NAIST-IS-DD1061018

Doctoral Dissertation

Techniques for Improving Transition-based Dependency Parsing Algorithms

Katsuhiko Hayashi

March 15th, 2013

Department of Information Processing
Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Katsuhiko Hayashi

Techniques for Improving Transition-based Dependency Parsing Algorithms[*]

Katsuhiko Hayashi

Abstract

The transition systems for dependency analysis usually employ the shift-reduce parsing algorithm, which offers linear-time complexity, and is much faster than the graph-based parsers. In recent years, the beam search and dynamic programming algorithms have been proposed for improving the transition systems. However, the transition-based algorithms empirically have lower accuracy than the graph-based algorithms. This thesis describes some methods for improving the accuracy of the transition-based dependency parsing algorithms.

To improve the accuracy of the previous transition systems, we propose a novel top-down transition system for dependency analysis. Its deductive system is similar to that of the "Earley" parsing algorithm for the context-free grammars or that of the "Head-corner" parsing algorithm for head context-free grammars. However, unlike the Earley and Head-corner parsers, the proposed parser is data-driven and use no explicit grammar rules. Therefore, our proposed parser performs the "Earley" prediction by using not grammar rules but a statistically learned prediction model. Unlike shift-reduce parsers, the top-down parser can integrate information of an overall input sentence into the parser model through the top-down prediction. We empirically show that the proposed parser achieves the state-of-the-art accuracy, but its time complexity $O(n^2)$ for a sentence length n is worse than that of the shift-reduce parser.

The proposed top-down transition system has another important advantage over the previous transition systems. There are no spurious ambiguity problems in it. By applying the essence of the non-spuriousity to a bottom-up arc-standard transition system, we also propose a novel bottom-up transition-based shift-reduce parsing algorithm.

[*]Doctoral Dissertation, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD1061018, March 15th, 2013.

This algorithm has not only no spurious ambiguity problems but also the linear-time complexity. Therefore, it works faster than the top-down parsing algorithm.

The dynamic programming algorithm can be applicable to the proposed shift-reduce parsing algorithm. The algorithm with dynamic programming can produce not only *k*-best outputs but also packed derivation forests which efficiently keep an exponential number of derivations. The non-spuriosity gurantees that there is an one-to-one correnspondence between a derivation and an output tree, and the *k*-best lists or packed forests do not contain any non-unique trees. Through experiments, we also show that the elimination of spurious ambiguity makes reranking algorithms work better.

Keywords:

Dependency Parsing, Incremental Parsing, Head-corner Parsing, Packed Forest, Statistical Machine Translation

Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisor Prof. Yuji Matsumoto for the continuous support of my Ph.D study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Satoshi Nakamura, Associate Prof. Masashi Shimbo, and Assistant Prof. Mamoru Komachi, for their encouragement, insightful comments, and hard questions.

My sincere thanks also goes to Dr. Taro Watanabe, Dr. Jun Suzuki, Hajime Tsukada, and Dr. Tsutomu Hirao for offering me the internship opportunities in their groups and leading me working on diverse exciting projects.

I thank my fellow labmates in Computational Linguistics Group: Kiso Tetsuo, Shuhei Kondo and Katsumasa Yoshikawa for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last three years.

Last but not the least, I would like to thank my family: my parents Yoshihiko Hayashi and Mayumi Hayashi, for giving birth to me at the first place and supporting me spiritually throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Understanding of natural language syntax and parsing is a very important problem for many useful natural language processing (NLP) applications such as machine translation [83, 85, 32, 61, 62]. Researchers in the computational linguistics have studied syntactic formalisms to represent the structure of natural language sentences, and in recent years lexicalized grammar formalisms have received an increasing amount of attention from both theoretical and practical standpoints.

Dependency grammar is a kind of lexicalized grammar formalism, has largely developed especially in Europe [60], and syntactic dependency representations of sentences have a long history in theoretical linguistics [28, 22, 38, 29, 76, 2, 18].

Recently, the computational parsing community has found renewed interest in dependency parsing due to its efficient computational properties and ability to naturally model non-nested constructions, which is important in freer-word order languages such as Czech, Dutch, and German.

A dependency representation consists of lexical entities (words, here) linked by binary asymmetrical relations, and can be defined as a labeled (or unlabeled) directed graph. To make the graph a rooted tree, some constraints such as the *single-head*, *acyclicity* and *connectedness* are imposed on it. Many practical systems for dependency parsing also assume the *projectivity* constraint [28] so that if word A depends on word B, then all words between A and B are also subordinate to B. Free word order languages do not conform the projectivity, but non-projective dependency parsing with higher-order features is usually computationally more expensive than projective parsing [59]. A projective dependency tree is examplified in Figure 1.1. Dependency

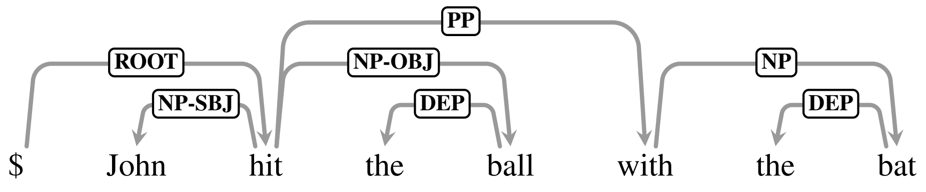

Figure 1.1: An example of a labeled dependency tree

arcs in the labeled dependency tree are labeled by functional categories like NP-OBJ, NP-SBJ and DEP.

There exist both grammar-driven and data-driven methods for dependency parsing. In the grammar-driven approach, Hays [28] and Gaifman [22] formulated a dependency grammar very close to context-free grammar (CFG) [1], and the well-known CKY and Earley parsing algorithms [41, 84, 15] can be used for Hays and Gaifman's dependency grammar parsing [30, 52]. Bilexical grammar [18] subsumes many lexical dependency grammars such as head automaton grammar [2], and an efficient dynamic programming algorithm exists for bilexical grammar parsing [21, 18].

On the other hand, in the NLP community, many recent studies have focused on data-driven dependency parsing [48]. The head-driven CFG parsing systems [9, 10] exploit the dependency relations for the construction of a CFG-style constituent representation, and score parse trees with a generative model. Unlike head-driven CFG parsing, "pure" data-driven parsers use a statistically learned model solely from annotated data and do not use any explicit grammar rules. This means that the "pure" data-driven parsing systems produce the form of a dependency structure directly. Henceforth, when we use the word "data-driven" in this thesis, it means "pure data-driven".

There are two main approaches to data-driven dependency parsing. One is the graph-base approach [20, 19, 57] and the other is the transition-based approach [82, 65, 23]. While graph-based parsers have quadratic or higher time complexity, transition-based parsers typically offer linear-time complexity. Therefore, the transition-based parsers run faster than the graph-based parsers in practice. Though the current state-of-the-art graph-based [46, 73, 86] systems outperform the transition-based systems [37, 88], the practical usefulness of the latter has drawn more and more attention into the NLP researchers.

We give an overview of the graph-based and transition-based dependency parsing approaches, and subsequently describe joint approach of the two methods.

1.2 Data-driven Dependency Parsing

1.2.1 Graph-based Dependency Parsing

Eisner [19] proposed a generative model of dependency parsing and the cubic chart (Eisner-CKY) parsing algorithm for projective dependency analysis. Though not explicitly called graph-based, Eisner-CKY algorithm is viewed as an arc-factored graph-based parser in which all parameters are associated with individual dependency arcs.

This algorithm has established the foundation of both second-order graph-based [58, 5] and third-order graph-based parsers [46]. The third-order algorithm considers substructures containing three dependencies, and it achieves current state-of-the-art accuracy. However, it requires $O(n^4)$ time complexity. Recently, vine-pruning technique has been proposed to improve the efficiency of the third-order graph-based parser [73]. Zhang and McDonald [86] applied cube-pruning [35] to approximate the third-order graph-based parsing inference by (k-best) Eisner-CKY algorithm.

Non-projective dependency parsing algorithm for an arc-factored model was proposed by McDonald et al. [57]. They adapted the Chu-Liu-Edmonds maximum spanning tree (MST) algorithm [7, 16], to find the highest scoring dependency tree. The MST parsing algorithm has the time complexity of $O(n^2)$ but is not capable of taking higher-order features into account.

Riedel and Clarke [72] used (Incremental) Integer Linear Programming to allow the use of global language knowledges as constrains for non-projective dependency parsing. Koo et al. [47] solved non-projective dependency parsing by dual decomposition, in which subgradient method minimizes the dual objective of MST and dynamic programming parsing models.

1.2.2 Transition-based Dependency Parsing

The deterministic transition-based approach was first proposed by Kudo and Matsumoto [49, 50] and Yamada and Matsumoto [82]. These systems process the input sentence of length n from left to right repeatedly as long as new dependencies are added. As the result, the worst case time complexity becomes $O(n^2)$.

Nivre [65, 66, 68] proposed more efficient deterministic algorithms that construct a dependency tree in a single pass over the input sentence. Their so-called *arc-standard* and *arc-eager* shift-reduce algorithms get larger popurality than the other deterministic approaches due to their efficiency.

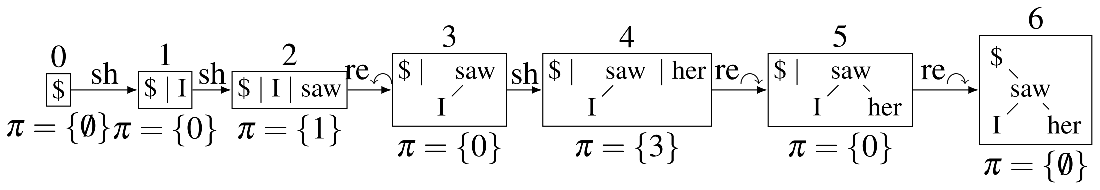
Figure 1.2: A transition sequence of the deterministic arc-standard shift-reduce dependency parser

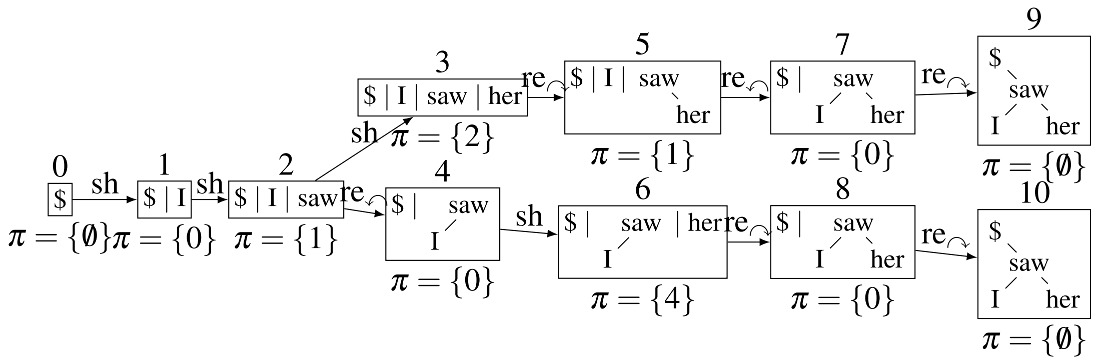
Figure 1.3: A transition lattice of the arc-standard shift-reduce dependency parser with beam search

Figure 1.2 shows a parsing process of the arc-standard shift-reduce parsing system for unlabeled projective dependency analysis. This system has three types of transitions, *shift* (sh), *reduce*$_\curvearrowleft$ (re$_\curvearrowleft$) and *reduce*$_\curvearrowright$ (re$_\curvearrowright$). The parser chooses the highest scoring transition from a configuration to the next configuration, and the final configuration produces a complete dependency tree. However, because the choice is made on local information, the output tree may be a local optimal solution.

In order to alleviate the local optimum problem of the deterministic algorithms, many practical systems [87, 37] use the beam search algorithm that develops *k* configurations at the same position in parallel, as shown in Figure 1.3. A more principled dynamic programming algorithm was proposed by Huang and Sagae [37]. These techniques have improved the transition-based systems significantly with little loss of efficiency.

To handle non-projective dependency structures, some transition-based algorithms have been proposed such as Covington's algorithm [14, 67] and Nivre's sorting algorithm [69]. The former needs $O(n^2)$ time complexity, and the latter needs the worst-case time complexity of $O(n^2)$. Attardi's *m*-degree reduction algorithm [3] works in linear time, but it can only produce restricted non-projective structures.

1.2.3 Joint Graph/Transition-based Parsing

There is also an alternative approach that integrates graph-based and transition-based models [74, 87, 70, 54, 4].

Martins et al. [54] formulated their approach as stacking of parsers such that the output of the first-stage parser is provided to the second as guide features. In particular, they used a transition-based parser for the first stage and a graph-based parser for the second stage.

The joint graph-based and transition-based approach [87, 4] uses an arc-eager shift-reduce parser with a joint graph-based and transition-based model. This approach improves parsing accuracy significantly, but the large beam size of the shift-reduce parser harms its efficiency.

Sagae and Lavie [74] showed that combining the outputs of graph-based and transition-based parsers can improve parsing accuracies. The parser combination reparses an input sentence by the Eisner-CKY or MST algorithm, maximizing votes of the output of the *m* initial parsers.

1.3 Contribution of the Thesis

Parsing accuracy of the current transition-based systems [37, 88] is empirically lower than that of the state-of-the-art graph-based systems [46, 73, 86]. How can we bridge the performance gap between the transition-based and the graph-based systems?

Here, we bring up three problems on the previous transition-based approaches:

- **Lack of top-down (or head-corner) approaches**: all existing transition-based data-driven dependency parsers are bottom-up parsing algorithms. There is no top-down approach in the data-driven dependency parsing community; by top-down we mean that the parser starts with the head word (usually, the main verb) of the sentence and constructs a parse tree in a *head-first* manner. This approach intuitively makes sense because the main verb will tell the parser what kinds of subjects and objects to look for. The observation suggests that top-down transition-based dependency parsing is interesting from not only theoretical but also practical perspectives.

- **Spurious ambiguity problem**: if a dependency parsing system has so-called spurious ambiguity, its many different derivations may produce the same dependency parse tree for an input sentence. For example, two configurations 9 and 10 in Figure 1.3 produce the same dependency tree. This example shows that the arc-standard shift-reduce parsing system has the spurious ambiguity problem. If spurious derivations are not pruned and kept in the k-best beams, it leads to wasted use of the beam as well as duplicated processing.

- **How to integrate graph-based models with transition-based models**: an easy way to improve the transition-based models is to integrate graph-based models with them. Previous work on the joint graph/transition-based approach has a problem with efficiency. Therefore, a more efficient joint algorithm is greatly desired.

This thesis addresses three open problems of the previous transition-based systems. All contribution of this thesis on these probelms is related with improving the data-driven transition-based systems for unlabeled projective dependency parsing.

1.3.1 Lack of Top-down Approaches

Head-corner parsing was proposed by Kay [42], and this constructs a parse tree through a head-first walk of it. This thesis presents a data-driven head-corner parser, following the current trends of dependency parsing.

However, the two parsing formalisms are not compatible; the original head-corner parser needs head CFG grammar rules [78], but data-driven approach does not use any grammar rules.

To overcome this problem, we propose weighted prediction which substitutes grammar rule-based prediction. For a practical use, we formalize it as transition-based parsing, but it takes $O(n^2)$ time complexity.

Experiments on English Penn Treebank data [53] show that while the proposed top-down parser is slower than the shift-reduce dependency parsers, it outperforms them and works fast enough for practical use.

1.3.2 Spurious Ambiguity Problem

The previous transition-based dependency parsing systems have the spurious ambiguity problem, i.e. two different derivations may produce the same dependency parse tree. As mentioned above, the arc-standard transition system for projective dependency analysis has spurious ambiguity.

One solution to eliminate the spurious ambiguity is to give priority to the construction of left arcs over that of right arcs (or vice versa). Our proposed top-down parser does not have any spurious ambiguity problem. This is because the parser always explores all left dependents of a head word before right dependents.

By taking the essence of the elimination of spurious ambiguity into an arc-standard transition system, we also propose a transition-based dependency parsing algorithm without spurious ambiguity, which turns out to be more efficient than the top-down parser. The time complexity of the proposed parser is linear.

Our experiments empirically show that it works as fast as the previous transition systems and also achieves higher accuracy than them.

1.3.3 How to integrate graph-based models with transition-based models

The beam search shift-reduce parser can output *m* parse trees. If combined with dynamic programming, it can efficiently store exponential number of parse trees in a "packed forest" [81]. Packed forests may potentially contain better trees than the 1-best of the baseline parser.

In this thesis, we propose a forest reranking algorithm with higher-order graph-based features, which works on packed forests produced by the dynamic programming shift-reduce parser. Like stacking approach, our proposed algorithm can use guide features from 1-best output from the baseline shift-reduce parser.

Compared with previous joint approaches, the proposed algorithm has the following advantages:

- Unlike the conventional stacking approach, the first-stage shift-reduce parser prunes the search space of the second-stage graph-based parser.
- In contrast to joint transition-based/graph-based approaches [87, 4] which require large beam size and make dynamic programming impractical, our two-stage approach can integrate two models with little loss of efficiency.

Experimental results show that the proposed reranking algorithm achieves about the same parsing accuracy as third-order graph-based algorithm. It also works faster than many previous dependency parsing systems. In addition, elimination of spurious ambiguity from the arc-standard shift-reduce parser improves the efficiency and accuracy of our approach.

1.4 Thesis Outline

In the rest of this thesis, we will present our proposed algorithms more precisely, and will show their usufulness through some experiments. The outline of this thesis is as follows:

- In Chapter 2, we define some basic notions of a projective dependency analysis in order to introduce our proposed algorithms.
- In Chapter 3, we present a top-down dependency parsing algorithm, and show some experimental results.
- In Chapter 4, we propose a novel shift-reduce dependency parsing algorithm without spurious ambiguity. The algorithm is derived by applying a property of the top-down parser presented in Chapter 3 to an arc-standard transition system. Then we empirically show the usefullness through some experiments.
- In Chapter 5, we describe a forest reranking algorithm, and experiments show that it achieves accuracy as high as the state-of-the-art graph-based parsing systems and runs faster.

Finally, we consider the implications we can draw from the application of the proposed algorithms to other NLP tasks such as machine translation.

Chapter 2

Basics

The example tree in Figure 2.1 is a projective dependency graph. We insert an artificial root node \$ at the beginning of each sentence as a unique root of the graph. This is a standard convention to simplify both theoretical definitions and computational implementations. Our proposed parsing algorithms in this thesis output only projective dependency graphs.

In this chapter, we define the projective dependency graph and the transition system for its analysis. These definitions are necessary to define our proposed parsing algorithms. Then, we describe two commonly used transition-based parsers for the projective dependency analysis, called “arc-standard” and “arc-eager” parsing.

In addtion, we introduce two techniques, “beam search” and “dynamic programming”, to improve the transition-based parsers. We also describe a discriminative learning algorithm, called Structured Perceptron, and it is widely used for learning of the discriminative models of the transition-based parsing systems.

2.1 Notational Convention

2.1.1 Definition of Projective Dependency Graph

A dependency graph is defined as follows.

Definition 2.1.1 (Dependency Graph) *Given an input sentence* $W = \mathbf{n}_0 \ldots \mathbf{n}_n$ *where* $\mathbf{n}_0$ *is a special root node \$, a directed graph is denoted by* $G_W = (V_W, A_W)$ *where* $V_W = \{0, 1, \ldots, n\}$ *is a set of (indices of) nodes and* $A_W \subseteq V_W \times V_W$ *is a set of directed arcs. The set of arcs is a set of pairs* (x, y) *where* x *is a head and* y *is a dependent*

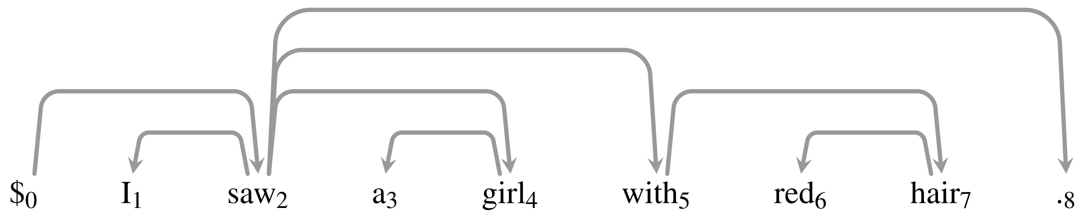

Figure 2.1: An example of an unlabeled projective dependency tree for a sentence "I saw a girl with red hair."

of x. $x \rightarrow^* z$ denotes a path from x to a node z. A directed graph $G_W = (V_W, A_W)$ is **well-formed** *if and only if:*

- **ROOT:** *There is no node x such that $(x, 0) \in A_W$.*
- **SINGLE-HEAD:** *If $(x, y) \in A_W$ then there is no node x' such that $(x', y) \in A_W$ and $x' \neq x$.*
- **ACYCLICTY:** *There is no subset of arcs $\{(x_0, x_1), (x_1, x_2), \ldots, (x_{l-1}, x_l)\} \subseteq A_W$ such that $x_0 = x_l$.*

We call an **well-formed** *directed graph a* **dependency graph**.

