i, \mathbf{x}_{i,j}), \;\; \mathbf{u}^-_{i,j} \equiv (\mathbf{x}_{i,j}, \tilde{\mathbf{x}}_i)$$

SVMs were used to maximize the margin between positive samples and negative samples, which in turn was proportional to the margin between the best parse of each sentence and the rest of the N-best parses.

In summary, in the works on reranking, the margin is defined as the distance between the best candidate and the rest. The reranking problem is reduced to a classification problem by using pairwise samples implicitly or explicitly.

6.1.2 Ranking and Ordinal Regression

Ranking is an important learning task in the field of machine learning, and it looks similar to the reranking tasks in NLP. It is interesting to us whether the methods previously used in ranking could be applied to reranking.

Obviously, ranking is a problem that lies between classification and regression. In some previous works, ranking was reduced to a classification problem by using pairwise *items* as training samples (Herbrich et al., 2000), by item we mean a candidate in the N-best list. This will increase the data complexity from $O(n)$ to $O(n^2)$ in the case of full set of pairs, where n is the size of the candidate list. To avoid this, a subset of the pairs are utilized. However, this results in the loss of some information in the training data for reranking. In some other approaches (Crammer and Singer, 2001b), extra biases are introduced to avoid using pairwise items as samples; each bias represents the boundary of two neighboring ranks on the score metric. But the use of extra biases prevents it from being used in reranking tasks, as we will explain later.

Crammer and Singer (2001b) proposed the *PRank* algorithm, a perceptron based ranking algorithm. In their framework each instance is associated with a rank, which is an integer from 1 to k. The goal of their ranking algorithm is to predict the correct rank of each instance. The PRank algorithm is a variant of the perceptron algorithm. The difference is that the PRank algorithm maintains a set of biases which are used as boundaries between two neighboring ranks.

PRank works very well for the ranking problems in which each sample is associated with a rank. However, due to the introduction of a set of biases, they are constrained from their use in other ranking-like problems. For example, the PRank algorithm cannot be trained on the data associated with a partial order instead of order on ranks. Furthermore, as we will show later, the PRank algorithm cannot handle the reranking problems.

Herbrich et al. (2000) proposed a margin based approach for ranking, or *ordinal regression* as they called it in their paper. In their framework, each training sample is associated with a rank which is an integer. The target function is required to maximize the margins between the samples of neighboring ranks. The Support Vector Machines (Vapnik, 1998) (SVMs) were used to compute the linear function maximizing the margins. In contrast to PRank, rank boundaries were not used explicitly in the training. Their approach is implemented by using pairwise samples for training. For example, we have the feature vectors for two samples, $\mathbf{u}$ and $\mathbf{v}$, where the sample of $\mathbf{u}$ ranks i and the sample of $\mathbf{v}$ ranks $i+1$, then $\mathbf{u}-\mathbf{v}$ is used as a positive sample and $\mathbf{v}-\mathbf{u}$ is used a negative sample. The *Preference* kernel was used to incorporate kernels defined on the space of single items. In SVM^{light}, a well known SVM package, the ranking module is implemented in this way (Joachims, 2002).

The underlying assumption of ordinal regression is that samples between consecutive ranks are separable. This may become a problem in the case that ranks are unreliable because ranking is too fine. This is just what happens in machine translation reranking. On the other hand, if ranking is rather sparse, the size of generated training samples will be very large. Suppose there are n samples evenly distributed on k ranks. The total number of pairwise samples in (Herbrich et al., 2000) is roughly n^2/k.

To sum up, in the previous works on ranking or ordinal regression, the margin is defined as the distance between two consecutive ranks. Two approaches have been used. One is to extend the perceptron algorithm by using multiple biases to represent the boundaries between every two consecutive ranks. The other approach is to reduce the ranking problem to a classification problem by using pairwise samples.

6.2 Ranks and Margins

Our initial goal is to adapt ranking algorithms to reranking. However, we note that there are a few fundamental differences between ranking and reranking, which make it hard to use ranking directly for reranking. On the other hand, both ranking and reranking employ pairwise samples to transform the original problem into a classification problem. We will examine these issues in the this section.

6.2.1 Locality of Ranks

First, in the previous works on ranking, ranks are defined on the whole training and test data. Thus we can define boundaries between consecutive ranks on the whole data. In the reranking problem, ranks are defined over a *group* of the samples in the data set. For example, in the parse reranking problem, the rank of a parse is only the rank among all the parses for the same sentence. The training data include 36,000 sentence, with an average of over 27 parses per sentence (Collins, 2000).

As a result, we cannot use the PRank algorithm in the reranking task, since there are no global ranks or boundaries for all the samples. If we introduce auxiliary variables for the boundaries for each group, the number of the parameters will be as large as the number of samples. Obviously this is not a good idea. However, the approach of using pairwise samples still works. By pairing up two samples, we actually compute the relative distance between these two samples with respect to the scoring metric. In the training phase, we are only interested in whether the relative distance is positive or negative, and we do not need to compare it with any specific numbers.

To sum up, the locality of ranks in the reranking problem leads us to the approach of using pairwise samples.

6.2.2 Density of Ranks

In the previous works on ranking, the number of samples is much larger than the number of ranks. For example, the Information Retrieval (IR) data used in (Herbrich et al., 2000) has only 3 ranks (“document is relevant”, “document is partially relevant”, and “irrelevant document”), the synthetic data sets used in (Herbrich et al., 2000; Crammer and Singer, 2001b) have 5 ranks, and the EachMovie dataset used in (Crammer and Singer, 2001b; Harrington, 2003) has 6 ranks. In these applications, difference between samples of different ranks is significant, so it is natural to define margin as distance between consecutive ranks.

However, this approach is problematic if ranks become denser and the samples of consecutive ranks become linearly inseparable. For example, in the IR data used in (Herbrich et al., 2000), we are not only interested in whether a document is relevant or not, but also the extent to which the document is relevant. In this case, pairwise samples defined only on the consecutive ranks are not enough for training, since ranks within a small range is unstable; it is hard to say if a document ranked i is more relevant than a document ranked $i+1$.

However, ranks in a large range is reliable; a document ranked 5 is more relevant than a document ranked 15. So we may want to use more pairwise samples. For example, we may use (doc_5, doc_{15}) as a positive sample, and use (doc_{15}, doc_5) as a negative sample. The extreme case is pairing up all the samples. This will increase the size of sample space by N times, where N is size of a group. To sum up, in the case of dense ranks, the full pairwise samples or a subset of the full pairwise samples are required for training.

Obviously, the ranks in reranking are *dense*. Each training group has a ranked list of N-best candidate. For parse reranking $N = 27$ on average (Collins, 2000), and for machine translation reranking $N = 1000$ or more (Och et al., 2004). Therefore, pairwise samples defined only on consecutive ranks are not enough in the training of reranking.

6.2.3 Full Pairwise for Reranking

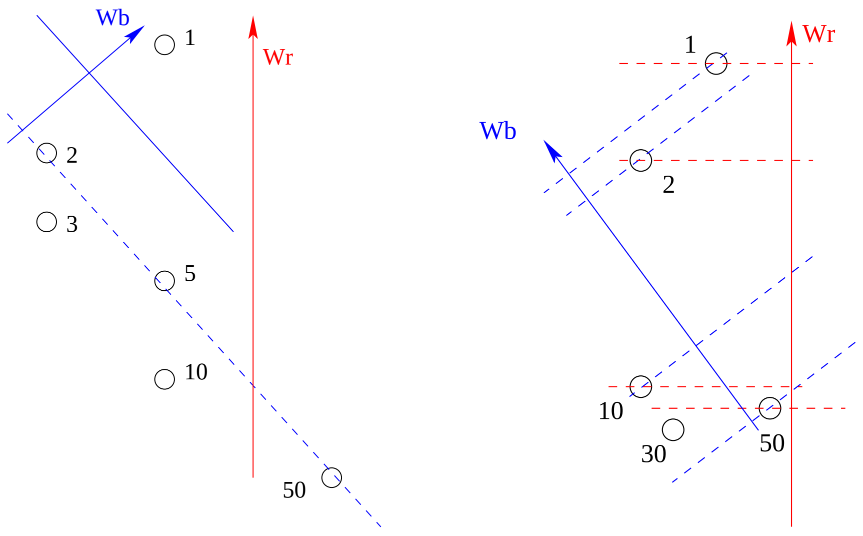
As we have shown above, in the previous works on reranking, we search for the hyperplane that separates the best candidate from the rest of the candidates within each group. Given a group, let $\mathbf{r}_i$ be the i^{th} best item within the group. If we only look for the hyperplane to separate the best one from the rest, we, in fact, discard the order information of $\mathbf{r}_2 ... \mathbf{r}_N$. For example, we did not employ the information that $\mathbf{r}_2$ is better than $\mathbf{r}_{50}$ in the training, as shown in Figure 6.1.a.

We may regard the weight vector $\mathbf{w}$ as a score metric, which assigns the highest score to the best item within each group, i.e. $\mathbf{r}_1$. So the question is whether it should assign the second highest score to $\mathbf{r}_2$, the third highest score to $\mathbf{r}_3$, and so on so forth, since $\mathbf{w}$ is a score metric. Therefore, in Figure 6.1.a, $\mathbf{w}_r$ is more preferable than $\mathbf{w}_b$ since $\mathbf{w}_r$ is able to keep the order of items with respect to their goodness.