Definition 2.1.2 (PROJECTIVITY) *A dependency graph $G_W = (V_W, A_W)$ is* **projective** *if and only if, for every arc $(x, y) \in A_W$ and node z in $x < z < y$ or $y < z < x$, there is a path $x \rightarrow^* z$ or $y \rightarrow^* z$.*

If a projective dependency graph is connected, we call it a *dependency tree*, and if not, a *dependency forest*. The algorithms we will introduce in Section 2.2 are defined for a class of the dependency graphs.

The graph of Figure 2.1 is defined as the following dependency graph G:

$$V = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$$
$$A = \{(0, 2), (2, 1), (2, 4), (4, 3), (2, 5), (5, 7), (7, 6), (2, 8)\}.$$

It is easy to see that the graph G is a *dependency graph*, and all nodes in V are connected. Moreover, all arcs in A satisfy the *projectivity* condition because there are no crossing arcs in G. Therefore, the graph G is a projective dependency tree.

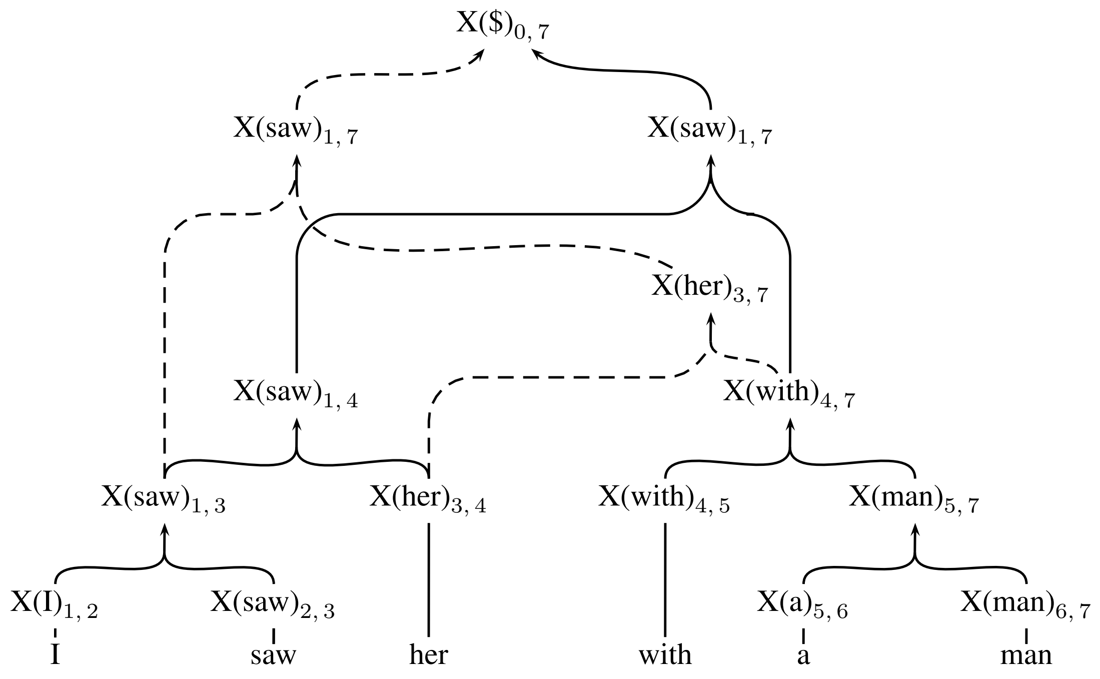

Figure 2.2: An example of packed dependency forest

2.1.2 Definition of Packed Dependency Forests as Hypergraph

Dynamic programming algorithm can efficiently encode an exponential number of parse trees as *packed dependency forest*. We define the packed dependency forest as a hypergraph. We defined a hypergraph as follows:

Definition 2.1.3 (Hypergraph) *A directed hypergraph is a pair $H = \langle V, E\rangle$, where V is the set of vertices and E is the set of hyperedges. Each hyperedge $e \in E$ is a pair $e = \langle T(e), h(e)\rangle$, where $h(e) \in V$ is its head vertex and $T(e) \in V^+$ is an ordered list of tail vertices. $BS(v)$ is the set of incoming hyperedges $\{e \in E | h(e) = v\}$ of a vertex v. When the size of the list is two, we call the hyperedge binary, and when the size is one, call it unary. t is a root node of H.*

Figure 2.2 depicts a sample packed dependency forest defined as the hypergraph. A vertex v is associated with its head word and span information. For example, a vertex X(saw)$_{1,4}$ in Figure 2.2 contains its head word “saw”and span from 1 to 4. The vertex X(saw)$_{1,4}$ has an incoming binary hyperedge with a list of two tail vertices X(saw)$_{1,3}$

and X(her)$_{3,4}$. In the packed dependency forest, each binary hyperedges corresponds to a dependency arc. X(\$)$_{0,7}$ is a root node.

2.1.3 Transition System for Projective Dependency Parsing

A transition system is defined as follows:

Definition 2.1.4 (Transition System) *A transition system for dependency parsing is a quadruple $S = (C, T, I, C_t)$, where C is a set of configurations defined below, and T is a finite set of transitions, each of which is a partial function $t : C \to C$, I is an initialization function, mapping an input sentence to a unique initial configuration, and $C_t \subseteq C$ is a set of terminal configurations.*

A configuration is defined as follows:

Definition 2.1.5 (Configuration) *A configuration is a triple (α, β, A). Symbols α and β are disjoint lists of nodes from $V_{\mathbf{W}}$, which are called stack and buffer respectively.*

We denote the stack with its topmost element to the right, and the buffer with its first element to the left. For example, $\alpha|\mathbf{n}_i$ denotes some stack with its topmost element $\mathbf{n}_i$ and $\mathbf{n}_i|\beta$ does some buffer with the first element $\mathbf{n}_i$. In addtion, we denote empty stack and buffer as $[]$.

We write a stack α of a configuration c as $\alpha(c)$, and a buffer β of c as $\beta(c)$. If t is a transition and c_1, c_2 are configurations such that $t(c_1) = c_2$, we write

$$t : \frac{c_1}{c_2}\ pc \tag{2.1}$$

as a deduction step, where pc denotes preconditions.

A computation is defined as follows:

Definition 2.1.6 (Computation) *A computation of S on $\mathbf{W}$ is a sequence $\gamma = c_0, \ldots, c_m$, $m \geq 0$, of configurations in which each configuration is obtained as the value of the preceding one under some transition.*

We write $G(\gamma)$ for a dependency graph which is produced from a computation γ. We call γ *complete* computation whenever $c_0 = I(\mathbf{W})$ and $c_m \in C_t$.

We use the above definitions to define the transition-based dependency parsing algorithms in the following sections.

2.1.4 Spurious Ambiguity

For a complete computation $\gamma = c_0, \ldots, c_m$, we denote as $DT(\gamma)$ the unique dependency tree (or forest) consisting of nodes $V_{\mathbf{W}}$ and all arcs in the final configuration c_m. A spurious ambiguity of a transition system S is defined as the following:

Definition 2.1.7 (Spurious Ambiguity) *A transition system has spurious ambiguity if, for some pair of complete computations γ and γ' with $\gamma \neq \gamma'$, it has $DT(\gamma) = DT(\gamma')$.*

2.1.5 Correctness of the Transition-based System

We define the *correctness* of the transition system:

Definition 2.1.8 (Correctness) *Let $S = (C, T, I, C_t)$ be a transition system for a dependency parsing. If S is correct for a class of dependency graphs, S satisfies the following two properties (sound and complete):*

- **Soundness:** *S is sound for a class of dependency graphs if and only if, for every sentence $\mathbf{W}$ and every computation $\gamma = c_0, \ldots, c_m$ for $\mathbf{W}$ in S, the parse graph $G(\gamma)$ is contained in a class of dependency graphs.*

- **Completeness:** *S is complete for a class of dependency graphs if and only if, for every sentence $\mathbf{W}$ and every dependency graph $G_{\mathbf{W}}$ for $\mathbf{W}$ in a class of dependency graphs, there is a computation $\gamma = c_0, \ldots, c_m$ for $\mathbf{W}$ in S such that $G(\gamma) = G_{\mathbf{W}}$.*

We use the above definition for proving the correctess of our proposed top-down transition-based parsing system for projective dependency graphs.

2.2 Transition-based Parsing Algorithms

2.2.1 Arc-Standard Shift-Reduce Parsing

A configuration for the arc-standard shift-reduce dependency parsing algorithm is defined as the following:

$$\ell : (\alpha, \beta, A)\ pc$$

where ℓ is a step size. For an input $W = \mathbf{n}_0, \ldots, \mathbf{n}_n$, the initial configuration c_0 (axiom) and the terminal configuration c_{2n} (goal) are defined in Figure 2.3.

$$\begin{array}{rl}
\text{input:} & \mathbf{W} = \mathbf{n}_0 \ldots \mathbf{n}_n \\
\text{axiom}(c_0)\text{:} & 0 : ([\mathbf{n}_0], \mathbf{n}_1 | \ldots | \mathbf{n}_n, ()) \\
\text{shift:} & \dfrac{\ell : (\alpha, \mathbf{n}_i | \beta, A)}{\ell + 1 : (\alpha | \mathbf{n}_i, \beta, A)} \; i < n \\
\text{reduce}_{\curvearrowleft}\text{:} & \dfrac{\ell : (\alpha | \mathbf{n}_j | \mathbf{n}_i, \beta, A)}{\ell + 1 : (\alpha | \mathbf{n}_i, \beta, A \cup (i, j))} \; i \neq 0 \\
\text{reduce}_{\curvearrowright}\text{:} & \dfrac{\ell : (\alpha | \mathbf{n}_j | \mathbf{n}_i, \beta, A)}{\ell + 1 : (\alpha | \mathbf{n}_j, \beta, A \cup (j, i))} \\
\text{goal}(c_{2n})\text{:} & 2n : ([\mathbf{n}_0], [], A)
\end{array}$$

Figure 2.3: The arc-standard transition-based deductive system for projective dependency graphs

As in Figure 2.3, the transition set T for the arc-standard algorithm contains three types of transitions:

- The transition reduce$_{\curvearrowleft}$ adds a dependency arc (j, i) to A, where i is the index of a node on top of the stack α and j is the index of the second node in the stack α. In addition, the second node is removed from α. This has as a precondition that the token i is not the artificial root node index 0.
- The transition reduce$_{\curvearrowright}$ adds a dependency arc (i, j) to A, where i is the index of a node on top of the stack α and j is the index of the second node in the stack α. In addition, the first node is removed from α.
- Transition shift removes the first node $\mathbf{n}_i$ in the buffer β and pushes it on top of the stack α.

The worst-case time complexity of the arc-standard algorithm is $O(n)$, where n is the length of the input sentence. The arc-standard transition-based system is correct for a class of the dependency forests. The proof of the correctness is omitted because it is out of scope here.

The arc-standard transition-based system has spurious ambigutiy. For examle, there exist the two following complete computations for an input $\mathbf{W} = \mathbf{n}_0, \mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3$, which produce the same dependency tree with a set of arcs $((0,2),(2,1),(2,3))$:

- shift, shift, reduce$_{\curvearrowleft}$, shift, reduce$_{\curvearrowright}$, reduce$_{\curvearrowright}$

$$\begin{array}{rl}
\text{input:} & \mathbf{W} = \mathbf{n}_0 \dots \mathbf{n}_n \\
\text{axiom}(c_0)\text{:} & 0 : ([\mathbf{n}_0], \mathbf{n}_1|\dots|\mathbf{n}_n, ()) \\
\text{shift:} & \dfrac{\ell : (\alpha, \mathbf{n}_i|\beta, A)}{\ell + 1 : (\alpha|\mathbf{n}_i, \beta, A)} \; i < n \\
\text{leftarc:} & \dfrac{\ell : (\alpha|\mathbf{n}_i, \mathbf{n}_j|\beta, A)}{\ell + 1 : (\alpha, \mathbf{n}_j|\beta, A \cup (j,i))} \; i \neq 0 \wedge \neg\exists k[(k,i) \in A] \\
\text{rightarc:} & \dfrac{\ell : (\alpha|\mathbf{n}_i, \mathbf{n}_j|\beta, A)}{\ell + 1 : (\alpha|\mathbf{n}_i|\mathbf{n}_j, \beta, A \cup (i,j))} \; \neg\exists k[(k,j) \in A] \\
\text{reduce:} & \dfrac{\ell : (\alpha|\mathbf{n}_i, \beta, A)}{\ell + 1 : (\alpha, \beta, A)} \; \exists k[(k,i) \in A] \\
\text{goal}(c_m)\text{:} & m : (\_, [], A)
\end{array}$$

Figure 2.4: The arc-eager transition-based deductive system for projective dependency graphs: _ means "take anything".

- shift, shift, shift, reduce$_{\curvearrowright}$, reduce$_{\curvearrowleft}$, reduce$_{\curvearrowright}$.

Therefore, the arc-standard transition-based system has spurious ambiguity.

2.2.2 Arc-Eager Shift-Reduce Parsing

A configuration for the arc-eager shift-reduce dependency parsing algorithm is defined as well as that for the arc-standard one:

$$\ell : (\alpha, \beta, A) \; pc$$

where ℓ is a step size. For an input $\mathbf{W} = \mathbf{n}_0, \dots, \mathbf{n}_n$, the initial configuration c_0 (axiom) and the terminal configuration c_m (goal) are defined in Figure 2.4.

The transition set T for the arc-eager transition-based dependency parsing system is defined in Figure 2.4 and contains four types of transitions:

- The transition leftarc adds a dependency arc (j,i) to A, where i is the index of a node on top of the stack α and j is the index of the first node in the buffer β. In addition, this action pops the stack α. They have as a precondition that the token i is not the artificial root node 0 and does not already have a head.

- The transition rightarc adds a dependency arc (i, j) to A, where i is the index of a node on top of the stack α and j is the index of the first node in the buffer β. In addition, this action removes the first node $\mathbf{n}_j$ in the buffer β and push it on top of the stack α. They have as a precondition that the token j does not already have a head.

- The transition reduce pops the stack β and is subject to the precondition that the top token has a head.

- The transition shift removes the first node $\mathbf{n}_i$ in the buffer β and pushes it on top of the stack α.

The worst-case time complexity of the arc-standard algorithm is $O(n)$, where n is the length of the input sentence. The arc-eager transition-based system is correct for a class of dependency forests. However, it is not sound for a class of dependency trees. This means that the arc-eager transition system does not always produce a dependency tree.

The arc-eager transition-based system also has spurious ambiguity. For examle, there exist the two following complete computations for an input $\mathbf{W} = \mathbf{n}_0, \mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3, \mathbf{n}_4, \mathbf{n}_5$, which produce the same dependency tree with a set of arcs $((0,2),(2,1),(2,3),(2,5),(5,4))$:

- shift, leftarc, rightarc, rightarc, shift, leftarc, reduce, rightarc, reduce, reduce

- shift, leftarc, rightarc, rightarc, reduce, shift, leftarc, rightarc, reduce, reduce

Therefore, the arc-eager transition-based system has spurious ambiguity.

2.2.3 Discriminative Model for Transition Systems

2.2.4 Non-Greedy Extension

In order to enhance the greedy (deterministic) transition-based system to non-greedy (nonderministic) one, beam search algorithm is often used. The beam seach transition-based system develops k configurations of the same step in parallel. The complexity of the algorithm is $O(nk)$ for an input whose length is n, which subsumes the greedy algorithm as a special case ($k = 1$).

As the beam search algorithm has to keep much more configurations than the greedy one, for more efficient implementation, we reduce the buffer β and a set of arcs A from

$$
\begin{array}{rl}
\text{input:} & \mathbf{W} = \mathbf{n}_0 \ldots \mathbf{n}_n \\
\text{axiom}(c_0)\text{:} & 0 : (0, 1, [\mathbf{n}_0]) \\
\text{shift:} & \dfrac{\ell : (i, \alpha)}{\ell + 1 : (i+1, \alpha|\mathbf{n}_i)} \; i < n \\
\text{reduce}_{\curvearrowleft}\text{:} & \dfrac{\ell : (i, \alpha|\mathbf{s}_1|\mathbf{s}_0)}{\ell + 1 : (i, \alpha|\mathbf{s}_1^{\curvearrowleft}\mathbf{s}_0)} \; \mathbf{s}_0.\mathrm{h} \neq 0 \\
\text{reduce}_{\curvearrowright}\text{:} & \dfrac{\ell : (i, \alpha|\mathbf{s}_1|\mathbf{s}_0)}{\ell + 1 : (i, \alpha|\mathbf{s}_1^{\curvearrowright}\mathbf{s}_0)} \\
\text{goal}(c_{2n})\text{:} & 2n : (n, [\mathbf{s}_0])
\end{array}
$$

Figure 2.5: The arc-standard transition-based dependency parsing deductive system for beam search: $\mathbf{s}_0$.h denotes a root node index of a tree $\mathbf{s}_0$. $\mathbf{a}^{\curvearrowright}\mathbf{b}$ denotes that a tree $\mathbf{b}$ is attached to a tree $\mathbf{a}$.

a configuration. Instead of them, we add an index to the definition of the configuration to indicate a current position of the topmost buffer element, and use not a node but a tree as the stack element. For example, in the case of the arc-standard transition system, we rewrite the configuration as the following:

$$\ell : (i, \alpha) \; pc$$

where i is an index of the topmost element of an input buffer. The arc-standard transition-based system is also rewritten as shown in Figure 2.5. The stack element $\mathbf{s}$ is a tree itself, and we write a root index of a tree $\mathbf{s}$ as $\mathbf{s}$.h. In order to produce an output tree the parser gathers arcs of the tree by backtracking from the terminal configuration to the initial configuration.

In fact, the parsing accuracies obtained by the beam search are considerably better than those obtained by the greedy algorithm. Therefore, in this thesis, our implemented transition-based systems employ the beam search algorithm as default.

2.2.5 Dynamic Programming

To improve the efficiency of the beam search (arc-standard) transition system, more principled dynamic programming solutions have been proposed [37]. The key observation for dynamic programming is to merge equivalent configurations in the same

$$\begin{array}{rl}
\text{input:} & \mathbf{W} = \mathbf{n}_0 \ldots \mathbf{n}_n \\
\text{axiom}(c_0)\text{:} & 0 : (0, 1, [\mathbf{n}_0]) : \emptyset \\
\text{shift:} & \dfrac{\overbrace{\ell : (\_, j, \alpha) : \_}^{p}}{\ell + 1 : (j, j+1, \alpha|\mathbf{n}_j) : (p)} \; i < n \\
\text{reduce}_{\curvearrowleft}\text{:} & \dfrac{\overbrace{\_ : (k, i, \alpha'|\mathbf{s}'_0) : \pi'}^{p} \quad \overbrace{\ell : (i+1, j, \alpha|\mathbf{s}_0) : \pi}^{q}}{\ell + 1 : (k, j, \alpha'|\mathbf{s}'_0 \curvearrowleft \mathbf{s}_0) : \pi'} \; \mathbf{s}_0.\text{h} \neq 0 \land p \in \pi \\
\text{reduce}_{\curvearrowright}\text{:} & \dfrac{\overbrace{\_ : (k, i, \alpha'|\mathbf{s}'_0) : \pi'}^{p} \quad \overbrace{\ell : (i+1, j, \alpha|\mathbf{s}_0) : \pi}^{q}}{\ell + 1 : (k, j, \alpha'|\mathbf{s}'_0 \curvearrowright \mathbf{s}_0) : \pi'} \; p \in \pi \\
\text{goal}(c_{2n})\text{:} & 2n : (0, n, [\mathbf{s}_0]) : (c_0)
\end{array}$$

Figure 2.6: The arc-standard transition-based dependency parsing deductive system for beam search with dynamic programming: _ means "don't care".

beam buffer. In the work [37], the configurations in the same beam are merged if they have the same feature values of a discriminative parsing model. Another important advantage of dynamic programming is to efficiently encode exponential number of parse outputs into *packed dependency forests* defined in Section 2.1.2.