In order to search for $\mathbf{w}_r$ instead of $\mathbf{w}_b$, we need to employ more ordering relations in the training. One solution is to use a set of items as *good* candidates, $\mathbf{r}_{good}$, and use the rest as bad candidates, $\mathbf{r}_{bad}$. For candidates $\mathbf{r}_{good}, \mathbf{r}_{bad}$ within the same group, $(\mathbf{r}_{good}, \mathbf{r}_{bad})$ is defined as a positive sample, and $(\mathbf{r}_{bad}, \mathbf{r}_{good})$ is defined a negative sample. In Section 6.3, we will use the $splitting$ model to capture this idea.

However, there is still a problem with this solution, i.e. it does not distinguish the differences between the good samples. In the case of machine translation for which N can be as large as 1000 or even more, $\mathbf{r}_1$ and $\mathbf{r}_{300}$ will be both regarded as good candidates for example, so the $splitting$ model does not try to assign a higher score to $\mathbf{r}_1$ than to $\mathbf{r}_{300}$.

As far as this aspect is concerned, we apply the ordinal regression model to reranking with full pairwise samples. That is to say, for every two candidates $\mathbf{r}_i$ and $\mathbf{r}_j$, $i < j$, within the same group, $(\mathbf{r}_i, \mathbf{r}_j)$ is used as a positive sample, and $(\mathbf{r}_j, \mathbf{r}_i)$ is used as a negative sample. However, in the next section we will show that ordinal regression is not a desirable model for reranking either.



a. Full Pairwise b. Uneven Margins

Figure 6.1: Comparison of score metrics

6.2.4 Uneven Margins

Reranking is not an ordinal regression problem. In reranking evaluation, we are only interested in the quality of the item with the highest score in each group, and we do not care the order of the other items. Therefore we cannot simply regard a reranking problem as an ordinal regression problem, since they have different loss functions.

Especially, we want to maintain a larger margin in items of high ranks and a smaller margin in items of low ranks. For example, in Figure 6.1.b, $\mathbf{w}_b$ meets the condition of ordinal regression by keeping the order of all candidates within a group. However, there is a small margin between $\mathbf{r}_1$ and $\mathbf{r}_2$, which means that if we are given two candidates similar to $\mathbf{r}_1$ and $\mathbf{r}_2$ in the test set, $\mathbf{w}_b$ is very likely to switch their order. However, $\mathbf{w}_r$ maintains a large margin between $\mathbf{r}_1$ and $\mathbf{r}_2$. Hence, $\mathbf{w}_r$ is more desirable even it fails to keep the order of some bad candidates, i.e. $\mathbf{r}_{30}$ and $\mathbf{r}_{50}$, but they are far away from the top candidates. According to the loss function, we are penalized if we switch the order of $\mathbf{r}_1$ and $\mathbf{r}_2$, but not for $\mathbf{r}_{30}$ and $\mathbf{r}_{50}$.

In order to handle this, we will introduce the idea of uneven margins in the training. The technique of uneven margins has been previously employed in binary classification on unbalanced data (Li et al., 2002), for which there are many more negative samples than positive samples.

Uneven margins were also used in the optimization of Max-Margin Markov Network (MMMN) in (Taskar et al., 2003) [2]. It maximizes the distance between the gold standard hypothesis and all the other hypotheses with uneven margins proportional to the loss of each incorrect hypothesis, which is similar to the learning criteria of previous classification-like reranking algorithms. The optimization criterion of MMMN is as follows

$$\begin{array}{rl} \text{minimize} & ||\mathbf{w}|| \\ \text{such that} & \forall i,j \quad \mathbf{w}\cdot\tilde{\mathbf{x}}_i - \mathbf{w}\cdot\mathbf{x}_{i,j} \geq L(\mathbf{x}_{i,j}), \end{array}$$

where $\tilde{\mathbf{x}}_i$ is the gold standard for the i^{th} sentence, and $\mathbf{x}_{i,j}$ is the j^{th} hypothesis for the i^{th} sentence. This framework of optimization originates from the multi-class classification model developed by Crammer and Singer (2001a). Recently, the use of uneven margins were also incorporated in other perceptron like learning algorithms, for example, as in MIRA with hinge loss (Crammer, 2004).

Ordinal Regression with Uneven Margins (ORUM) is more desirable for reranking. Specifically, we search for a linear function separating items in each group with uneven margins, for example

$$margin(\mathbf{r}_1,\mathbf{r}_{30}) > margin(\mathbf{r}_1,\mathbf{r}_{10}) > margin(\mathbf{r}_{21},\mathbf{r}_{30}) \tag{6.1}$$

It is not difficult to exhibit a function that satisfies (6.1), for example

$$g(\mathbf{r}_i,\mathbf{r}_j) = \frac{1}{i} - \frac{1}{j} \tag{6.2}$$

Obviously, there are many other candidates for the function of margin ratio, for example, like the product of g and the difference of losses. It would be interesting to investigate

[2]The model of uneven margins for ordinal regression was developed during the JHU Summer Workshop 2003. It was independent of the work of MMMN.

which margin function is theoretically desirable. However, in this chapter, we will simply use the margin function as in (6.2) in our experiments.

6.3 Models for Ranking and Reranking

6.3.1 Problem Definition

We first formalize the reranking problem.

Item $\mathbf{x} \in \mathcal{R}^d$ is a d-dimensional vector on the real number. It represents a single parse, or a single translation in applications.

Group $\mathbf{c} = (\mathbf{x}_1, \ldots, \mathbf{x}_k) \in (\mathcal{R}^d)^k$ is a vector of k items. It represents a ranked list of parses or translations for the same source sentence in applications.

Rank $\mathbf{r}_\mathbf{c} = (y_1, \ldots, y_k) \in \mathcal{N}^k$, where y_i is the rank of item $\mathbf{x}_i$ in $\mathbf{c}$, $1 \le y_i \le k$.

Learning Let $\mathcal{X}$ be the space of groups, $\mathcal{X} = (\mathcal{R}^d)^k$, and let $\mathcal{Y}$ be the space of ranks, $\mathcal{Y} = \mathcal{N}^k$. The hypothesis class $\mathcal{H}$ is $\mathcal{X} \to \mathcal{Y}$. A learning algorithm $\mathcal{L}((\mathcal{X} \times \mathcal{Y})^m)$ takes m *i.i.d.* drawn training groups associated with ranks from $\mathcal{X} \times \mathcal{Y}$ according to distribution $\mathcal{D}_{\mathcal{X} \times \mathcal{Y}}$, and outputs a hypothesis $h \in \mathcal{H}$.

Margin is a function on $\mathcal{X} \times \mathcal{Y} \times \mathcal{H} \to \mathcal{R}$.

6.3.2 Splitting

In this framework, we will first propose a *splitting* model, which is similar to previous works on reranking, but in a more general form.

Let the training samples be

$$S = \{(\mathbf{x}_{i,j}, y_{i,j}) \mid 1 \le i \le m,\ 1 \le j \le k\},$$

where m is the number of groups and k is the length of ranks for each group.

Let $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$ be a linear function, where $\mathbf{x}$ is an item, and $\mathbf{w}$ is a weight vector. We construct a hypothesis function $h_f : \mathcal{X} \rightarrow \mathcal{Y}$ with f as follows.

$$h_f(\mathbf{x}_1, ...\mathbf{x}_k) = rank(f(\mathbf{x}_1), ..., f(\mathbf{x}_k)),$$

where *rank* is a function that returns the vector of ranks for the input vector. For example $rank(4.2, 9.0, 6.5, 8.8) = (4, 1, 3, 2)$.

An *r-splitting* algorithm searches for a linear function $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$ that successfully splits the top r-ranked items from the rest of the items in every group. Let $\mathbf{y}^f$ = $(y_1^f, ..., y_k^f)$ = $h_f(\mathbf{x}_1, ...\mathbf{x}_k)$ for any linear function f. We look for a function f such that

$$y_i^f \leq r \quad \text{if} \quad y_i \leq r \tag{6.3}$$

$$y_i^f > r \quad \text{if} \quad y_i > r \tag{6.4}$$

Suppose there exists a linear function f satisfying (6.3) and (6.4), we say $\{(\mathbf{x}_{i,j}, y_{i,j})\}$ is *r-splittable* by f. Furthermore, we can define the *splitting margin* γ for group $\mathbf{c}_i$ as follows.

$$\gamma(f, r, i) = \min_{j: y_{i,j} \leq r} f(\mathbf{x}_{i,j}) - \max_{j: y_{i,j} > r} f(\mathbf{x}_{i,j})$$

The *minimal splitting margin*, γ^{split}, for f and r is defined as follows.

$$\gamma^{split}(f, r) = \min_i \gamma(f, r, i) = \min_i (\min_{y_{i,j} \leq r} f(\mathbf{x}_{i,j}) - \max_{y_{i,j} > r} f(\mathbf{x}_{i,j}))$$

Obviously, if $r = 1$, the *r-splitting* searches for a linear function f that can successfully separate the best item from the rest in every group, which is similar to some previous reranking algorithms (Collins, 2000; Collins and Duffy, 2002; Shen and Joshi, 2003).