A configuration for the beam search arc-standard transition system with dynamic programming is defined as follows:

$$\ell : (i, j, \alpha) : \pi$$

where $[i, j]$ is the span of the topmost element in the stack α, and π is a set of pointers to the *predictor configurations*, each of which is a configuration just before pushing the root node of $\mathbf{s}_0$ ($\mathbf{s}_0$) into the stack α.

The transition-based system is defined in Figure 2.6. In a shift step, if configuration p generates configuration q (called "predictor configuration"), then p is added onto π. When two equivalent shifted configurations get merged, their predictor configurations get combined. By using the predictor configuration q, a reduce step is defined with the following deduction step which is extended from the deduction step in Equation 2.1.

$$\frac{c_q \quad c_p}{c_r} \; pc \tag{2.2}$$

where c_q is a predictor configuration of p, and p is the preceding configuration of the resulting configuration c_r. In a reduction step, the configuration c_q tries to combine with every predictor configuration p in π, and the resulting configuration inherits the predictor configurations set π' from p.

The key to the above dynamic programming is behind the concept of *push computation*. We define the push computation as the following:

Definition 2.2.1 (Push Computation) *A computation $\gamma = c_0, \ldots, c_m$, which satisfies the following two properties, is* ***p****ush computation.*

- *The initial stack $\alpha(c_0)$ is not modified during the computation, and is not even exposed after the first transition: For every $1 \leq i \leq m$, there exists a non-empty stack α_i such that $\alpha(c_i) = \alpha(c_0)|\alpha_i$.*
- *The overall effect of the computation is to push a single node to the stack: The stack $\alpha(c_m)$ can be written as $\alpha(c_m) = \alpha(c_0)|h$, for some $h \in \mathbf{W}$.*

We can build larger push computations by means of two binary operations $f_{\mathrm{l}a}$ and $f_{\mathrm{r}a}$, defined as follows. Let $\gamma_1 = c_{10}, \ldots, c_{1m_1}$ and $\gamma_2 = c_{20}, \ldots, c_{2m_2}$ be push computations on the same input $\mathbf{W}$ such that $c_{1m_1} = c_{20}$. Then we can build large push computation by applying the function $f_{\mathrm{r}a}$ to the two smaller push computations:

$$f_{\mathrm{r}a}(\gamma_1, \gamma_2) = c_{10}, \ldots, c_{1m_1}, c_{21}, \ldots, c_{2m_2}, c$$

where c is obtained by applying the reduce$_{\curvearrowright}$ to c_{2m_2}. The operation $f_{\mathrm{l}a}$ is defined analogously.

In the work [51], Kuhlmann et al. mapped a push computation $\gamma = c_0, \ldots, c_m$ to the items of a tabulation-based deductive system [77], whose item form is $[i, h, j]$, as follows:

$$\begin{aligned}
\beta(c_0) &= \mathbf{n}_i|\beta \\
\beta(c_m) &= \mathbf{n}_j|\beta \\
\alpha(c_m) &= \alpha(c_0)|h,
\end{aligned}$$

and they proposed a dynamic programming tabulation (chart) parsing algorithm for the arc-standard transition-based dependency parsing.

Huang and Sagae's arc-standard shift-reduce parser uses not a chart table but a graph-structured stack [81] for dynamic programming. In fact, as well as the above

Algorithm 1 Structured Perceptron Algorithm

```
1: Input: data $D = (x^t, y^t)_{t=1}^N$
2: Output: weight vector $\mathbf{w}$
3: Let $\Delta\mathbf{f}(x,y,z) = \mathbf{f}(x,y) - \mathbf{f}(x,z)$
4: repeat
5:     for $(x,y) \in D$ do
6:         $\hat{z} = \text{argmax}_{z \in \mathscr{Z}} \mathbf{w} \cdot \mathbf{f}(x,z)$
7:         if $\hat{z} \neq y$ then
8:             $\mathbf{w} = \mathbf{w} + \Delta\mathbf{f}(x,y,\hat{z})$
9:         end if
10:    end for
11: until convergence
```

mappings, we can map a push computation to the item of Huang and Sagae's system as follows:

$$\begin{aligned}\beta(c_0) &= \mathbf{n}_i|\beta \\ \beta(c_m) &= \mathbf{n}_j|\beta \\ \alpha(c_m) &= \alpha(c_0)|\mathbf{s}_0.\text{h},\end{aligned}$$

where $\mathbf{s}_0$ is the topmost element of stack α. Each configuration has pointers to the previous push computations in π, and in a reduction step, the larger push computation is constructed by combining the current configuration (push computation) with one configuration (push computation) of them in π.

2.3 Discriminative Learning Algorithm

2.3.1 Structured Perceptron

The transition-based dependency parsing system usually decides the next transition from a configuration by using a discriminative structured prediction model conditioned on the parse history of the configuration. Structured perceptron [12] has been widely used for learning the discriminative model of the transition-based parsing system.

The algorithm 1 shows the structured perceptron algorithm. $D = (x,y)_{t=1}^N$ is training data which contains N training instances. Each training instance is a pair of an input x

Algorithm 2 Averaged Structured Perceptron Algorithm

```
1: Input: data $D = (x^t, y^t)_{t=1}^N$
2: Output: weight vector $\mathbf{w}$
3: Let $\Delta\mathbf{f}(x,y,z) = \mathbf{f}(x,y) - \mathbf{f}(x,z)$
4: $c \leftarrow 1$
5: repeat
6:     for $(x,y) \in D$ do
7:         $\hat{z} = \mathrm{argmax}_{z \in \mathscr{Z}} \mathbf{w} \cdot \mathbf{f}(x,z)$
8:         if $\hat{z} \neq y$ then
9:             $\mathbf{w} = \mathbf{w} + \Delta\mathbf{f}(x,y,\hat{z})$
10:            $\mathbf{w}_a = \mathbf{w}_a + c\Delta\mathbf{f}(x,y,\hat{z})$
11:        end if
12:        $c \leftarrow c + 1$
13:    end for
14: until convergence
15: $\mathbf{w} = \mathbf{w} - \mathbf{w}_a/c$
```

and its correct structure y. Whenever the predicted $\hat{z}$ for x differs from y, the structured perceptron algorithm updates the weights $\mathbf{w}$ as follows:

$$\mathbf{w} = \mathbf{w} + \Delta\mathbf{f}(x,y,\hat{z}) \tag{2.3}$$

where $\mathbf{f}$ is a feature function and $\Delta\mathbf{f}(x,y,\hat{z})$ is $\mathbf{f}(x,y) - \mathbf{f}(x,\hat{z})$.

One solution to reduce weight overfitting is weight averaging. We show the averaged structured perceptron algorithm in Algorithm 2. Weight averaging is accomplished by modifying the standard structured perceptron algorithm so that the final weights returned are the average of all weight vectors encountered during the algorithm.

2.3.2 Early Update for Structured Perceptron

The averaged structured perceptron with early update [13] is a variant on the structured perceptron that deals with the issue that the argmax in line 7 of Algorithm 2 may not be analytically available. The idea is to relace argmax with a beam search algorithm. The early update heuristic updates weights in the place where the correct sequence falls off the beam buffer. Huang et al. [36] prooved the convergence properties of the averaged structured perceptron with early update.

Chapter 3

Top-down Transition-based Dependency Parsing

3.1 Motivation

The transition-based parsing algorithms are widely used for dependency analysis due to their efficiency. However, these parsers have one major problem in that they can handle only local information, and cannot use the information of the whole input sentence for the transition decision on a configuration.

The previous study [39] pointed out that the drawbacks of the previous transition-based parsing systems could be resolved by incorporating top-down information such as root finding. But the root finding approach is not systematic in the sense that the essence of the algorithm is not schematized into the transition system.

As we mentioned in Chapter 1, there is no top-down data-driven parsing algorithm. This thesis presents an $O(n^2)$ top-down transition-based parsing algorithm. The proposed algorithm determines all dependency relations of an input sentence top-down in a head-first manner.

The parsing system is very similar to the Earley parser [15] for CFG or the head-corner parser [42] for head CFG, but has the following difference. The Earley prediction is tied to a particular grammar rule, but the proposed algorithm is data-driven, following the current trends of dependency parsing.

To do the prediction without any grammar rules, we introduce a weighted prediction that is to predict child nodes from parent nodes with a statistically learned model. Through the weighted prediction, the parser can consider the information of the whole sentence into determining the dependency relations.

To improve parsing flexibility in deterministic parsing, the proposed top-down parser uses a beam search algorithm with dynamic programming [37]. The complexity becomes $O(n^2b)$ where b is the beam size.

To reduce prediction errors, we propose a lookahead technique based on the *FIRST* function, inspired by the LL(1) parser [1].

Experimental results show that the proposed top-down parser achieves better results than other data-driven parsing algorithms.

3.2 Related Work

Alshawi [2] proposed a head automaton which recognizes an input sentence top-down. Eisner and Satta [21] showed that there is a cubic-time parsing algorithm in the formalism of the head automaton grammars, which can be equivalently converted into split-head bilexical context-free grammars (SBCFGs) [55, 40]. Although our proposed algorithm does not employ the formalism of SBCFGs, it creates left children before right children, implying that it is free from spurious ambiguity just like parsing algorithms on the SBCFGs. Head-corner parsing algorithm [42] creates dependency trees top-down, and our algorithm has similar spirit to it.

Nivre [64, 65] proposed two transition-based algorithm, known as "arc-standard" and "arc-eager" algorithms. The arc-standard algorithm can be regarded as an application of the classical shift-reduce algorithm to dependency analysis. The arc-eager algorithm processes right-dependent top-down, but this does not involve the prediction of lower nodes from higher nodes. Therefore, the arc-eager algorithm is a totally bottom-up algorithm [65]. Zhang [87] proposed a combination approach of the transition-based algorithm with graph-based algorithm [58], which is the same as our combination model of stack-based and prediction models.

3.3 Deductive System

Our top-down parsing system has four actions: prediction$_{\curvearrowleft}$ (pred$_{\curvearrowleft}$), prediction$_{\curvearrowright}$ (pred$_{\curvearrowright}$), scanner (scan) and completer (comp). It formally uses the following configuration:

$$\ell : (i, h, j, \alpha) : \pi$$

$$
\begin{array}{rl}
\text{input:} & \mathbf{W} = \mathbf{n}_0 \ldots \mathbf{n}_n \\
\text{axiom}(c_0)\text{:} & 0 : (1, 0, n+1, \mathbf{n}_0) : \emptyset \\
\text{pred}_{\curvearrowleft}\text{:} & \dfrac{\overbrace{\ell : (i, h, j, \mathbf{s}_d | \ldots | \mathbf{s}_0) : \_}^{\text{conf } p}}{\ell + 1 : (i, k, h, \mathbf{s}_{d-1} | \ldots | \mathbf{s}_0 | \mathbf{n}_k) : (p)} \; \exists k : i \le k < h \\
\text{pred}_{\curvearrowright}\text{:} & \dfrac{\overbrace{\ell : (i, h, j, \mathbf{s}_d | \ldots | \mathbf{s}_0) : \_}^{\text{conf } p}}{\ell + 1 : (i, k, j, \mathbf{s}_{d-1} | \ldots | \mathbf{s}_0 | \mathbf{n}_k) : (p)} \; \exists k : i \le k < j \wedge h < i \\
\text{scan:} & \dfrac{\ell : (i, h, j, \mathbf{s}_d | \ldots | \mathbf{s}_0) : \pi}{\ell + 1 : (i+1, h, j, \mathbf{s}_d | \ldots | \mathbf{s}_0) : \pi} \; i = h \\
\text{comp:} & \dfrac{\overbrace{\_ : (\_, h', j', \mathbf{s}'_d | \ldots | \mathbf{s}'_0) : \pi'}^{\text{conf } q} \quad \overbrace{\ell : (i, h, j, \mathbf{s}_d | \ldots | \mathbf{s}_0) : \pi}^{\text{conf } p}}{\ell + 1 : (i, h', j', \mathbf{s}'_d | \ldots | \mathbf{s}'_1 | \mathbf{s}'_0 {}^{\curvearrowleft} \mathbf{s}_0) : \pi'} \; q \in \pi, \; h < i \\
\text{goal}(c_{3n})\text{:} & 3n : (n+1, 0, n+1, \mathbf{s}_0) : \emptyset
\end{array}
$$

Figure 3.1: The non-weighted deductive system of top-down dependency parsing algorithm: _ means "don't care".

where ℓ is step size, i is the index of a word on the top of an input buffer, h is the index of a root word of the top tree on stack, j is the index to indicate the right limit ($j-1$ inclusive) of predict$_{\curvearrowright}$, α is a stack of trees $\mathbf{s}_d | \ldots | \mathbf{s}_0$ where $\mathbf{s}_0$ is the top tree and d is the feature window size for dynamic programming [37], and π is a set of pointers to **predictor configurations** which are the configurations just before putting word h onto the stack.

The deductive system of the top-down algorithm is shown in Figure 3.1. The axiom of this system is a configuration with a stack initialized by the root symbol \$. At each step, the system applies one action to each configuration selected from applicable actions. Each of three kinds of actions, pred, scan, and comp, occurs n times on an input sentence of length n, and this system takes $3n$ steps for a complete analysis.

Unlike a conventional shift-reduce parser which directly moves a word from buffer onto stack in a left to right direction, our top-down system splits the process into two steps of prediction and scan. A prediction step puts a word onto stack from the middle of an input string. Action pred$_{\curvearrowleft}$ selects a word k from words ranged in $i \le k < h$ when

step	config	stack	buffer	action	config information
0	c_0	$\$_0$	I_1 saw_2 a_3 girl_4	–	$(1,0,5) : \emptyset$
1	c_1	$\$_0 \vert \mathrm{saw}_2$	I_1 saw_2 a_3 girl_4	$\mathrm{pred}_{\curvearrowright}$	$(1,2,5) : (c_0)$
2	c_2	$\mathrm{saw}_2 \vert \mathrm{I}_1$	I_1 saw_2 a_3 girl_4	$\mathrm{pred}_{\curvearrowleft}$	$(1,1,2) : (c_1)$
3	c_3	$\mathrm{saw}_2 \vert \mathrm{I}_1$	saw_2 a_3 girl_4	scan	$(2,1,2) : (c_1)$
4	c_4	$\$_0 \vert \mathrm{I}_1 \curvearrowleft \mathrm{saw}_2$	saw_2 a_3 girl_4	comp	$(2,2,5) : (c_0)$
5	c_5	$\$_0 \vert \mathrm{I}_1 \curvearrowleft \mathrm{saw}_2$	a_3 girl_4	scan	$(3,2,5) : (c_0)$
6	c_6	$\mathrm{I}_1 \curvearrowleft \mathrm{saw}_2 \vert \mathrm{girl}_4$	a_3 girl_4	$\mathrm{pred}_{\curvearrowright}$	$(3,4,5) : (c_5)$
7	c_7	$\mathrm{girl}_4 \vert \mathrm{a}_3$	a_3 girl_4	$\mathrm{pred}_{\curvearrowleft}$	$(3,3,4) : (c_6)$
8	c_8	$\mathrm{girl}_4 \vert \mathrm{a}_3$	girl_4	scan	$(4,3,4) : (c_6)$
9	c_9	$\mathrm{I}_1 \curvearrowleft \mathrm{saw}_2 \vert \mathrm{a}_3 \curvearrowleft \mathrm{girl}_4$	girl_4	comp	$(4,4,5) : (c_5)$
10	c_{10}	$\mathrm{I}_1 \curvearrowleft \mathrm{saw}_2 \vert \mathrm{a}_3 \curvearrowleft \mathrm{girl}_4$		scan	$(5,4,5) : (c_5)$
11	c_{11}	$\$_0 \vert \mathrm{I}_1 \curvearrowleft \mathrm{saw}_2 \curvearrowright \mathrm{girl}_4$		comp	$(5,2,5) : (c_0)$
12	c_{12}	$\$_0 \curvearrowright \mathrm{saw}_2$		comp	$(5,0,5) : \emptyset$

Figure 3.2: Stages of the top-down deterministic parsing process for a sentence "I saw a girl". We follow the convention and write the stack with its topmost element to the right, and the buffer with its first element to the left. In this example, we set the window size d to 1, and write the descendants of trees on stack elements s_0 and s_1 within depth 1.

$i < h$ which means that left words ranged in $i \leq h$ have not been processed yet. Action $\mathrm{pred}_{\curvearrowright}$ selects a word k from words ranged in $i \leq k < j$ when $h < i < j$ which means that the partial parse of words from 1 to h has been completed. The word on the top of buffer is scanned when it is equal to the root word of c_0 and input words are scanned in a left to right direction.

Action comp creates a directed arc from the root word of c'_0 on a predictor configuration q to that of c_0 on a current configuration c when $h < i$ which means that the word h has been already scanned[1]. In other words, action comp does not create an arc with words which have not been scanned yet. Predicting a word which has been predicted already is prevented by that both preconditions of $\mathrm{pred}_{\curvearrowleft}$ and $\mathrm{pred}_{\curvearrowright}$ do not include the root word h of s_0 and words from 1 to $i-1$ which have been scanned already.

[1]In a single root tree, the special root symbol \$ has exactly one child. Therefore, we do not apply comp action to a configuration when the condition of it satisfies $c_1.\mathrm{h} = \$ \wedge \ell \neq 3n$.

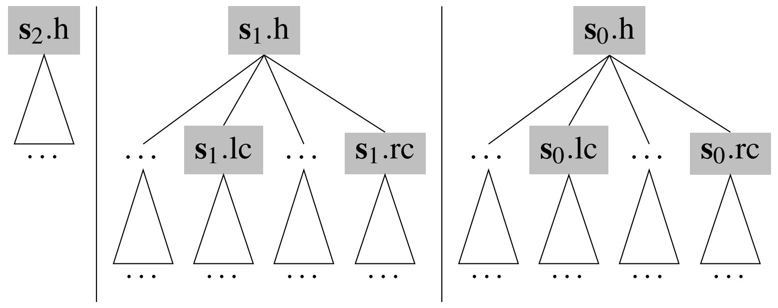

Figure 3.3: Feature window of trees on stack S: The window size d is set to 2. Each $\mathbf{x}$.h, $\mathbf{x}$.lc and $\mathbf{x}$.rc denotes root, left and right child nodes of a stack element $\mathbf{x}$.

3.4 Statistical Parsing Models

3.4.1 Stack-based Model

The proposed algorithm employs a stack-based model for scoring hypothesis. The score of the model is defined as follows:

$$sc_{\mathrm{s}}(i,h,j,\alpha) = \theta_{\mathrm{s}} \cdot \mathbf{f}_{\mathrm{s},act}(i,h,j,\alpha) \tag{3.1}$$

where θ_{s} is a weight vector, $\mathbf{f}_{\mathrm{s}}$ is a feature function, and *act* is one of the applicable actions to a configuration $\ell : (i,h,j,\alpha) : \pi$. We use a set of feature templates shown in Table 3.1. As shown in Figure 3.3, left children $\mathbf{s}_0.\mathbf{l}$ and $\mathbf{s}_1.\mathbf{l}$ of trees on stack for extracting features are different from those in the work [37] because in our parser the left children are generated from left to right.

As mentioned in Section 3.1, we apply beam search and dynamic programming techniques to our top-down parser. Algorithm 1 shows the our beam search algorithm in which top most b configurations are preserved in a buffer $buf[\ell]$ in each step. In line 10 of Algorithm 3, equivalent configurations in the step ℓ are merged following the idea of dynamic programming. Two configurations (i,h,j,α) and (i',h',j',α') in the step ℓ are equivalent, notated $(i,h,j,\alpha) \sim (i',h',j',\alpha')$, iff

$$\mathbf{f}_{\mathrm{s},act}(i,h,j,\alpha) = \mathbf{f}_{\mathrm{s},act}(i',h',j',\alpha'). \tag{3.2}$$

When two equivalent predicted configurations are merged, their predictor configurations in π get combined.