6.3.3 Ordinal Regression on Groups

In this section we will model a more complicated problem, ordinal regression on groups. Let $\mathbf{x}_{i,j}, \mathbf{x}_{i,l}$ be two items where $y_{i,j} < y_{i,l}$. It means that the rank of $\mathbf{x}_{i,j}$ of higher than the rank of $\mathbf{x}_{i,l}$. We are interested in finding a weight vector $\mathbf{w}$, such that

$$\mathbf{w} \cdot \mathbf{x}_{i,j} > \mathbf{w} \cdot \mathbf{x}_{i,l} + \tau, \text{ if } y_{i,j} < y_{i,l}$$

Let the training samples be

$$S = \{(\mathbf{x}_{i,j}, y_{i,j}) \mid 1 \leq i \leq m,\ 1 \leq j \leq k\},$$

where m is the number of groups and k is the length of ranks for each group. Let $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$. We say the f is an ordinal regression function for the training set S if

$$\mathbf{w} \cdot \mathbf{x}_{i,j} > \mathbf{w} \cdot \mathbf{x}_{i,l}, \text{ if } y_{i,j} < y_{i,l},$$

for $1 \leq i \leq m,\ 1 \leq j,l \leq k$

Suppose f is an ordinal regression function for the training set S, the *regression margin* for group $\mathbf{c}_i$ is defined as follows

$$\gamma(f,i) = \min_{y_{i,j} < y_{i,l}} f(\mathbf{x}_{i,j}) - f(\mathbf{x}_{i,l})$$

The *minimal regression margin*, γ^{order}, for f is defined as follows.

$$\gamma^{order}(f) = \min_i \gamma(f,i)$$

6.3.4 Pairwise Classification

The two models described in the previous two sections can be generalized as a pairwise classification problem. Let the training samples be

$$S = \{(\mathbf{x}_{i,j}, y_{i,j}) \mid 1 \leq i \leq m,\ 1 \leq j \leq k\},$$

where m is the number of groups and k is the length of ranks for each group.

Let $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$. We say the training data is *separable* with respect to T by f if

$$\mathbf{w} \cdot \mathbf{x}_{i,j} > \mathbf{w} \cdot \mathbf{x}_{i,l}, \text{ if } (y_{i,j}, y_{i,l}) \in T,$$

for $1 \leq i \leq m,\ 1 \leq j,l \leq k$, and $T \subseteq \mathcal{K} \times \mathcal{K}$ is a partial order on $\mathcal{K}$, where $\mathcal{K} = \{1, 2, ..., k\}$.

It is not difficult to see that both splitting and ordinal regression can be reduce to the model given above. For splitting,

$$(y_{i,j}, y_{i,l}) \in T \iff y_{i,j} \leq r < y_{i,l},$$

and for ordinal regression

$$(y_{i,j}, y_{i,l}) \in T \iff y_{i,j} < y_{i,l}.$$

Furthermore, this model can be transformed into a binary classification problem as follows. $\forall (y_{i,j}, y_{i,l}) \in T, \mathbf{x}_{i,j} - \mathbf{x}_{i,l}$ is defined as a positive sample, and $\mathbf{x}_{i,l} - \mathbf{x}_{i,j}$ is defined as a negative sample. Therefore, the model above is equivalent to finding a linear function separating the positive and negative samples. The classification margin is equivalent to splitting and regression margin respectively.

6.3.5 Pairwise Classifi cation with Uneven Margins

Let the training samples be

$$S = \{(\mathbf{x}_{i,j}, y_{i,j}) \mid 1 \leq i \leq m,\ 1 \leq j \leq k\},$$

where m is the number of groups and k is the length of ranks for each group.

Let $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$, where $||\mathbf{w}|| = 1$. We say the training data is *separable* with respect to T by f with margin τ weighted with margin function g if

$$\mathbf{w} \cdot (\mathbf{x}_{i,j} - \mathbf{x}_{i,l}) > g(y_{i,j}, y_{i,l})\tau, \text{ if } (y_{i,j}, y_{i,l}) \in T, \tag{6.5}$$

for $1 \leq i \leq m,\ 1 \leq j, l \leq k,\ T \subseteq \mathcal{K} \times \mathcal{K}$ is a partial order on $\mathcal{K}$, and $g \in \mathcal{K}^2 \rightarrow \mathcal{R}$ is a margin function, where $\mathcal{K} = \{1, 2, ..., k\}$.

6.3.6 Single Group

These models for multi-group samples can be easily reduced to 1-group case, which can be used to model the ranking problem.

Item $\mathbf{x} \in \mathcal{R}^d$ is a d-dimensional vector on the real number.

Rank $y \in \mathcal{N}$, where $1 \leq y \leq k$ for some fixed number k.

Algorithm 6 Pairwise Classification with Uneven Margins

```
Require: ${(\mathbf{x}_{i,j}, y_{i,j})}$ is separable with respect to $T$.
Require: a margin function $g$
Require: a positive learning margin $\tau$.
 1: $t \leftarrow 0$, initialize $\mathbf{w}^0$;
 2: repeat
 3:   for ($i = 1, \ldots, m$) do
 4:     for ($1 \leq j, l \leq k$) do
 5:       if ($(y_{i,j}, y_{i,l}) \in T$ and $\mathbf{w}^t \cdot (\mathbf{x}_{i,j} - \mathbf{x}_{i,l}) \leq g(y_{i,j}, y_{i,l})\tau$) then
 6:         $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t + g(y_{i,j}, y_{i,l})(\mathbf{x}_{i,j} - \mathbf{x}_{i,l})$
 7:         $t \leftarrow t + 1$
 8:       end if
 9:     end for
10:   end for
11: until no updates made in the outer for loop
```

Learning Let $\mathcal{X}$ be the space of items, $\mathcal{X} = \mathcal{R}^d$, and let $\mathcal{Y}$ be the space of ranks, $\mathcal{Y} = \mathcal{N}$. The hypothesis class $\mathcal{H}$ is $\mathcal{X} \rightarrow \mathcal{Y}$. A learning algorithm $\mathcal{L}((\mathcal{X} \times \mathcal{Y})^m)$ takes m *i.i.d.* drawn training groups associated with ranks from $\mathcal{X} \times \mathcal{Y}$ according to distribution $\mathcal{D}_{\mathcal{X} \times \mathcal{Y}}$, and outputs a hypothesis $h \in \mathcal{H}$.

Margin is a function defined on $\mathcal{X} \times \mathcal{Y} \times \mathcal{H} \rightarrow \mathcal{R}$.

In the case of a single group, items, instead of groups, are drawn *i.i.d.* with respect to some distribution.

6.4 Training Algorithms

In this section, we will introduce perceptron like algorithms to employ uneven margins for reranking.

6.4.1 Perceptron over Full Pairwise Samples

Algorithm 6 is used to solve the Pairwise Classification with Uneven Margins (PCUM) model proposed in Section 6.3.5. The idea of Algorithm 6 is as follows. For every two items $\mathbf{x}_{i,j}$ and $\mathbf{x}_{i,l}$, if

- $(y_{i,j}, y_{i,l}) \in T$, and
- the weight vector $\mathbf{w}$ can not successfully separate $\mathbf{x}_{i,j}$ and $\mathbf{x}_{i,l}$ with a learning margin $g(y_{i,j}, y_{i,l})\tau$,

then we need to update $\mathbf{w}$ with the addition of $g(y_{i,j}, y_{i,l})(\mathbf{x}_{i,j} - \mathbf{x}_{i,l})$.

In Section 6.5 we will give the theoretical justification for algorithm 6.

If T is the full order, the size of T is $k(k-1)/2$, where k is the size of a group. If we represent all the pairwise sample explicitly, the data complexity will be $dmk(k-1)/2$, where d is the dimensionality of a sample. In machine translation reranking (Och et al., 2004), we have roughly 1000 source sentences, each of which has 1000 best translations, and each candidate is represented with a vector of 20 real-valued features. In this case, the size of pairwise samples is roughly 40G bytes. Thus, we can only compute pairwise samples dynamically in our algorithm.

In Algorithm 6, for each iteration of a group, we need execute the dot product operation $|T|$ times, which is usually $O(k^2)$. Thus the complexity of outer iteration is $O(k^2 d)$. In the next section, we will give an alternative algorithm, which reduces the complexity of outer iteration to $O(k^2 + kd)$.

6.4.2 Fast Perceptron Training

Algorithm 7 is similar to Algorithm 6 except that the updating operation is executed on the group level instead of sample level. For each iteration of a group, we first compute $\mathbf{w} \cdot \mathbf{x}_{i,j}$ for each item $\mathbf{x}_{i,j}$. Then we use these values to check whether an update on $\mathbf{w}$ is required for each pair in T. However, the update is not executed immediately. Instead, for each item, there is a valuable $u_{i,j}$ that records all the updates. After all the pairs are checked, updating is executed once and for all.