	Features Templates
(1)	$\mathbf{s}_0$.h.w $\mathbf{s}_0$.h.t $\mathbf{s}_0$.h.w $\circ$ $\mathbf{s}_0$.h.t
	$\mathbf{s}_1$.h.w $\mathbf{s}_1$.h.t $\mathbf{s}_1$.h.w $\circ$ $\mathbf{s}_1$.h.t
	$\mathbf{q}_0$.h.w $\mathbf{q}_0$.h.t $\mathbf{q}_0$.h.w $\circ$ $\mathbf{q}_0$.h.t
(2)	$\mathbf{s}_0$.h.w $\circ$ $\mathbf{s}_1$.h.w $\mathbf{s}_0$.h.t $\circ$ $\mathbf{s}_1$.h.t
	$\mathbf{s}_0$.h.t $\circ$ $\mathbf{q}_0$.h.t $\mathbf{s}_0$.h.w $\circ$ $\mathbf{s}_0$.h.t $\circ$ $\mathbf{s}_1$.h.t
	$\mathbf{s}_0$.h.t $\circ$ $\mathbf{s}_1$.h.w $\circ$ $\mathbf{s}_1$.h.t $\mathbf{s}_0$.h.w $\circ$ $\mathbf{s}_1$.h.w $\circ$ $\mathbf{s}_1$.h.t
	$\mathbf{s}_0$.h.w $\circ$ $\mathbf{s}_0$.h.t $\circ$ $\mathbf{s}_1$.h.w $\mathbf{s}_0$.h.w $\circ$ $\mathbf{s}_0$.h.t $\circ$ $\mathbf{s}_1$.h.w $\circ$ $\mathbf{s}_1$.h.t
(3)	$\mathbf{s}_0$.h.t $\circ$ $\mathbf{q}_0$.h.t $\circ$ $\mathbf{q}_1$.h.t $\mathbf{s}_1$.h.t $\circ$ $\mathbf{s}_0$.h.t $\circ$ $\mathbf{q}_0$.h.t
	$\mathbf{s}_0$.h.w $\circ$ $\mathbf{q}_0$.h.t $\circ$ $\mathbf{q}_1$.h.t $\mathbf{s}_1$.h.t $\circ$ $\mathbf{s}_0$.h.w $\circ$ $\mathbf{q}_0$.h.t
(4)	$\mathbf{s}_1$.h.t $\circ$ $\mathbf{s}_1$.lc.t $\circ$ $\mathbf{s}_0$.h.t $\mathbf{s}_1$.h.t $\circ$ $\mathbf{s}_1$.rc.t $\circ$ $\mathbf{s}_0$.h.t
	$\mathbf{s}_1$.h.t $\circ$ $\mathbf{s}_1$.lc.t $\circ$ $\mathbf{s}_0$.h.t $\mathbf{s}_1$.h.t $\circ$ $\mathbf{s}_1$.rc.t $\circ$ $\mathbf{s}_0$.h.t
	$\mathbf{s}_1$.h.t $\circ$ $\mathbf{s}_1$.lc.t $\circ$ $\mathbf{s}_0$.h.t $\mathbf{s}_1$.h.t $\circ$ $\mathbf{s}_1$.rc.t $\circ$ $\mathbf{s}_0$.h.t
(5)	$\mathbf{s}_0$.h.t $\circ$ $\mathbf{s}_1$.h.t $\circ$ $\mathbf{s}_2$.h.t

Table 3.1: Features template for the stack-based model of the top-down transition-based parsing system: w and t denote a word and a part-of-speech tag.

3.4.2 Weighted Prediction

The step 0 in Figure 3.2 shows an example of prediction for a head node "$\$_0$", where the node "saw$_2$" is selected as its child node. To select a probable child node, we define a statistical model for the prediction. In this thesis, we integrate the score from a graph-based model [58] which directly models dependency links. The score of the first-order model is defined as the relation between a child node **c** and a head node **h**:

$$sc_{\mathrm{p}}(\mathbf{h},\mathbf{c}) = \theta_{\mathrm{p}} \cdot \mathbf{f}_{\mathrm{p}}(\mathbf{h},\mathbf{c}) \tag{3.3}$$

where θ_{p} is a weight vector and $\mathbf{f}_{\mathrm{p}}$ is a features function. Using the score sc_{p}, the top-down parser selects a probable child node in each prediction step.

When we apply beam search to the top-down parser, then we no longer use $\exists$ but $\forall$ on $\text{pred}_{\curvearrowleft}$ and $\text{pred}_{\curvearrowright}$ in Figure 3.1. Therefore, the parser may predict many nodes as an appropriate child from a single configuration, causing many predicted configurations. This may cause the beam buffer to be filled only with the configurations, and these may exclude other configurations, such as scanned or completed configurations. Thus, we limit the number of predicted configurations from a single configuration by **prediction size** implicitly in line 10 of Algorithm 3.

Algorithm 3 Top-down Parsing with Beam Search

```
1:  input $W = \mathbf{n}_0, \ldots, \mathbf{n}_n$
2:  $start \leftarrow \langle 1, 0, n+1, \mathbf{n}_0 \rangle$
3:  $buf[0] \leftarrow \{start\}$
4:  for $\ell \leftarrow 1 \ldots 3n$ do
5:      $hypo \leftarrow \{\}$
6:      for each $conf$ in $buf[\ell - 1]$ do
7:          for $act \leftarrow$applicableAct$(conf)$ do
8:              $newconfs \leftarrow$actor$(act, conf)$
9:              addAll $newconfs$ to $hypo$
10:         end for
11:     end for
12:     add top $b$ configurations to $buf[\ell]$ from $hypo$
13: end for
14: return  best candidate from $buf[3n]$
```

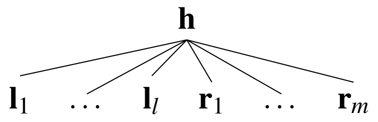

Figure 3.4: An example of tree structure: Each **h**, **l**_ and **r**_ denotes head, left and right child nodes.

To improve the prediction accuracy, we introduce a more sophisticated model. The score of the sibling second-order model is defined as the relationship between **c**, **h** and a sibling node **sib**:

$$sc_{\mathrm{p}}(\mathbf{h}, \mathbf{sib}, \mathbf{c}) = \theta_{\mathrm{p}} \cdot \mathbf{f}_{\mathrm{p}}(\mathbf{h}, \mathbf{sib}, \mathbf{c}). \tag{3.4}$$

The first- and sibling second-order models are the same as the definitions of the work [58], except the scoring factors of the sibling second-order model. The scoring factors for a tree structure in Figure 3.4 are defined as follows:

$$sc_{\mathrm{p}}(\mathbf{h}, -, \mathbf{l}_1) + \sum_{y=1}^{l-1} sc_{\mathrm{p}}(\mathbf{h}, \mathbf{l}_y, \mathbf{l}_{y+1}) + sc_{\mathrm{p}}(\mathbf{h}, -, \mathbf{r}_1) + \sum_{y=1}^{m-1} sc_{\mathrm{p}}(\mathbf{h}, \mathbf{r}_y, \mathbf{r}_{y+1}).$$

This is different from the work [58] in that the scoring factors for left children are calculated from left to right, while those in [58]'s definition are calculated from right to left. This is because our top-down parser generates left children from left to right.

Note that the score of weighted prediction model in this section is incrementally calculated by using only the information on the current configuration, thus the condition of configuration merging in Equation 3.2 remains unchanged.

3.4.3 Weighted Deductive System

We extend deductive system to a weighted one, and introduce **forward score** and **inside score** [79, 37]. The forward score is the total score of a sequence from an initial configuration to the end configuration. The inside score is the score of a top tree $\mathbf{s}_0$ in stack α. We define these scores using a combination of stack-based model and weighted prediction model. The forward and inside scores of the combination model are as follows:

$$\begin{cases} sc^{\mathrm{fw}} = sc_{\mathrm{s}}^{\mathrm{fw}} + sc_{\mathrm{p}}^{\mathrm{fw}} \\ sc^{\mathrm{in}} = sc_{\mathrm{s}}^{\mathrm{in}} + sc_{\mathrm{p}}^{\mathrm{in}} \end{cases} \tag{3.5}$$

where $sc_{\mathrm{s}}^{\mathrm{fw}}$ and $sc_{\mathrm{s}}^{\mathrm{in}}$ are a forward score and an inside score for stack-based model, and $sc_{\mathrm{p}}^{\mathrm{fw}}$ and $sc_{\mathrm{p}}^{\mathrm{in}}$ are a forward score and an inside score for weighted prediction model. We add the following tuple of scores to a configuration:

$$(sc_{\mathrm{s}}^{\mathrm{fw}}, sc_{\mathrm{s}}^{\mathrm{in}}, sc_{\mathrm{p}}^{\mathrm{fw}}, sc_{\mathrm{p}}^{\mathrm{in}}).$$

For each action, we define how to efficiently calculate the forward and inside scores[2]. In either case of $\mathrm{pred}_{\curvearrowleft}$ or $\mathrm{pred}_{\curvearrowright}$,

$$\frac{(sc_{\mathrm{s}}^{\mathrm{fw}}, \_, sc_{\mathrm{p}}^{\mathrm{fw}}, \_)}{(sc_{\mathrm{s}}^{\mathrm{fw}} + \lambda, 0, sc_{\mathrm{p}}^{\mathrm{fw}} + sc_{\mathrm{p}}(\mathbf{s}_0.\mathbf{h}, \mathbf{n}_k), 0)}$$

where

$$\lambda = \begin{cases} \theta_{\mathrm{s}} \cdot \mathbf{f}_{\mathrm{s,pred}_{\curvearrowleft}}(i, h, j, \alpha) & \text{if } \mathrm{pred}_{\curvearrowleft} \\ \theta_{\mathrm{s}} \cdot \mathbf{f}_{\mathrm{s,pred}_{\curvearrowright}}(i, h, j, \alpha) & \text{if } \mathrm{pred}_{\curvearrowright} \end{cases} \tag{3.6}$$

In the case of scan,

$$\frac{(sc_{\mathrm{s}}^{\mathrm{fw}}, sc_{\mathrm{s}}^{\mathrm{in}}, sc_{\mathrm{p}}^{\mathrm{fw}}, sc_{\mathrm{p}}^{\mathrm{in}})}{(sc_{\mathrm{s}}^{\mathrm{fw}} + \xi, sc_{\mathrm{s}}^{\mathrm{in}} + \xi, sc_{\mathrm{p}}^{\mathrm{fw}}, sc_{\mathrm{p}}^{\mathrm{in}})}$$

[2]For brevity, we present the formula not by second-order model as equation 3.4 but a first-order one for weighted prediction.

where

$$\xi = \theta_{\mathrm{s}} \cdot \mathbf{f}_{\mathrm{s,scan}}(i, h, j, \alpha). \tag{3.7}$$

In the case of comp,

$$\frac{(sc'^{\mathrm{fw}}_{\mathrm{s}}, sc'^{\mathrm{in}}_{\mathrm{s}}, sc'^{\mathrm{fw}}_{\mathrm{p}}, sc'^{\mathrm{in}}_{\mathrm{p}}) \qquad (sc^{\mathrm{fw}}_{\mathrm{s}}, sc^{\mathrm{in}}_{\mathrm{s}}, sc^{\mathrm{fw}}_{\mathrm{p}}, sc^{\mathrm{in}}_{\mathrm{p}})}{\begin{aligned}(sc'^{\mathrm{fw}}_{\mathrm{s}} + sc^{\mathrm{in}}_{\mathrm{s}} + \mu, sc'^{\mathrm{in}}_{\mathrm{s}} + sc^{\mathrm{in}}_{\mathrm{s}} + \mu,&\\ sc'^{\mathrm{fw}}_{\mathrm{p}} + sc^{\mathrm{in}}_{\mathrm{p}} + sc_{\mathrm{p}}(\mathbf{s}'_0.\mathbf{h}, \mathbf{s}_0.\mathbf{h}),&\\ sc'^{\mathrm{in}}_{\mathrm{p}} + sc^{\mathrm{in}}_{\mathrm{p}} + sc_{\mathrm{p}}(\mathbf{s}'_0.\mathbf{h}, \mathbf{s}_0.\mathbf{h}))&\end{aligned}}$$

where

$$\mu = \theta_{\mathrm{s}} \cdot \mathbf{f}_{\mathrm{s,comp}}(i, h, j, \alpha) + \theta_{\mathrm{s}} \cdot \mathbf{f}_{\mathrm{s,pred\_}}(\_, h', j', \alpha'). \tag{3.8}$$

Pred_ takes either pred$_{\curvearrowleft}$ or pred$_{\curvearrowright}$. Beam search is performed based on the following linear order for the two configurations p and p' at the same step, which have $(sc^{\mathrm{fw}}, sc^{\mathrm{in}})$ and $(sc'^{\mathrm{fw}}, sc'^{\mathrm{in}})$ respectively:

$$p \succ p' \text{ iff } sc^{\mathrm{fw}} > sc'^{\mathrm{fw}} \text{ or } sc^{\mathrm{fw}} = sc'^{\mathrm{fw}} \wedge sc^{\mathrm{in}} > sc'^{\mathrm{in}}. \tag{3.9}$$

We prioritize the forward score over the inside score since forward score pertains to longer action sequence and is better suited to evaluate hypothesis configurations than inside score [63].

3.5 "First" Function for a Lookahead

Top-down backtrack parser usually reduces backtracking by precomputing the set FIRST$(\cdot)$ [1]. We define the set FIRST$(\cdot)$ for our top-down dependency parser:

$$\mathrm{FIRST(t')} = \{\mathbf{ld}.\mathrm{t} | \mathbf{ld} \in \mathrm{lmdescendant(Tree, t')Tree} \in \mathrm{Corpus}\}$$

where t' is a POS-tag, Tree is a correct dependency tree which exists in Corpus, a function lmdescendant(Tree, t') returns the set of the leftmost descendant node **ld** of each nodes in Tree whose POS-tag is t', and **ld**.t denotes a POS-tag of **ld**. Though our parser does not backtrack, it looks ahead when selecting possible child nodes at the prediction step by using the function FIRST. In case of pred$_{\curvearrowleft}$:

$$\frac{\forall k : i \leq k < h \wedge \mathbf{n}_i.\mathrm{t} \in \mathrm{FIRST}(\mathbf{n}_k.\mathrm{t}) \\ \overbrace{\ell : (i, h, j, \mathbf{s}_d | \ldots | \mathbf{s}_0) : \_}^{p}}{\ell + 1 : (i, k, h, \mathbf{s}_{d-1} | \ldots | \mathbf{s}_0 | \mathbf{n}_k \rangle : (p)}$$

where $\mathbf{n}_i$.t is a POS-tag of the node $\mathbf{n}_i$ on the top of the buffer, and $\mathbf{n}_k$.t is a POS-tag in kth position of an input nodes. The case for $\text{pred}_{\curvearrowright}$ is the same. If there are no nodes which satisfy the condition, our top-down parser creates new configurations for all nodes, and pushes them into *hypo* in line 9 of Algorithm 3.

3.6 Experiments

3.6.1 Experimental Setups

Experiments were performed on the English Penn Treebank data and the Chinese CoNLL-06 data. For the English data, we split WSJ part of it into sections 02-21 for training, section 22 for development and section 23 for testing. We used the standard head rules [82] to convert phrase structure to dependency structure. For the Chinese data, we used the information of words and fine-grained POS-tags for features. To compare our algorithm with second-order Eisner-Satta algorithm, we used MSTParser[3].

We used an early update version of averaged perceptron algorithm [13] for training of shift-reduce and top-down parsers. A set of feature templates in [37] were used for the stack-based model, and a set of feature templates in [58] were used for the second-order prediction model. The weighted prediction and stack-based models of top-down parser were jointly trained.

3.6.2 Results on English Data

During training, we fixed the prediction size and beam size to 5 and 16, respectively, judged by preliminary experiments on development data. After 25 iterations of perceptron training, we achieved 92.94 unlabeled accuracy for top-down parser with the FIRST function and 93.01 unlabeled accuracy for shift-reduce parser on development data by setting the beam size to 8 for both parsers and the prediction size to 5 in top-down parser. These trained models were used for the following testing.

We compared our top-down parsing algorithm with other data-driven parsing algorithms in Table 3.2. The top-down parser achieved comparable unlabeled accuracy with others, and outperformed them on the sentence complete rate. On the other hand,

[3]http://www.seas.upenn.edu/ strctlrn/MSTParser/MSTParser.html

	time	accuracy	complete	root
McDonald06 (2nd)	0.15	91.5	42.1	–
Koo10 [45]	–	93.04	–	–
Hayashi11 [26]	0.3	92.89	–	–
2nd-MST*	0.13	92.3	43.7	96.0
Goldberg10 [23]	–	89.7	37.5	91.5
Kitagawa10 [43]	–	91.3	41.7	–
Zhang08 (Sh beam 64)	–	91.4	41.8	–
Zhang08 (Sh+Graph beam 64)	–	92.1	45.4	–
Huang10 (beam+DP)	0.04	92.1	–	–
Huang10* (beam 8+DP)	0.03	92.3	43.5	96.0
Huang10* (beam 16+DP)	0.06	92.27	43.7	96.0
Huang10* (beam 32+DP)	0.10	92.26	43.8	96.1
Zhang11 (beam 64) [88]	–	93.07	49.59	–
top-down* (beam 8+pred 5+DP)	0.07	91.7	45.0	94.5
top-down* (beam 16+pred 5+DP)	0.12	92.3	45.7	95.7
top-down* (beam 32+pred 5+DP)	0.22	92.5	**45.9**	96.2
top-down* (beam 8+pred 5+DP+FIRST)	0.07	91.9	45.0	95.1
top-down* (beam 16+pred 5+DP+FIRST)	0.19	92.4	45.3	96.2
top-down* (beam 32+pred 5+DP+FIRST)	0.33	92.6	45.5	96.6

Table 3.2: Results for test data: Time measures the parsing time per sentence in seconds. Accuracy is an unlabeled attachment score, complete is a sentence complete rate, and root is a correct root rate. $*$ indicates our experiments.

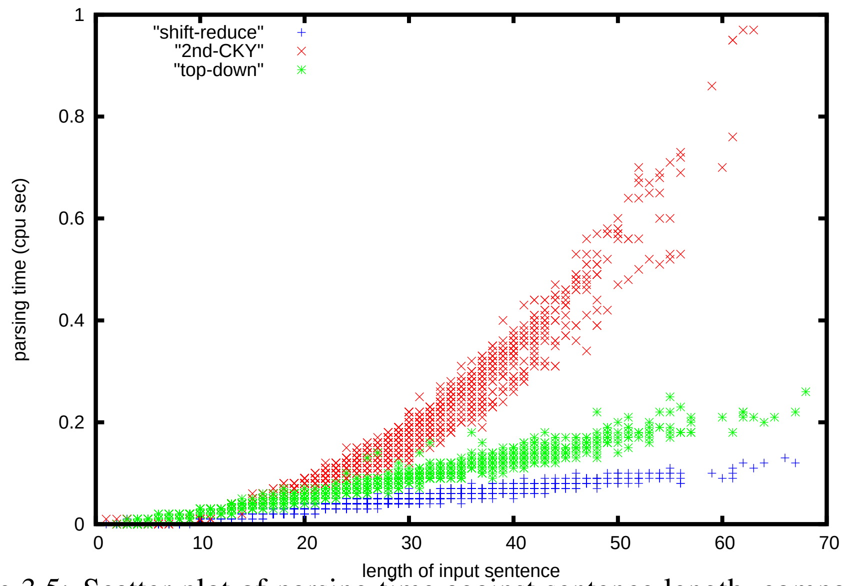

Figure 3.5: Scatter plot of parsing time against sentence length, comparing with top-down, 2nd-MST and shift-reduce parsers (beam size: 8, pred size: 5)

	accuracy	complete	root
oracle (sh+mst)	94.3	52.3	97.7
oracle (top+sh)	94.2	51.7	97.6
oracle (top+mst)	93.8	50.7	97.1
oracle (top+sh+mst)	94.9	55.3	98.1

Table 3.3: Oracle score, choosing the highest accuracy parse for each sentence on test data from results of top-down (beam 8, pred 5) and shift-reduce (beam 8) and MST(2nd) parsers in Table 3.2.

the top-down parser was less accurate than shift-reduce parser (Huang10*) on the correct root measure. In step 0, top-down parser predicts a child node, a root node of a complete tree, using little syntactic information, which may lead to errors in the root node selection. Therefore, we think that it is important to seek more suitable features for the prediction in future work.

Figure 3.5 presents the parsing time against sentence length. Our proposed top-down parser is theoretically slower than shift-reduce parser and Figure 3.5 empirically indicates the trends. The dominant factor comes from the score calculation, and we will leave it for future work. Table 3.3 shows the oracle score for test data, which is the score of the highest accuracy parse selected for each sentence from results of several

	accuracy	complete	root
top-down (beam:8, pred:5)	90.9	80.4	93.0
shift-reduce (beam:8)	90.8	77.6	93.5
2nd-MST	91.4	79.3	94.2
oracle (sh+mst)	94.0	85.1	95.9
oracle (top+sh)	93.8	84.0	95.6
oracle (top+mst)	93.6	84.2	95.3
oracle (top+sh+mst)	94.7	86.5	96.3

Table 3.4: Results for Chinese Data (CoNLL-06)

Table 3.5: Sentences which include relative cluases (482 sentences)

	unlabeled accuracy	complete	root
top-down	**92.1**	**33.0**	**95.6**
shift-reduce	91.8	29.6	94.6
2nd-CKY	91.9	29.9	94.1

parsers. This indicates that the parses produced by each parser are different from each other. However, the gains obtained by the combination of top-down and second-MST parsers are smaller than other combinations. This is because top-down parser uses the same features as second-MST parser, and these are more effective than those of stack-based model. It is worth noting that as shown in Figure 3.5, our $O(n^2b)$ ($b = 8$) top-down parser is much faster than $O(n^3)$ Eisner-Satta CKY parsing.

3.6.3 Results on Chinese Data

We also experimented on the Chinese data. Following English experiments, shift-reduce parser was trained by setting beam size to 16, and top-down parser was trained with the beam size and the prediction size to 16 and 5, respectively. Table 3.4 shows the results on the Chinese test data when setting beam size to 8 for both parsers and prediction size to 5 in top-down parser. The trends of the results are almost the same as those of the English results.

Table 3.6: Parsing Accuracies for longer sentences than 30.

	unlabeled accuracy	complete	root
top-down	91.0	18.0	94.0
shift-reduce	**91.5**	15.5	**95.5**
2nd-CKY	91.3	**18.3**	94.9

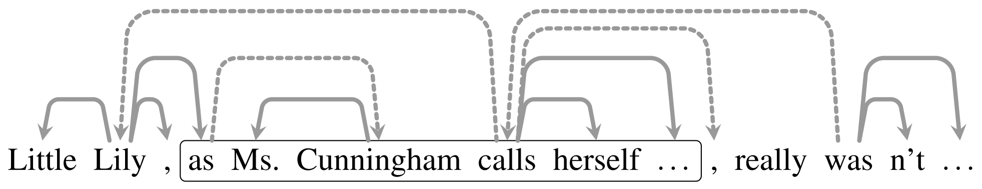
Little Lily , as Ms. Cunningham calls herself ... , really was n't ...

Figure 3.6: A parsing result of shift-reduce parser for a clause sentence

3.6.4 Error Analysis

We analyzed the results on English data. To compare the trends of the top-down parsing results with those of other parsing results under an equal condition, we set the hyperparameters of the top-down parser to achieve almost the same results with other parsers (92.3 unlabeled accuracy). This results in the beam width of 12 and the prediction size of 5. Table 3.5 shows the parsing accuracies for sentences containing relative clauses. For these sentences, the results of the top-down parser are better than those of other parsers. The difference between the accuracies of the top-down and 2nd-CKY parsers is statistically significant by Mcnemar's test (0.01). For shift-reduce and CKY parsers, it is difficult to parse the sentences which include relative clauses and more than one verbs because they use only local information of the input sentence. On the other hand, the top-down parser can incorporate global information of the overall input sentence, and parses those sentences better than the others. In Figure 3.6, we show a result of the shift-reduce parser for a clause sentence, and in Figure 3.7, show that of the top-down parser for the sentence. This sentence has a clause (squred framing words), and these results clearly indicate that the top-down parser parses it well while the shift-reduce parser fails to parse it (dotted arcs indicate the mistakes).

On the other hand, we investigate sentences which the top-down parser fails to parse correcly. Table 3.6 shows the parsing accuracies for longer sentences than 30 words. It is obvious from this result that the top-down parser is worse at parsing long sentences than other parsers. This is because the longer the input sentence is, the more the

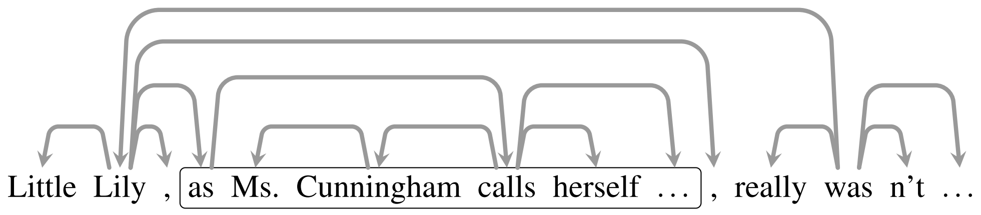
Little Lily , as Ms. Cunningham calls herself . . . , really was n't . . .

Figure 3.7: A parsing result of top-down parser for a clause sentence

prediction of the top-down parser is, and it is more likely to fail to predict dependents of a head correctly. Especially, the root correct rate of the top-down parser is much worse than that of other parsers. In the initial step, the top-down parser needs to select (predict) the root word from all words in the input sentence, and the root prediction for long sentences makes the results of the top-down parser be worse than those of the other parsers.

3.7 Summary

In this chapter, we have presented a novel head-driven parsing algorithm and empirically shown that it is as practical as other dependency parsing algorithms. Our head-driven parser has potential for handling non-projective structures better than other non-projective dependency algorithms [58, 3, 70, 46]. We are in the process of extending our head-driven parser for non-projective structures as our future work.

Chapter 4

Transition-based Dependency Parsing System without Spurious Ambiguity

4.1 Motivation

The proposed top-down parser in Chapter 3 has no spurious ambigutities because all left arcs of a head are always created before its right arcs are created. Additionaly, this algorithm always returns not a dependency forest but a dependency tree, while the arc-eager transition-based dependency parsing algorithm is not sound for a class of dependency trees. These are strong advantages of the proposed top-down dependency parsing algorithm over the previous transition-based dependency parsing algorithms. However, it needs higher time complexity than them.

In this chapter, we propose a transition-based dependency parsing algorithm without spurious ambiguity. This algorithm is more efficient than the top-down dependency parsing algorithm of Chapter 3.

To accomplish this, we apply the essence of the top-down parsing algorithm to the arc-standard transition-based parsing algorithm.

We first introduce a bottom-up transition-based parsing system without spurious ambiguity, and then describe a method to extract packed dependency forests from the parser's outputs. We investigate its usefulness by comparing its output packed dependency forests with those of the usual arc-standard transition-based parsing (which has spurious ambiguities).

4.2 Related Work

Alshawi's split head automaton removes the spurious ambiguity of the classical lexicalized CKY parsing [11] because each constructions of left and right arcs is separated from each other.

On the other hand, in recent years, Cohen et al. [8] have proposed a method to eliminate the spurious ambiguity of shift-reduce transition-based systems, which cover existing systems such as the arc-standard transition-based and Attardi's non-projective transition-based systems [3]. This method introduces some boolean values on each stack elements as constraints to perform reductions as early as they become available in a computation. They proved that the non-spurious arc-standard transition-based system S' is equivalent with the original arc-standard transition-based system S from the following point of view:

- For each complete computation γ' of S', there is a complete computation γ of S such that $DT(\gamma) = DT(\gamma')$.
- For each complete computation γ of S, there is a complete computation γ' of S' such that $DT(\gamma) = DT(\gamma')$.

However, the system S' is not sound for a class of dependency trees, and this means that S' sometimes produces not a dependency tree but a dependency forest. A novel transition-based parsing algorithm proposed in this chapter is not only a non-spurious ambiguity system but also sound for a class of dependency trees.

4.3 Bottom-up Transition-based System using Scanner Action for Eliminating Spurious Ambiguity

One solution to remove spurious ambiguity of dependency parsing is to give priority to the construction of left arcs over that of right arcs (or vice versa) [17]. For example, an Head-corner dependency parser [27] attaches all the left dependents to a word before the right dependents. The parser uses a scan action to stop the construction of left arcs.

We apply this idea to the arc-standard transition system and show the resulting transition system in Figure 4.1. A configuration of the system is defined as follows:

$$\ell : (i, j, \alpha) : \pi \tag{4.1}$$

$$
\begin{array}{rl}
\text{axiom}(c_0): & 0 : (0, 1, \mathbf{w}_0) : \emptyset \\
\text{goal}(c_{3n}): & 3n : (0, n, \mathbf{s}_0) : \emptyset \\
\text{shift}: & \dfrac{\overbrace{\ell : (\_, j, \mathbf{s}_d|\mathbf{s}_{d-1}|\ldots|\mathbf{s}_1|\mathbf{s}_0) : \_}^{\text{conf } p}}{\ell+1 : (j, j+1, \mathbf{s}_{d-1}|\ldots|\mathbf{s}_0|\mathbf{w}_j^*) : (p)} \; j < n \\
\text{scan}: & \dfrac{\ell : (i, j, \mathbf{s}_d|\mathbf{s}_{d-1}|\ldots|\mathbf{s}_1|\mathbf{s}_0^*) : \pi}{\ell+1 : (i, j, \mathbf{s}_d|\mathbf{s}_{d-1}|\ldots|\mathbf{s}_1|\mathbf{s}_0) : \pi} \\
\text{reduce}_{\curvearrowleft}: & \dfrac{\overbrace{\_ : (i, j, \mathbf{s}'_d|\mathbf{s}'_{d-1}|\ldots|\mathbf{s}'_0|\mathbf{s}'_0) : \pi'}^{\text{conf } p} \quad \overbrace{\ell : (j, k, \mathbf{s}_d|\mathbf{s}_{d-1}|\ldots|\mathbf{s}_1|\mathbf{s}_0^*) : \pi}^{\text{conf } q}}{\ell+1 : (i, k, \mathbf{s}'_d|\mathbf{s}'_{d-1}|\ldots|\mathbf{s}'_1|\mathbf{s}'_0 \curvearrowleft \mathbf{s}_0^*) : \pi'} \; \mathbf{s}'_0.\text{h}.\mathbf{w} \neq \mathbf{w}_0 \wedge p \in \pi \\
\text{reduce}_{\curvearrowright}: & \dfrac{\overbrace{\_ : (i, j, \mathbf{s}'_d|\mathbf{s}'_{d-1}|\ldots|\mathbf{s}'_1|\mathbf{s}'_0) : \pi'}^{\text{conf } p} \quad \overbrace{\ell : (j, k, \mathbf{s}_d|\mathbf{s}_{d-1}|\ldots|\mathbf{s}_1|\mathbf{s}_0) : \pi}^{\text{conf } q}}{\ell+1 : (i, k, \mathbf{s}'_d|\mathbf{s}'_{d-1}|\ldots|\mathbf{s}'_1|\mathbf{s}'_0 \curvearrowright \mathbf{s}_0) : \pi'} \; p \in \pi
\end{array}
$$

Figure 4.1: The dynamic programming arc-standard transition-based deductive system without spurious ambiguity

and this is the same as that of the arc-standard shift-reduce dependency parsing algorithm with dynamic programming, but has the following difference. Our system uses the $*$ symbol to indicate that the root node of the topmost element on stack has not been scanned yet.

The shift and reduce$_{\curvearrowright}$ actions can be used only when the root of the topmost element on the stack has already been scanned, and all left arcs are always attached to the head before the head is scanned. This separate construction of left and right arcs enables the system to eliminate the spurious ambiguity problem. Note that henceforth, we call parsers, *k*-best lists, forests, and such with spurious ambiguity "spurious"and without spurious ambiguity "non-spurious".

4.4 Non-Spurious of the Proposed System

To prove that our proposed system has no spurious ambiguity problems, we need to show that different complete computations of it for an input $\mathbf{W}$ always produce diffrent trees, i.e., if $\gamma_1 \neq \gamma_2$ are its complete computations for input $\mathbf{W}$, then $DT(\gamma_1) \neq DT(\gamma_2)$.

To do so, following Cohen et al. [8], we write γ_1 as $\alpha c_1 \beta_1$ and γ_2 as $\alpha c_2 \beta_2$, with α the common prefix among both two computations, and c_1, c_2 configurations such that $c_1 \neq c_2$. Note that α cannot be empty, since both computations must at least have the axiom in common. We call c_0 the last configuration in α, and t_1, t_2 the transitions that produce c_1, c_2 respectively from c_0.

We consider the following two cases.

- Case 1: t_1 and t_2 are scan and reduce$_{\curvearrowleft}$ actions respectively. If the reduce$_{\curvearrowleft}$ action (t_2) creates an arc (x, y) ($x, y \in \mathbf{W}$ and $y < x$), the arc cannot be created in the other computation γ_1 because after scanning the head word x of the topmost stack element of c_0, it cannot reduce its left words including y. Therefore, $DT(\gamma_1) \neq DT(\gamma_2)$.

- Case 2: t_1 and t_2 are shift and reduce$_{\curvearrowright}$ actions respectively. If each head word of the topmost and second stack elements of c_0 is x, y, after shifting a word $x+1$, x get reduced by its right words ($x <$), or x reduces its right words and then get reduced by one of its left words including y. Therefore, $DT(\gamma_1) \neq DT(\gamma_2)$.

4.5 Extraction of Packed Dependency Forests

The dynamic programming can be applied to the arc-standard transition-based system without spurious ambiguity, and the system is able to produce packed dependency forests. The arc-standard shift-reduce parser without spurious ambiguity takes $3n$ steps to finish parsing, and the additional n scan actions add surplus vertices and (unary) hyperedges to a packed forest as shown in Figure 4.2.

However, it is easy to remove them from the packed forest because a consequent configuration of a scan action has a unique antecedent configuration and all the hyperedges going out from a vertex corresponding to the consequent configuration can be attached to the vertex corresponding to the antecedent configuration. A scan weight of the removed unary hyperedge is added to each weight of the hyperedges attached to the antecedent.

Note that the consequent (scanned) configuration produced by a scan action will be never merged with other configurations in the same step by dynamic programming because of the following reasons:

- If two scanned configurations in the ℓ step were merged, their antecedent configurations in the $\ell - 1$ step must be merged previously.

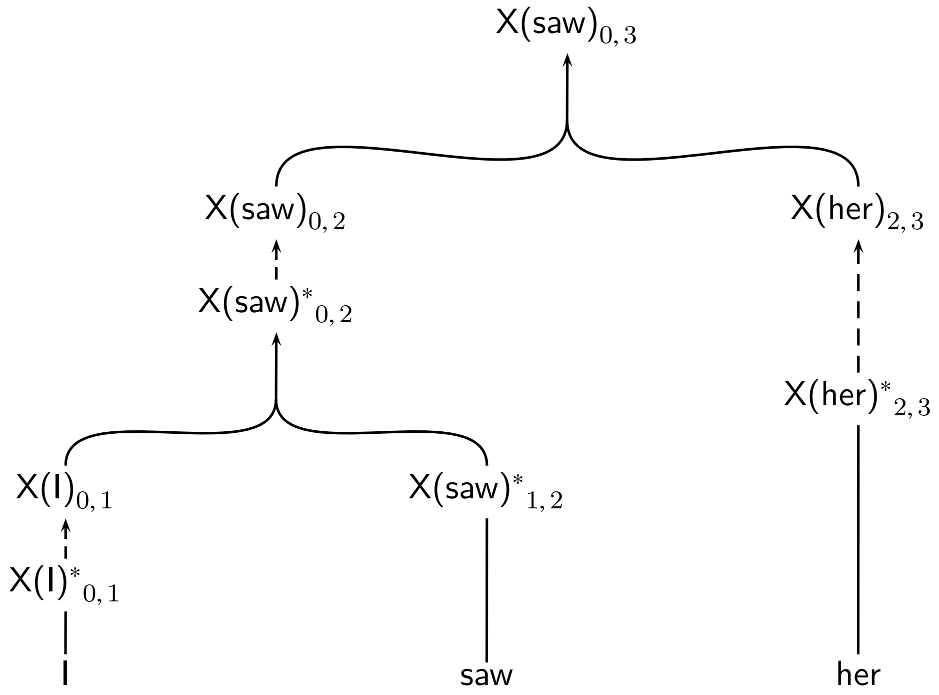

Figure 4.2: An example of a dependency derivation tree produced by the non-spurious transition-based system: the $*$ symbol on some vertices indicates that head words of them have not been scanned yet.

- The root node of the topmost element in stack of a scanned configuration does not contain the $*$ symbol, but those of shifted and left reduced condigurations contain it.
- Each index j of scanned and right reduced configurations is always different from each other because the scanned configuration has not processed right indexed words from the index of the root node of the topmost element in the stack yet, while the right reduced configuration has processed them already.

These guarantee that a consequent configuration of a scan action has a unique antecedent configuration even in case of using dynamic programming.

		8	16	32	64	128	256
sp.	accuracy	**85.0**	85.0	85.0	84.9	84.9	84.8
	milli sec.	**0.016**	**0.03**	**0.059**	**0.121**	**0.273**	**0.711**
non-sp.	accuracy	84.9	85.0	85.0	84.9	84.9	84.8
	milli sec.	0.018	0.034	0.068	0.142	0.334	0.858

Table 4.1: Unlabeled accuracies and parsing times (+forest dumping times, milli sec.) for parsing the 3019 test sentences with spurious shift-reduce and proposed shift-reduce parser (non-sp) using several beam sizes: the parsing accuracies are reported as unlabeled accuracy including punctuations.

4.6 Experiments (baseline vs. proposed)

4.6.1 Experimental Setups

Experiments were performed on the English Penn Treebank (PTB) data with automatic part-of-speech tags. We used Stanford POS tagger[1] with a model used 10-way jackknifing to tag training data. The tagging accuracies on training was 97.1. We randomly selected three disjoint sets of 5000, 2000, and 3019 sentences from the WSJ part of PTB, and used the set of 5000 sentences for training and that of 2000 and 3019sentences for testing. While the 2000 sentence have average 22.2 words, the 3019 sentences have average 46.7 words and are more difficult to parse them than shorter sentences. We used the standard head rules [82] to convert phrase structure to dependency structure. We used an early update version of averaged perceptron algorithm [11] for training of the original dynamic programming arc-standard transition-based system and the proposed dynamic programming arc-standard transition-based system without spurious ambigutiy.

4.6.2 Comparison of arc-standard transition-based parsing with and without spurious ambiguity: Parsing Accuracy and Time

We evaluate parsing accuracies and cpu times (milli sec.) for parsing 3019 and 2000 test data with each baseline and proposed shift-reduce dependency parser with several beam sizes, and the results are shown in Table 4.1 and 4.2. The results for 3019

[1]http://nlp.stanford.edu/software/tagger.shtml

		8	16	32	64	128	256
sp.	accuracy	89.5	89.5	89.6	89.5	89.4	89.4
	milli sec.	**0.006**	**0.012**	**0.024**	**0.048**	**0.1**	**0.22**
non-sp.	accuracy	89.5	89.6	89.6	**89.6**	**89.6**	**89.6**
	milli sec.	0.007	0.014	0.028	0.051	0.119	0.294

Table 4.2: Unlabeled accuracies and parsing times (+forest dumping times, milli sec.) for parsing the 2000 test sentences with spurious shift-reduce and proposed shift-reduce parser (non-sp) using several beam sizes: the parsing accuracies are reported as unlabeled accuracy including punctuations.

		8	16	32	64	128	256
sp.	oracle acc.	**86.9**	**88.1**	**89.1**	**90.1**	90.9	91.6
	% of distinct trees	96.0	94.4	92.7	90.9	88.8	86.7
non-sp.	oracle acc.	86.8	88.0	89.0	90.0	90.9	**91.7**
	% of distinct trees	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

Table 4.3: Unlabeled oracle accuracies and average percentages of distinct trees in *k*-best lists for parsing the 3019 test sentences with spurious shift-reduce and proposed shift-reduce parser (non-sp) using several beam sizes: the oracle accuracies are reported as unlabeled accuracy including punctuations.

sentences (long sentences) show that the proposed parser is slower than the baseline parser. It is likely that this is caused by the additional *n* scan actions. On the other hand, the results for 2000 sentences (short sentences) show that the parsing times of the proposed parser are almost the same as those of the baseline parser even in the case of setting beam size larger. We conjecture from these results that the parsing speed of the proposed parser is not very sensitive to a length of an input sentence unless the length is extremely long. The results also show that the parsing acuuracies of the proposed parser are almost the same as those of the baseline parser.

		8	16	32	64	128	256
	oracle acc.	**89.5**	**91.4**	**93.2**	94.5	95.7	96.6
sp.	# of hyperedges	152.7	300.3	612.7	1277.0	2714.3	5868.7
	% of distinct trees	76.4	73.3	76.3	80.1	82.5	84.2
	oracle acc.	89.0	91.3	93.1	94.5	**95.8**	**96.7**
non-sp.	# of hyperedges	**138.8**	**277.7**	**571.7**	**1211.2**	**2632.3**	**5829.8**
	% of distinct trees	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

Table 4.4: Unlabeled oracle accuracies and average percentages of distinct output trees in 1000-best trees extracted from packed forests produced by baseline shift-reduce and proposed shift-reduce parser (non-sp) using several beam sizes (by parsing 3019 long sentences): the parsing accuracies are reported as unlabeled accuracy including punctuations.