It is easy to show that the complexity of each iteration of a group is $O(k^2 + kd)$. In Section 6.5 we will show that Algorithm 7 is theoretically correct, and in the experimental

Algorithm 7 Fast Pairwise Classification with Uneven Margins

```
Require: $\{(\mathbf{x}_{i,j}, y_{i,j})\}$ is separable with respect to $T$.
Require: a margin function $g$
Require: a positive learning margin $\tau$.
 1: $t \leftarrow 0$, initialize $\mathbf{w}^0$;
 2: repeat
 3:    for ($i = 1, \ldots, m$) do
 4:       for ($j = 1, \ldots, k$) do
 5:          compute $\mathbf{w}^t \cdot \mathbf{x}_{i,j}$ for all $j$'s;
 6:          $u_j \leftarrow 0$;
 7:       end for
 8:       for ($1 \leq j, l \leq k$ ) do
 9:          if (($y_{i,j}, y_{i,l}) \in T$ and $\mathbf{w}^t \cdot (\mathbf{x}_{i,j} - \mathbf{x}_{i,l}) \leq g(y_{i,j}, y_{i,l})\tau$) then
10:             $u_j \leftarrow u_j + g(y_{i,j}, y_{i,l})$; $u_l \leftarrow u_l - g(y_{i,j}, y_{i,l})$;
11:          end if
12:       end for
13:       if ($\sum_j |u_j| > 0$) then
14:          $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t + \sum_j u_j \mathbf{x}_{i,j}$;
15:          $t \leftarrow t + 1$;
16:       end if
17:    end for
18: until no updates made in the outer for loop
```

section we will show the superiority of Algorithm 7 over Algorithm 6.

6.5 Theoretical Justification

In the previous subsections, we have proposed two perceptron based large margin algorithms for splitting and ordinal regression models. Now we show that these two algorithms stop in finite steps if the training data is separable, and present a lower bound of the resulting margins.

6.5.1 Justification for Algorithm 1

Theorem 6.1 gives us an upper bound on the number of steps needed for Algorithm 6 to stop, if the training data is separable. The proof for Theorem 6.1 is given the Appendix B.

Theorem 6.1 *Suppose the training samples* $\{(\mathbf{x}_{i,j}, y_{i,j})\}$ *are separable with respect to T by a linear function defined on the weight vector* $\mathbf{w}^*$ *with a margin* γ *weighted on margin function g as in (6.5), where* $||\mathbf{w}^*|| = 1$. *Let* $R = \max_{i,j} ||\mathbf{x}_{i,j}||$ *and* $\lambda = \min_{i,j,l}\ g(y_{i,j}, y_{i,l})$. *Then we have*

a *Algorithm 6 stops in t steps of updates, where*

$$t \le \frac{2\tau + 4R^2}{\lambda^2\gamma^2} \tag{6.6}$$

b *The resulting weight vector separates any two item* $\mathbf{x}_{i,j}$ *and* $\mathbf{x}_{i,l}$, *where* $y_{i,j} < y_{i,l}$, *with margin*

$$\gamma_{i,j,l} \ge g(y_{i,j}, y_{i,l})\gamma\frac{\tau}{2\tau + 4R^2} \tag{6.7}$$

According to (6.7), if $\tau >> R$, Algorithm 6 stops with resulting margins at least half of the optimum margins, $g(y_{i,j}, y_{i,l})\gamma$. This result can be viewed as an extension of (Krauth and Mezard, 1987), as described in Section 1.3.2.

6.5.2 Justifi cation for Algorithm 2

For Algorithm 7, we will show that if the training data is separable, the algorithm will stop in a finite number of step with a resulting margin on the training data. For the sake of simplicity, we will only take a weak version of the algorithm, by assuming $g(y_{i,j}, y_{i,l}) \equiv 1$ for all i, j, l. However, the theorem and the proof given here can be extended to the original algorithm.

Theorem 6.2 *Suppose the training samples* $\{(\mathbf{x}_{i,j}, y_{i,j})\}$ *are separable by a linear function defined on the weight vector* $\mathbf{w}^*$ *with a margin* γ, *where* $||\mathbf{w}^*||$=1. *Let* R =$\max_{i,j} ||\mathbf{x}_{i,j}||$. *Then we have*

a *Algorithm 7 makes at most* $\frac{2\tau + k^2R^2}{\gamma^2}$ *mistakes on the pairwise samples during the training.*

b Algorithm 7 stops in t steps of updates, where

$$t \leq \frac{2\tau + k^2R^2}{\gamma^2} \tag{6.8}$$

c The resulting margin on the training data is at least

$$\gamma\frac{\tau}{2\tau + k^2R^2} \tag{6.9}$$

The proof of Theorem 6.2 is given in the Appendix B.

It should be noted that Algorithm 7 needs more iterations for convergence than Algorithm 6 theoretically, while the former runs much faster in each iteration. Experiments given in the next section show that the running time for Algorithm 7 is shorter, and the result is more stable.

6.5.3 Inseparable Data

The theorems given in the previous two sections apply to separable data only. One way to handle inseparable data is to make the training data artificially separable, which is called the λ-trick as described in (Herbrich, 2002).

The idea of the λ-trick is the following. Suppose we have m groups, and for each group we have k hypotheses. We augment each sample $\mathbf{x}_{i,j}$ with a vector $\sqrt{\lambda}\mathbf{e}_{ik+j}$, where parameter $\lambda > 0$ and unit vector $\mathbf{e}_{ik+j} \in R^{mk}$. It is easy to verify that, if λ is large enough, the augmented samples are always separable. However, we always try to use a relatively small value for λ. Intuitively, the weight for an augmented dimension denotes the weight updates in learning. We will use this technique in our experiments.

Another way to handle inseparable data is to use voted perceptron in (Freund and Schapire, 1999), as described in Section 1.3.2. However, our experiments show no improvement by using both voted perceptron and λ-trick.

6.6 Experiments and Analysis

In this section, we show the experimental results on two NLP tasks, parse reranking and discriminative reranking for machine translation, which were partially reported in (Shen and Joshi, 2004) and (Shen et al., 2004) respectively. A single Pentium III 1.13GHz CPU with 2GB memory is used for all experiments. The algorithms are coded in Java, running on the Linux operating system.

6.6.1 Parse Reranking

For parse reranking, we use the same data set as described in (Collins, 2000). Section 2-21 of the WSJ Penn Treebank (PTB) (Marcus et al., 1994) are used as training data, and Section 23 is used for testing. The training data contains around 40,000 sentences, each of which has 27 distinct parses on average. Of the 40,000 training sentences, the first 36,000 are used to train Perceptrons. The remaining 4,000 sentences are used as development data for parameter estimation, such as the number of rounds of iteration in training. The 36,000 training sentences contain 1,065,620 parses in total.

We use the same feature set as in Collins (Collins, 2000). Features are defined on the fragments of CFG parse trees. There are 521,498 features in all.

We use Algorithm 7 in the first set of experiments. By using different settings of the pairwise samples and the margins, we design the experiments as follows.

- **1-splitting, CD02**: This setting is used to simulate the perceptron algorithm introduced in (Collins and Duffy, 2002).[3] Only one of the best parses for each sentence is used as the good parse and the other best parses are dropped. Other parses are used as bad parses.

- **r-splitting, even margins**: all the best parses are used as good parses. Other parses are used as bad parses.

[3]In (Collins and Duffy, 2002), the tree kernel is used as features on all tree segments. Here we use the feature set used in (Collins, 2000; Shen and Joshi, 2003).

section 23, $\leq$100 words (2416 sentences)			
model	recall%	precision%	f-score%
baseline (Collins, 1999)	88.1	88.3	88.2
1-splitting, CD02	89.2	89.8	89.5
r-splitting, even margins	89.1	89.8	89.5
r-splitting, uneven margins	89.3	90.0	89.6
ordinal regression, even margins	88.1	87.8	88.0
ordinal regression, uneven margins	89.5	90.0	**89.8**

Table 6.1: Experimental Results

- **r-splitting, uneven margins**: all the best parses are used as good parses. The margin function $g(\mathbf{r}_i, \mathbf{r}_j) = \frac{1}{i} - \frac{1}{j}$, which was previously defined in (6.2), is employed.

- **ordinal regression, even margins**: $T = \{(j,l) \mid j < l\}$.

- **ordinal regression, uneven margins**: $T = \{(j,l) \mid j < l\}$. The margin function defined in (6.2) is employed.

By estimating the number of rounds of iterations on the development data, we obtain the results on the test data as shown in Table 6.1. Ordinal regression with uneven margins achieves the best result in f-score. It verifies that using more pairs in training is helpful for the reranking problem. Uneven margins are crucial for employing full pairwise models to reranking.

We compare the performance of Algorithm 1 and Algorithm 2 on our best model, ordinal regression with uneven margins. Figure 2 shows the comparison of the learning curves of these two algorithms. For Algorithm 6, we compute the score of all pairwise samples, $\mathbf{w}^t \cdot \mathbf{x}_{i,j} - \mathbf{w}^t \cdot \mathbf{x}_{i,l}$, on the fly.