4.6.3 Comparison of arc-standard transition-based parsing with and without spurious ambiguity: Oracle Accuracy on Packed Forests and K-best lists

We compare oracle accuracies in k-best lists produced by baseline and proposed dependency parsers with several beam sizes. The results in Table 4.3 show that while all output trees in k-best lists produced by the proposed parser are unique, k-best lists produced by the baseline parser have some non-unique output trees in them. However, when setting beam size k smaller, the oracle accuracies in k-best lists produced by the proposed parser are slightly less than those in k-best lists produced by the proposed parser. Table 4.4 show the statistics of packed forests produced by each baseline and proposed parser. The oracle accuracies of packed forests without spurious ambiguity are also less than those of packed forests with spurious ambiguity when setting beam size k smaller. We guess that this comes from the fact that the proposed parser needs to set beam size larger than the baseline parser.

4.7 Summary

In this chapter, we have presented a novel arc-standard shift-reduce dependency parser without spurious ambiguity, inspired by the top-down dependency parser proposed in Chapter 3. In the next chapter, we'll use non-spurious packed forests produced by our proposed parser in this chapter to a dependency forest reranking approach, and show that the non-spuriosity is important to improve its accuracy and efficiency.

Chapter 5

Dependency Forest Reranking

5.1 Introduction

There are two main approaches to data-driven dependency parsing. One is a graph-based approach and the other is a transition-based approach.

In the graph-based approach, global optimization algorithms find the highest-scoring tree with locally factored models [57]. While third-order graph-based models achieve state-of-the-art accuracy, it has $O(n^4)$ time complexity for a sentence of length n. Recently, some pruning techniques have been proposed to improve the efficiency of third-order models [73, 86].

The transition-based approach usually employs shift-reduce parsing algorithms with linear-time complexity [68]. However, it greedily chooses the transition with the highest score and the resulting transition sequence is not always globally optimal. The beam search algorithm improves parsing flexibility in deterministic parsing [87, 88], and dynamic programming makes beam search more efficient [37].

There is also an alternative approach that integrates graph-based and transition-based models [74, 87, 54]. Martins et al. [54] formulated their approach as stacking of parsers where the output of the first-stage parser is provided to the second as guide features. In particular, they used a transition-based parser for the first stage and a graph-based parser for the second stage. The main drawback of the stacking approach is that the efficiency of the transition-based parser is sacrificed because the second-stage employs full parsing.

This thesis proposes an efficient stacked parsing method through discriminative reranking with higher-order graph-based features, which works on the forests output by the first-stage dynamic programming shift-reduce parser and integrates non-local features efficiently with cube-pruning [35]. The advantages of our method are as follows:

- Unlike the conventional stacking approach, the first-stage shift-reduce parser prunes the search space of the second-stage graph-based parser.
- In addition to guide features, the second-stage graph-based parser can employ the scores of the first-stage parser which cannot be incorporated in standard graph-based models.
- In contrast to joint transition-based/graph-based approaches [87, 4] which require large beam size and make dynamic programming impractical, our two-stage approach can integrate both models with little loss of efficiency.

In addition, elimination of so-called spurious ambiguity from the arc-standard shift-reduce parser improves the efficiency and accuracy of our approach.

5.2 Related Work

5.2.1 How to Handle Spurious Ambiguity

The graph-based approach employs Eisner and Satta [21]'s algorithm where spurious ambiguities are eliminated by the notion of split head automaton grammars [2].

However, the arc-standard transition-based parser has the spurious ambiguity problem. Cohen et al. [8] proposed a method to eliminate the spurious ambiguity of shift-reduce transition systems, which covers existing systems such as the arc-standard and non-projective transition-based parsers [3]. While many transition sequences, which produce the same dependency tree, remain in Cohen's system and it rules non-canonical sequences out during parsing, in our non-spurious system, there exists just one transition sequence which produces the tree.

The arc-eager shift-reduce parser also has a spurious ambiguity problem. Goldberg and Nivre [24] attacked this problem by not only training with a canonical transition sequence but also with alternate optimal transitions that are calculated dynamically for a current state.

5.2.2 Methods to Improve Dependency Parsing

Higher-order features like third-order dependency relations are essential to improve dependency parsing accuracy [45, 73, 86]. A reranking approach is one effective solution to introduce rich features to a parser model in the context of constituency parsing [6, 33].

Hall [25] applied a *k*-best maximum spanning tree algorithm to non-projective dependency analysis, and showed that *k*-best discriminative reranking improves parsing accuracy in several languages. Sangati [75] proposed a *k*-best dependency reranking algorithm using a third-order generative model, and Hayashi et al. [26] extended it to a forest algorithm. Though forest reranking requires some approximations to integrate non-local features, it can explore larger search space than *k*-best reranking.

The stacking approach [70, 54] uses the output of one dependency parser to provide guide features for another. Stacking improves the parsing accuracy of second stage parsers on various language datasets.

The joint graph-based and transition-based approach [87, 4] uses an arc-eager shift-reduce parser with a joint graph-based and transition-based model. Though it improves parsing accuracy significantly, the large beam size of the shift-reduce parser harms its efficiency.

Sagae and Lavie [74] showed that combining the outputs of graph-based and transition-based parsers can improve parsing accuracies.

5.3 Arc-Standard Shift-Reduce Parsing

We use a beam search shift-reduce parser with dynamic programming as our baseline system. Figure 2.3 shows it as a deductive system [77]. A configuration is defined as the following:

$$\ell : (i, j, \mathbf{s}_d|\mathbf{s}_{d-1}|\dots|\mathbf{s}_1|\mathbf{s}_0) : \pi$$

where ℓ is the step size, $[i, j]$ is the span of the topmost stack element $\mathbf{s}_0$, $\mathbf{s}_d|\mathbf{s}_{d-1}|\dots|\mathbf{s}_1$ shows a stack with d elements at the top, where d is the window size used for defining features. The axiom is initialized with an input sentence of length n, $x = \mathbf{w}_0 \dots \mathbf{w}_n$ where $\mathbf{w}_0$ is a special root symbol $\$_0$. The system takes $2n$ steps for a complete analysis.

$$\begin{array}{rl}
\text{axiom}(c_0): & 0:(0,1,\mathbf{w}_0):\emptyset \\
\text{goal}(c_{2n}): & 2n:(0,n,\mathbf{s}_0):\emptyset \\
\text{shift}: & \dfrac{\overbrace{\ell:(\_,j,\mathbf{s}_d|\mathbf{s}_{d-1}|\ldots|\mathbf{s}_1|\mathbf{s}_0):\_}^{\text{conf } p}}{\ell+1:(j,j+1,\mathbf{s}_{d-1}|\ldots|\mathbf{s}_0|\mathbf{w}_j):(p)}\; i<n \\
\text{reduce}_{\curvearrowleft}: & \dfrac{\overbrace{\_:(i,j,\mathbf{s}'_d|\mathbf{s}'_{d-1}|\ldots|\mathbf{s}'_1|\mathbf{s}'_0):\pi'}^{\text{conf } p} \quad \overbrace{\ell:(j,k,\mathbf{s}_d|\mathbf{s}_{d-1}|\ldots|\mathbf{s}_1|\mathbf{s}_0):\pi}^{\text{conf } q}}{\ell+1:(i,k,\mathbf{s}'_d|\mathbf{s}'_{d-1}|\ldots|\mathbf{s}'_1|\mathbf{s}'_0{\curvearrowleft}\mathbf{s}_0):\pi'}\; \mathbf{s}'_0.\mathrm{h}.\mathbf{w}\neq\mathbf{w}_0\wedge p\in\pi \\
\text{reduce}_{\curvearrowright}: & \dfrac{\overbrace{\_:(i,j,\mathbf{s}'_d|\mathbf{s}'_{d-1}|\ldots|\mathbf{s}'_1|\mathbf{s}'_0):\pi'}^{\text{conf } p} \quad \overbrace{\ell:(j,k,\mathbf{s}_d|\mathbf{s}_{d-1}|\ldots|\mathbf{s}_1|\mathbf{s}_0):\pi}^{\text{conf } q}}{\ell+1:(i,k,\mathbf{s}'_d|\mathbf{s}'_{d-1}|\ldots|\mathbf{s}'_1|\mathbf{s}'_0{\curvearrowright}\mathbf{s}_0):\pi'}\; p\in\pi
\end{array}$$

Figure 5.1: The arc-standard transition-based dependency parsing system with dynamic programming: _ means "don't care". $\mathbf{a}{\curvearrowright}\mathbf{b}$ denotes that a tree $\mathbf{b}$ is attached to a tree $\mathbf{a}$.

π is a set of pointers to the predictor configurations, each of which is the configuration just before shifting the root word of $\mathbf{s}_0$ into stack[1]. Dynamic programming merges equivalent configurations in the same step if they have the same feature values. We add the feature templates shown in Table 5.1 to Huang and Sagae's feature templates [37].

Dynamic programming not only makes the beam search shift-reduce parser more efficient but it also can produce a packed forest encoding an exponential number of dependency trees.

A packed dependency forest can be represented by a weighted (directed) hypergraph. A weighted hypergraph is a pair $H = \langle V, E\rangle$, where V is the set of vertices and E is the set of hyperedges. Each hyperedge $e \in E$ is a tuple $e = \langle T(e), h(e), f_e\rangle$, where $h(e) \in V$ is its head vertex, $T(e) \in V^+$ is an ordered list of tail vertices, and f_e is a weight for e.

Figure 5.2 shows an example of a packed forest. Each binary hyperedge corresponds to a reduce action, and each leaf vertex corresponds to a shift action. Each vertex also corresponds to a configuration, and parse histories on the configurations can be encoded into the vertices. In the example, information about the topmost stack element is attached to the corresponding vertex marked with a non-terminal symbol X.

[1]As mentioned in Chapter 2, Huang and Sagae's dynamic programming [37] is based on a notion of a *push computation* [51].

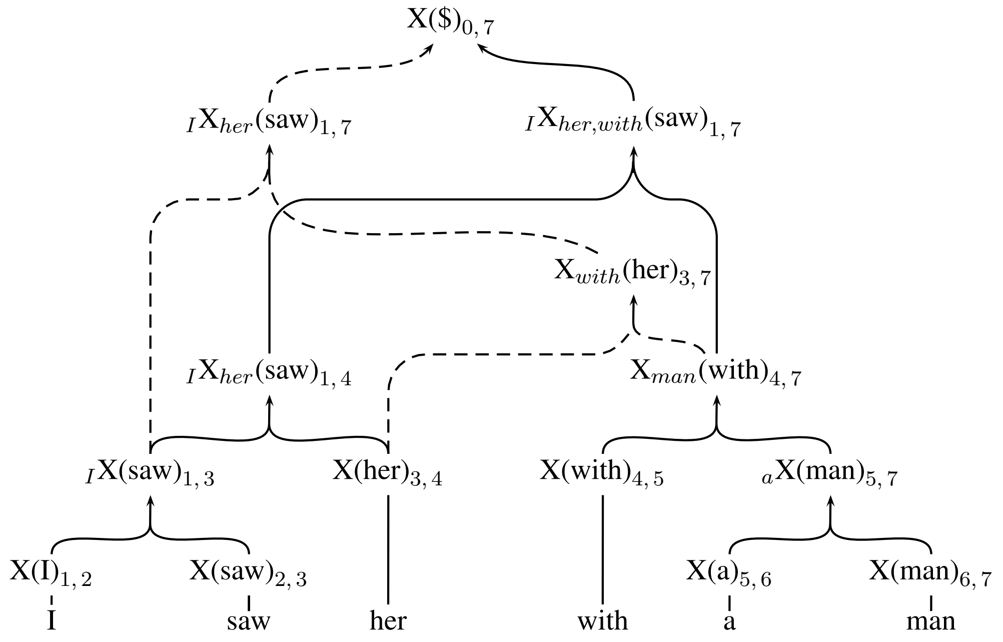

Figure 5.2: An example of packed dependency (derivation) forest: note that this forest is intuitively correct, but in practice dynamic programming shift-reduce parser produces more inefficient forests because of its hard condition of merging equivalent configurations.

Weights are omitted in the example. In practice, we attach each reduction weight to the corresponding hyperedge, and add the shift weight to the reduction weight when a shifted word is reduced.

5.4 Experiments (Spurious Ambiguity vs. Non-Spurious Ambiguity)

We conducted experiments on the English Penn Treebank (PTB) data to compare spurious and non-spurious shift-reduce parsers. We split the WSJ part of PTB into sections 02-21 for training, section 22 for development, and section 23 for test. We used the standard head rules [82] to convert phrase structure to dependency structure.

We used an early update version of averaged perceptron algorithm [13, 36] for training two shift-reduce dependency parsers. In all experiments, we fixed beam size to 12 for training both parsers.

$\mathbf{s}_0$.h.t $\circ$ $\mathbf{s}_0$.lc.t $\circ$ $\mathbf{s}_0$.lc2.t	$\mathbf{s}_0$.h.t $\circ$ $\mathbf{s}_0$.rc.t $\circ$ $\mathbf{s}_0$.rc2.t
$\mathbf{s}_1$.h.t $\circ$ $\mathbf{s}_1$.lc.t $\circ$ $\mathbf{s}_1$.lc2.t	$\mathbf{s}_1$.h.t $\circ$ $\mathbf{s}_1$.rc.t $\circ$ $\mathbf{s}_1$.rc2.t
$\mathbf{s}_0$.h.t $\circ$ $\mathbf{s}_0$.lc.t $\circ$ $\mathbf{s}_0$.lc2.t $\circ$ $\mathbf{q}_0$.t	
$\mathbf{s}_0$.h.t $\circ$ $\mathbf{s}_0$.rc.t $\circ$ $\mathbf{s}_0$.rc2.t $\circ$ $\mathbf{q}_0$.t	
$\mathbf{s}_0$.h.t $\circ$ $\mathbf{s}_1$.h.t $\circ$ $\mathbf{q}_0$.t $\circ$ $\mathbf{q}_1$.t	
$\mathbf{s}_0$.h.w $\circ$ $\mathbf{s}_1$.h.t $\circ$ $\mathbf{q}_0$.t $\circ$ $\mathbf{q}_1$.t	

Table 5.1: Additional feature templates for shift-reduce parsers: **q** denotes input queue. h, lc and rc are head, leftmost child and rightmost child of a stack element **s**. lc2 and rc2 denote the second leftmost and rightmost children. t and w are a part-of-speech (POS) tag and a word.

Table 5.2 shows experimental results for parsing the development and test datasets with each of the spurious and non-spurious shift-reduce parsers using several beam sizes. Parsing accuracies were evaluated by both unlabeled accuracy scores (UAS) with and without punctuations. The parsing times were measured on an Intel Core i7 2.8GHz. The average cpu time (per sentence) includes that of dumping packed forests. This result indicates that the non-spurious shift-reduce parser achieves better accuracies than the spurious shift-reduce parser without loss of efficiency.

Figure 5.3 shows oracle unlabeled accuracies of spurious *k*-best lists, non-spurious *k*-best lists, spurious forests, and non-spurious forests. We extract an oracle tree from each packed forest using the forest oracle algorithm [33]. Both forests produce much better results than the *k*-best lists, and non-spurious forests have almost the same oracle accuracies as spurious forests.

However, as shown in Table 5.3, spurious forests encode a number of non-unique dependency trees while all dependency trees in non-spurious forests are distinct from each other.

5.5 Decoding Algorithms for Hypergraph Search

5.5.1 Generalized Viterbi Algorithm

We introduce a dynamic programming algorithm for the highest derivation search problem on a hypergraph $H = (V, E)$.

First, we define a derivation with back-pointers $\hat{D}$ of a vertex v as a tuple $\langle e, \mathbf{j} \rangle$ such that $e \in BS(e)$ and $\mathbf{j} \in 1, 2, \ldots, k^{|e|}$. There is one-to-one correspondence between the

			8	16	32
dev.	spurious	UAS (w/o punc.)	92.5 / 93.5	**92.7** / 93.6	92.6 / 93.6
		milli sec. (per sent.)	0.01	**0.017**	0.03
	non-sp.	UAS (w/o punc.)	92.5 / **93.6**	92.6 / 93.6	92.6 / 93.6
		milli sec. (per sent.)	0.01	0.018	0.03
test	spurious	UAS (w/o punc.)	92.7 / 93.3	92.7 / 93.3	92.7 / 93.3
		milli sec. (per sent.)	0.01	**0.017**	0.03
	non-sp.	UAS (w/o punc.)	**92.8 / 93.4**	**92.9 / 93.5**	**92.9 / 93.5**
		milli sec. (per sent.)	0.01	0.018	0.03

			64	128	256
dev.	spurious	UAS (w/o punc.)	92.6 / 93.6	92.6 / 93.6	92.6 / 93.6
		milli sec. (per sent.)	**0.06**	0.13	0.25
	non-sp.	UAS (w/o punc.)	92.6 / 93.6	92.6 / 93.6	92.6 / 93.6
		milli sec. (per sent.)	0.07	0.13	0.25
test	spurious	UAS (w/o punc.)	92.8 / 93.3	92.8 / 93.3	92.7 / 93.3
		milli sec. (per sent.)	0.06	0.13	0.25
	non-sp.	UAS (w/o punc.)	**92.9 / 93.5**	**92.9 / 93.5**	**92.9 / 93.5**
		milli sec. (per sent.)	0.06	0.13	0.25

Table 5.2: Unlabeled accuracy scores (UAS) and parsing times (+forest dumping times, milli sec. per sentence) for parsing development (WSJ22) and test (WSJ23) data with spurious shift-reduce and proposed shift-reduce parser (non-sp.) using several beam sizes.

derivation with back-pointers and derivations of v:

$$\langle e, (j_1, \ldots, j_{|e|}) \rangle \sim \langle e, (D_{j_1}(T_1(e)), \ldots, D_{j_{|e|}}(T_{|e|}(e))) \rangle \tag{5.1}$$

where $D_i(v)$ denotes the i-th best derivation of v.

By using a scoring function s, an ordering on derivations is defined as $\hat{D}' \leq \hat{D}$ if $s(\hat{D}') \leq s(\hat{D})$.

Algorithm 4 show the 1-best Viterbi algorithm. This traverses the hypergraph in topological order, and calculates the 1-best derivation of each vertex v. If the arity of the hyperedge is constant, the time complexity of this algorithm is $O(|E|)$.

beam size	8	32	128
% of distinct trees (/10)	93.5	94.8	95.0
% of distinct trees (/100)	81.8	84.9	87.2
% of distinct trees (/1000)	70.6	73.1	77.6
% of distinct trees (/10000)	62.1	64.3	65.6

Table 5.3: The percentages of distinct dependency trees in 10, 100, 1000 and 10000 best trees extracted from spurious forests with several beam sizes.

Algorithm 4 Viterbi Algorithm

```
1: Input: a hypergraph $H = \langle V, E \rangle$
2: Output: 1-best derivation $\hat{D}_1(t)$
3: for $v \in V$ in topoligical order do
4:     for $e \in BS(v)$ do
5:         $\hat{D}_1(v) \leftarrow \max_{\leq}(\hat{D}_1(v), \langle e, \mathbf{1} \rangle)$
6:     end for
7: end for
```

5.5.2 *K*-best Generalized Viterbi Algorithm

Huang and Zhang [34] generalizes the 1-best Viterbi algorithm to extract *k*-best derivations. We introduce their "algorithm 2" here, and show it in Algorithm 5.

We denote $D_1(v), \ldots, D_k(v)$ as a vector $\mathbf{D}(v)$, and the *k*-best derivations search problem is to find $\mathbf{D}(t)$ where t is a root node of a hypergraph H. As well as the 1-best Viterbi algorithm, Huang and Zhang's algorithm 2 traverses the hypergraph in topological order (FindKbest).

First, the algorithm enumerates 1-best derivation for each incoming hyperedge in $BS(v)$, and insert each of the 1-best derivation into $cand[v]$ (GetCandidate).

Then, it calls the AppendNext function until getting *k*th-best derivations for v or $cand[v]$ becomes empty. The AppendNext function pops the best derivation $\langle e, \mathbf{j} \rangle$ from $cand[v]$ and insert it into $\mathbf{D}(v)$ ($\mathbf{p}$), and then its $|e|$ next best elements $\{\langle e, \mathbf{j}^l \rangle | 1 \leq l \leq |e|\}$ are inserted into $cand[v]$ by using $\mathbf{d}_i$ whose elements are all 0 except $d_i^i = 1$.

The overall time complexity of this algorithm is $O(|E| + |V| k \log k)$.