Algorithm 6 converges faster than Algorithm 7 in terms of rounds, which is consistent with the theoretical justification. However, Algorithm 7 converges faster in terms of running time. The results of f-scores are similar, But the result of Algorithm 7 is more stable; it does not over-fit the training data even after 3000 rounds of iteration, while Algorithm

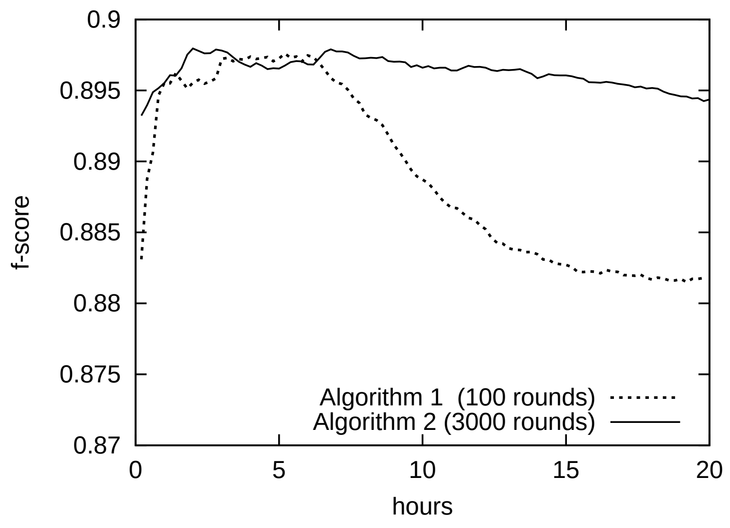


Figure 6.2: Learning curves of Algorithm 1 and Algorithm 2 on PTB Section 23

6 over-trains quickly in terms of rounds. We think the reason is that Algorithm 6 updates the weight vector whenever there is a classification error on a pairwise sample, thus it is more likely to drop into a local optimum. While Algorithm 7 to some extent alleviates the ill-posed problem[4] of the perceptron algorithm.

6.6.2 Discriminative Reranking for Machine Translation

We provide experimental results on the NIST 2003 Chinese-English large data track evaluation. We use the data set used in (Och et al., 2004). The training data consists of about 170M English words, on which the baseline translation system is trained. The training data is also used to build language models which are used to define feature functions on various syntactic levels. The development data consists of 993 Chinese sentences. Each Chinese sentence is associated with 1000-best English translations generated by the baseline MT system. The development data set is used to estimate the parameters for the feature functions for the purpose of reranking. The test data consists of 878 Chinese sentences. Each Chinese sentence is associated with 1000-best English translations too. The test set is used

[4]The result of the algorithm depends on the order in which the training samples are used.

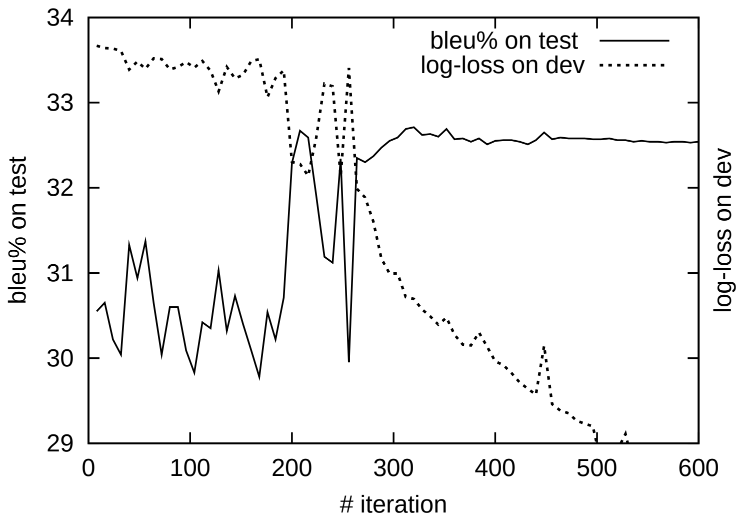


Figure 6.3: Splitting with uneven margins

to assess the quality of the reranking output.

In (Och et al., 2004), aggressive search was used to combine features (Och, 2003). After combining about a dozen features, the BLEU score (Papineni et al., 2001) did not improve any further, and the score was 32.9%. In our experiments, we will use the top 20 individual features developed by Och et al. (2004). Algorithm 2 is used with the three different settings as follows.

- **best vs rest**: $T = \{(1, j) \mid j > 1\}$ and even margin.
- **splitting, uneven margins**: $T = \{(j,l) \mid j <= 300 \text{ and } l >= 700\}$. The margin function defined in (6.2) is employed.
- **ordinal regression, uneven margins**: $T = \{(j,l) \mid j * 2 < l \text{ and } j + 20 < l\}$. The margin function defined in (6.2) is employed.

The best vs. rest setting is used to simulate the previous perceptron reranking algorithm (Collins and Duffy, 2002). It converged in 7 iterations, and achieved BLEU score of 30.9%. The learning curve of the Splitting and the ORUM is shown in figure 6.3 and

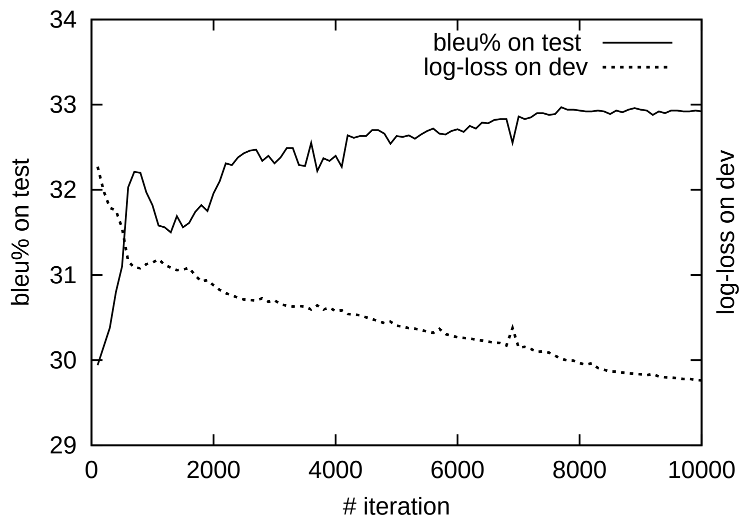


Figure 6.4: Ordinal regression with uneven margins

6.4. The Splitting algorithm achieved BLEU score of 32.6%, and the ORUM algorithm achieved BLEU score of **32**.**9**%, which is significantly better than the best vs. rest model.

According to the learning curves, we notice that whenever the log-loss on the development set decreases, the BLEU score on the test set increases. It shows the generalization capability of these two algorithms.

6.7 Summary

To sum up, we have proposed a general framework for ranking and reranking. In this framework, we have proposed two variants of perceptron, which are trained on pairwise samples. Using these two algorithms, we can employ more pairwise samples, which are useful in training a ranker or reranker. We also keep the data complexity unchanged and make the training efficient with these algorithms.

Using these two algorithms, we investigated the margin selection problem for the reranking tasks. By using uneven margins on ordinal regression, we achieves an f-score of

89.8% on sentences with $\leq$ 100 words in Section 23 of Penn Treebank. In machine translation reranking, our perceptron-like algorithm matches the state-of-the-art discriminative MT reranking system reported in (Och, 2003).

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Statistical LTAG parsing is a well known hard problem due to its increased computational complexity as compared to CFG parsing. In this work, we investigated two aspects of the problem, the structure and the algorithm.

As to the data structure, we introduce LTAG-spinal, a variant of LTAG with very desirable linguistic, computational and statistical properties. As to the algorithm, we not only explore various parsing strategies, but also investigate the reranking approach. We achieved state-of-the-art results in both approaches.

To sum up, we have accomplished the following achievements in this research.

- We introduced a new formalism, LTAG-spinal, which is weakly equivalent to LTAG and has desirable linguistic, computational and statistical properties.

- We extracted an LTAG-spinal Treebank extracted from the PTB with Propbank annotation. This treebank provides a desirable resource for parsing and semantic role labeling.

- We implemented a left-to-right incremental parser for LTAG-spinal. The incremental parser achieved an f-score of 89.3% on LTAG dependency on section 23 of the

LTAG-spinal Treebank.

- We implemented a bidirectional LTAG-spinal dependency parser. The bidirectional parser achieved an f-score of 90.5% on LTAG dependency on the same data set as the left-to-right parser.

- We proposed a novel graph-based incremental construction algorithm, which could be applied to many structure prediction problems in NLP, e.g. semantic role labeling.

- We also proposed a novel discriminative reranking algorithm, Ordinal Regression with Uneven Margins, which could be applied many reranking problems in NLP, including LTAG parse reranking.

7.2 Future Work

- Semantic Role Labeling

  The domain of locality property of LTAG makes derivation tree a very attractive representation for Semantic Role Labeling (SRL). In addition, we can handle most of the discontinuous arguments which are troublesome for the CFG based representation. As described in Chapter 5, the LTAG-spinal treebank shows desirable compatibility with Propbank annotations It seems to us that high quality LTAG parsing result will be very useful in the SRL task on Propbank.

- Semantic Parsing

  LTAG derivation trees reveal deep syntactic relations. It will be desirable to compute formal semantics of a sentence over an LTAG derivation. In previous works (Zettlemoyer and Collins, 2005), machine learning methods have been successfully used to map sentences to logic form. We will explore the learning methods of mapping LTAG derivation trees to logic form.

- Improvement on graph-based incremental construction

  We will investigate the methods of making graph-based incremental construction more efficient. In addition to the greedy search approach, we are also interested in the methods of precise search with dynamic programming. We will build a generic tool for this algorithm, which could be used in other NLP learning algorithms, for example, like semantic role labeling.

- Incorporating POS Tagging and Chunking

  The bidirectional parsing algorithm can be viewed as a natural extension of the chunking-attachment approach for parsing. We are interested in the methods of incorporating POS tagging and chunking into this discriminative learning framework. We hope to build an end-to-end system centering on LTAG derivation trees.