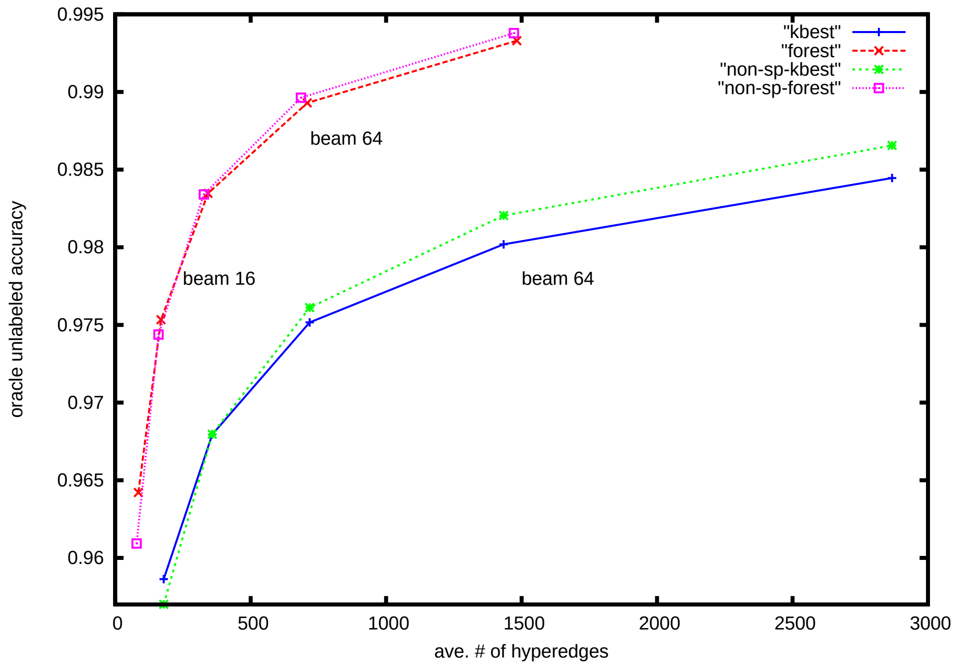

Figure 5.3: Each plot shows oracle unlabeled accuracies of spurious k-best lists, spurious forests, and non-spurious forests. The oracle accuracies are evaluated using unlabeled accuracy including punctuation.

5.6 Forest Reranking

5.6.1 Discriminative Reranking Model

We define a reranking model based on the graph-based features as the following:

$$\hat{y} = \operatorname*{argmax}_{y \in H} \alpha \cdot \mathbf{f}_{\mathrm{g}}(x, y) \tag{5.2}$$

where α is a weight vector, $\mathbf{f}_{\mathrm{g}}$ is a feature vector (g indicates "graph-based"), and H is a dependency forest.

This model assumes a hyperedge factorization which induces a decomposition of the feature vector as the following:

$$\alpha \cdot \mathbf{f}_{\mathrm{g}}(x, y) = \sum_{e \in y} \alpha \cdot \mathbf{f}_{\mathrm{g},e}(e). \tag{5.3}$$

The search problem can be solved by simply using the (generalized) Viterbi algorithm [44].

When using non-local features, the hyperedge factorization is redefined to the following:

$$\alpha \cdot \mathbf{f}_{\mathrm{g}}(x,y) = \sum_{e \in y} \alpha \cdot \mathbf{f}_{\mathrm{g},e}(e) + \alpha \cdot \mathbf{f}_{\mathrm{g},e,N}(e) \tag{5.4}$$

where $\mathbf{f}_{\mathrm{g},e,N}$ is a non-local feature vector. Though the cube-pruning algorithm [35] is an approximate decoding technique based on a k-best Viterbi algorithm, it can calculate the non-local scores efficiently.

The baseline score can be taken into the reranker as a linear interpolation:

$$\hat{y} = \operatorname*{argmax}_{y \in H} \beta \cdot \mathbf{sc}_{\mathrm{tr}}(x,y) + \alpha \cdot \mathbf{f}_{\mathrm{g}}(x,y) \tag{5.5}$$

where $\mathbf{sc}_{\mathrm{tr}}$ is the score from the baseline parser (tr indicates "transition-based"), and β is a scaling factor.

5.6.2 Features for Discriminative Model

Local Features

While the inference algorithm is a simple Viterbi algorithm, the discriminative model can use all tri-sibling features and some grand-sibling features[2] [45] as a local scoring factor in addition to the first- and sibling second-order graph-based features. This is because the first stage shift-reduce parser uses features described in Section 5.3 and this information can be encoded into vertices of a hypergraph.

The reranking model also uses guide features extracted from the 1-best tree predicted by the first stage shift-reduce parser. We define the guide features as first-order relations like those used in Nivre [70] though our parser handles only unlabeled and projective dependency structures. We summarize the features for discriminative reranking model as the following:

- First- and second-order features: these features are the same as those used in MST parser[3].

- Grand-child features: we define tri-gram POS features with POS tags of grand parent, parent, and rightmost or leftmost child.

[2]The grand-child and grand-sibling features can be used only when interacting with the leftmost or rightmost child and sibling. In experiments, the grand-sibling features were not effective in improving parsing accuracy. Therefore, in case of local reranking, we used only grand-child features.

[3]`http://www.seas.upenn.edu/~strctlrn/MSTParser/MSTParser.html`

- Tri-sibling features: we define tri-gram features with three POS-tags of child, sibling, and tri-sibling. We also define tri-gram features with one word and two POS tags of the above.
- Guide feaures: we define a feature indicating whether an arc from the child to the parent is present in the 1-best tree predicted by the first-stage shift-reduce parser, conjoined with the POS tags of the parent and child.
- PP-Attachment features: when a parent word is a preposition, we define tri-gram features with the parent word and POS tags of grand parent and rightmost child.

Non-local Features

To define richer features as a non-local factor, we extend a local reranking algorithm to augment each *k*-best item with all child vertices of its head vertex[4]. Information about all children enables the reranker to calculate the following features when reducing the head vertex:

- Grand-child features: we define tri-gram features with one word and two POS tags of grand parent, parent, and child.
- Grand-sibling features: we define 4-gram POS features with POS tags of grand parent, parent, sibling and child. We also define coordination features with POS tags of grand parent, parent and child when the sibling word is a coordinate conjunction.
- Valency features: we define a feature indicating the number of children of a head, conjoined with each of its word and POS tag.

When using the non-local features, we removed the local grand-child features from the model.

5.6.3 Oracle for Discriminative Training

A discriminative reranking model is trained on packed forests by using their oracle trees as correct. Therefore, more accurate oracles are essential to train a discriminative

[4]If each item is augmented with richer information, even features based on the entire subtree can be defined.

	sp.	**non-sp.**
ave. # of hyperedges	141.9	133.3
ave. # of vertices	199.1	187.6
ave. % of distinct trees	82.5	100.0
1-best UAS (w/ punc.)	92.5	92.6
oracle UAS (w/ punc.)	100.0	100.0

Table 5.4: Comparison of spurious (sp.) and non-spurious (non-sp.) forests: each forest is produced by baseline and proposed shift-reduce parsers using beam size 12 for 39832 training sentences with gold POS tags.

reranker	**pre-comp.**	**training**
spurious	16.4 min.	34.9 min.
non-spurious	15.5 min.	32.9 min.
spurious non-local	17.3 min.	64.3 min.
non-spurious non-local	16.2 min.	60.3 min.

Table 5.5: Training times on both spurious and non-spurious packed forests (beam 12): pre-comp. denotes cpu time for feature extraction and attaching features to all hyperedges. The non-local models were trained setting *k*-best size of cube-pruning to 5, and non-local features were calculated on-the-fly while training.

reranking model well. While large size forests have much more accurate oracles than small size forests, large forests have too many hyperedges to train a discriminative model on them, as shown in Figure 5.3. The usual forest reranking algorithms [33, 26] remove low quality hyperedges from large forests by using inside-outside forest pruning. However, producing large forests and pruning them are computationally very expensive. Instead, we propose a simpler method to produce small forests which have more accurate oracles by forcing beam search shift-reduce parser to keep correct state in the beam buffer. As the result, a correct tree can be always encoded in a produced packed forests.

	sp.	non-sp.
ave. # of hyperedges	127.0	119.1
ave. # of vertices	178.6	168.5
ave. % of distinct trees	82.4	100.0
1-best UAS (w/ punc.)	92.8	92.9
oracle UAS (w/ punc.)	97.0	97.0

Table 5.6: Comparison of spurious (sp.) and non-spurious (non-sp.) forests: each forest is produced by baseline and proposed shift-reduce parsers using beam size 12 for test data (WSJ23) with gold POS tags.

5.7 Experiments (Discriminative Reranking)

5.7.1 Experimental Setting

Following [33], the training set (WSJ02-21) is split into 20 folds, and each fold is parsed by each of the spurious and non-spurious shift-reduce parsers using beam size 12 with the model trained on sentences from the remaining 19 folds, dumping the outputs as packed forests.

The reranker is modeled by either equation (5.2) or (5.5). By our preliminary experiments using development data (WSJ22), we modeled the reranker as equation (5.2) when training but as equation (5.5) when testing[5] (i.e., the scores of the first-stage parser are not considered during training of the reranking model). This prevents the discriminative reranking features from *under-training* [80, 31].

A discriminative reranking model is trained on the packed forests by using the averaged perceptron algorithm with 5 iterations. When training non-local reranking models, we set *k*-best size of cube-pruning to 5.

5.7.2 Test with Gold POS tags

We show the comparison of dumped spurious and non-spurious packed forests for training data in Table 5.4. Both oracle accuracies are 100.0 due to the method described in Section 5.6.3. The 1-best accuracy of the non-spurious forests is higher than that of

[5]The scaling factor β was tuned by minimum error rate training (MERT) algorithm [71] using development data. The MERT algorithm is suited to tune low-dimensional parameters. The β was set to about 1.2 in case of local reranking, and to about 1.5 in case of non-local reranking.

system	w/ rerank.	milli sec. (per sent.)	UAS (w/o punc.)
sr (12)	–	0.011	92.8 / 93.3
(8)	w/ local	0.009 + 0.0056	93.03 / 93.69
(12)	w/ local	0.011 + 0.0079	93.03 / 93.68
(32)	w/ local	0.03 + 0.019	93.07 / 93.67
(64)	w/ local	0.06 + 0.039	93.0 / 93.61
(12, *k*=3)	w/ non-local	0.011 + 0.0085	93.17 / 93.78
(64, *k*=3)	w/ non-local	0.06 + 0.046	93.19 / 93.78
non-sp sr (12)	–	0.012	92.9 / 93.5
(8)	w/ local	0.01 + 0.005	93.05 / 93.73
(12)	w/ local	0.012 + 0.0074	93.21 / 93.87
(32)	w/ local	0.031 + 0.0184	93.22 / 93.84
(64)	w/ local	0.061 + 0.0375	93.23 / 93.83
(12, *k*=3)	w/ non-local	0.012 + 0.0083	93.28 / 93.9
(64, *k*=3)	w/ non-local	0.061 + 0.045	93.39 / 93.96

Table 5.7: Unlabeled accuracy scores and cpu times per sentence (parsing+reranking) when parsing and reranking test data (WSJ23) with gold POS tags: shift-reduce parser is denoted as sr (beam size, *k*: *k*-best size of cube pruning).

the spurious forests. As we expected, the results show that there are many non-unique dependency trees in the spurious forests. The spurious forests also get larger than the non-spurious forests.

Table 5.5 shows how long training on spurious and non-spurious forests took on an Opteron 8356 2.3GHz. It is clear from the results that training on non-spurious forests is more efficient than that on spurious forests.

Table 5.6 shows the statistics of spurious and non-spurious packed forests dumped by shift-reduce parsers using beam size 12 for test data. The trends are similar to those for training data shown in Table 5.4. We show the results of the forest reranking algorithms for test data in Table 5.7. Each spurious and non-spurious shift-reduce parser produces packed forests using four beam sizes 8, 12, 32, and 64. The reranking on non-spurious forests achieves better accuracies and is slightly faster than that on spurious forests consistently.

system	tok./sec.	UAS (o/ punc.)
sr (12)	2130	92.5
w/ local (12)	1290	92.8
non-sp sr (12)	1950	92.6
w/ local (12)	1300	92.98
w/ non-local (12, *k*=1)	1280	93.1
w/ non-local (12, *k*=3)	1180	**93.12**
w/ non-local (12, *k*=12)	1060	**93.12**
Huang10 sr (8)	782	92.1
Rush12 sr (16)	4780	92.5
Rush12 sr (64)	1280	92.7
Koo10	–	93.04
Rush12 third	20	93.3
Rush12 vine	4400	93.1
H-Zhang12 third	50	92.81
H-Zhang12 (label)	220	93.06
Y-Zhang11 (64, label)	680	92.9
Bohnet12 (80, label)	120	93.39

Table 5.8: Comparison with other systems: the results were evaluated on testing data (WSJ23) with automatic POS tags: label means labeled dependency parsing and the cpu times of our systems were taken on Intel Core i7 2.8GHz.

5.7.3 Test with Automatic POS tags

To compare the proposed reranking system with other systems, we evaluate its parsing accuracy on test data with automatic POS tags. We used the Stanford POS tagger[6] with a model trained on sections 02-21 to tag development and test data, and used 10-way jackknifing to tag training data. The tagging accuracies on training, development, and test data were 97.1, 97.2, and 97.5.

Table 5.8 lists the accuracy and parsing speed of our proposed systems together with results from related work. The difference of the parsing time does not represent the efficiency of the algorithm directly because each system was implemented in different programming language and the times were measured on different environments.

The accuracy of local reranking on non-spurious forests is the best among unlabeled

[6]`http://nlp.stanford.edu/software/tagger.shtml`

feature \ UAS	**w/ guide.**	**o/ guide.**
	92.98	92.86
Linear (first)	89,330	89,215
CorePos (first)	1,047,948	1,053,796
TwoObs (first)	1,303,911	1,325,990
Sibling (second)	290,291	292,849
Trip (second)	19,333	19,267
Grand-child	16,975	16,951
Guide	4,934	–
Tri-sibling	277,770	279,720
PP-Attachment	32,695	32,993
total	3,083,187	3,110,781

Table 5.9: Accuracy and the number of non-zero weighted features of the local reranking models with and without guide features: the first- and second-order features are named for MSTParser.

shift-reduce parsers, but slightly behind the third-order graph-based systems [45, 86, 73]. It is likely that the difference comes from the fact that our local reranking model can define only some of the grand-child related features.

To define all grand-child features and other non-local features, we also experimented with the non-local reranking algorithm on non-spurious packed forests. It achieved almost the same accuracy as the previous third-order graph-based algorithms. Moreover, the computational overhead is very small when setting *k*-best size of cube-pruning small.

5.7.4 Analysis

One advantage of our reranking approach is that guide features can be defined as in stacked parsing. To analyze the effect of the guide features on parsing accuracy, we remove the guide features from baseline reranking models with and without non-local features used in Section 5.7.3. The results are shown in Table 5.9 and 5.10. The parsing accuracies of the baseline reranking models are better than those of the models without guide features though the number of guide features is not large. Additionally, each

	w/ guide.	**o/ guide.**
UAS / feature	**93.12**	93.04
Linear (first)	88,634	88,934
CorePos (first)	1,035,897	1,045,242
TwoObs (first)	1,274,834	1,301,103
Sibling (second)	284,341	288,796
Trip (second)	19,201	19,219
Guide	4,916	–
Tri-sibling	272,418	276,025
PP-Attachment	32,085	32,577
Grand-child	718,064	730,663
Grand-sibling	72,865	73,103
Valency	49,262	49,677
total	3,852,517	3,905,339

Table 5.10: Accuracy and the number of non-zero weighted features of the non-local reranking models with and without guide features: the first- and second-order features are named for MSTParser.

model with guide features is smaller than that without guide features. This indicates that stacking has a good effect on training the models.

To further investigate the effects of guide features, we tried to define unlabeled versions of the second-order guide features used in [54, 56]. However, these features did not produce good results, and investigation to determine the cause is an important future work.

We also examined parsing errors in more detail. Table 5.11 shows root and sentence complete rates of three systems, the non-spurious shift-reduce parser, local reranking, and non-local reranking. The two reranking systems outperform the shift-reduce parser significantly, and the non-local reranking system is the best among them.

Part of the difference between the shift-reduce parser and reranking systems comes from the correction of coordination errors. Table 5.12 shows the head correct rate, recall, precision, F-measure and complete rate of coordination structures, by which we mean the head and siblings of a token whose POS tag is CC. The head correct rate denotes how correct a head of the CC token is. The recall, precision, F-measure are measured by counting arcs between the head and siblings. When the head of the

system	UAS	root	comp.
non-sp sr	92.6	95.8	45.6
local	92.98	96.1	48.1
non-local	**93.12**	**96.3**	**48.2**

Table 5.11: Unlabeled accuracy, root correct rate, and sentence complete rate: these scores are measured on test data (WSJ23) without punctuations.

	non-sp sr	local	non-local
head correct	87.73	**88.97**	88.83
recall	82.38	**84.35**	84.11
precision	83.07	**84.57**	83.98
F-measure	82.72	**84.46**	84.05
comp.	62.92	64.52	**65.18**

Table 5.12: Head correct rate, recall, precision, F-measure, and complete rate of coordination strutures: these are measured on test data (WSJ23).

CC token is incorrect, all arcs of the coordination structure are counted as incorrect. Therefore, the recall, precision, F-measure are greatly affected by the head correct rate, and though the complete rate of non-local reranking is higher than that of local reranking, the results of the first three measures are lower.

We assume that the improvements of non-local reranking over the others can be mainly attributed to the better prediction of the structures around the sentence root because most of the non-local features are useful for predicting these structures. Table 5.13 shows the recall, precision and F-measure of grand-child structures whose grand parent is a sentence root symbol \$. The results support the above assumption. The root correct rate directly influences on prediction of the overall structures of a sentence, and it is likely that the reduction of root prediction errors brings better results.

5.7.5 Experiments on Chinese

We also experiment on the Penn Chinese Treebank (CTB5). Following Huang and Sagae [37], we split it into training (secs 001-815 and 1001-1136), development (secs 886-931 and 1148-1151), and test (secs 816-885 and 1137-1147) sets, and use the head rules of Zhang's work [87]. The training set is split into 10 folds to dump packed

system	recall	precision	F-measure
non-sp sr	91.58	92.5	92.04
local	91.96	92.95	92.45
non-local	**92.44**	**93.07**	**92.75**

Table 5.13: Recall, precision, and F-measure of grand-child structures whose grand parent is an artificial root symbol: these are measured on test data (WSJ23).

system	UAS	root	comp.
sr (12)	85.3	78.6	33.4
w/ non-local (12, *k*=3)	85.8	79.4	34.2
non-sp sr (12)	85.3	78.4	33.7
w/ non-local (12, *k*=3)	**85.9**	**79.6**	**34.3**

Table 5.14: Results on Chinese Treebank data (CTB5): evaluations are performed without punctuations.

forests for training of reranking models.

We set the beam size of both spurious and non-spurious parsers to 12, and the number of perceptron training iterations to 25 for the parsers and to 8 for both rerankers. Table 5.14 shows the results for the test sets. As we expected, reranking on non-spurious forests outperforms that on spurious forests.

5.8 Summary

We have presented a discriminative forest reranking algorithm for dependency parsing. This can be seen as a kind of joint transition-based and graph-based approach because the first-stage parser is a shift-reduce parser and the second-stage reranker uses a graph-based model.

Additionally, we have proposed a dynamic programming arc-standard transition-based dependency parser without spurious ambiguity, along with a heuristic that encodes the correct tree in the output packed forest for reranker training, and shown that forest reranking works well on packed forests produced by the proposed parser.

To improve the accuracy of reranking, we will engage in feature engineering. We need to further investigate effective higher-order guide and non-local features. It also seems promising to extend the unlabeled reranker to a labeled one because labeled information often improves unlabeled dependency accuracy.

In this thesis, we adopt a reranking approach, but a rescoring approach is more promising to improve efficiency because it does not have the overhead of dumping packed forests.

Algorithm 5 *k*-best Viterbi Algorithm

```
1: Input: a hypergraph $H = \langle V, E \rangle$
2: Output: $k$-best derivations $\mathbf{D}(t) = \{D_1(t) \ldots D_k(t)\}$
3: function FindKBest($k$)
4: for $v \in V$ in topological order do
5:     KBest($v$, $k$)
6: end for
7: end function
8:
9: function KBest($v$, $k$)
10: GetCandidates($v$, $k$)
11: while $|\hat{\mathbf{D}}(v)| < k$ and $|cand[v]| > 0$ do
12:     AppendNext($cand[v]$, $\hat{\mathbf{D}}(v)$)
13: end while
14: end function
15:
16: function GetCandidates($v$, $k$)
17: $temp \leftarrow \{\langle e, \mathbf{1} \rangle | e \in BS(v)\}$
18: $cand[v] \leftarrow$ the top $k$ elements in $temp$
19: HEAPIFY($cand[v]$)
20: end function
21:
22: function AppendNext($cand$, $\mathbf{p}$)
23: $\langle e, \mathbf{1} \rangle \leftarrow$ Extract-Min($cand$)
24: append $\langle e, \mathbf{1} \rangle$ to $\mathbf{p}$
25: for $i \leftarrow 1 \ldots |e|$ do
26:     $\mathbf{j}$'$\leftarrow \mathbf{j} + \mathbf{d_i}$
27:     if $j'_i \leq |\hat{\mathbf{D}}(T_i(e))|$ and $\langle \mathbf{j}', e \rangle \notin cand$ then
28:         insert($cand, \langle \mathbf{j}', e \rangle$)
29:     end if
30: end for
31: end function
```

Chapter 6

Conclusions

The transition-based approach is one of the most promising approaches for data-driven dependency parsing because of its computational efficiency. However, its search algorithm is incremental and non-optimal, and causes more parse errors than the state-of-the-art graph-based algorithms.

In this thesis, we have brought up three problems on existing transition-based parsing algorithms for unlabeled projective dependency analysis, and addressed these problems for improving the transition-based systems.

In Chapter 3, we have presented a data-driven top-down (head-coner) parser, and shown its usefulness through experiments on English data. This study is a first approach to data-driven top-down dependency parsing, and also brings a new direction of studies on data-driven dependency parsing.

In Chapter 4, by applying non-spuriosity property of the proposed top-down parser to the arc-standard shift-reduce parser, we have proposed a novel shift-reduce parsing algorithm without spurious ambiguity. This parser not only works as fast as the previous shift-reduce parsing algorithms but also achieves higher accuracy.

In Chapter 5, we have presented a forest reranking algorithm which can combine higher-order graph-based models efficienty with transition-based models. This reranking approach achieves the state-of-the-accuracy and works much faster than the higher-order graph-based parser. In addition, we have showed that our proposed shift-reduce parser in Chapter 4 improves the efficiency and accuracy of reranking.

As we mentioned in the first line of Chapter 1, many NLP applications such as machine translation [62, 61] use a syntax tree of a sentence as their input. Because these applications need to return their outputs to the users as fast as possible, a not only highly accurate but also fast parser must be required.

Through step-by-step problem solving, in this thesis, we have accomplished improving the parsing accuracy of previous transition-based systems with little loss of their efficiency. We believe that the proposed parsing systems in this thesis are useful for the many NLP applications and applied to them.

However, there remain several promising and interesting future works on our approaches. Here, we refer to the following two future works.

- Extention to handle non-projective dependency trees.
- Extention to handle labeled dependency arcs.

All algorithms proposed in this thesis are for unlabeled projective dependency parsing. Extentions of two transition-based parsers proposed in Chapters 3 and 4 to handle non-projective dependency structures might be very interesting from algorithmic perspectives. It also seems promising to extend our algorithms to handle labeled dependency arcs because labeled information often improves dependency parsing accuracy.

In future, we plan to address these futher problems.

Appendix

Correctness of the Top-down Transition-based Parsing Algorithm for a class of Projective Dependency Graphs

To prove the *correctness* of the system in Figure 3.1 for the **projective** dependency graph, we use the proof strategy of [68]. The correct deductive system is both *sound* and *complete*.

Theorem Appendix.1 *The deductive system in Figure 3.1 is correct for the class of dependency forest.*

Proof Appendix.1 *To show soundness, we show that $G_{p_0} = (V_W, \emptyset)$, which is a directed graph defined by the axiom, is well-formed and projective, and that every transition preserves this property.*

- **ROOT***: The node 0 is a root in G_{p_0}, and the node 0 is on the top of stack of p_0. The two pred actions put a word onto the top of stack, and predict an arc from root or its descendant to the child. The comp actions add the predicted arcs which include no arc of $(x, 0)$.*
- **SINGLE-HEAD***: G_{p_0} is single-head. A node y is no longer in stack and queue after a comp action creates an arc (x, y). The node y cannot make any arc (x', y) after the removal.*
- **ACYCLICITY***: G_{p_0} is acyclic. A cycle is created only if an arc (x, y) is added when there is a directed path $y \to^* x$. The node x is no longer in stack and queue when the directed path $y \to^* x$ was made by adding an arc (l, x). There is no chance to add the arc (x, y) on the directed path $y \to^* x$.*
- **PROJECTIVITY***: G_{p_0} is projective. Projectivity is violated by adding an arc (x, y) when there is a node l in $x < l < y$ or $y < l < x$ with the path to or from the outside of the span x and y. When $pred_{\curvearrowright}$ creates an arc relation from x to y, the node y cannot be scanned before all nodes l in $x < l < y$ are scanned and*

completed. When pred$_{\curvearrowleft}$ *creates an arc relation from x to y, the node y cannot be scanned before all nodes k in $k < y$ are scanned and completed, and the node x cannot be scanned before all nodes l in $y < l < x$ are scanned and completed. In those processes, the node l in $x < l < y$ or $y < l < x$ does not make a path to or from the outside of the span x and y, and a path $x \rightarrow^* l$ or $y \rightarrow^* l$ is created.* $\square$

To show completeness, we show that for any sentence W, and dependency forest $G_W = (V_W, A_W)$, there is a transition sequence $C_{0,m}$ such that $G_{p_m} = G_W$ by an inductive method.

- *If $|W| = 1$, the projective dependency graph for W is $G_W = (\{0\}, \emptyset)$ and $G_{p_0} = G_W$.*
- *Assume that the claim holds for sentences with length less or equal to t, and assume that $|W| = t+1$ and $G_W = (V_W, A_W)$. The subgraph $G_{W'}$ is defined as $(V_W - t, A^{-t})$ where $A^{-t} = A_W - \{(x,y) | x = t \vee y = t\}$. If G_W is a dependency forest, then $G_{W'}$ is also a dependency forest. It is obvious that there is a transition sequence for constructing G_W except arcs which have a node t as a head or a dependent*[1]*. There is a state $p_q = q : \langle i, x, t+1 \rangle : \_$ for i and x ($0 \leq x < i < t+1$). When x is the head of t, pred*$_{\curvearrowright}$ *to t creates a state $p_{q+1} = q+1 : \langle i, t, t+1 \rangle : \{p_q\}$. At least one node y in $i \leq y < t$ becomes the dependent of t by pred*$_{\curvearrowleft}$ *and there is a transition sequence for constructing a tree rooted by y. After constructing a subtree rooted by t and spaned from i to t, t is scaned, and then comp creates an arc from x to t. It is obvious that the remaining transition sequence exists. Therefore, we can construct a transition sequence $C_{0,m}$ such that $G_{p_m} = G_W$.* $\square$

The deductive sysmtem in Figure 3.1 is both sound and complete. Therefore, it is correct. $\square$

[1]This transition sequence is defined for $G_{W'}$, but it is possible to be regarded as the definition for G_W as long as the transition sequence is indifferent from the node t.

Bibliography

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1: Parsing. Prentice-Hall, 1972.

[2] H. Alshawi. Head automata for speech translation. In *Proc. the ICSLP*, 1996.

[3] G. Attardi. Experiments with a multilanguage non-projective dependency parser. In *Proc. of the 10th Conference on Natural Language Learning*, pages 166–170, 2006.

[4] B. Bohnet and J. Kuhn. The best of bothworlds – a graph-based completion model for transition-based parsers. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 77–87, 2012.

[5] X. Carreras. Experiments with a higher-order projective dependency parser. In *Proc. the CoNLL-EMNLP*, pages 957–961, 2007.

[6] E. Charniak and M. Johnson. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 173–180, 2005.

[7] Y.-J. Chu and T.-H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14(1396-1400):270, 1965.

[8] S. B. Cohen, C. Gómez-Rodríguez, and G. Satta. Elimination of spurious ambiguity in transition-based dependency parsing. Technical report, 2012.

[9] M. Collins. Three generative, lexicalised models for statistical parsing. In *Proc. the 35th ACL*, pages 16–23, 1997.

[10] M. Collins. *Head-driven statistical models for natural language parsing*. PhD thesis, University of Pennsylvania, 1999.

[11] M. Collins. Discriminative reranking for natural language parsing. In *Proc. the ICML*, 2000.

[12] M. Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 1–8, 2002.

[13] M. Collins and B. Roark. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL'04)*, 2004.

[14] M. A. Covington. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th annual ACM southeast conference*, pages 95–102. Citeseer, 2001.

[15] J. Earley. An efficient context-free parsing algorithm. *Communications of the Association for Computing Machinery*, 13(2):94–102, 1970.

[16] J. Edmonds. Optimum branchings. *Journal of Research of the National Bureau of Standards B*, 71:233–240, 1967.

[17] J. Eisner. Bilexical grammars and a cubic-time probabilistic parser. In *Proceedings of the 5th International Workshop on Parsing Technologies (IWPT)*, pages 54–65, 1997.

[18] J. Eisner. Bilexical grammars and their cubic-time parsing algorithms. *Advances in Probabilistic and Other Parsing Technologies*, 16:29–61, 2000.

[19] J. M. Eisner. An empirical comparison of probability models for dependency grammar. In *Technical Report*, pages 1–18, 1996.

[20] J. M. Eisner. Three new probabilistic models for dependency parsing: An exploration. In *Proc. the 16th COLING*, pages 340–345, 1996.

[21] J. M. Eisner and G. Satta. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, pages 457–464, 1999.

[22] H. Gaifman. Dependency systems and phrase-structure systems. *Information and control*, 8(3):304–337, 1965.

[23] Y. Goldberg and M. Elhadad. An efficient algorithm for easy-first non-directional dependency parsing. In *Proc. the HLT-NAACL*, pages 742–750, 2010.

[24] Y. Goldberg and J. Nivre. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of the 24rd International Conference on Computational Linguistics (Coling 2012)*, 2012.

[25] K. Hall. K-best spanning tree parsing. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 392–399, 2007.

[26] K. Hayashi, T. Watanabe, M. Asahara, and Y. Matsumoto. The third-order variational reranking on packed-shared dependency forests. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1479–1488, 2011.

[27] K. Hayashi, T. Watanabe, M. Asahara, and Y. Matsumoto. Head-driven transition-based parsing with top-down prediction. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*, pages 657–665, 2012.

[28] D. G. Hays. Dependency theory: A formalism and some observations. *Language*, 40(4):511–525, 1964.

[29] P. Hellwig. Dependency unification grammar. In *Proceedings of the 11th coference on Computational linguistics*, pages 195–198. Association for Computational Linguistics, 1986.

[30] T. Holan, V. Kuboň, and M. Plátek. A prototype of a grammar checker for czech. In *Proceedings of the fifth conference on Applied natural language processing*, pages 147–154. Association for Computational Linguistics, 1997.

[31] K. Hollingshead and B. Roark. Reranking with baseline system scores and ranks as features. In *CSLU-08-001, Center for Spoken Language Understanding, Oregon Health and Science University*, 2008.

[32] L. Huang. Statistical syntax-directed translation with extended domain of locality. In *In Proc. AMTA 2006*, pages 66–73, 2006.

[33] L. Huang. Forest reranking: Discriminative parsing with non-local features. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics*, pages 586–594, 2008.

[34] L. Huang and D. Chiang. Better k-best parsing. In *Proceedings of the 11th International Conference on Parsing Technologies (IWPT'05)*, pages 53–64, 2005.

[35] L. Huang and D. Chiang. Forest rescoring: Faster decoding with integrated language models. In *Proceedings of the 45th Annual Meeting of the Association of Computat ional Linguistics*, pages 144–151, 2007.

[36] L. Huang, S. Fayong, and Y. Guo. Structured perceptron with inexact search. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 142–151, 2012.

[37] L. Huang and K. Sagae. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL'10)*, pages 1077–1086, 2010.

[38] R. A. Hudson. *Word grammar*. Blackwell Oxford, 1984.

[39] H. Isozaki, H. Kazawa, and T. Hirao. A deterministic word dependency analyzer enhanced with preference learning. In *Proc. the 21st COLING*, pages 275–281, 2004.

[40] M. Johnson. Transforming projective bilexical dependency grammars into efficiently-parsable CFGs with unfold-fold. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL'07)*, pages 168–175, 2007.

[41] T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical report, DTIC Document, 1965.

[42] M. Kay. Head driven parsing. In *Proc. the IWPT*, 1989.

[43] K. Kitagawa and K. Tanaka-Ishii. Tree-based deterministic dependency parsing — an application to nivre's method —. In *Proc. the 48th ACL 2010 Short Papers*, pages 189–193, July 2010.

[44] D. Klein and C. D. Manning. Parsing and hypergraphs. In *Proceedings of the 7th International Workshop on Parsing Technologies*, 2001.

[45] T. Koo and M. Collins. Efficient third-order dependency parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL'10)*, pages 1–11, 2010.

[46] T. Koo, A. M. Rush, M. Collins, T. Jaakkola, and D. Sontag. Dual decomposition for parsing with non-projective head automata. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1288–1298, 2010.

[47] T. Koo, A. M. Rush, M. Collins, T. Jaakkola, and D. Sontag. Dual decomposition for parsing with non-projective head automata. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1288–1298, October 2010.

[48] S. Kübler, R. T. McDonald, and J. Nivre. *Dependency Parsing*. Morgan & Claypool Publishers, 2009.

[49] T. Kudo and Y. Matsumoto. Japanese dependency structure analysis based on support vector machines. In *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics-Volume 13*, pages 18–25. Association for Computational Linguistics, 2000.

[50] T. Kudo and Y. Matsumoto. Japanese dependency analysis using cascaded chunking. In *proceedings of the 6th conference on Natural language learning-Volume 20*, pages 1–7. Association for Computational Linguistics, 2002.

[51] M. Kuhlmann, C. Gómez-Rodríguez, and G. Satta. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, pages 673–682, 2011.

[52] V. Lombardo and L. Lesmo. An earley-type recognizer for dependency grammar. In *Proc. the 16th COLING*, pages 723–728, 1996.

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

[54] T. Martins, F. André, D. Das, N. A. Smith, and E. P. Xing. Stacking dependency parsers. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 157–166, 2008.

[55] D. McAllester. A reformulation of eisner and satta's cubic time parser for split head automata grammars. 1999. http://ttic.uchicago.edu/ dmcallester/.

[56] D. McClosky, W. Che, M. Recasens, M. Wang, R. Socher, and C. D. Manning. Stanford 's system for parsing the english web. In *Proceedings of First Workshop on Syntactic Analysis of Non-Canonical Language (SANCL) at NAACL 2012*, 2012.

[57] R. McDonald, K. Crammer, and F. Pereira. Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 91–98, 2005.

[58] R. McDonald and F. Pereira. Online learning of approximate dependency parsing algorithms. In *Proc. EACL*, pages 81–88, 2006.

[59] R. McDonald and G. Satta. On the complexity of non-projective data-driven dependency parsing. In *Proc. of IWPT*, pages 121–132, 2007.

[60] I. A. Melčuk. *Dependency syntax: theory and practice*. State University of New York Press, 1988.

[61] H. Mi and L. Huang. Forest-based translation rule extraction. In *Proc. of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 206–214, Honolulu, Hawaii, 2008.

[62] H. Mi, L. Huang, and Q. Liu. Forest-based translation. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 192–199, Columbus, Ohio, June 2008.

[63] M.-J. Nederhof. Weighted deductive parsing and knuth's algorithm. *Computational Linguistics*, 29:135–143, 2003.

[64] J. Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the 10th International Conference on Parsing Technologies (IWPT'03)*, pages 149–160, 2003.

[65] J. Nivre. Incrementality in deterministic dependency parsing. In *Proc. the ACL Workshop Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57, 2004.

[66] J. Nivre. *Inductive Dependency Parsing of Natural Language Text*. PhD thesis, School of Mathematics and Systems Engineering, Växjö University, 2005.

[67] J. Nivre. Constraints on non-projective dependency parsing. In *Eleventh Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 73–80, 2006.

[68] J. Nivre. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34:513–553, 2008.

[69] J. Nivre. Sorting out dependency parsing. *Advances in Natural Language Processing*, pages 16–27, 2008.

[70] J. Nivre and R. McDonald. Integrating graph-based and transition-based dependency parsers. In *Proceedings of ACL-08: HLT*, pages 950–958, 2008.

[71] F. J. Och. Minimum error rate training in statistical machine translation. In *Proc. the 41st ACL*, pages 160–167, 2003.

[72] S. Riedel and J. Clarke. Incremental integer linear programming for non-projective dependency parsing. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 129–137, July 2006.

[73] A. Rush and S. Petrov. Vine pruning for efficient multi-pass dependency parsing. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 498–507, 2012.

[74] K. Sagae and A. Lavie. Parser combination by reparsing. In *Proc. HLT*, pages 129–132, 2006.

[75] F. Sangati, W. Zuidema, and R. Bod. A generative re-ranking model for dependency parsing. In *Proceedings of the 11th International Conference on Parsing Technologies (IWPT'09)*, pages 238–241, 2009.

[76] P. Sgall, E. Hajicová, and J. Panevová. *The meaning of the sentence in its semantic and pragmatic aspects*. Springer, 1986.

[77] S. M. Shieber, Y. Schabes, and F. C. N. Pereira. Principles and implementation of deductive parsing. *J. Log. Program.*, 24(1&2):3–36, 1995.

[78] K. Sikkel and R. op den Akker. Predictive head-corner chart parsing. *Recent Advances in Parsing Technology. Kluwer Academic, Netherlands*, pages 169–182, 1996.

[79] A. Stolcke. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–201, 1995.

[80] C. Sutton, M. Sindelar, and A. McCallum. Reducing weight undertraining in structured discriminative learning. In *Conference on Human Language Technology and North American Association for Computational Linguistics (HLT-NAACL)*, 2006.

[81] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publisher, 1986.

[82] H. Yamada and Y. Matsumoto. Statistical dependency analysis with support vector machines. In *Proceedings of the 10th International Conference on Parsing Technologies (IWPT'03)*, pages 195–206, 2003.

[83] K. Yamada and K. Knight. A syntax-based statistical translation model. In *Proceedings of 39th Annual Meeting of the Association for Computational Linguistics*, pages 523–530, Toulouse, France, July 2001.

[84] D. H. Younger. Recognition and parsing of context-free languages in time ¡ i¿ n¡/i¿¡ sup¿ 3¡/sup¿. *Information and control*, 10(2):189–208, 1967.

[85] D. Yuan and M. Palmer. Machine translation using probabilistic synchronous dependency insertion grammars. In *Proceedings of the 42rd Annual Meeting of the Association of Computational Linguistics*, pages 541–548, Ann Arbor, Michigan, June 2005.

[86] H. Zhang and R. McDonald. Generalized higher-order dependency parsing with cube pruning. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 320–331, 2012.

[87] Y. Zhang and S. Clark. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 562–571, 2008.

[88] Y. Zhang and J. Nivre. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193, 2011.

List of Publication

International Journal Papers

- Katsuhiko Hayashi, Shuhei Kondo, Yuji Matsumoto. Efficient Stacked Dependency Parsing by Forest Reranking. *Transactions of the Association for Computational Linguistics*, 2013.

Domestic Journal Papers

- Katsuhiko Hayashi, Taro Watanabe, Hajime Tsukada, Hideki Isozaki, and Seiichi Yamamoto. Max-margin Learning for Statistical Machine Translation. *Transactions of the Japanese Society for Artificial Intelligence*, Vol.25, No.5, pp.560-569, July 2010 (in Japanese).

International Conference

- Katsuhiko Hayashi, Taro Watanabe (NICT), Masayuki Asahara and Yuji Matsumoto. Head-driven Transition-based Parsing with Top-down Prediction. *In Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Jeju, South Korea, pp.657-665, July 2012, oral.

- Katsuhiko Hayashi, Taro Watanabe (NICT), Masayuki Asahara, and Yuji Matsumoto. Third-order variational reranking on packed-shared dependency forests. *In Proceedings of the Conference on Empirical Methods in Natural Language Processing*, Edinburgh, Scotland, UK, pp.1479-1488, July 2011, oral.

- Katsuhiko Hayashi, Hajime Tsukada (NTT), Katsuhito Sudoh (NTT), Kevin Duh (NTT), and Seiichi Yamamoto (Doshisha Univ). Hierarchical phrase-based machine translation with word-based reordering model. *In Proceedings of the 23rd International Conference on Computational Linguistics (COLING 2010)*, Beijing, China, pp.439-446, August 2010, oral.

Other Publications

- Katsuhiko Hayashi, Taro Watanabe, Masayuki Asahara, and Yuji Matsumoto. Split Head Automata for Dependency Parsing. *In Information Processing Society of Japan SIG notes, NL-206*, pp.1-8, May, 2012. (In Japanese)

Awards

- Young Scientist Award of Information Processing Society of Japan SIG notes, NL-206.