Appendix A

LTAG-spinal

Proof of Theorem 2.1

We show Theorem 2.1 by the following two lemmas.

Lemma A.1 *For any LTAG grammar G, there exists a weakly equivalent LTAG grammar G', such that, any node in an elementary tree is either on the spines or a child of a spinal node. G' is called a* **slim** *grammar.*

Proof We show it by construction. Suppose elementary tree T is not *slim*. Let node B be an internal node and B is a child of a spine node, as shown in Figure A.1. We decompose T into a lexicalized tree T_x and a non-lexicalized tree T_y, by replacing the subtree rooted on B with the node with a unique label X. Therefore, T_y must attach to node X. We remove T from G, put T_x into G, and put T_y into U, a set used collect non-lexicalized elementary trees.

We repeat this operation until all the elementary trees in G are *slim*. Then we lexicalize the trees in U with the *slim* elementary trees in G. For example, as to T_y in Figure A.1, we could lexicalize this tree by attaching all the B_1-rooted elementary trees in G to the B_1 node. Then we put all the lexicalized trees into G, and set U to empty.

We repeat the whole procedure if there is non-slim trees in G. This loop is guaranteed to stop, since the maximal number of nodes which are neither spinal nodes or children of

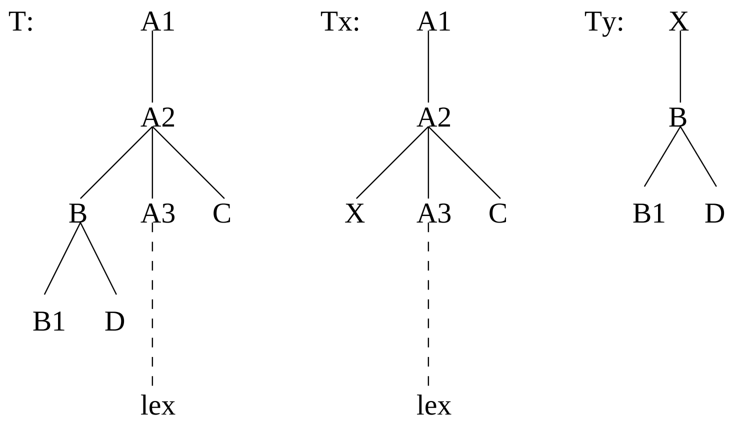


Figure A.1: Replacing elementary trees in G with slim trees

a spinal node in an elementary tree always decreases.

The resulting G is the *slim* grammar G' that we are searching for.

It is easy to verify that, at each step, $G \cup U$ generates the same string language. So we have shown G and G' are weakly equivalent. ▮

Lemma A.2 *For any slim LTAG grammar G, there exists a weakly equivalent LTAG-spinal grammar G_s.*

Proof We show it by recursively replacing a non-spinal elementary tree with a set of of elementary trees which generate the same language. We divide it into two cases, initial trees and auxiliary trees.

- Suppose we have an initial tree T_a which is not spinal. We can decompose T_a into a lexicalized tree T_x and a non-lexicalized tree T_y so as to remove nodes which are not on the spine one by one, for example, node B_1 as shown in Figure A.2.

  The idea is that we insert a node with a unique label X_1 into the spine, where X_1 requires obligatory adjunction. In this way, T_y has to adjoins to T_x whenever T_x is used. The combined tree is designed to replace T_a in G.

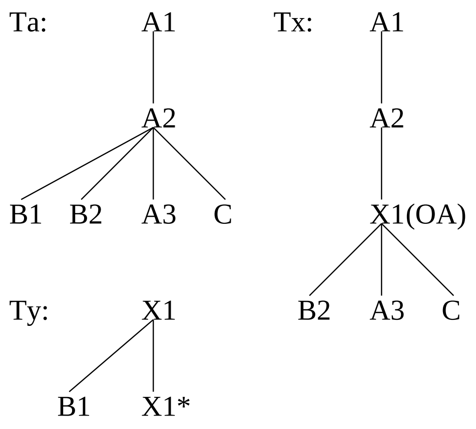


Figure A.2: Spinalization for initial trees

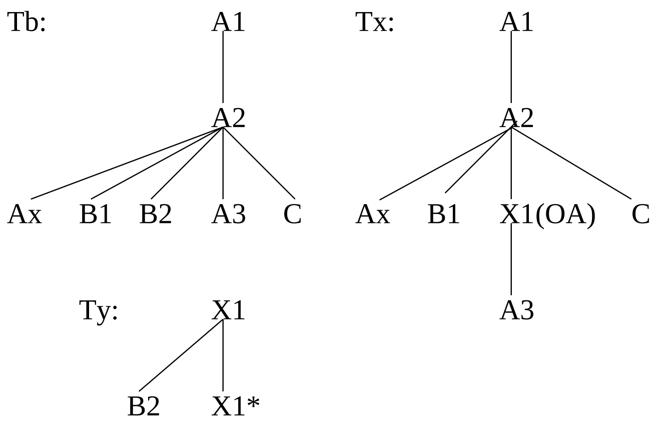


Figure A.3: Spinalization for auxiliary trees

We remove T_a from G, add T_x in G, and put T_y into U, a set used collect non-lexicalized elementary trees.

- As to an auxiliary tree T_b which is not spinal. We remove nodes which are neither on the lexical spine nor on the recursive spine one by one, in a way similar to the case of initial trees. What is a little bit different is the case that we need to remove a node which is a child of the lowest node of the shared spine, between the anchor and the foot of the auxiliary tree, as shown in Figure A.3.

  Here, A_3 dominates the anchor and A_x dominates the foot node. B_2 is a child of A_2, the lowest node in the shared spine, and B_2 is between A_3 and A_x.

  We remove T_b from G, put T_x into G, and put T_y into U.

We recursively replace elementary trees in G until all the trees in G are in the spinal form. For each non-lexicalized tree T_y in U, we generate a set of lexicalized trees for T_y by expanding the single non-terminal in T_y with all compatible initial tree in G, and put them in G.

The resulting grammar G is the spinal grammar G_s that we are searching for.

It is easy to verify that, at each step, $G \cup U$ generates the same string language. So we have shown that $\mathcal{L}$(LTAG) $\subseteq \mathcal{L}$(LTAG-spinal). ▮

By combining Lemma A.1 and Lemma A.2, we have $\mathcal{L}$(LTAG) $\subseteq \mathcal{L}$(LTAG-spinal). On the other hand, it is straightforward to show that $\mathcal{L}$(LTAG) $\supseteq \mathcal{L}$(LTAG-spinal), since attachment can be simulated by adjunction so that LTAG-spinal can be viewed as a sub-set of LTAG. Therefore,

$$\mathcal{L}(\text{LTAG}) = \mathcal{L}(\text{LTAG-spinal}).$$

Theorem 2.1 holds.

Appendix B

Ordinal Regression with Uneven Margins

PROOF OF THEOREM 6.1

Proof: We show it by bounding $||\mathbf{w}^t||^2$ from above and below. Suppose $\{(\mathbf{x}_{i,j}, y_{i,j})\}$ is separable with respect to T with margin γ weighted on a margin function g as in (6.5), and $g(a,b) = -g(b,a)$ if $b < a$. The training data can be noted as $S = \{(\mathbf{u}_r, v_r)\}$ with margin weight $\{\lambda_r\}$ where

$$\begin{aligned}
\mathbf{u}_r &= \mathbf{x}_{i,j} - \mathbf{x}_{i,l}, \\
v_r &= \begin{cases} 1 & \text{if } (y_{i,j}, y_{i,l}) \in T, \\ -1 & \text{if } (y_{i,l}, y_{i,j}) \in T, \end{cases} \\
\lambda_r &= v_r g(y_{i,j}, y_{i,l}),
\end{aligned}$$

for i, j, l s.t. $(y_{i,j}, y_{i,l}) \in T$ or $(y_{i,l}, y_{i,j}) \in T$. Therefore, we have

$$v_r(\mathbf{w}^* \cdot \mathbf{u}_r) > \lambda_r \gamma, \tag{B.1}$$

where $1 \leq r \leq |S|, ||\mathbf{w}^*||^t = 1$. Thus

According to algorithm 6,

$$\mathbf{w}^t = \mathbf{w}^{t-1} + v^t \lambda^t \mathbf{u}^t \text{ , if } v^t \mathbf{w}^{t-1} \cdot \mathbf{u}^t \leq \lambda^t \tau$$

Thus

$$\begin{aligned}||\mathbf{w}^t||^2 &= (\mathbf{w}^{t-1} + v^t\lambda^t\mathbf{u}^t)^2 \\ &= ||\mathbf{w}^{t-1}||^2 + 2\lambda^t v^t \mathbf{w}^{t-1}\cdot\mathbf{u}^t + \lambda^t\lambda^t||\mathbf{u}^t||^2 \\ &\leq ||\mathbf{w}^{t-1}||^2 + 2\lambda^t\lambda^t\tau + \lambda^t\lambda^t(2R)^2 \\ &\leq \sum_{p=1}^{t}\lambda^p\lambda^p(2\tau+4R^2) \qquad \text{(B.2)}\end{aligned}$$

On the other hand,

$$\begin{aligned}\mathbf{w}^*\cdot\mathbf{w}^t &= \mathbf{w}^*\cdot(\mathbf{w}^{t-1}+v^t\lambda^t\mathbf{u}^t) = \mathbf{w}^*\cdot\mathbf{w}^{t-1} + v^t\lambda^t\mathbf{w}^*\cdot\mathbf{u}^t \\ &> \mathbf{w}^*\cdot\mathbf{w}^{t-1} + \lambda^t\lambda^t\gamma \;>\; \sum_{p=1}^{t}\lambda^p\lambda^p\gamma \qquad \text{(B.3)}\end{aligned}$$

Combining (B.2) and (B.3), we have

$$\begin{aligned}(\sum_{p=1}^{t}\lambda^p\lambda^p\gamma)^2 &< ||\mathbf{w}^*\cdot\mathbf{w}^t||^2 = ||\mathbf{w}^t||^2 \leq \sum_{p=1}^{t}\lambda^p\lambda^p(2\tau+4R^2) \\ (\sum_{p=1}^{t}\lambda^p\lambda^p) &\leq \frac{\sum_{p=1}^{t}\lambda^p\lambda^p(2\tau+4R^2)}{\sum_{p=1}^{t}\lambda^p\lambda^p\gamma^2} = \frac{2\tau+4R^2}{\gamma^2} \qquad \text{(B.4)}\end{aligned}$$

Let $\lambda_{min} = \min_r \lambda_r$. With (B.4), we have

$$t \;\leq\; \frac{\sum_{p=1}^{t}\lambda^p\lambda^p}{\lambda_{min}^2} \leq \frac{2\tau+4R^2}{\lambda_{min}^2\gamma^2} \qquad \text{(B.5)}$$

Therefore claim *a* of Theorem 6.1 holds. Now we show claim *b*. With (B.2) and (B.4) we have,

$$\begin{aligned}||\mathbf{w}^t||^2 &\leq \sum_{p=1}^{t}\lambda^p\lambda^p(2\tau+4R^2) \leq \frac{2\tau+4R^2}{\gamma^2}(2\tau+4R^2) \\ &= (\frac{2\tau+4R^2}{\gamma})^2, \text{ so} \\ ||\mathbf{w}^t|| &\leq \frac{2\tau+4R^2}{\gamma} \qquad \text{(B.6)}\end{aligned}$$

Then the resulting margin for $\mathbf{x}_r$ given by $\mathbf{w}^t$ is bounded from below.

$$\gamma_r^t = \frac{v_r\mathbf{w}^t\cdot\mathbf{u}_r}{||\mathbf{w}^t||} \geq \frac{\lambda_r\tau}{||\mathbf{w}^t||} \geq \lambda_r\gamma\frac{\tau}{2\tau+4R^2} \qquad \text{(B.7)}$$

Therefore claim *b* of Theorem 6.1 holds. ∎

PROOF OF THEOREM 6.2

Proof: We show it by bounding $||\mathbf{w}^t||^2$ from above and below. Suppose $\{(\mathbf{x}_{i,j}, y_{i,j})\}$ is *separable* by $\mathbf{w}^*$ with margin γ, so

$$\mathbf{w}^* \cdot \mathbf{x}_{i,j} - \mathbf{w}^* \cdot \mathbf{x}_{i,l} > \gamma, \tag{B.8}$$

where $1 \leq i \leq m, 1 \leq j,l \leq k$. Let us consider each updating $\mathbf{w}^t = \mathbf{w}^{t-1} + \sum_s u_s \mathbf{x}_{i,s}$ for some fixed i. According to the algorithm, we have

$$1 \leq \frac{1}{2}\sum_s |u_s| \leq \frac{k(k-1)/2}{2} < \frac{k^2}{4} \tag{B.9}$$

$$\sum_s u_s \mathbf{x}_{i,s} = \sum_{(j,l)\in S_t} \mathbf{x}_{i,j} - \mathbf{x}_{i,l}, \tag{B.10}$$

where $S_t = \{(j,l) \mid 1 \leq j,l \leq k, \mathbf{w}^t \cdot (\mathbf{x}_{i,j} - \mathbf{x}_{i,l}) \leq \tau\}$, $|S_t| = \frac{1}{2}\sum_s |u_s|$ for each iteration. Here $\frac{1}{2}\sum_s |u_s|$ represents the number of mistakes made by $\mathbf{w}^t$ on sentence i at the t^{th} updating. We define e_t as $e_t \equiv \frac{1}{2}\sum_s |u_s|$.

We first bound $||\mathbf{w}^t||^2$ from below. We have

$$\begin{aligned}
\mathbf{w}^* \cdot \mathbf{w}^t &= \mathbf{w}^* \cdot \mathbf{w}^{t-1} + \sum_s u_s \mathbf{w}^* \cdot \mathbf{x}_s \\
&= \mathbf{w}^* \cdot \mathbf{w}^{t-1} + \sum_{(j,l)\in S_{t-1}} \mathbf{w}^*(\mathbf{x}_{i,j} - \mathbf{x}_{i,l}) \\
&\geq \mathbf{w}^* \cdot \mathbf{w}^{t-1} + \sum_{(j,l)\in S_{t-1}} \gamma \\
&= \mathbf{w}^* \cdot \mathbf{w}^{t-1} + e_{t-1}\gamma \\
&\geq \sum_{p=1}^{t-1} e_p \gamma \\
||\mathbf{w}^t||^2 &= ||\mathbf{w}^*||^2 ||\mathbf{w}^t||^2 \geq (\mathbf{w}^* \cdot \mathbf{w}^t)^2 \geq (\sum_{p=1}^{t-1} e_p)^2 \gamma^2 \qquad \text{(B.11)}
\end{aligned}$$

Then we bound $||\mathbf{w}^t||^2$ from above.

$$\begin{aligned}
||\mathbf{w}^t||^2 &= ||\mathbf{w}^{t-1}||^2 + 2\mathbf{w}^{t-1}\sum_s u_s \mathbf{x}_{i,s} + ||\sum_s u_s \mathbf{x}_{i,s}||^2 \\
&= ||\mathbf{w}^{t-1}||^2 + 2\sum_{(j,l)\in S_{t-1}} \mathbf{w}^{t-1} \cdot (\mathbf{x}_{i,j} - \mathbf{x}_{i,l}) + ||\sum_s u_s \mathbf{x}_{i,s}||^2
\end{aligned}$$

$$
\begin{aligned}
&\leq ||\mathbf{w}^{t-1}||^2 + 2e_{t-1}\tau + ||\sum_s u_s \mathbf{x}_{i,s}||^2 \\
&\leq ||\mathbf{w}^{t-1}||^2 + 2e_{t-1}\tau + 4e_{t-1}^2 R^2 \\
&\leq 2\sum_{p=1}^{t-1} e_p \tau + 4\sum_{p=1}^{t-1} e_p^2 R^2 \qquad \text{(B.12)}
\end{aligned}
$$

Combining (B.11) and (B.12), we have

$$
(\sum_{p=1}^{t} e_p)^2 \gamma^2 \leq 2\sum_{p=1}^{t} e_p \tau + 4\sum_{p=1}^{t} e_p^2 R^2 \qquad \text{(B.13)}
$$

Therefore, the number of mistakes made in the first t updates is

$$
\begin{aligned}
\sum_{p=1}^{t} e_p &\leq \frac{2\sum_{p=1}^{t} e_p \tau}{\sum_{p=1}^{t} e_p \gamma^2} + \frac{4\sum_{p=1}^{t} e_p^2 R^2}{\sum_{p=1}^{t} e_p \gamma^2} \\
&\leq \frac{2\tau}{\gamma^2} + \frac{4(k^2/4)\sum_{p=1}^{t} e_p R^2}{\sum_{p=1}^{t} e_p \gamma^2} \\
&= \frac{k^2 R^2 + 2\tau}{\gamma^2} \qquad \text{(B.14)}
\end{aligned}
$$

Since $\forall p, 1 \leq e_p$, thus $t \leq \sum_{p=1}^{t} e_p$. Therefore Algorithm 7 stops after t updates, where

$$
t \leq \sum_{p=1}^{t} e_p \leq \frac{k^2 R^2 + 2\tau}{\gamma^2}
$$

Therefore claim *a* and *b* of Theorem 6.2 hold. For claim *c*, we use the same technique as in the proof of Theorem 6.1, and we have

$$
\gamma^t \geq \gamma \frac{\tau}{2\tau + k^2 R^2} \qquad \blacksquare
$$

Bibliography

Christopher M. Bishop. 1996. *Neural Networks for Pattern Recognition*. Oxford University Press.

L. Bottou. 1991. *Une approche thérique de l'apprentissage connexionniste: Applications à la reconnaissance de la parole*. Ph.D. thesis, Université de Paris XI.

C. Chelba and F. Jelinek. 2000. Structured language modeling. *Computer Speech and Language*, 14(4):283–332.

J. Chen. 2001. *Towards Efficient Statistical Parsing using Lexicalized Grammatical Information*. Ph.D. thesis, University of Delaware.

D. Chiang. 2000. Statistical Parsing with an Automatically-Extracted Tree Adjoining Grammar. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Michael Collins and Nigel Duffy. 2002. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*.

M. Collins and B. Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL)*.

M. Collins. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania.

M. Collins. 2000. Discriminative reranking for natural language parsing. In *Proceedings of the 17th International Conference on Machine Learning*.

M. Collins. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference of Empirical Methods in Natural Language Processing*.

M. Collins. 2004. Parameter estimation for statistical parsing models: Theory and practice of distribution-free methods. In H. Bunt, J. Carroll, and G. Satta, editors, *New Developments in Parsing Technology*. Kluwer Academic Publishers.

K. Crammer and Y. Singer. 2001a. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of Machine Learning Research*.

K. Crammer and Y. Singer. 2001b. PRanking with Ranking. In *Proceedings of the 15th Annual Conference Neural Information Processing Systems*.

K. Crammer and Y. Singer. 2003. Ultraconservative online algorithms for multiclass problems. *Journal of Machine Learning Research*, 3:951–991.

K. Crammer. 2004. *Online Learning of Complex Categorial Problems*. Ph.D. thesis, Hebrew Univeristy of Jerusalem.

H. Daumé III and D. Marcu. 2005. Learning as search optimization: Approximate large margin methods for structured prediction. In *ICML 2005*.

R. Frank. 2002. *Phrase Structure Composition and Syntactic Dependencies*. The MIT Press.

Y. Freund and R. E. Schapire. 1999. Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296.

E. F. Harrington. 2003. Online Ranking/Collaborative Filtering Using the Perceptron Algorithm. In *Proceedings of the 20th International Conference on Machine Learning*.

J. Henderson. 2000. A neural network parser that handles sparse data. In *IWPT 2000*.

J. Henderson. 2003. Generative versus discriminative models for statistical left-corner parsing. In *IWPT 2003*.

Ralf Herbrich, Thore Graepel, and Klaus Obermayer. 2000. Large margin rank boundaries for ordinal regression. In *Advances in Large Margin Classifiers*, pages 115–132. The MIT Press.

R. Herbrich. 2002. *Learning Kernel Classifiers: Theory and Algorithms*. The MIT Press.

Thorsten Joachims. 2002. Optimizing search engines using clickthrough data. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*.

A. K. Joshi and Y. Schabes. 1997. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 69 – 124. Springer-Verlag.

A. K. Joshi and B. Srinivas. 1994. Disambiguation of super parts of speech (or supertags): Almost parsing. In *Proceedings of COLING '94: The 15th Int. Conf. on Computational Linguistics*.

A. K. Joshi, L. S. Levy, and M. Takahashi. 1975. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1).

A. K. Joshi, T. Becker, and O. Rambow, 2002. *Complexity of scrambling: A new twist to the competence performance distinction*, chapter Tree-Adjoining Grammars. Univ. of Chicago Prress.

A. K. Joshi. 1985. Tree adjoining grammars: How much context sensitivity is required to provide a reasonable structural description. In I. Karttunen, D. Dowty, and A. Zwicky, editors, *Natural Language Parsing*, pages 206–250. Cambridge University Press.

M. Kay, 1980. *Readings in Natural Language Processing*, chapter Algorithmic schemata and data structures in syntactic processing. Morgan Kaufmann.

D. Klein and C. Manning. 2003. A* parsing: Fast exact viterbi parse selection. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*.

W. Krauth and M. Mezard. 1987. Learning algorithms with optimal stability in neural networks. *Journal of Physics A*, 20:745–752.

A. Kroch and A. K. Joshi. 1985. The linguistic relevance of tree adjoining grammar. Report MS-CIS-85-16. CIS Department, University of Pennsylvania.

J. Lafferty, A. McCallum, and F. Pereira. 2001. Conditional random fields: Probabilistic models for stgmentation and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning*.

Yaoyong Li, Hugo Zaragoza, Ralf Herbrich, John Shawe-Taylor, and Jaz Kandola. 2002. The perceptron algorithm with uneven margins. In *Proceedings of the 19th International Conference on Machine Learning*.

D. Magerman. 1995. Statistical decision-tree models for parsing. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1994. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

R. McDonald, K. Crammer, and F. Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43th Annual Meeting of the Association for Computational Linguistics (ACL)*.

A. B. J. Novikoff. 1962. On convergence proofs on perceptrons. In *The Symposium on the Mathematical Theory of Automata*, volume 12.

Franz Josef Och, Daniel Gildea, Sanjeev Khudanpur, Anoop Sarkar, Kenji Yamada, Alex Fraser, Shankar Kumar, Libin Shen, David Smith, Katherine Eng, Viren Jain, Zhen Jin, and Dragomir Radev. 2004. A smorgasbord of features for statistical machine translation. In Susan Dumais, Daniel Marcu, and Salim Roukos, editors, *Proceedings of the 2004 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 161–168, Boston, MA, USA, May.

F. J. Och. 2003. Minimum error rate training for statistical machine translation. In Erhard W. Hinrichs and Dan Roth, editors, *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 160–167, Sapporo, Japan, July.

M. Palmer, D. Gildea, and P. Kingsbury. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31(1).

K. Papineni, S. Roukos, and T. Ward. 2001. Bleu: a method for automatic evaluation of machine translation. IBM Research Report, RC22176.

S. Pradhan, K. Hacioglu, V. Krugler, W. Ward, J. Martin, and D. Jurafsky. 2004. Shallow semantic parsing using support vector machines. In *Proceedings of the 2004 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*.

C. Prolo. 2003. *LR Parsing for Tree Adjoining Grammars and its Application to Corpus-based Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania.

V. Punyakanok and D. Roth. 2001. The use of classifiers in sequential inference. In *Proceedings of the 15th Annual Conference Neural Information Processing Systems*.

V. Punyakanok, D. Roth, W. Yih, and D. Zimak. 2005. Learning and Inference over Constrained Output. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*.

A. Ratnaparkhi. 1997. A linear observed time statistical parser based on maximum entropy models. In *Proceedings of the 2nd Conference of Empirical Methods in Natural Language Processing*.

B. Roark. 2001. Probabilistic top-down parsing and language modeling. *Computational Linguistics*, 27(2):249–276.

F. Rosenblatt. 1958. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408.

A. Sarkar and A. K. Joshi. 1996. Coordination in tree adjoining grammars: Formalization and implementation. In *Proceedings of COLING '96: The 16th Int. Conf. on Computational Linguistics*.

G. Satta and O. Stock. 1994. Bi-Directional Context-Free Grammar Parsing for Natural Language Processing. *Artificial Intelligence,* 69(1-2).

Y. Schabes and R. C. Waters. 1995. A cubic-time, parsable formalism that lexicalizes context-free grammar without changing the trees produced. *Computational Linguistics*, 21(4).

R. E. Schapire, Y. Freund, P. Bartlett, and W. S. Lee. 1997. Boosting the margin: a new explanation for the effectiveness of voting methods. In *Proceedings of the 14th International Conference on Machine Learning*, pages 322–330.

L. Shen and A. K. Joshi. 2003. An SVM based voting algorithm with application to parse reranking. In *Proceedings of the 7th Conference on Computational Natural Language Learning*.

L. Shen and A. K. Joshi. 2004. Flexible margin selection for reranking with full pairwise samples. In K. Su and J. Tsujii, editors, *Proceedings of the 1st International Joint Conference of Natural Language Processing*, pages 467–474, Sanya, Hainan Island, China, March.

L. Shen, A. Sarkar, and A. K. Joshi. 2003. Using ltag based features in parse reranking. In *Proceedings of the 2003 Conference of Empirical Methods in Natural Language Processing*.

L. Shen, A. Sarkar, and F. J. Och. 2004. Discriminative reranking for machine translation. In Susan Dumais, Daniel Marcu, and Salim Roukos, editors, *Proceedings of the 2004 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 177–184, Boston, MA, USA, May.

B. Srinivas. 1997. Performance evaluation of supertagging for partial parsing. In *IWPT 1997*.

P. Sturt and V. Lombardo. 2005. Processing coordinated structures: Incrementality and connectedness. *Cognitive Science, to appear*.

B. Taskar, C. Guestrin, and D. Koller. 2003. Max-margin markov networks. In *Proceedings of the 17th Annual Conference Neural Information Processing Systems*.

B. Taskar, D. Klein, M. Collins, D. Koller, and C. Manning. 2004. Max-margin parsing. In *Proceedings of the 2004 Conference of Empirical Methods in Natural Language Processing*.

Vladimir N. Vapnik. 1998. *Statistical Learning Theory*. John Wiley.

K. Vijay-Shanker. 1987. *A study of Tree Adjoining Grammar*. Ph.D. thesis, University of Pennsylvania.

William Woods. 1976. Parsers in speech understanding systems. Technical Report 3438, Vol. 4, 1–21, Bolt, Beranek and Newman Inc.

F. Xia and T. Bleam. 2000. A corpus-based evaluation of syntactic locality in tags. In *TAG+5*.

F. Xia. 2001. *Automatic Grammar Generation From Two Different Perspectives*. Ph.D. thesis, University of Pennsylvania.

XTAG-Group. 2001. A lexicalized tree adjoining grammar for english. Technical Report 01-03, IRCS, Univ. of Pennsylvania.

P. Xu, C. Chelba, and F. Jelinek. 2002. A study on richer syntactic dependencies for structured language modeling. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*.

H. Yamada and Y. Matsumoto. 2003. Statistical dependency analysis with Support Vector Machines. In *IWPT 2003*.

L. Zettlemoyer and M. Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars.