Optimization of Natural Language Processing Components for Robustness and Scalability

by

Jinho D. Choi

B.A., Coe College, 2002

M.S.E., University of Pennsylvania, 2003

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado Boulder in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2012

This thesis entitled:
Optimization of Natural Language Processing Components for Robustness and Scalability
written by Jinho D. Choi
has been approved for the Department of Computer Science

Martha Palmer

James Martin

Wayne Ward

Bhuvana Narasimhan

Joakim Nivre

Nicolas Nicolov

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Choi, Jinho D. (Ph.D., Computer Science)

Optimization of Natural Language Processing Components for Robustness and Scalability

Thesis directed by Dr. Martha Palmer

This thesis focuses on the optimization of NLP components for robustness and scalability. Three kinds of NLP components are used for our experiments, a part-of-speech tagger, a dependency parser, and a semantic role labeler. For part-of-speech tagging, dynamic model selection is introduced. Our dynamic model selection approach builds two models, domain-specific and generalized models, and selects one of them during decoding by comparing similarities between lexical items used for building these models and input sentences. As a result, it gives robust tagging accuracy across corpora and shows fast tagging speed. For dependency parsing, a new transition-based parsing algorithm and a bootstrapping technique are introduced. Our parsing algorithm learns both projective and non-projective transitions so it can generate both projective and non-projective dependency trees yet shows linear time parsing speed on average. Our bootstrapping technique bootstraps parse information used as features for transition-based parsing, and shows significant improvement for parsing accuracy. For semantic role labeling, a conditional higher-order argument pruning algorithm is introduced. A higher-order pruning algorithm improves the coverage of argument candidates and shows improvement on the overall F1-score. The conditional higher-order pruning algorithm also noticeably reduces average labeling complexity with minimal reduction in F1-score.

For all experiments, two sets of training data are used; one is from the Wall Street Journal corpus, and the other is from the OntoNotes corpora. All components are evaluated on 9 different genres, which are grouped separately for in-genre and out-of-genre experiments. Our experiments show that our approach gives higher accuracies compared to other state-of-the-art NLP components, and runs fast, taking about 3-4 milliseconds per sentence for processing all three components. All components are publicly available as an open source project, called ClearNLP. We believe that this project is beneficial for many NLP tasks that need to process large-scale heterogeneous data.

Dedication

This thesis is dedicated to my parents, Youngsang Choi and Kyungsook Lee, my American grandparents, Thomas and Elizabeth Janik, and my sister, Jeany Choi, who have loved, supported, and encouraged me ever since I was born. It is also dedicated to my prayer partner, Kyungshim Lee, and my future wife, Juyoung Choi, who have prayed for me constantly.

Acknowledgements

We gratefully acknowledge the support of the following grants. Any contents expressed in this material are those of the authors and do not necessarily reflect the views of any grant.

- The National Science Foundation Grants IIS-0325646, Domain Independent Semantic Parsing, CISE-CRI-0551615, Towards a Comprehensive Linguistic Annotation, CISE-CRI 0709167, Collaborative: A Multi-Representational and Multi-Layered Treebank for Hindi/Urdu, CISE-IIS-RI-0910992, Richer Representations for Machine Translation.
- A grant from the Defense Advanced Research Projects Agency (DARPA/IPTO) under the GALE program, DARPA/CMO Contract No. HR0011-06-C-0022, subcontract from BBN, Inc.
- A subcontract from the Mayo Clinic and Harvard Children's Hospital based on a grant from the ONC, 90TR0002/01.
- Strategic Health Advanced Research Project Area 4: Natural Language Processing.

Contents

Tables

Figures

Chapter 1

Introduction

1.1 Research questions and objectives

Unlike earlier stages of Natural Language Processing (NLP) where it was used by certain groups of people for specific purposes (e.g., DARPA using NLP for machine translation),[1] the application of NLP has expanded to everyday computing and has also broadened to a general audience. For example, Google and Bing Translators are developed using an NLP technique called statistical machine translation, and they provide an online service that instantly translates texts and web pages.[2] IBM Watson is based on an NLP application called question-answering, and has shown outstanding performance in answering Jeopardy type questions (`www.watson.ibm.com`). NLP is no longer in the future; it has become a present-day reality.

As NLP components mature, more attention is drawn to the practical aspects of these components. In this respect, two things must be evaluated. First, NLP components should be tested for robustness in handling heterogeneous data.[3] Many recent NLP components take statistical learning approaches, which usually perform well when training and testing data are from similar sources. However, the performance degrades increasingly as the discrepancy between the training and testing data becomes larger. Thus, NLP components need to be evaluated on data from several different sources to ensure their robustness. Second, NLP components should be tested for scalability in handling a large amount of data. Lately, these components have been used for processing data

[1] DARPA: Defense Advanced Research Projects Agency.
[2] Google Translator: `translate.google.com`, Bing Translator: `translator.bing.com`
[3] The term "heterogeneous data" is used to indicate a mixture of data collected from several different sources.

that are not fixed but growing indefinitely. The scope of this data type can be as large as the entire content of the web. Thus, NLP components need to be evaluated for speed and complexity to ensure their scalability.

The objective of this thesis is to answer the following research question: "how can NLP components be optimized for better robustness and scalability". Three NLP components are used in experiments to answer this question: a part-of-speech tagger, a dependency parser, and a semantic role labeler. A part-of-speech tagger assigns lexical categories to word tokens, a dependency parser finds dependency relations between word pairs, and a semantic role labeler finds semantic arguments of predicates. Since there are many NLP tasks highly dependent on output generated by these components (e.g., information retrieval, machine translation, question-answering), it is important to optimize them for robustness as well as scalability so that they can provide reliable information without becoming bottlenecks for other tasks.

Three goals are to be accomplished in this research. First, to prepare gold-standard data from several different sources, which can be used to evaluate system performance for both in-genre and out-of-genre experiments. This step involves automatic generation of dependency trees with semantic roles. Second, to develop a part-of-speech tagger, a dependency parser, and a semantic role labeler showing robust results across this data. This step involves dynamic model selection for part-of-speech tagging, feature bootstrapping for dependency parsing, and higher-order argument pruning for semantic role labeling. Third, to reduce the expected running time of these components while retaining good accuracy. This step involves optimization of tagging, parsing, and labeling algorithms.

Although experiments are done in English, all three components use statistical, data-driven learning approaches so they can be easily ported to other languages with some feature engineering. Moreover, they already have been used for several projects and are publicly available as an open source project called ClearNLP (`clearnlp.googlecode.com`). We believe this research helps make NLP components available for more applications and brings these components closer to a greater number of users.

1.2 Background

1.2.1 Part-of-speech tagging

Part-of-speech tagging is the task of assigning a lexical category known as a "part-of-speech" (e.g., noun, verb) to a word. Part-of-speech tagging is useful for two reasons. First, it can disambiguate meanings of words belonging to multiple lexical categories. For example, a word *spring* belongs to two lexical categories, noun and verb, and means either *the season of growth* or *to move forward by leaps and bounds* when used as a noun or a verb, respectively.[4] Thus, the meaning of *spring* can only be disambiguated by identifying its part-of-speech in the context of the current sentence. Second, part-of-speech tagging provides back-off features of lexical items for NLP components. Lexical items are perhaps the most important features for many NLP components; however, they are often biased to the source data. Using only lexical items as features can be problematic when NLP components, trained on this particular source data, are used to process data from different sources where lexical items are very different from ones in the training data. Part-of-speech tags group lexical items into certain grammatical categories and provide more generalized features to NLP components, which improves robustness in handling new data.

Part-of-speech tagging is performed as a preprocessing step to many NLP tasks such as named entity recognition, NP chunking, dependency parsing, etc. It has been used widely for various applications and is considered one of the most useful NLP techniques in existence.[5]

1.2.2 Dependency parsing

Dependency parsing is the task of finding a dependency structure for an input sentence. A dependency refers to either a syntactic or semantic relation between a pair of word tokens. There are syntactic dependencies like subject or object and semantic dependencies like locative or temporal (see Appendix D for more details about dependency labels). Dependency parsing is often compared to constituent parsing, which groups word tokens into constituents and groups these constituents

[4] The definitions of *spring* are captured from WordNet (`http://wordnet.princeton.edu`).
[5] See Table A.2 for a list of Penn part-of-speech tags.

again into bigger constituents (e.g., phrases, clauses); thus, the final output becomes a syntactic tree (a constituent structure). Figure 1.1 shows an example of constituent and dependency structures. Unlike a constituent structure, also known as a phrase structure, there is no phrasal node in a dependency structure: each node in a dependency structure represents a word token, and each node is dependent on exactly one other node except for the root (e.g., *bought* in Figure 1.1).

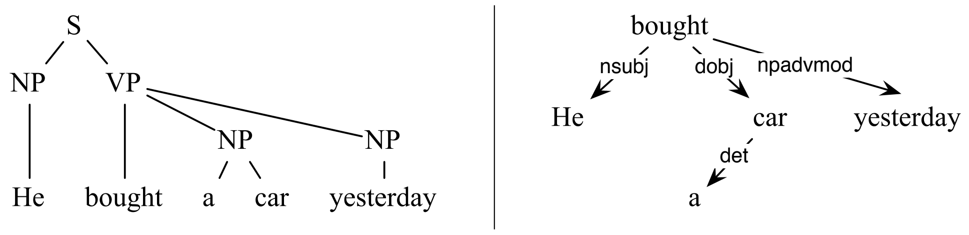


Figure 1.1: An example of constituent (left) and dependency (right) structures. Tables A.3 and 2.3 show more details about these phrase level tags and dependency labels, respectively.

Dependency structures have several advantages over phrase structures. First, phrase structures are more affected by word order in that different phrase structure rules need to be generated for phrases with the same meaning but a different word order. Dependency structures are less affected by word order, which makes them more suitable for representing flexible word order languages (e.g., Bulgarian, Czech, Hindi, Korean). Second, many dependencies are applicable cross-linguistically such that a similar set of dependencies can be applied to multiple languages with minor modifications. This language independent aspect was one of the main reasons for CoNLL to choose dependency parsing as a multilingual parsing task (Buchholz and Marsi, 2006; Nivre et al., 2007; Hajič et al., 2009). Third, automatic dependency parsing is much faster than automatic constituent parsing, which makes it more applicable for systems requiring fast processing of large-scale data.[6]

These advantages have resulted in considerable recent interest in applying dependency parsing to many NLP tasks such as machine translation (Shen et al., 2008), sentiment analysis (Councill et al., 2010), question-answering (Cui et al., 2005), etc. It is especially beneficial for those who need to deal with a vast amount of data in many different languages.

[6] Dependency parsing takes about $0.003 \sim 0.157$ seconds per sentence whereas constituent parsing takes about $0.25 \sim 0.77$ seconds per sentence (Cer et al., 2010).

1.2.3 Semantic role labeling

Semantic role labeling is the task of identifying arguments of a predicate and labeling the arguments with the semantic roles they play with respect to the predicate. In theory, a predicate can be any lexical category (e.g., verb, noun) that forms its own predicate argument structure. An argument is a participant of an event (or state) denoted by the predicate. Semantic roles, also known as thematic roles, are the underlying relations between the predicate and its arguments. In Figure 1.2, a verb predicate *open* forms an argument structure that consists of two arguments, *He* as an agent and *the door* as a theme, and two adjuncts, *with his foot* as instrumental and *at ten* as temporal (see Appendix B for more details about semantic roles).

Semantic role labeling has sparked much interest because it provides helpful information for several NLP tasks (Shen and Lapata, 2007; Liu and Gildea, 2010). For example, to answer a question like "How did he open the door?", analyzing the syntactic trees in Figure 1.2 can only give a clue to the answer (e.g., the answer may be found from one of PPs or PREP dependencies), but not the exact answer. Semantic roles can give more fine-grained information with respect to the focus of the question, in this case, the instrument of the event, *with his foot*.

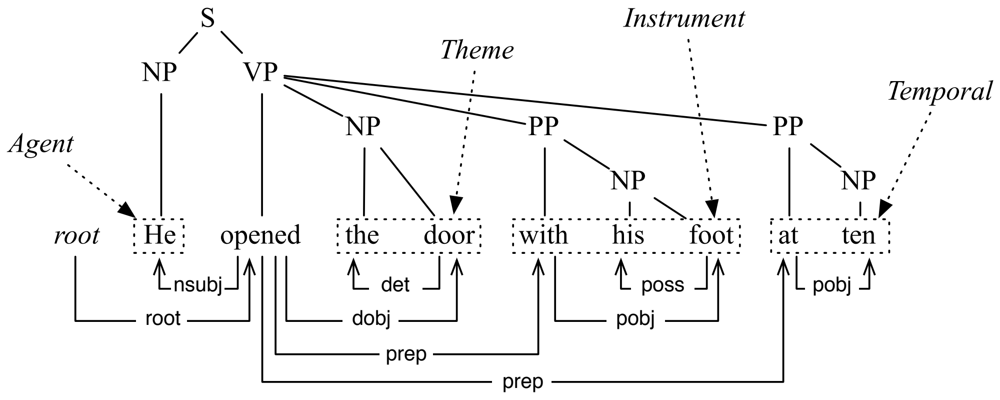


Figure 1.2: An example of constituent (top) and dependency (bottom) trees labeled with semantic roles. See Table B.2 for more details about these semantic role labels.

In the context of machine translation resources, parallel sentences from different languages, which are translations of each other, often form the same predicate argument structures even when their syntactic structures appear to be different. In Figure 1.3, the surface locations of the agent *him* and the recipient *her* are reversed between English and Chinese, but their semantic roles stay the same. This implies that predicate argument structures can provide more generalizations useful for multilingual tasks such as machine translation than syntactic structures.

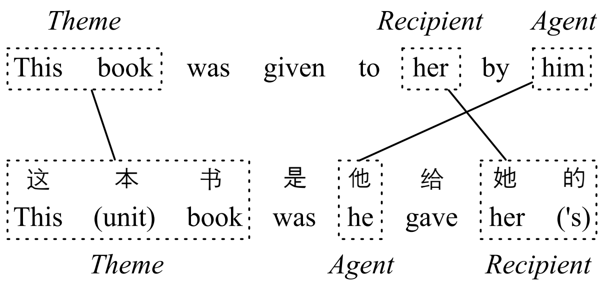

Theme
Recipient
Agent
This book was given to her by him
这 本 书 是 他 给 她 的
This (unit) book was he gave her ('s)
Theme
Agent
Recipient

Figure 1.3: An example of English and Chinese parallel sentences labeled with semantic roles. The lines show many-to-many (block) word alignments between the parallel sentences.

For our experiments, we use the PropBank style semantic role labels, which are coarse-grained on a surface level (e.g., `ARG0`, `ARG1`) but can be mapped to more fine-grained labels (e.g., agent, theme) through predefined mappings (Palmer et al., 2005).

1.3 Overall framework

Figure 1.4 shows the overall framework of this research. Prior to the development of the NLP components, manually annotated constituent trees are taken from Treebanks and automatically converted to dependency trees (Chapter 2). Semantic roles from PropBanks are added to dependency trees during this conversion (Section 6.2.3). Once the conversion is done, dependency trees with semantic roles are divided into training and evaluation sets. Given the training set, three models are built in pipelines: part-of-speech tagging (Chapter 4), dependency parsing (Chapter 5), and semantic role labeling (Chapter 6). During training, models in later pipelines are trained on automatic outputs generated by previous models. For example, a semantic role labeling model is trained on data consisting of automatic outputs from both part-of-speech tagging and dependency parsing models. Finally, all models are tested on the evaluation set for their robustness and scalability (Chapter 3).

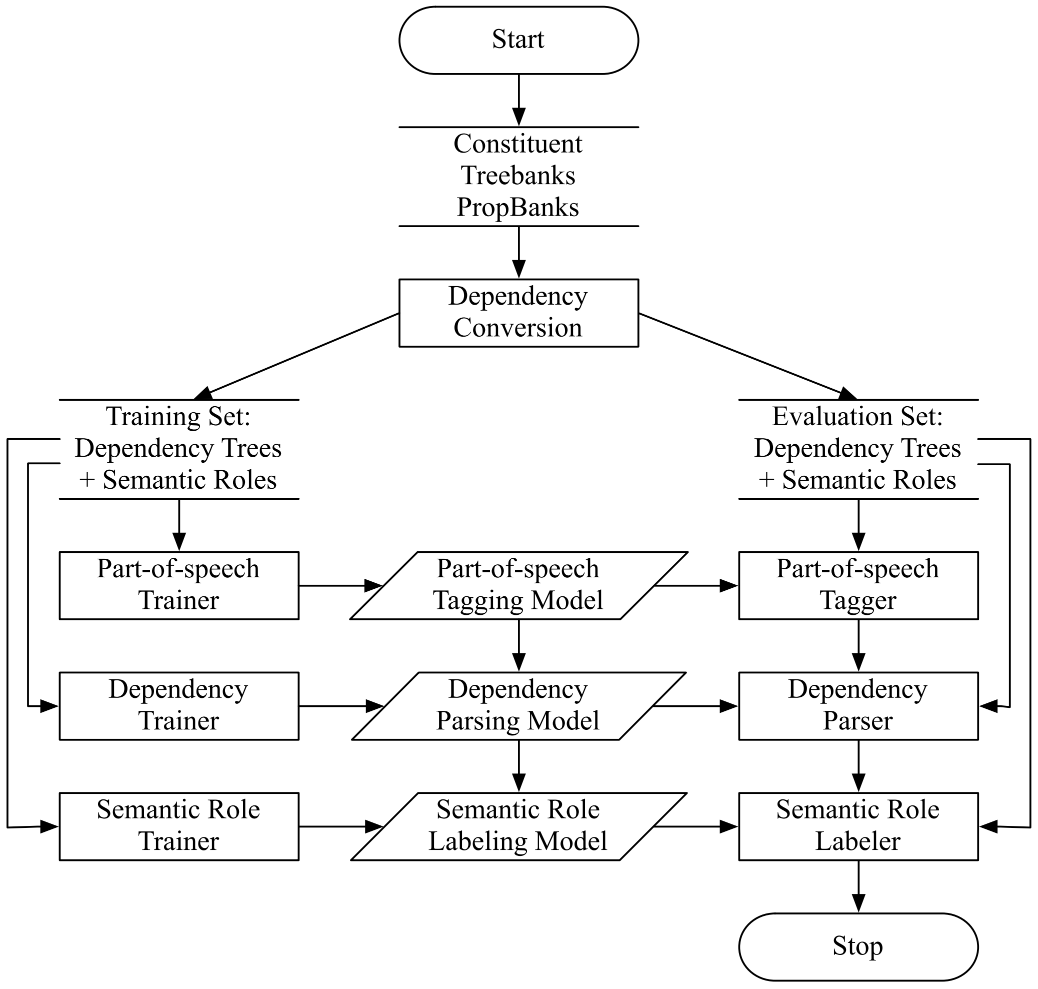


Figure 1.4: The overall framework.

1.4 Thesis statement

The goal of this thesis is to improve the robustness and scalability of three NLP components, a part-of-speech tagger, a dependency parser, and a semantic role labeler. The term ‘robustness’ is used here to indicate how well NLP components respond to test data that varies from their training data. Robustness is evaluated by measuring accuracies on both in-genre and out-of-genre data. By building a generalized model for part-of-speech tagging, bootstrapping parse information for dependency parsing, and applying higher-order argument pruning for semantic role labeling, we improve the robustness of these three NLP components. The term ‘scalability’ is used here to indicate how fast NLP components run when processing large scale data. Scalability is evaluated by measuring the expected running times and speeds.[7] By adapting dynamic model selection for part-of-speech tagging, optimizing the engineering of transition-based parsing algorithms for dependency parsing, and applying conditional higher-order argument pruning for semantic role labeling, we improve the scalability of these three components.

[7] The expected running time is how fast an algorithm is expected to run on average in terms of complexity and the speed is how fast a system actually runs on a real machine.

Chapter 2

Dependency Conversion

2.1 Introduction

2.1.1 Motivation

Most current state-of-the-art dependency parsers take various statistical learning approaches (Mcdonald and Pereira, 2006; Nivre, 2008; Huang and Sagae, 2010; Rush and Petrov, 2012). The biggest advantage of statistical parsing is found in the ability to adapt to new data without modifying the parsing algorithm. Statistical parsers can be trained on data from new domains, genres, or languages as long as they are provided with sufficiently large training data from the new sources. On the other hand, this is also the biggest drawback for statistical parsing because annotating such large training data is manually intensive work that is costly and time consuming.

Although a few manually annotated dependency Treebanks are available for English (Rambow et al., 2002; Čmejrek et al., 2004), constituent Treebanks are still more dominant (Marcus et al., 1993; Weischedel et al., 2011). It has been shown that the Penn Treebank style constituent trees can reliably be converted to dependency trees using head-finding rules and heuristics (Johansson and Nugues, 2007; de Marneffe and Manning, 2008a; Choi and Palmer, 2010b). By automatically converting these constituent trees to dependency trees, statistical dependency parsers have access to a larger amount of training data. Few tools are available for constituent to dependency conversion. Two of the most popular ones are the LTH and the Stanford dependency converters.[1] The LTH

[1] The LTH dependency converter: `http://nlp.cs.lth.se/software/treebank_converter/`
The Stanford dependency converter: `http://nlp.stanford.edu/software/stanford-dependencies.shtml`

converter had been used to provide English data for the CoNLL'07-09 shared tasks (Nivre et al., 2007; Surdeanu et al., 2008; Hajič et al., 2009). The LTH converter makes several improvements over its predecessor, Penn2Malt,[2] by adding syntactic and semantic dependencies retained from function tags (e.g., `PRD`, `TMP`) and producing long-distance dependencies caused by empty categories or gapping relations.[3] The Stanford converter was used for the SANCL'12 shared task (Petrov and McDonald, 2012), and is perhaps the most widely used dependency converter at the moment. The Stanford converter gives fine-grained dependency labels useful for many NLP tasks. Appendix C shows descriptions of the CoNLL and the Stanford dependency labels generated by these two tools.

Both converters perform well for most cases; however, they are somewhat customized to the Penn Treebank (mainly to the Wall Street Journal corpus; see Marcus et al. (1993)), so do not work as well when applied to different corpora. For example, the OntoNotes Treebank (Weischedel et al., 2011) contains additional constituent tags not used by the Penn Treebank (e.g., `EDITED`, `META`), and shows occasional departures from the Penn Treebank guidelines (e.g., inserting `NML` phrases, separating hyphenated words; see Figure 2.1). These new formats affect the ability of existing tools to find correct dependencies, motivating us to aim for a more resilient approach.

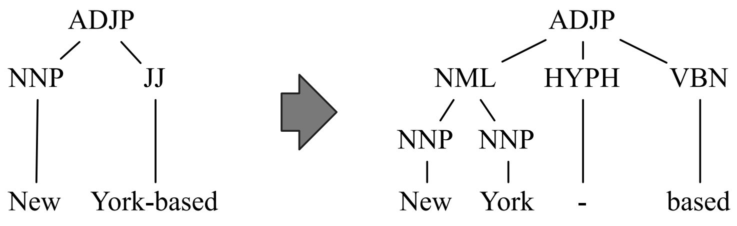


Figure 2.1: Structural differences in the Penn Treebank (left) and the OntoNotes Treebank (right). The hyphenated word is tokenized, `HYPH`, and the nominal phrase is grouped, `NML`, in the OntoNotes.

Producing more informative trees provides additional motivation. The Stanford converter generates dependency trees without using information such as function tags (Section A.1), empty categories (Section 2.2), or gapping relations (Section 2.5.1), which is provided in manually annotated but not in automatically generated constituent trees. This enables the Stanford converter to generate

[2] Penn2Malt: `http://stp.lingfil.uu.se/~nivre/research/Penn2Malt.html`
[3] The term "long-distance dependency" is used to indicate dependency relations between words that are not within the same domain of locality.

the same kind of dependencies given either manually or automatically generated constituent trees. However, it sometimes misses important details such as long-distance dependencies, which can be retrieved from empty categories, or produces unclassified dependencies that can be disambiguated by function tags. This becomes an issue when this converter is used for generating dependency trees for training because statistical parsers trained on these trees would not reflect these details.

The dependency conversion described here takes the Stanford dependency approach as the core structure and integrates the CoNLL dependency approach to add long-distance dependencies, to enrich important relations like object predicates, and to minimize unclassified dependencies. The Stanford dependency approach is taken for the core structure because it gives more fine-grained dependency labels and is currently used more widely than the CoNLL dependency approach. For our conversion, head-finding rules and heuristics are completely reanalyzed from the previous work to handle constituent tags and relations not introduced by the Penn Treebank. Our conversion has been evaluated with several different constituent Treebanks (Marcus et al., 1993; Nielsen et al., 2010; Weischedel et al., 2011; Verspoor et al., 2012) and showed robust results across these corpora.

2.1.2 Background

2.1.2.1 Dependency graph

A dependency structure can be represented as a directed graph. For a given sentence $s = w_1, \ldots, w_n$, where w_i is the i'th word token in the sentence, a dependency graph $G_s = (V_s, E_s)$ can be defined as follows:

$$\begin{aligned} V_s &= \{w_0 = \text{root}, w_1, \ldots, w_n\} \\ E_s &= \{(w_i \xrightarrow{r} w_j) : i \neq j, w_i \in V_s, w_j \in V_s - \{w_0\}, r \in R_s\} \\ R_s &= \text{A subset of all dependency relations in } s \end{aligned}$$

$w_i \xrightarrow{r} w_j$ is a directed edge from w_i to w_j with a label r, which implies that w_i is the head of w_j with a dependency relation r. A dependency graph is considered *well-formed* if it satisfied all of the following properties:

- **Root**: there must be a unique vertex, w_0, with no incoming edge.

  $\neg[\exists k.\ (w_0 \leftarrow w_k)]$

- **Single head**: each vertex $w_{i>0}$ must have at most one incoming edge.

  $\forall i.\ [i > 0 \Rightarrow \forall j.\ [(w_i \leftarrow w_j) \Rightarrow \neg[\exists k.\ (k \neq j) \wedge (w_i \leftarrow w_k)]]]$

- **Connected**: there must be an undirected path between any two vertices.[4]

  $[\forall i, j.\ (w_i - w_j)]$, where $w_i - w_j$ indicates an undirected path between w_i and w_j.

- **Acyclic**: a directed path between any two vertices must not be cyclic.

  $\neg[\exists i, j.(w_i \ {}^{*}\!\!\leftarrow w_j) \wedge (w_i \rightarrow^{*} w_j)]$, where $w_i \rightarrow^{*} w_j$ indicates a directed path from w_i to w_j.

Sometimes, *projectivity* is also considered a property of a well-formed dependency graph. When projectivity is considered, no crossing edge is allowed when all vertices are lined up in linear-order and edges are drawn above the vertices (Figure 2.2). Preserving projectivity can be useful because it enables regeneration of the original sentence from its dependency graph without losing the word order. More importantly, it reduces parsing complexity to $O(n)$ (Nivre and Scholz, 2004).

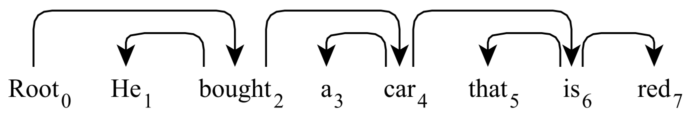


Figure 2.2: An example of a projective dependency graph.

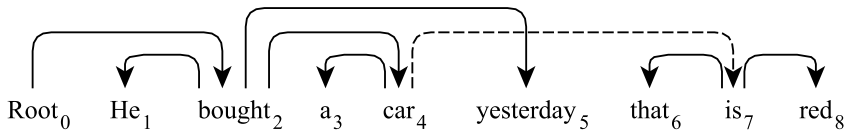


Figure 2.3: An example of a non-projective dependency graph. The dependency between *car* and *is* is non-projective because it crosses the dependency between *bought* and *yesterday*.

Although preserving projectivity has a few advantages, non-projective dependencies are often required, especially in flexible word order languages, to represent correct dependencies (Nivre, 2008). Even in rigid word order languages such as English, non-projective dependencies are necessary to

[4] An 'undirected path' implies a path between two vertices, regardless of their directionality.

represent long-distance dependencies. In Figure 2.3, there is no way of describing the dependency relations for both "*bought* → *yesterday*" and "*car* → *is*" without having their edges cross. Because of such cases, projectivity is dropped from the properties of a well-formed dependency graph for this research.

A well-formed dependency graph, with or without the projective property, satisfies all of the conditions for a tree structure, so is called a 'dependency tree'.

2.1.2.2 Types of empty categories

Empty categories are syntactic units, usually nominal phrases, that appear in the surface form to signal the canonical locations of syntactic elements in its deep structure (Cowper, 1992; Chomsky, 1995). Table 2.1 shows a list of empty categories used in constituent Treebanks for English. Some of these empty categories have overloaded meanings. For instance, `*PRO*` indicates empty subjects caused by different pro-drop cases (e.g., control, imperative, nominalization). See Bies et al. (1995); Taylor (2006) for more details about these empty categories.

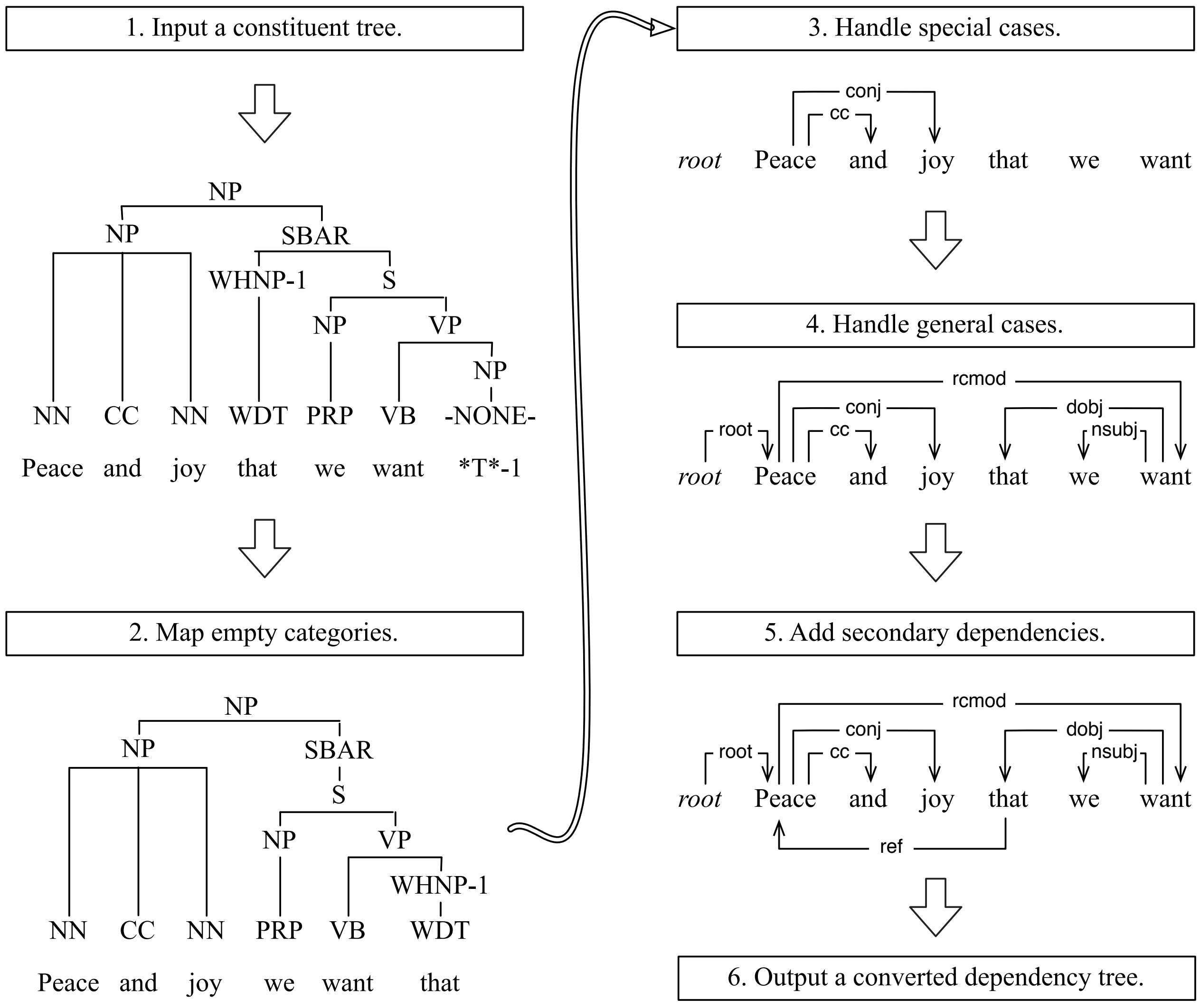
Type	**Description**
`*PRO*`	Empty subject of pro-drop (e.g., control, ECM, imperative, nominalization)
`*T*`	Trace of *wh*-movement and topicalization
`*`	Trace of subject raising and passive construction
`0`	Null complementizer
`*U*`	Unit (e.g., $)
`*ICH*`	Pseudo-attach: Interpret Constituent Here
`*?*`	Placeholder for ellipsed material
`*EXP*`	Pseudo-attach: EXPletives
`*RNR*`	Pseudo-attach: Right Node Raising
`*NOT*`	Anti-placeholder in template gapping
`*PPA*`	Pseudo-attach: Permanent Predictable Ambiguity

Table 2.1: A list of empty categories used in constituent Treebanks for English.

2.1.3 Overview

Figure 2.4 shows the overview of our constituent to dependency conversion. Given a constituent tree, empty categories are mapped to their antecedents first (step 2; see Section 2.2). This step relocates phrasal nodes regarding certain kinds of empty categories that may cause generation of non-projective dependencies.[5] Once empty categories are mapped, special cases such as apposition, coordination, or small clauses are handled next (step 3; see Sections 2.3.2 to 2.3.4). Finally, general cases are handled using head-finding rules and heuristics (step 4; see Section 2.3.1 and Section 2.4).

Secondary dependencies are added as a separate layer of this dependency tree (step 5; see Section 2.5). Additionally, syntactic and semantic function tags in the constituent tree are preserved as features of individual nodes in the dependency tree (not shown in Figure 2.4; see Section A.1).



1. Input a constituent tree.
NP
NP
SBAR
WHNP-1
S
NP
VP
NP
NN CC NN WDT PRP VB -NONE-
Peace and joy that we want *T*-1
2. Map empty categories.
NP
NP
SBAR
S
NP
VP
WHNP-1
NN CC NN PRP VB WDT
Peace and joy we want that
3. Handle special cases.
conj
cc
root Peace and joy that we want
4. Handle general cases.
rcmod
root
conj
cc
dobj
nsubj
root Peace and joy that we want
5. Add secondary dependencies.
rcmod
root
conj
cc
dobj
nsubj
root Peace and joy that we want
ref
6. Output a converted dependency tree.

Figure 2.4: The overview of constituent to dependency conversion.

[5] Although phrases in constituency trees are relocated, word order in dependency trees remains the same.

2.2 Mapping empty categories

Most long-distance dependencies can be represented without using empty categories in dependency structure. In English, long-distance dependencies are caused by certain linguistic phenomena such as *wh*-movement, topicalization, discontinuous constituents, etc. It is difficult to find long-distance dependencies during automatic parsing because they often introduce dependents that are not within the same domain of locality, resulting in non-projective dependencies (McDonald and Satta, 2007; Koo et al., 2010; Kuhlmann and Nivre, 2010).

Four types of empty categories are used to represent long-distance dependencies during our conversion: `*T*`, `*RNR*`, `*ICH*`, and `*PPA*` (see Table 2.1). Note that the CoNLL dependency approach used `*EXP*` to represent extraposed elements in expletive constructions, which is not used in our approach because the annotation of `*EXP*` is somewhat inconsistent across different corpora.

2.2.1 Wh-movement

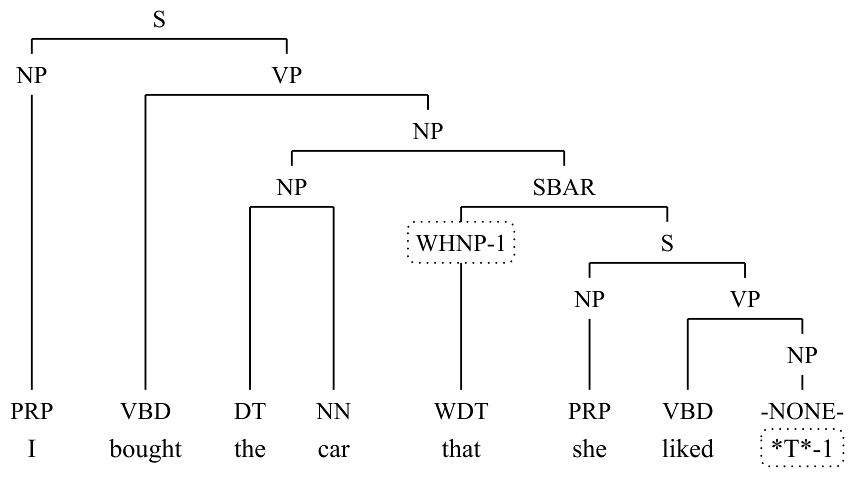


Figure 2.5: An example of *wh*-movement.

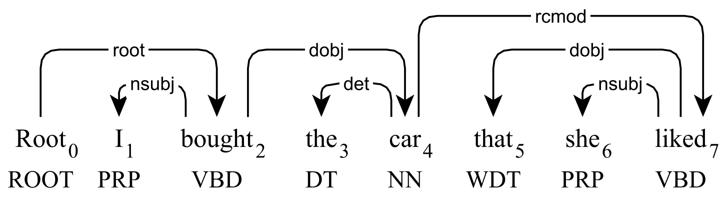


Figure 2.6: A dependency tree converted from the constituent tree in Figure 2.5

Wh-movement is represented by `*T*` in constituent trees. In Figure 2.5, `WHNP-1` is moved from the object position of the subordinate verb *liked* and leaves a trace, `*T*-1`, at its original position. Figure 2.6 shows a dependency tree converted from the constituent tree in Figure 2.5. The dependency of `WHNP-1` is derived from its original position so that it becomes a direct object of *liked* (`DOBJ`; Section D.2.2).

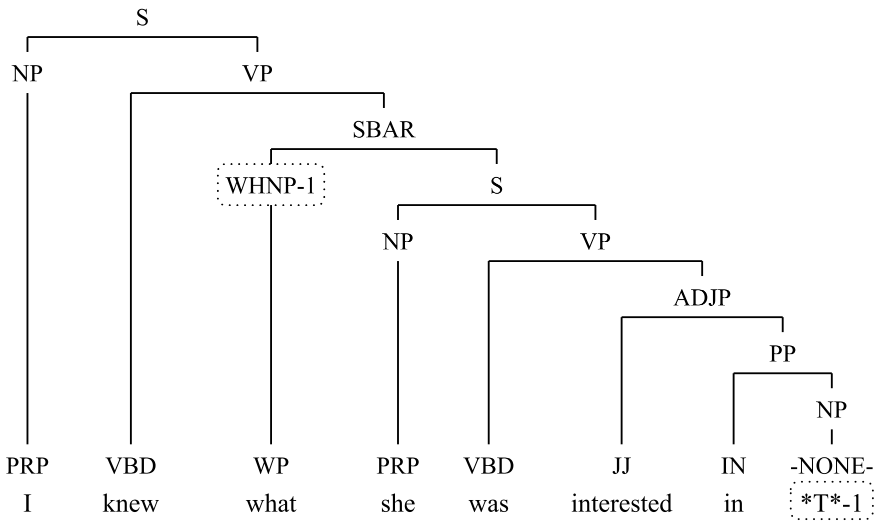


Figure 2.7: Another example of *wh*-movement.

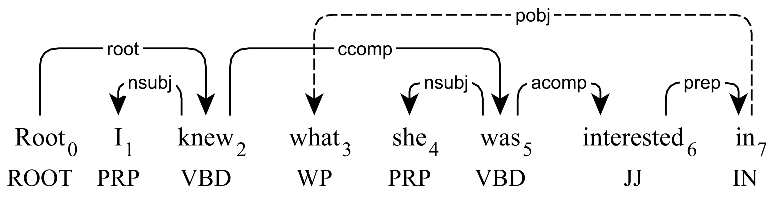


Figure 2.8: A dependency tree converted from the constituent tree in Figure 2.7. The dependency derived from the *wh*-movement, `POBJ`, is indicated by a dotted line.

Wh-complementizers can be moved from several positions. In Figure 2.7, `WHNP-1` is moved from the prepositional phrase, `PP`, so in Figure 2.8, the complementizer *what* becomes an object of the preposition *in* (`POBJ`; Section D.8.2). Notice that the `POBJ` dependency is non-projective; it crosses the dependency between *knew* and *was*. This is a typical case of a non-projective dependency caused by *wh*-movement.

2.2.2 Topicalization

Topicalization is also represented by `*T*`. In Figure 2.9, `S-1` is moved from the subordinate clause, `SBAR`, and leaves a trace behind. In Figure 2.10, the head of `S-1`, *liked*, becomes a dependent of the matrix verb *seemed* (`ADVCL`; Section D.5.1), and the preposition *like* becomes a dependent of the subordinate verb *liked* (`MARK`; Section D.5.3). The `MARK` dependency is non-projective such that it crosses the dependency between *Root* and *seemed*.

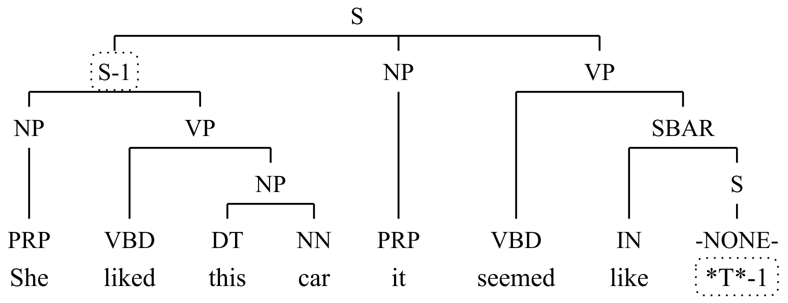


Figure 2.9: An example of topicalization.

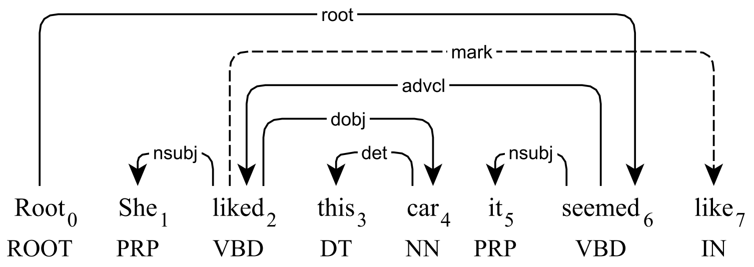


Figure 2.10: A dependency tree converted from the constituent tree in Figure 2.9. The dependency derived from the topicalization, `MARK`, is indicated by a dotted line.

There are a few cases where `*T*` mapping causes cyclic dependency relations. In Figure 2.11, `*T*-1` is mapped to `S-1` that is an ancestor of itself. Thus, the head of `S-1`, *bought*, becomes a dependent of the subordinate verb *said* while the head of the subordinate clause, *said*, becomes a dependent of the matrix verb *bought*. Since this creates a cyclic relation in the dependency tree, such traces are ignored during our conversion (Figure 2.12).

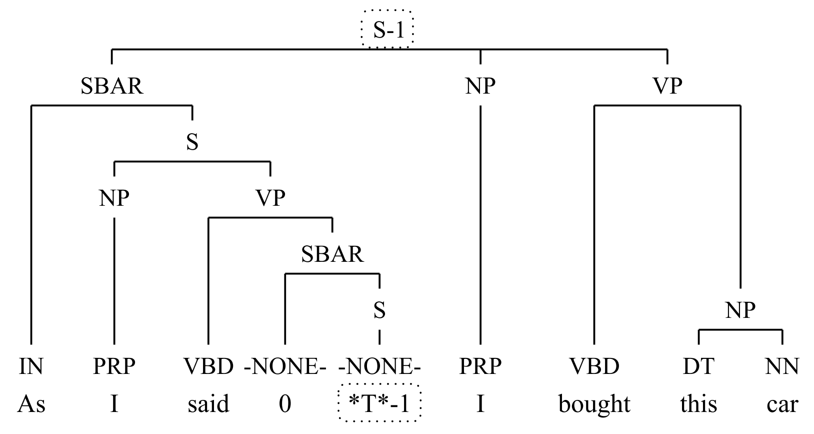

S-1
SBAR
NP
VP
S
NP
VP
SBAR
S
NP
IN PRP VBD -NONE- -NONE- PRP VBD DT NN
As I said 0 *T*-1 I bought this car

Figure 2.11: An example of topicalization, where a topic movement creates a cyclic relation.

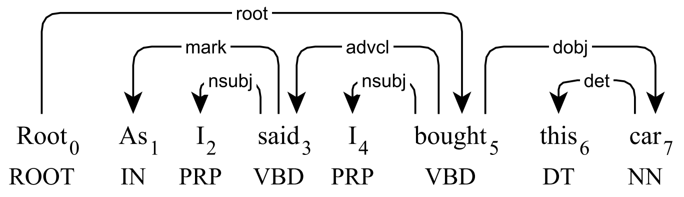

root
mark
nsubj
advcl
nsubj
dobj
det
Root$_0$ As$_1$ I$_2$ said$_3$ I$_4$ bought$_5$ this$_6$ car$_7$
ROOT IN PRP VBD PRP VBD DT NN

Figure 2.12: A dependency tree converted from the constituent tree in Figure 2.11.

2.2.3 Right node raising

Right node raising occurs in coordination where a constituent is governed by multiple parents that are not on the same level (Levine, 1985). Right node raising is represented by `*RNR*` in constituent trees. In Figure 2.13, `NP`-1 should be governed by both `PP`-1 and `PP`-2, where `*RNR*`-1's are located. Making `NP`-1 dependents of both `PP`-1 and `PP`-2 breaks the **single head** property (Section 2.1.2.1); instead, the dependency of `NP`-1 is derived from its closest `*RNR*`-1 in our conversion. In Figure 2.14, *her* becomes a dependent of the head of `PP`-2, *in*. The dependency between *her* and the head of `PP`-1, *for*, is preserved as a secondary dependency, `REF` (referent; see Section 2.5). Thus, *her* is a dependent of only `PP`-2 in our dependency tree while the dependency to `PP`-2 can still be retrieved through the secondary dependency.[6]

[6] Secondary dependencies are not commonly used in dependency structures. These are dependencies derived from gapping relations, referent relations, right node raising, and open clausal subjects, which may break tree properties (Section 2.5). During our conversion, secondary dependencies are preserved in a separate layer so they can be learned either jointly or separately from other dependencies.

Note that the CoNLL dependency approach makes *her* a dependent of the head of PP-1, which creates a non-projective dependency (the dependency between *for* and *her* in Figure 2.15). This non-projective dependency is avoided in our approach without losing any referential information.

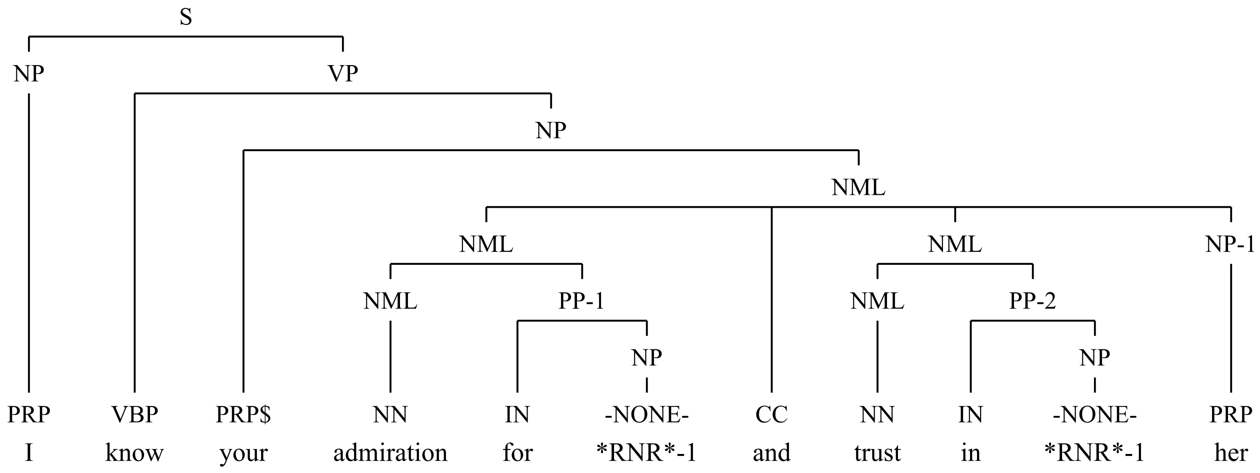

S
NP
VP
NP
NML
NML
NML
NP-1
NML
PP-1
NML
PP-2
NP
NP
PRP VBP PRP$ NN IN -NONE- CC NN IN -NONE- PRP
I know your admiration for *RNR*-1 and trust in *RNR*-1 her

Figure 2.13: An example of right node raising.

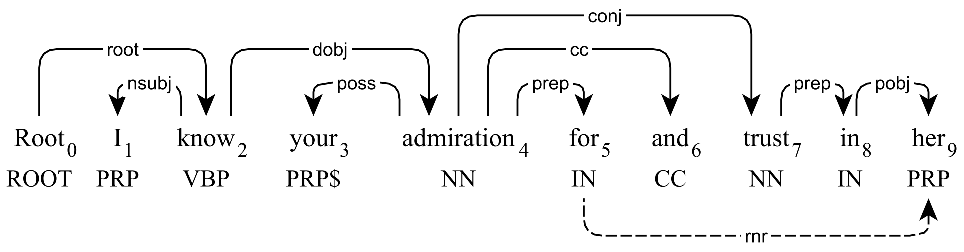

conj
root
dobj
cc
nsubj
poss
prep
prep
pobj
Root0 I1 know2 your3 admiration4 for5 and6 trust7 in8 her9
ROOT PRP VBP PRP$ NN IN CC NN IN PRP
rnr

Figure 2.14: A dependency tree converted from the constituent tree in Figure 2.13. The secondary dependency, RNR, is added to a separate layer to preserve tree properties.

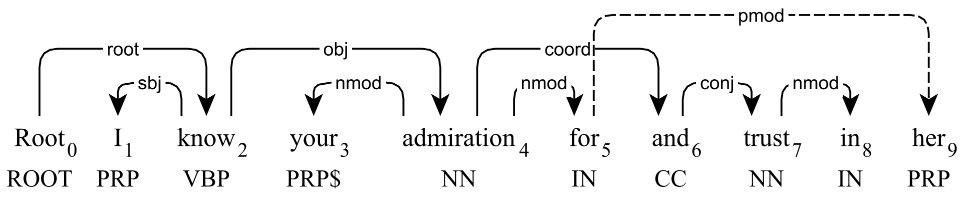

pmod
root
obj
coord
sbj
nmod
nmod
conj
nmod
Root0 I1 know2 your3 admiration4 for5 and6 trust7 in8 her9
ROOT PRP VBP PRP$ NN IN CC NN IN PRP

Figure 2.15: A CoNLL style dependency tree converted from the constituent tree in Figure 2.13. The dependency caused by right node raising, PMOD, is indicated by a dotted line.

2.2.4 Discontinuous constituent

A discontinuous constituent is a constituent that is separated from its original position by some intervening material. The original position of a discontinuous constituent is indicated by `*ICH*` in constituent trees. In Figure 2.16, `PP`-1 is separated from its original position, `*ICH*`-1, by the adverb phrase, `ADVP`. Thus, in Figure 2.17, the head of the prepositional phrase, *than*, becomes a prepositional modifier (`PREP`; Section D.8.3) of the head of the adjective phrase (`ADJP`-2), *expensive*. The `PREP` dependency is non-projective; it crosses the dependency between *is* and *now*.

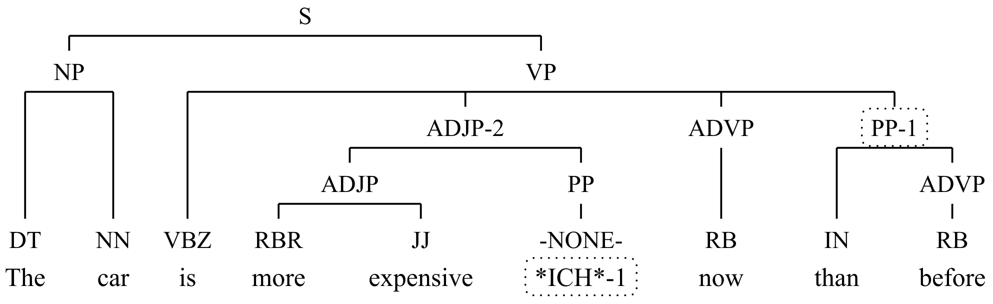


Figure 2.16: An example of discontinuous constituents.

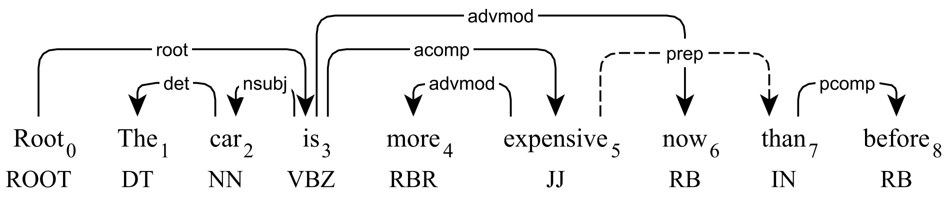


Figure 2.17: A dependency tree converted from the constituent tree in Figure 2.16. The dependency derived from the `*ICH*` movement, `PREP`, is indicated by a dotted line.

2.3 Finding dependency heads

2.3.1 Head-finding rules

ADJP	r	JJ*|VB*|NN*;ADJP;IN;RB|ADVP;CD|QP;FW|NP;*
ADVP	r	VB*;RP;RB*|JJ*;ADJP;ADVP;QP;IN;NN;CD;NP;*
CONJP	l	CC;VB*;NN*;TO|IN;*
EDITED	r	VP;VB*;NN*|PRP|NP;IN|PP;S*;*
EMBED	r	S*;FRAG|NP;*
FRAG	r	VP;VB*;-PRD;S|SQ|SINV|SBARQ;NN*|NP;PP;SBAR;JJ*|ADJP;RB|ADVP;INTJ;*
INTJ	l	VB*;NN*;UH;INTJ;*
LST	l	LS|CD;NN;*
META	l	NP;VP|S;*
NAC	r	NN*;NP;S|SINV;*
NML	r	NN*|NML;CD|NP|QP|JJ*|VB*;*
NP	r	NN*|NML;NX;PRP;FW;CD;NP;-NOM;QP|JJ*|VB*;ADJP;S;SBAR;*
NX	r	NN*;NX;NP;*
PP	l	RP;TO;IN;VB*;PP;NN*;JJ;RB;*
PRN	r	VP;NP;S|SBARQ|SINV|SQ;SBAR;*
PRT	l	RP;PRT;*
QP	r	CD;NN*;JJ;DT|PDT;RB;NP|QP;*
RRC	l	VP;VB*;-PRD;NP|NN*;ADJP;PP;*
S	r	VP;VB*;-PRD;S|SQ|SINV|SBARQ;SBAR;NP;PP;*
SBAR	r	VP;S|SQ|SINV;SBAR*;FRAG|NP;*
SBARQ	r	VP;SQ|SBARQ;S|SINV;FRAG|NP;*
SINV	r	VP;VB*;MD;S|SINV;NP;*
SQ	r	VP;VB*;SQ;S;MD;NP;*
UCP	l	*
VP	l	VP;VB*;MD|TO;JJ*|NN*|IN;-PRD;NP;ADJP|QP;S;*
WHADJP	r	JJ*|VBN;WHADJP|ADJP;*
WHADVP	r	RB*|WRB;WHADVP;*
WHNP	r	NN*;WP|WHNP;NP|NML|CD;JJ*|VBG;WHADJP|ADJP;DT;*
WHPP	l	IN|TO;*
X	r	*

Table 2.2: Head-finding rules. `l`/`r` implies the search direction for the leftmost/rightmost constituent. `*`/`+` implies 0/1 or more characters and `-TAG` implies any POS tag with the specific function tag. `|` implies a logical OR and `;` is a delimiter between POS tags. Each rule gives higher precedence to the left (e.g., `VP` takes the highest precedence in `S`).

Table 2.2 shows head-finding rules (henceforth, headrules) derived from various constituent Treebanks. For each phrase (or clause) in a constituent tree, the head of the phrase is found by using its headrules, and all other nodes in the phrase become dependents of the head. This procedure goes on recursively until every constituent in the tree becomes a dependent of one other constituent, except for the top constituent, which becomes the root of the dependency tree. A dependency tree generated by this procedure is guaranteed to be well-formed (Section 2.1.2.1), and may or may not be non-projective, depending on how empty categories are mapped (Section 2.2).

Notice that the headrules in Table 2.2 give information about which constituents can be the heads, but do not show which constituents cannot be the heads. Some constituents are more likely to be dependents than heads. In Figure 2.18, both *Three times* and *a week* are noun phrases under another noun phrase. According to our headrules, the rightmost noun phrase, `NP-TMP`, is chosen to be the head of this phrase. However, `NP-TMP` is actually an adverbial modifier of `NP-H` (`NPADVMOD`; Section D.5.5); thus, `NP-H` should be the head of this phrase instead. This indicates that extra information is required to retrieve correct heads for this kind of phrases.

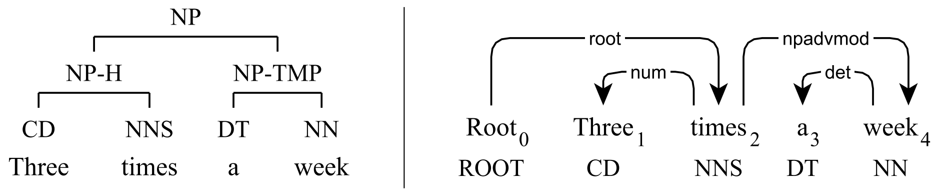


Figure 2.18: An example of a noun phrase modifying another noun phrase.

Algorithm 2.1 : *getHead*(N, R)

Input: An ordered list N of constituent nodes that are siblings,
The headrules R of the parent of nodes in N.
Output: The head constituent of N with respect to R.
All other nodes in N become dependents of the head.

```
1: if the search direction of $R$ is $r$ then $N$.reverse()        # the 2nd column in Table 2.2
2: for $flag$ in $\{0 \dots 3\}$ do
3:   for $tags$ in $R$ do                                         # e.g,. $tags \leftarrow$ NN*|NML
4:     for $node$ in $N$ do
5:       if ($flag = getHeadFlag(node)$) and ($node$ is $tags$) then
6:         $head \leftarrow node$
7:         break the highest for-loop
8: for $node$ in $N$ do
9:   if $node \neq head$ then
10:    $node$.head $\leftarrow head$
11:    $node$.label $\leftarrow getDependencyLabel(node, node.\text{parent}, head)$     # Section 2.4.2
12: return $head$
```

The *getHead*(N, R) method in Algorithm 2.1 finds the head of a phrase (lines 2-7) and makes all other constituents in the phrase dependents of the head (lines 8-11). The input to the method is the ordered list of children N and the corresponding headrules R of the phrase. The *getHeadFlag*(C) method in Algorithm 2.2 returns the head-flag of a constituent C, which indicates the dependency precedence of C: the lower the flag is, the sooner C can be chosen as the head. For example, `NP-TMP` in Figure 2.18 is skipped during the first iteration (line 2 in Algorithm 2.1) because it has the adverbial function tag `TMP`, so gets a flag of 1 (line 1 in Algorithm 2.2). Alternatively, `NP-H` is not skipped because it gets a flag of 0. Thus, `NP-H` becomes the head of this phrase.

Algorithm 2.2 : *getHeadFlag*(C)

```
 Input:  A constituent $C$.
Output:  The head-flag of $C$, that is either 0, 1, 2, or 3.
1: if hasAdverbialTag($C$) return 1                                # Section D.5
2: if isMetaModifier($C$) return 2                                 # Section D.10.4
3: if ($C$ is an empty category) or isPunctuation($C$) return 3    # Section D.10.8
4: return 0
```

The following sections describe heuristics to resolve special cases such as apposition, coordination, and small clauses, where correct heads cannot always be retrieved by headrules alone.

2.3.2 Apposition

Apposition is a grammatical construction where multiple noun phrases are placed side-by-side and later noun phrases give additional information about the first noun phrase. For example, in a phrase "*John, my brother*", both *John* and *my brother* are noun phrases such that *my brother* gives additional information about its preceding noun phrase, *John*. The *findApposition*(C) method in Algorithm 2.3 makes each appositional modifier a dependent of the first noun phrase in a phrase (lines 8-9). An appositional modifier is either a noun phrase without an adverbial function tag (line 5), any phrase with the function tag `HLN|TTL` (headlines or titles; line 6), or a reduced relative clause containing a noun phrase with the function tag `PRD` (non-`VP` predicate; line 7).

Algorithm 2.3 : *findApposition*(C)

```
   Input:  A constituent $C$.
  Output:  True if $C$ contains apposition; otherwise, False.
 1: if ($C$ is not NP|NML) or ($C$ contains NN*) or ($C$ contains no NP) return False
 2: let $f$ be the first NP|NML in $C$ that contains no POS   # skip possession modifier
 3: $b \leftarrow$ False
 4: for $s$ in all children of $C$ preceded by $f$ do
 5:   if (($s$ is NML|NP) and (not hasAdverbialTag($s$)))     # Section D.5
 6:       or ($s$ has HLN|TTL)
 7:       or (($s$ is RRC) and ($s$ contains NP-PRD)) then
 8:     $s$.head $\leftarrow f$
 9:     $s$.label $\leftarrow$ APPOS
10:     $b \leftarrow$ True
11: return $b$
```

2.3.3 Coordination

Several approaches have been proposed for coordination representation in dependency structure. The Stanford dependency approach makes the leftmost conjunct the head of all other conjuncts and conjunctions. The Prague dependency approach makes the rightmost conjunction the head of all conjuncts and conjunctions (Čmejrek et al., 2004). The CoNLL dependency approach makes each preceding conjunct or conjunction the head of its following conjunct or conjunction.

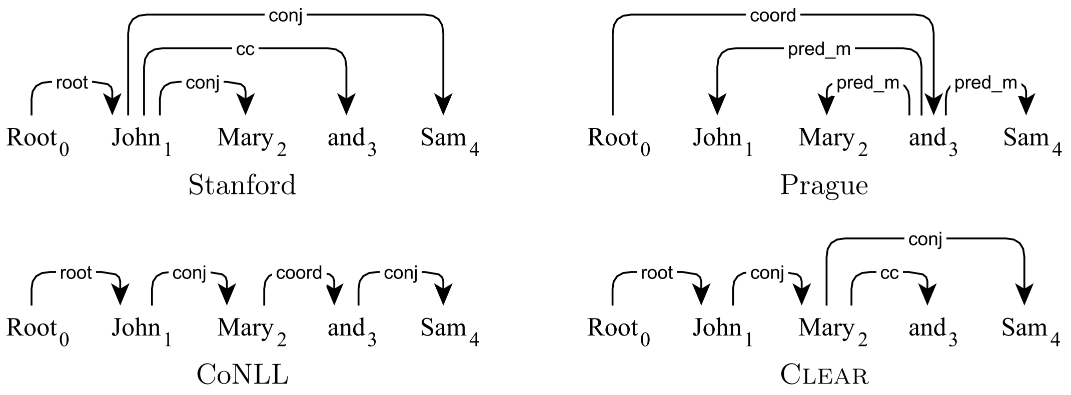

conj
cc
root
conj
Root0 John1 Mary2 and3 Sam4
Stanford
coord
pred_m
pred_m
pred_m
Root0 John1 Mary2 and3 Sam4
Prague
root
conj
coord
conj
Root0 John1 Mary2 and3 Sam4
CoNLL
conj
root
conj
cc
Root0 John1 Mary2 and3 Sam4
Clear

Figure 2.19: Different ways of representing coordination in dependency structure.

Our conversion takes an approach similar to the CoNLL dependency approach, which had been shown to work better for transition-based dependency parsing (Nilsson et al., 2006). There is one small change in our approach such that conjunctions do not become the heads of conjuncts (Clear in Figure 2.19). This way, conjuncts are always dependents of their preceding conjuncts whether or not conjunctions exist in between.

Algorithm 2.4 : *getCoordinationHead*(C, R)

```
   Input:  A constituent $C$ and the headrule $R$ of $C$.
  Output:  The head of the leftmost conjunct in $C$ if exists; otherwise, null.

 1: if not $containsCoordination(C)$ return null
 2: $p \leftarrow getConjunctHeadPattern(C)$
 3: $pHead \leftarrow$ null                                        # previous conjunct head
 4: $isPatternFound \leftarrow$ False
 5: let $f$ be the first child of $C$
 6: for $c$ in all children of $C$ do
 7:   if $isCoordinatingConjunction(c)$ or ($c$ is ,|:) then       # Section D.6.2
 8:     if $isPatternFound$ then
 9:       let $S$ be a sub-span of $C$ from $f$ to $c$ (exclusive)
10:       $pHead \leftarrow getConjunctHead(S, R, pHead)$
11:       $c$.head $\leftarrow pHead$
12:       $c$.label $\leftarrow getDependencyLabel(c, C, pHead)$      # Section 2.4.2
13:       $isPatternfound \leftarrow$ False
14:       let $f$ be the next sibling of $c$ in $C$
15:     elif $pHead \neq$ null then
16:       let $S$ be a sub-span of $C$ from $f$ to $c$ (inclusive)
17:       for $s$ in $S$ do
18:         $s$.head $\leftarrow pHead$
19:         $s$.label $\leftarrow getDependencyLabel(s, C, pHead)$    # Section 2.4.2
20:       let $f$ be the next sibling of $c$ in $C$
21:   elif $isConjunctHead(c, C, p)$ then $isPatternFound \leftarrow$ True   # a conjunct is found
22: if $pHead =$ null return null                                   # no conjunct is found
23: let $S$ be a sub-span of $C$ from $f$ to $c$ (inclusive)
24: if $S$ is not empty then $getConjunctHead(S, R, pHead)$
25: return the head of the leftmost conjunct
```

The *getCoordinationHead*(C) method in Algorithm 2.4 finds dependencies between conjuncts and returns the head of the leftmost conjunct in C. The algorithm begins by checking if C is coordinated (line 1). For each constituent in C, the algorithm checks if it matches the conjunct head pattern of C (line 21), which varies by C's phrase type. For instance, only a non-auxiliary verb or a verb phrase can be a conjunct head in a verb phrase (see *getConjunctHeadPattern*(C) in Algorithm 2.6). When a coordinator (a conjunction, comma, or colon) is encountered, a sub-span is formed (line 9). If the span includes at least one constituent matching the conjunct head pattern, it is considered a new conjunct and the head of the conjunct is retrieved by the headrule of C (line 10). The head of the current conjunct becomes a dependent of the head of its preceding conjunct if it exists (see *getConjunctHead*$(S, R, pHead)$ in Algorithm 2.8). If there is no constituent matching the pattern, all constituents within the span become dependents of the head of the previous conjunct if it exists (lines 16-19). This procedure goes on iteratively until all constituents in C are encountered. Note that the *getCoordinationHead*(C, R) method is called before the *findApposition*(C) method in Algorithm 2.3; thus, a constituent can be a conjunct or an appositional modifier, but not both.

The *containsCoordination*(C) method in Algorithm 2.5 decides whether a constituent C is coordinated. C is coordinated if it is an unlike coordinated phrase (line 1), a noun phrase containing a constituent with the function tag `ETC` as the rightmost child (line 2-4), or contains a conjunction followed by a conjunct (lines 5-9).

Algorithm 2.5 : *containsCoordination*(C)

```
 Input:   Constituent $C$.
Output:   True if $C$ contains coordination; otherwise, False.
1: if $C$ is UCP return True                                    # unlike coordinated phrase
2: if ($C$ is NML|NP) and ($C$ contains -ETC) then              # et cetera (etc.)
3:   let $e$ be a child of $N$ with -ETC
4:   if $e$ is the rightmost element besides punctuation return True
5: for $f$ in all children of $C$ do                            # skip pre-conjunctions
6:   if not (isCoordinatingConjunction($f$) or isPunctuation($f$)) then   # App. D.6.2, D.10.8
7:     break
8: let $N$ be all children of $C$ preceded by $f$
9: return $N$ contains CC|CONJP
```

The *getConjunctHeadPattern*(C) method in Algorithm 2.6 returns a pattern that matches potential conjunct heads of C. In theory, a verb phrase should contain at least one non-auxiliary verb or a verb phrase that matches the pattern (`VP|VB`$^\flat$ in line 9); however, this is not always true in practice (e.g., `VP`-ellipsis, randomly omitted verbs in web-texts). Moreover, phrases such as unlike coordinated phrases, quantifier phrases, or fragments do not always show clear conjunct head patterns. The default pattern of `*` is used for these cases, indicating that any constituent can be the potential head of a conjunct in these phrases.

Algorithm 2.6 : *getConjunctHeadPattern*(C)

Input: A constituent C.

Output: The conjunct head pattern of C if exists; otherwise, the default pattern, `*`.
If C contains no child satisfying the pattern, returns the default pattern, `*`.
`VB`$^\flat$ implies a non-auxiliary verb (Section D.3).
`S`$^\flat$ implies a clause without an adverbial function tag (Section D.5).

```
1: if   $C$ is ADJP then $p \leftarrow$ ADJP|JJ*|VBN|VBG
2: elif $C$ is ADVP then $p \leftarrow$ ADVP|RB*
3: elif $C$ is INTJ then $p \leftarrow$ INTJ|UH
4: elif $C$ is PP then $p \leftarrow$ PP|IN|VBG
5: elif $C$ is PRT then $p \leftarrow$ PRT|RP
6: elif $C$ is NML|NP then $p \leftarrow$ NP|NML|NN*|PRP|-NOM
7: elif $C$ is NAC then $p \leftarrow$ NP
8: elif $C$ is NX then $p \leftarrow$ NX
9: elif $C$ is VP then $p \leftarrow$ VP|VB$^\flat$
10: elif $C$ is S then $p \leftarrow$ S$^\flat$|SINV|SQ|SBARQ
11: elif $C$ is SQ then $p \leftarrow$ S$^\flat$|SQ|SBARQ
12: elif $C$ is SINV then $p \leftarrow$ S$^\flat$|SINV
13: elif $C$ is SBAR* then $p \leftarrow$ SBAR*
14: elif $C$ is WHNP then $p \leftarrow$ NN*|WP
15: elif $C$ is WHADJP then $p \leftarrow$ JJ*|VBN|VBG
16: elif $C$ is WHADVP then $p \leftarrow$ RB*|WRB|IN
17: if ($p$ is not found) or ($C$ contains no $p$) return *
18: return $p$
```

A pattern p retrieved by the *getConjunctHeadPattern*(C) method in Algorithm 2.6 is used in the *isConjunctHead*(C, P, p) method in Algorithm 2.7 to decide whether a constituent C is a potential conjunct head of its parent P. No subordinating conjunction is considered a conjunct head in a subordinate clause (line 1); this rule is added to prevent a complementizer such as *whether* from being the head of a clause starting with expressions like *whether or not*. When the default pattern is used, the method accepts any constituent except for a few special cases (lines 3-7). The method returns `True` if C matches p (line 9).

Algorithm 2.7 : *isConjunctHead*(C, P, p)

```
  Input:  Constituents $C$ and $P$, where $P$ is the parent of $C$,
          and the conjunct head pattern $p$ of $P$.
 Output:  True if $C$ matches the conjunct head pattern; otherwise, False.
 1: if ($P$ is SBAR) and ($C$ is ID|DT) return False          # Section D.5.3
 2: if $p$ is * then                                           # the default pattern
 3:   if isPunctuation($C$) return False                     # Section D.10.8
 4:   if isInterjection($C$) return False                    # Section D.10.3
 5:   if isMetaModifier($C$) return False                    # Section D.10.4
 6:   if isParentheticalModifier($C$) return False           # Section D.10.5
 7:   if isAdverbialModifier($C$) return False               # Section D.5.2
 8:   return True
 9: if $C$ is $p$ return True
10: return False
```

Finally, the *getConjunctHead*$(S, R, pHead)$ method in Algorithm 2.8 finds the head of a conjunct S and makes this head a dependent of its preceding conjunct head, *pHead*. The head of S is found by the *getHead*(N, R) method in Algorithm 2.1 where R is the headrule of S's parent. The dependency label `CONJ` is assigned to this head except for the special cases of interjections and punctuation (lines 4-6).

Algorithm 2.8 The *getConjunctHead*$(S, R,$ *pHead*$)$ method.

```
  Input:  A constituent $C$, a sub-span $S$ of $C$, the headrule $R$ of $C$, and the previous conjunct
          head pHead in $C$.
 Output:  The head of $S$. All other nodes in $S$ become dependents of the head.
 1: cHead $\leftarrow$ getHead($S, R$)                                  # Section 2.3.1
 2: if pHead $\neq$ null then
 3:   cHead.head $\leftarrow$ pHead
 4:   if   isInterjection($C$) then cHead.label $\leftarrow$ INTJ      # Section D.10.3
 5:   elif isPunctuation($C$) then cHead.label $\leftarrow$ PUNCT      # Section D.10.8
 6:   else cHead.label $\leftarrow$ CONJ                              # Section D.6.1
 7: return cHead
```

2.3.4 Small clauses

Small clauses are represented as declarative clauses without verb phrases in constituent trees. Small clauses may not contain internal subjects. In Figure 2.20, both S-1 and S-2 are small clauses but S-1 contains an internal subject, *me*, whereas the subject of S-2 is controlled externally. This distinction is made because S-1 can be rewritten as a subordinate clause such as "*I am her friend*" whereas such a transformation is not possible for S-2. In other words, *me her friend* as a whole is an argument of *considers* whereas *me* and *her friend* are separate arguments of *calls*.

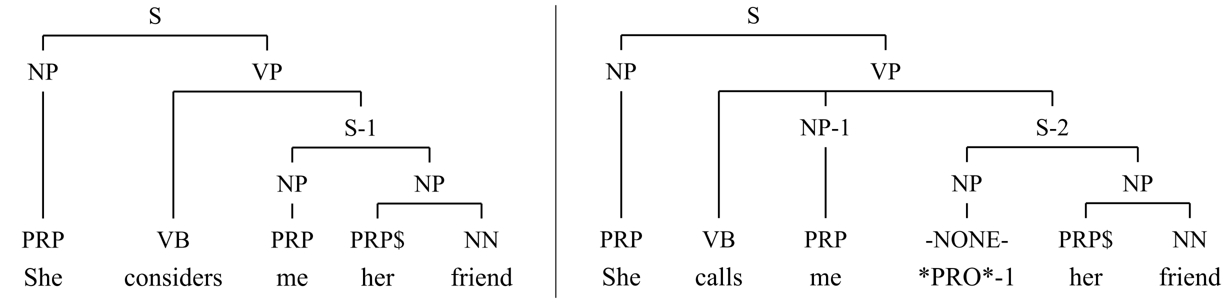


Figure 2.20: Examples of small clauses with internal (left) and external (right) subjects.

Figure 2.21 shows dependency trees converted from the trees in Figure 2.20. A small clause with an internal subject is considered a clausal complement (CCOMP; the left tree in Figure 2.20) whereas one without an internal subject is considered an object predicate (OPRD; the right tree in Figure 2.20), implying that it is a non-VP predicate of the object. This way, although *me* has no direct dependency to *friend*, their relation can be inferred through this label.

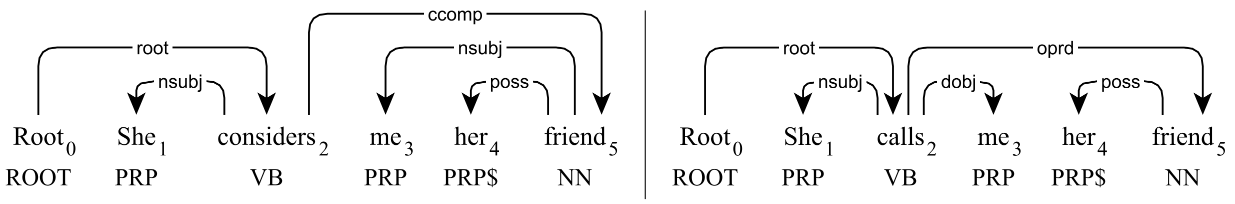


Figure 2.21: Dependency trees converted from the constituent trees in Figure 2.20.

Note that the CoNLL dependency approach uses the object predicate for both kinds of small clauses such that *me* and *her friend* become separate dependents of *considers*, as they are for *calls*. This analysis is not taken in our approach because we want our dependency trees to be consistent with the original constituent trees. Preserving the original structure makes it easier to integrate additional information to the converted dependency trees that has been already annotated on top of these constituent trees (e.g., semantic roles in PropBank).

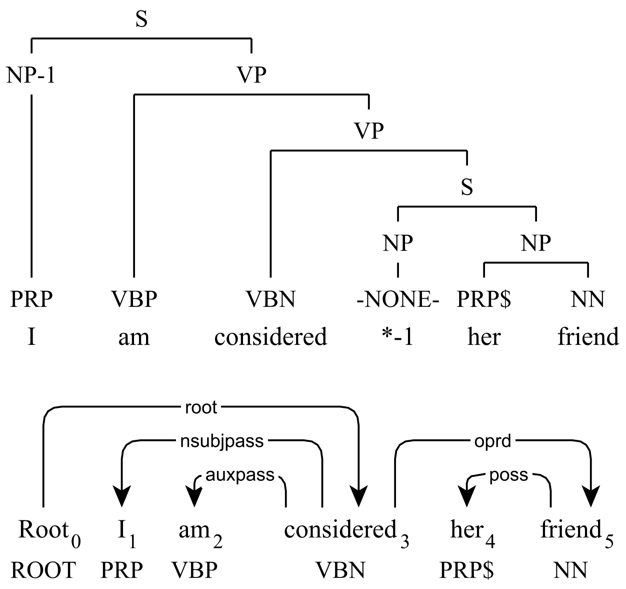


Figure 2.22: An example of a small clause in a passive construction.

For passive constructions, `OPRD` is applied to both kinds of small clauses because a dependency between the object and the non-`VP` predicate is lost by the `NP` movement. In Figure 2.22, *I* is moved from the object position to the subject position of *considered* (`NSUBJPASS`; Section D.1.6); thus, it is no longer a dependent of *friend.* The dependency between *I* and *friend* can be inferred through `OPRD` without adding more structural complexity to the tree.

2.4 Assigning dependency labels

2.4.1 Clear dependency labels

Table 2.3 shows a list of dependency labels, called the Clear dependency labels, generated by our dependency conversion. These labels are mostly inspired by the Stanford dependency approach, partially borrowed from the CoNLL dependency approach, and newly introduced by the Clear dependency approach to minimize unclassified dependencies. Appendix D shows detailed descriptions of the Clear dependency labels. Section 2.4.3 shows a comparison between the Clear and the Stanford dependencies.

Label	Description	Label	Description
ACOMP	Adjectival complement	NEG	Negation modifier
ADVCL	Adverbial clause modifier	NMOD*	Modifier of nominal
ADVMOD	Adverbial modifier	NN	Noun compound modifier
AGENT	Agent	NPADVMOD	Noun phrase as ADVMOD
AMOD	Adjectival modifier	NSUBJ	Nominal subject
APPOS	Appositional modifier	NSUBJPASS	Nominal subject (passive)
ATTR	Attribute	NUM	Numeric modifier
AUX	Auxiliary	NUMBER	Number compound modifier
AUXPASS	Auxiliary (passive)	OPRD*	Object predicate
CC	Coordinating conjunction	PARATAXIS	Parataxis
CCOMP	Clausal complement	PARTMOD	Participial modifier
COMPLM	Complementizer	PCOMP	Complement of a preposition
CONJ	Conjunct	POBJ	Object of a preposition
CSUBJ	Clausal subject	POSS	Possession modifier
CSUBJPASS	Clausal subject (passive)	POSSESSIVE	Possessive modifier
DEP	Unclassified dependent	PRECONJ	Pre-correlative conjunction
DET	Determiner	PREDET	Predeterminer
DOBJ	Direct object	PREP	Prepositional modifier
EXPL	Expletive	PRT	Particle
INFMOD	Infinitival modifier	PUNCT	Punctuation
INTJ**	Interjection	QUANTMOD	Quantifier phrase modifier
IOBJ	Indirect object	RCMOD	Relative clause modifier
MARK	Marker	ROOT	Root
META**	Meta modifier	XCOMP	Open clausal complement

Table 2.3: A list of the Clear dependency labels. Labels followed by * are borrowed from the CoNLL dependency approach. Labels followed by ** are newly introduced by the Clear dependency approach.

2.4.2 Dependency label heuristics

The *getDependencyLabel*(C, P, p) in Algorithm 2.10 assigns a dependency label to a constituent C by using function tags and inferring constituent relations between C, P, and p, where P is the parent of C and p is the head constituent of P. Heuristics described in this algorithm are derived from careful analysis of several constituent Treebanks (Marcus et al., 1993; Nielsen et al., 2010; Weischedel et al., 2011; Verspoor et al., 2012) and manually evaluated case-by-case. All supplementary methods are described in Appendix D. The *getSimpleLabel*(C) method in Algorithm 2.9 returns the dependency label of a constituent C if it can be inferred from the POS tag of C; otherwise, `null`.

Algorithm 2.9 : *getSimpleLabel*(C)

```
 Input:   A constituent $C$.
Output:   The dependency label of $C$ if it can be inferred from the POS tag of $C$;
          otherwise, null.

1:  let $d$ be the head dependent of $C$
2:  if $C$ is ADJP|WHADJP|JJ* return AMOD                  # Section D.10.1
3:  if $C$ is PP|WHPP return PREP                          # Section D.8.3
4:  if $C$ is PRT|RP return PRT                            # Section D.10.7
5:  if isPreCorrelativeConjunction($C$) return PRECONJ     # Section D.6.3
6:  if isCoordinatingConjunction($C$) return CC            # Section D.6.2
7:  if isParentheticalModifier($C$) return PARATAXIS       # Section D.10.5
8:  if isPunctuation($C|d$) return PUNCT                   # Section D.10.8
9:  if isInterjection($C|d$) return INTJ                   # Section D.10.3
10: if isMetaModifier($C$) return META                     # Section D.10.4
11: if isAdverbialModifier($C$) return ADVMOD              # Section D.5.2
12: return null
```

Algorithm 2.10 : *getDependencyLabel*(C, P, p)

```
 Input:  Constituents $C$, $P$, and $p$.
         $P$ is the parent of $C$, and $p$ is the head constituent of $P$.
Output:  The dependency label of $C$ with respect to $p$ in $P$.

1:  let $c$ be the head constituent of $C$
2:  let $d$ be the head dependent of $C$
3:  if hasAdverbialTag($C$) then                                          # Section D.5
4:    if $C$ is S|SBAR|SINV return ADVCL
5:    if $C$ is NML|NP|QP return NPADVMOD
6:  if (label $\leftarrow$ getSubjectLabel($C$)) $\neq$ null return label                # Section D.1
7:  if $C$ is UCP then
8:    $c$.add(all function tags of $C$)
9:    return getDependencyLabel($c, P, p$)
10: if $P$ is VP|SINV|SQ then
11:   if $C$ is ADJP return ACOMP
12:   if (label $\leftarrow$ getObjectLabel($C$)) $\neq$ null return label                # Section D.2
13:   if isObjectPredicate($C$) return OPRD                                 # Section D.2.4
14:   if isOpenClausalComplement($C$) return XCOMP                          # Section D.4.3
15:   if isClausalComplement($C$) return CCOMP                              # Section D.4.2
16:   if (label $\leftarrow$ getAuxiliaryLabel($C$)) $\neq$ null return label             # Section D.3
17: if $P$ is ADJP|ADVP then
18:   if isOpenClausalComplement($C$) return XCOMP                          # Section D.4.3
19:   if isClausalComplement($C$) return CCOMP                              # Section D.4.2
20: if $P$ is NML|NP|WHNP then
21:   if (label $\leftarrow$ getNonFiniteModifierLabel($C$)) $\neq$ null return label     # Section D.7
22:   if isRelativeClauseModifier($C$) return RCMOD                         # Section D.7.10
23:   if isClausalComplement($C$) return CCOMP                              # Section D.4.2
24: if isPossessionModifier($C, P$) return POSS                             # Section D.10.6
25: if (label $\leftarrow$ getSimpleLabel($C$)) $\neq$ null return label                  # Section 2.4.2
26: if $P$ is PP|WHPP return getPrepositionModifierLabel($C$)               # Section D.8
27: if ($C$ is SBAR) or isOpenClausalComplement($C$) return ADVCL           # Section D.4.3
28: if ($P$ is PP) and ($C$ is S*) return ADVCL
29: if $C$ is S|SBARQ|SINV|SQ return CCOMP
30: if $P$ is QP return ($C$ is CD) ? NUMBER : QUANTMOD
31: if ($P$ is NML|NP|NX|WHNP) or ($p$ is NN*|PRP|WP) then
32:   return getNounModifierLabel($C$)                                      # Section D.7
33: if (label $\leftarrow$ getSimpleLabel($c$)) $\neq$ null return label                  # Section 2.4.2
34: if $d$ is IN return PREP
35: if $d$ is RB* return ADVMOD
36: if ($P$ is ADJP|ADVP|PP) or ($p$ is JJ*|RB*) then
37:   if $C$ is NML|NP|QP|NN*|PRP|WP return NPADVMOD
38:   return ADVMOD
39: return DEP
```

2.4.3 Comparison to the Stanford dependency approach

Treating dependency trees generated by the Stanford dependency approach as gold-standard, the Clear dependency approach shows a labeled attachment score of 90.39%, an unlabeled attachment score of 95.39%, and a label accuracy of 93.01%. For comparison, the OntoNotes Treebank is used, which consists of various corpora in multiple genres (see Section 3.1 for more details about the OntoNotes Treebank). Out of 138K dependency trees generated by our conversion, 3.69% of them contain at least one non-projective dependency. Out of 2.6M dependencies, 3.62% are unclassified by the Stanford converter whereas 0.23% are unclassified by our approach, that is a 93.65% proportional reduction in error. A dependency is considered unclassified if it is assigned with the label, `DEP` (Section D.10.2). Table 2.4 shows a list of the top 40 dependency labels generated by our approach that are unclassified by the Stanford dependency approach.[7]

PUNCT	23.98	MARK	3.37	AMOD	0.83	PCOMP	0.49	PRECONJ	0.15
INTJ	14.18	PREP	1.97	NMOD	0.82	COMPLM	0.47	PREDET	0.14
APPOS	9.44	OPRD	1.86	NSUBJ	0.72	ACOMP	0.43	CSUBJ	0.12
META	7.25	ADVMOD	1.56	PARTMOD	0.70	NEG	0.39	INFMOD	0.12
NPADVMOD	6.86	XCOMP	1.07	NN	0.69	POBJ	0.29	IOBJ	0.09
CCOMP	6.71	PARATAXIS	1.06	QUANTMOD	0.67	DET	0.22	POSS	0.02
ADVCL	4.98	CONJ	1.06	CC	0.64	DOBJ	0.22	NUM	0.01
DEP	4.75	RCMOD	0.92	PRT	0.50	ATTR	0.18	AGENT	0.01

Table 2.4: A list of the Clear dependency labels that are unclassified by the Stanford dependency approach. The first column shows the unclassified Clear dependency labels and the second column shows their proportions to unclassified dependencies in the Stanford dependency approach (in %).

Table 2.5 shows mappings between the Clear and the Stanford dependency labels. Some labels in the Stanford dependency approach are not used in our conversion. For instance, multi-word expressions (`MWE`) are not used in our approach because it is not clear how to identify multi-word expressions systematically. Furthermore, purpose clause modifiers (`PURPCL`) and temporal modifiers (`TMOD`) are not included as dependencies but added as separate features of individual nodes in our dependency trees (see Section A.1 for more details about these additional features).

[7] The following options are used for the Stanford dependency conversion, which is the same setup that was used for the SANCL'12 shared task (Petrov and McDonald, 2012): `-basic -conllx -keepPunct -makeCopulaHead`.

Clear	Count	Stanford
ACOMP	20,325	ACOMP(98.19)
ADVCL	33,768	ADVCL(53.43), XCOMP(19.79), DEP(11.33), CCOMP(6.67), PARTMOD(6.04)
ADVMOD	101,134	ADVMOD(96.38)
AGENT	4,756	PREP(99.62)
AMOD	131,971	AMOD(97.93)
APPOS	17,869	APPOS(54.80), DEP(40.56)
ATTR	22,597	ATTR(81.87), NSUBJ(15.41)
AUX	106,428	AUX(99.98)
AUXPASS	19,289	AUXPASS(99.99)
CC	68,522	CC(99.26)
CCOMP	42,354	CCOMP(78.50), DEP(12.16), XCOMP(6.73)
COMPLM	13,130	COMPLM(94.94)
CONJ	61,270	CONJ(97.42)
CSUBJ	1,766	CSUBJ(92.19), DEP(5.32)
CSUBJPASS	72	CSUBJPASS(91.67), DEP(6.94)
DEP	4,046	DEP(90.06), NSUBJ(5.98)
DET	214,488	DET(99.82)
DOBJ	112,856	DOBJ(98.90)
EXPL	4,373	EXPL(99.20)
INFMOD	5,697	INFMOD(98.05)
INTJ	10,947	DEP(99.44)
IOBJ	2,615	IOBJ(86.16), DOBJ(10.48)
MARK	21,235	MARK(82.07), DEP(12.18), COMPLM(5.66)
META	5,620	DEP(99.00)
NEG	18,585	NEG(95.71)
NMOD	923	DEP(68.47), AMOD(30.23)
NN	149,201	NN(99.51)
NPADVMOD	21,267	TMOD(41.11), DEP(24.77), NPADVMOD(14.70), DOBJ(8.08), NSUBJ(5.01)
NSUBJ	208,934	NSUBJ(99.52)
NSUBJPASS	16,994	NSUBJPASS(99.82)
NUM	30,412	NUM(99.91)
NUMBER	3,456	NUMBER(98.96)
OPRD	2,855	DEP(49.91), ACOMP(26.90), XCOMP(22.42)
PARATAXIS	3,662	PARATAXIS(77.01), DEP(22.23)
PARTMOD	9,945	PARTMOD(94.17), DEP(5.39)
PCOMP	12,702	PCOMP(88.99), POBJ(7.98)
POBJ	222,115	POBJ(99.89)
POSS	45,156	POSS(99.91)
POSSESSIVE	16,608	POSSESSIVE(99.99)
PRECONJ	574	PRECONJ(76.83), DEP(20.56)
PREDET	2,409	PREDET(94.65), DEP(4.61)
PREP	231,742	PREP(97.52)
PRT	10,149	PRT(96.21), DEP(3.79)
PUNCT	280,452	PUNCT(93.39), DEP(6.56)
QUANTMOD	3,467	QUANTMOD(83.50), DEP(14.94)
RCMOD	22,781	RCMOD(96.28), DEP(3.09)
ROOT	132,225	ROOT(99.98)
XCOMP	25,909	XCOMP(89.61), CCOMP(7.13), DEP(3.17)

Table 2.5: Mappings between the Clear and the Stanford dependency labels. The Clear column show the Clear dependency labels. The Count column shows the count of each label. The Stanford column shows labels generated by the Stanford converter in place of the Clear dependency label with probabilities (in %); labels with less than 3% occurrences are discarded.

2.5 Adding secondary dependencies

Secondary dependencies are additional dependency relations derived from gapping relations (Section 2.5.1), relative clauses (Section 2.5.2), right node raising (Section 2.5.3), and open clausal complements (Section 2.5.4). These are separated from the other types of dependencies (Section 2.4) because they can break tree properties (e.g., single head, acyclic) when combined with the others. Preserving tree structure is important because most dependency parsing algorithms assume their input to be trees. Secondary dependencies give deeper representations that allow extraction of more complete information from the dependency structure.

2.5.1 `GAP`: gapping

Gapping is represented by co-indexes (with the `=` symbol) in constituent trees. Gapping usually happens in forms of coordination where some parts included in the first conjunct do not appear in later conjuncts (Jackendoff, 1971). In Figure 2.23, the first conjunct, `VP`-3, contains the verb *used*, which does not appear in the second conjunct, `VP`-4, but is implied for both `NP`=1 and `PP`=2. The CoNLL dependency approach makes the conjunction, *and*, the heads of both `NP`=1 and `PP`=2, and adds an extra label, `GAP`, to their existing labels (`ADV-GAP` and `GAP-OBJ` in Figure 2.25). Although this represents the gapping relations in one unified format, statistical dependency parsers perform poorly on these labels because they do not occur frequently enough and are often confused with regular coordination.

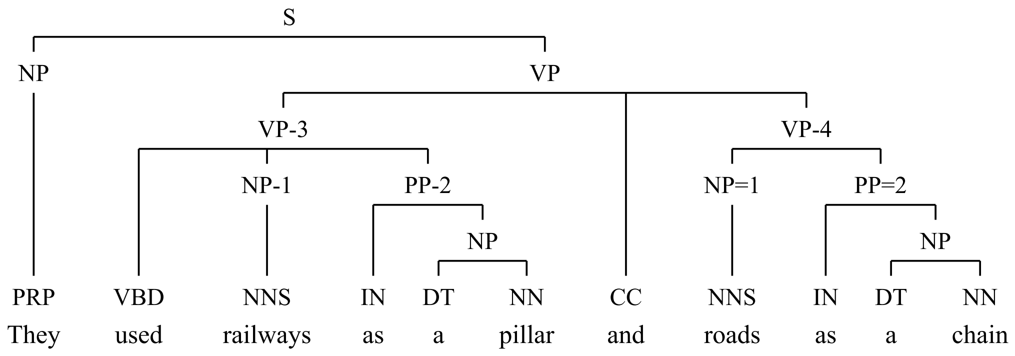


Figure 2.23: An example of a gapping relation.

In our approach, gapping is represented as secondary dependencies; this way, it can be trained separately from the other types of dependencies. The `GAP` dependencies in Figure 2.24 show how gapping is represented in our structure: the head of each constituent involving a gap (*road*, as_9) becomes a dependent of the head of the leftmost constituent not involving a gap (*railways*, as_4).

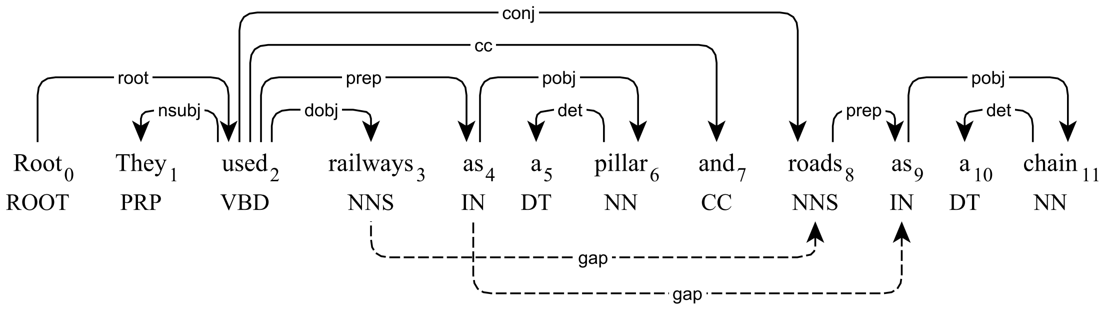

conj
cc
root
prep
pobj
pobj
nsubj
dobj
det
prep
det
Root0 They1 used2 railways3 as4 a5 pillar6 and7 roads8 as9 a10 chain11
ROOT PRP VBD NNS IN DT NN CC NNS IN DT NN
gap
gap

Figure 2.24: The Clear dependency tree converted from the constituent tree in Figure 2.23. The gapping relations are represented by the secondary dependencies, `GAP`.

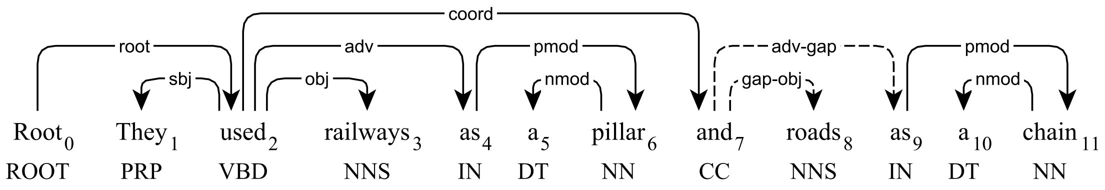

coord
root
adv
pmod
adv-gap
pmod
sbj
obj
nmod
gap-obj
nmod
Root0 They1 used2 railways3 as4 a5 pillar6 and7 roads8 as9 a10 chain11
ROOT PRP VBD NNS IN DT NN CC NNS IN DT NN

Figure 2.25: The CoNLL dependency tree converted from the constituent tree in Figure 2.23. The dependencies derived from the gapping relations, `ADV-GAP`, `GAP-OBJ`, are indicated by dotted lines.

2.5.2 `REF`: referent

A referent is the relation between a *wh*-complementizer in a relative clause and its referential head. In Figure 2.26, the relation between the complementizer *which* and its referent *Crimes* is represented by the `REF` dependency. Referent relations are represented as secondary dependencies because integrating them with other dependencies breaks the single-head tree property (e.g., *which* would have multiple heads in Figure 2.26). The *linkReferent*(C) method in Algorithm 2.11 finds a *wh*-complementizer and makes it a dependent of its referent. Note that referent relations are not provided in constituent trees; however, they are manually annotated in the PropBank as

LINK-SLC (Bonial et al., 2010, Chap. 1.8). This algorithm was tested against the PropBank annotation using gold-standard constituent trees and showed an F1-score of approximately 97%.

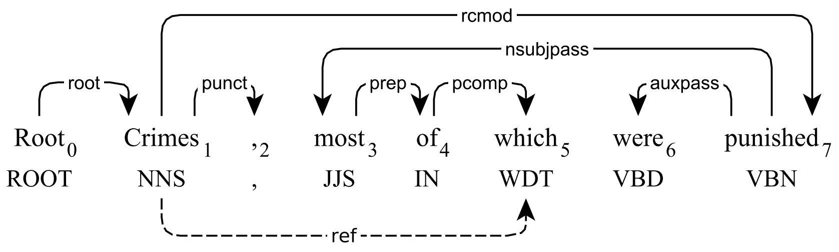

rcmod
nsubjpass
root
punct
prep
pcomp
auxpass
Root0 Crimes1 ,2 most3 of4 which5 were6 punished7
ROOT NNS , JJS IN WDT VBD VBN
ref

Figure 2.26: An example of a referent relation. The referent relation is represented by the secondary dependency, REF.

Algorithm 2.11 : *linkReferent*(C)

```
Input:  A constituent $C$.
 1: if $C$ is WHADVP|WHNP|WHPP then
 2:   let $c$ be the wh-complementizer of $C$
 3:   let $s$ be the topmost SBAR of $C$
 4:   if the parent of $s$ is UCP then $s \leftarrow s$.parent
 5:   if isRelativizer($c$) and ($s$ has no NOM) then
 6:     let $p$ be the parent of $s$
 7:     ref ← null
 8:     if $p$ is NP|ADVP then
 9:       let ref be the previous sibling of $s$ that is NP|ADVP, respectively
10:     elif $p$ is VP then
11:       let $t$ be the previous sibling of $s$ that has PRD
12:       if $s$ has CLF then ref ← $t$
13:       if ($C$ is WHNP) and ($t$ is NP) then ref ← $t$
14:       if ($C$ is WHPP) and ($t$ is PP) then ref ← $t$
15:       if ($C$ is WHADVP) and ($t$ is ADVP) then ref ← $t$
16:     if ref ≠ null then
17:       while ref has an antecedent do ref ← ref.antecedent
18:       $c$.rHead ← ref
19:       $c$.rLabel ← REF
```

Algorithm 2.12 : *isRelativizer*(C)

```
 Input:  A constituent $C$.
Output:  True if $C$ is a relativizer linked to some referent; otherwise, False.
1: return $C$ is 0|that|when|where|whereby|wherein|whereupon|which|who|whom|whose
```

2.5.3 RNR: right node raising

As mentioned in Section 2.2.3, missing dependencies caused by right node raising are preserved as secondary dependencies. In Figure 2.14 (page 19), *her* should be a dependent of both *for* and *in*; however, it is a dependent of only *for* in our structure because making it a dependent of both nodes breaks a tree property (e.g., *her* would have multiple heads). Instead, the dependency between *her* and *for* is preserved with the RNR dependency. Figure 2.28 shows another example of right node raising where the raised constituent, VP-2, is the head of the constituents that it is raised from, VP-4 and VP-5. In this case, *done* becomes the head of *can*$_2$ with the dependency label, RNR.

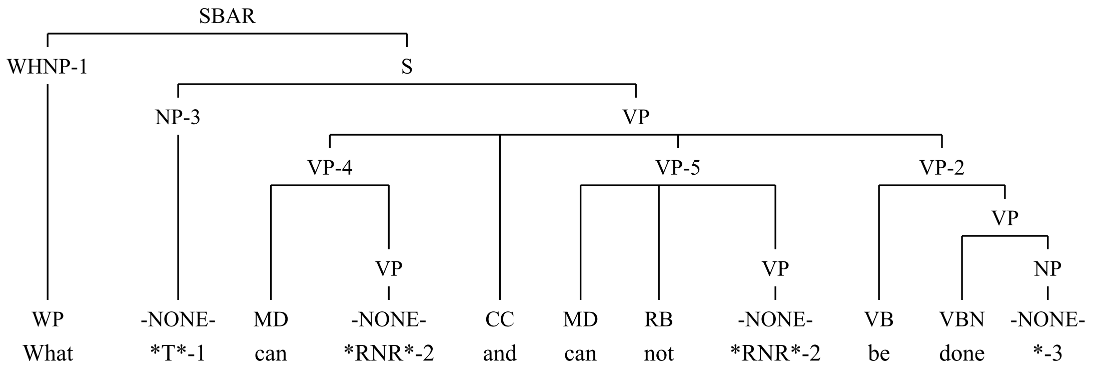


Figure 2.27: An example of right node raising where the raised constituent is the head.

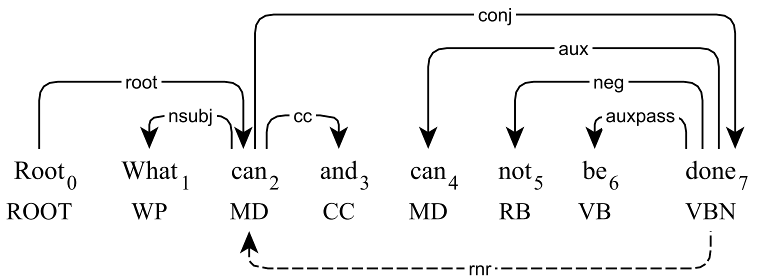


Figure 2.28: The dependency tree converted from the constituent tree in Figure 2.27. Right node raising is represented by the secondary dependency, RNR.

2.5.4 XSUBJ: open clausal subject

An open clausal subject is the subject of an open clausal complement (usually non-finite) that is governed externally. Open clausal subjects are often caused by raising and control verbs (Chomsky, 1981). In Figure 2.29, the subject of *like* is moved to the subject position of the raising verb *seemed* (subject raising) so that *She* becomes the syntactic subject of *seemed* as well as the open clausal subject of *like* (see Figure 2.30).

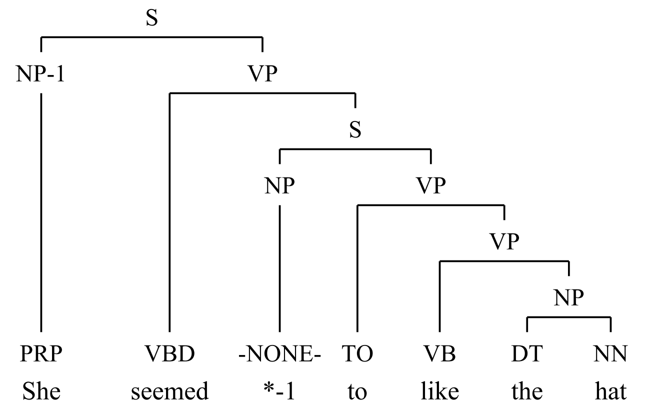


Figure 2.29: An example of an open clausal subject caused by a subject raising.

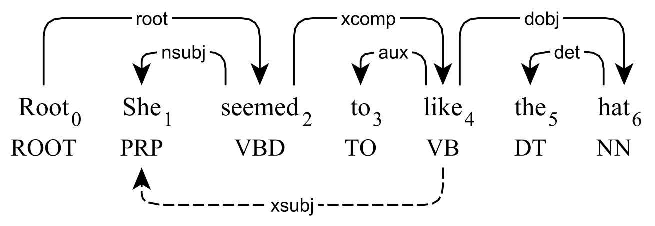


Figure 2.30: The dependency tree converted from the constituent tree in Figure 2.29. The open clausal subject is represented by the secondary dependency, XSUBJ.

In Figure 2.31, the subject of *wear* is shared with the object of the control verb *forced* (object control) so that *me* becomes the direct object of *forced* as well as the open clausal subject of *wear* (Figure 2.32). Alternatively, *me* in Figure 2.33 is not considered the direct object of *expected* but the subject of *wear*; this is a special case called "exceptional case marking (ECM)", which appears to be very similar to the object control case but is handled differently in constituent trees (see Taylor (2006) for more details about ECM verbs).

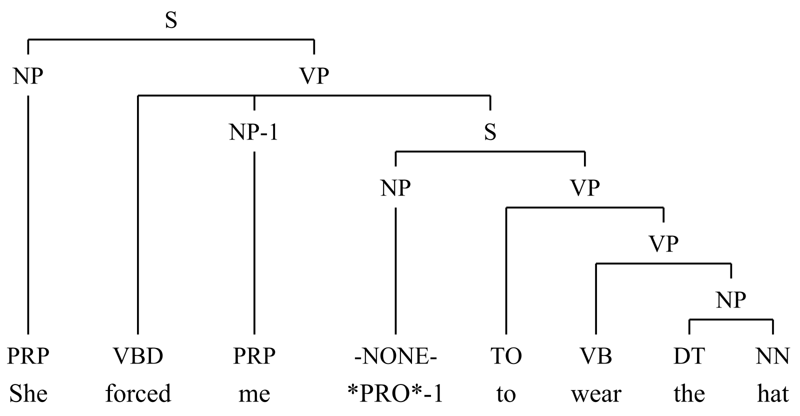

S
NP
VP
NP-1
S
NP
VP
VP
NP
PRP VBD PRP -NONE- TO VB DT NN
She forced me *PRO*-1 to wear the hat

Figure 2.31: An example of an open clausal subject caused by an object raising.

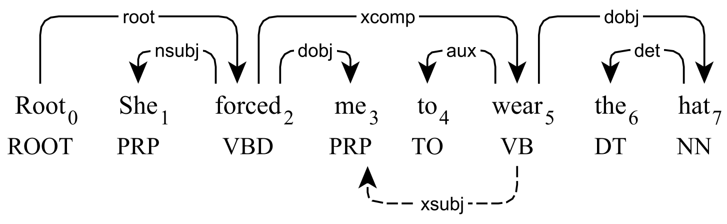

root
nsubj
xcomp
dobj
aux
dobj
det
Root0 She1 forced2 me3 to4 wear5 the6 hat7
ROOT PRP VBD PRP TO VB DT NN
xsubj

Figure 2.32: A dependency tree converted from the constituent tree in Figure 2.31. The open clausal subject is represented by the secondary dependency, XSUBJ.

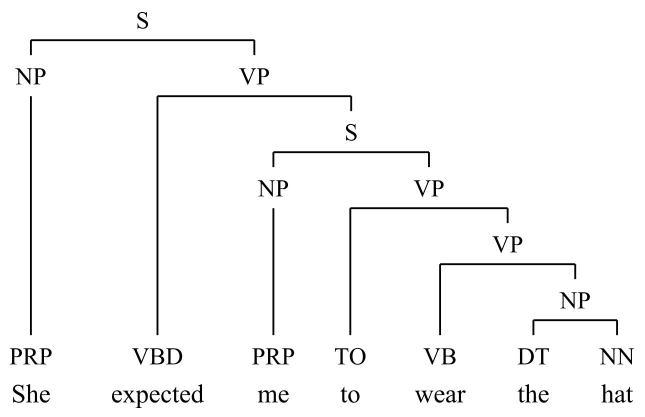


Figure 2.33: An example of exceptional case marking.

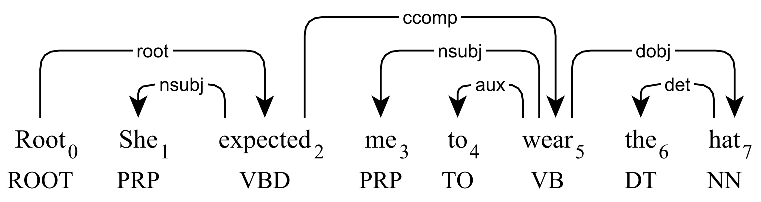

ccomp
root
nsubj
nsubj
aux
dobj
det
Root0 She1 expected2 me3 to4 wear5 the6 hat7
ROOT PRP VBD PRP TO VB DT NN

Figure 2.34: A dependency tree converted from the constituent tree in Figure 2.33.

2.6 Adding function tags

2.6.1 `SEM`: semantic function tags

When a constituent is annotated with a semantic function tag (`BNF`, `DIR`, `EXT`, `LOC`, `MNR`, `PRP`, `TMP`, and `VOC`; see Section A.1), the tag is preserved with the head of the constituent as an additional feature. In Figure 2.35, the subordinate clause `SBAR` is annotated with the function tag `PRP`, so the head of the subordinate clause, *is*, is annotated with the semantic tag in our representation (Figure 2.36). Note that the CoNLL dependency approach uses these semantic tags in place of dependency labels (e.g., the dependency label between *is* and *let* would be `PRP` instead of `ADVCL`). These tags are kept separate from the other kinds of dependency labels in our approach so they can be processed either during or after parsing. The semantic function tags can be integrated easily into our dependency structure by replacing dependency labels with semantic tags (Figure 2.37).

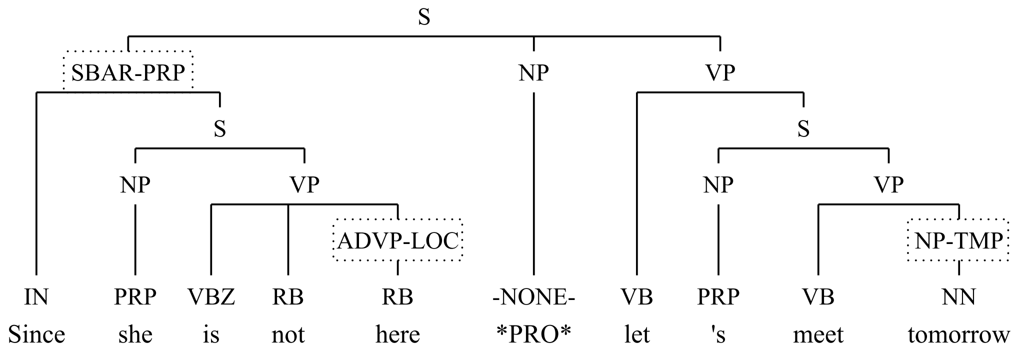


Figure 2.35: A constituent tree with semantic function tags. The phrases with the semantic function tags are indicated by dotted boxes.

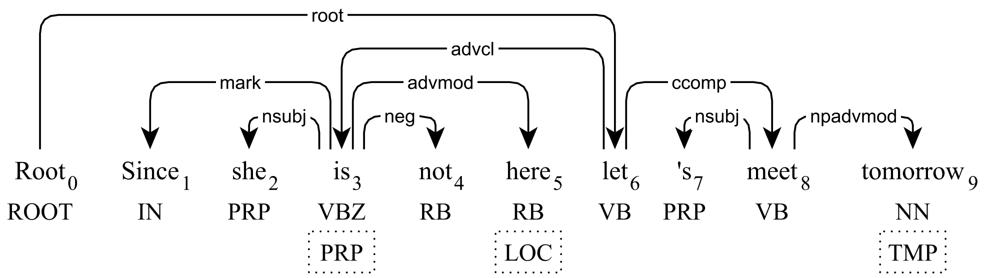


Figure 2.36: A dependency tree converted from the constituent tree in Figure 2.35. The function tags `PRP`, `LOC`, and `TMP` are preserved as additional features of *is*, *here*, and *tomorrow*, respectively.

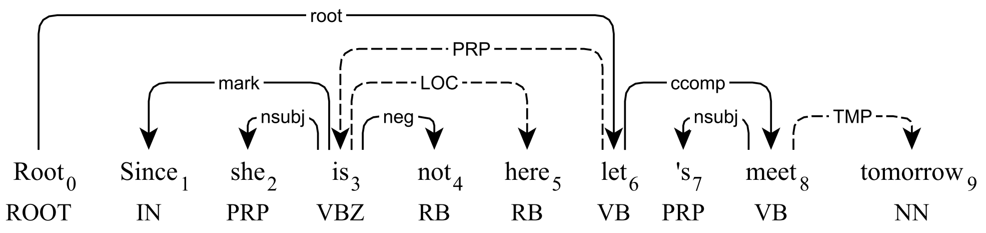

root
PRP
mark
LOC
ccomp
nsubj
neg
nsubj
TMP
Root0 Since1 she2 is3 not4 here5 let6 's7 meet8 tomorrow9
ROOT IN PRP VBZ RB RB VB PRP VB NN

Figure 2.37: Another dependency tree converted from the constituent tree in Figure 2.35. The function tags, `PRP`, `LOC`, and `TMP`, replace the original dependency labels, `ADVCL`, `ADVMOD`, and `NPADVMOD`.

2.6.2 `SYN`: syntactic function tags

When a constituent is annotated with one or more syntactic function tags (`ADV`, `CLF`, `CLR`, `DTV`, `NOM`, `PUT`, `PRD`, `RED`, and `TPC`; see Section A.1), all tags are preserved with the head of the constituent as additional features. In Figure 2.38, the noun phrase `NP-1` is annotated with the function tag `PRD` and `TPC` so the head of the noun phrase, *slap*, is annotated with both tags in our representation (Figure 2.39). Similarly to the semantic function tags (Section 2.6.1), syntactic function tags can also be integrated into our dependency structure by replacing dependency labels with syntactic tags.

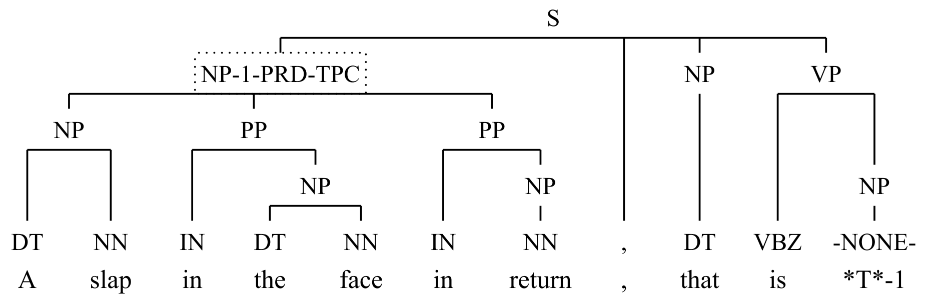

S
NP-1-PRD-TPC
NP
VP
NP
PP
PP
NP
NP
NP
DT NN IN DT NN IN NN , DT VBZ -NONE-
A slap in the face in return , that is *T*-1

Figure 2.38: A constituent tree with syntactic function tags. The phrase with the syntactic function tags is indicated by a dotted box.

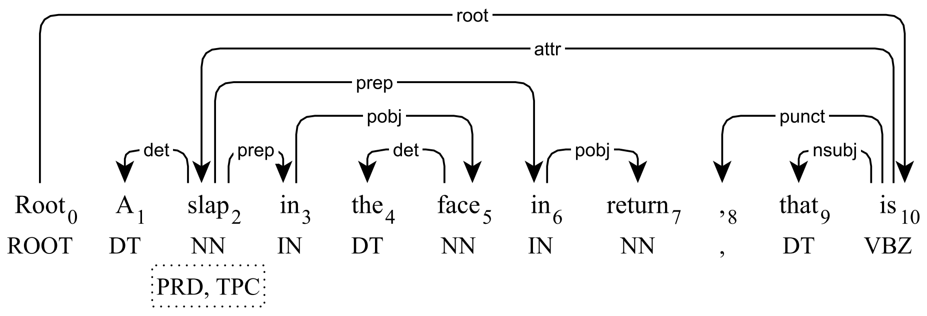

root
attr
prep
pobj
punct
det
prep
det
pobj
nsubj
Root0 A1 slap2 in3 the4 face5 in6 return7 ,8 that9 is10
ROOT DT NN IN DT NN IN NN , DT VBZ
PRD, TPC

Figure 2.39: A dependency tree converted from the constituent tree in Figure 2.38. The function tags, `PRD` and `TPC`, are preserved as additional features of *slap*.

Chapter 3

Experimental setup

All experiments for POS tagging (Section 4.7), dependency parsing (Section 5.6), and semantic role labeling (Section 6.5) use the same setup. For each task, several systems are run for comparison and two models are built for each system; one is trained on the Wall Street Journal corpus in OntoNotes, called the WSJ model, and the other is trained on all corpora in OntoNotes, called the OntoNotes model. These models are evaluated on corpora from six genres in OntoNotes and three genres in medical domains (Section 3.1). The WSJ models demonstrate how the systems perform when they are trained and evaluated on data from different genres. The OntoNotes models demonstrate how the systems perform when they are trained on a mixture of genres and evaluated on genres that are the same or different from the ones used for training. These experiments provide insights for those who want to build a single model to process data from different sources.

All models are trained by a machine learning algorithm called Liblinear (Section 3.2). For each system, both accuracies and speeds are measured. Accuracies are measured by the standard methods generally used for evaluating each task (see Table 3.1). Speeds are measured by running each system five times, cutting off the top and the bottom speeds, and averaging the middle three. All systems are evaluated on an Intel Xeon 2.57GHz using a single core, a Linux v2.6.18-308.11.1.el5, and a 64-Bit Java virtual machine v1.6.0_33. Note that the model loading times are excluded for speed comparison because they are fixed costs that can be pre-processed, whereas the data reading and the feature extraction times are included because they play an important role during decoding. The accuracy and speed results demonstrate the robustness and scalability of each system.

Task	Measurement	Equation
POS tagging	Accuracy	$Acc_p = \forall i.\ \lvert C_p(w_i)\rvert \,/\, \lvert w_i\rvert$
Dependency parsing	Labeled attachment score	$\text{LAS} = \forall i.\ \lvert C_{h\wedge l}(w_i)\rvert \,/\, \lvert w_i\rvert$
	Unlabeled attachment score	$\text{UAS} = \forall i.\ \lvert C_h(w_i)\rvert \,/\, \lvert w_i\rvert$
Semantic role labeling	Argument identification + Argument classification	
	- Precision	$P_{h\wedge l} = \forall i.\ \lvert C_{h\wedge l}(a_i)\rvert \,/\, \lvert a_i\rvert_s$
	- Recall	$R_{h\wedge l} = \forall i.\ \lvert C_{h\wedge l}(a_i)\rvert \,/\, \lvert a_i\rvert_g$
	- F1-score	$F_{h\wedge l} = 2 \cdot (P_{h\wedge l} \cdot R_{h\wedge l})/(P_{h\wedge l} + R_{h\wedge l})$

Table 3.1: Evaluation methods for different tasks. $|w_i|$: the total # of word tokens, $|a_i|_s$: the total # of system-generated arguments, $|a_i|_g$: the total # of gold-standard arguments, $C_p(w_i)$: a token with the correct POS tag, $C_h(w_i)$: a token with the correct dependency head, $C_{h\wedge l}(w_i)$: a token with the correct dependency head and label, $C_{h\wedge l}(a_i)$: an argument with the correct semantic head (predicate) and label (semantic role).

3.1 Corpora

For the WSJ models, the Wall Street Journal sections 2-21 from OntoNotes (Weischedel et al., 2011) are used for training. This training set consists of 30,060 sentences with 731,677 word tokens and 77,826 verb predicates. We use the WSJ corpus from OntoNotes instead of the original Penn Treebank (Marcus et al., 1993) because they use slightly different annotation guidelines which our evaluation data follow. Table 3.2 shows the distributions of all genres used for training the OntoNotes models. The latest version of OntoNotes, v4.99, is used for building these models.[1]

	BC	BN	MZ	NW	TC	WB	ALL
Tokens	214,178	228,019	165,400	1,047,377	89,278	238,760	1,983,012
Sentences	14,131	11,366	6,924	41,553	11,386	11,046	96,406
Predicates	27,860	27,187	18,898	106,295	13,720	19,735	213,695

Table 3.2: Distributions of training sets for the OntoNotes models. BC: broadcasting conversation, BN: broadcasting news, MZ: magazine, NW: newswire, TC: telephone conversion, WB: web-text, ALL: all genres combined.

For evaluation, six corpora from OntoNotes are used: the MSNBC broadcasting conversation (BC), the CNN broadcasting news (BN), the Sinorama news magazine (MZ), the WSJ newswire (NW), the Callhome telephone conversation (TC), and the GALE web-text (WB), and three corpora from

[1] The religious texts in OntoNotes v4.99, the Bible, are excluded from our experiments because they come from a very specific domain that should be trained and evaluated separately.

medical domains are used (Nielsen et al., 2010): the Mipacq clinical notes (MP), the Medpedia articles (MD), and the SHARP clinical notes (SH). These corpora are first evaluated individually then grouped together to demonstrate in-genre and out-of-genre experiments. For the WSJ models, the WSJ corpus is used for in-genre and all other corpora are used for out-of-genre experiments. For the OntoNotes models, all OntoNotes corpora are used for in-genre and all medical corpora are used for out-of-genre experiments.

Source	**OntoNotes**						**Medical**			
Genre	**BC**	**BN**	**MZ**	**NW**	**TC**	**WB**	**MD**	**MP**	**SH**	**ALL**
Tokens	31,704	31,328	32,120	39,590	32,444	34,707	34,022	35,721	33,291	304,927
Sentences	2,076	1,969	1,409	1,640	4,505	1,738	1,850	3,170	2,651	21,008
Predicates	4,429	4,001	3,849	4,138	5,408	3,673	4,356	4,244	2,182	36,280
Arguments	12,084	10,210	10,287	11,120	14,182	9,751	8,780	8,320	4,142	88,876

Table 3.3: Distributions of evaluation sets. BC: broadcasting conversation, BN: broadcasting news, MD: Medpedia, MP: Mipacq, MZ: magazine, NW: newswire, SH: SHARP, TC: telephone conversion, WB: web-text, ALL: all data combined.

Table 3.4 shows lexical similarities between the evaluation sets in Table 3.3. Given two sets of lexical types, S_i ans S_j, collected from corpora C_i and C_j, the lexical similarity of C_i against C_j is measured as $LS_{ij} = \frac{|S_i \cap S_j|}{|S_i|}$, where i and j indicate the indices of a column and a row in Table 3.4, respectively. For example, LS_{14} (that is the value in the BC column and the NW row) implies that out of 3,677 lexical types in BC, 50.88% of them intersect with lexical types in NW.

	BC	**BN**	**MZ**	**NW**	**TC**	**WB**	**MD**	**MP**	**SH**
BC	-	35.76	29.09	27.99	40.90	26.85	20.30	18.13	15.12
BN	48.74	-	36.78	35.26	47.66	33.08	25.75	22.57	18.34
MZ	43.95	40.77	-	33.84	47.42	32.81	28.48	23.83	20.02
NW	50.88	47.04	40.72	-	46.65	34.76	30.69	26.11	22.81
TC	31.90	27.28	24.48	20.01	-	19.74	16.64	15.19	11.98
WB	53.01	47.91	42.88	37.74	49.97	-	29.56	25.21	21.25
MD	30.79	28.66	28.59	25.59	32.36	22.70	-	33.89	28.82
MP	30.54	27.90	26.57	24.19	32.81	21.50	37.64	-	35.02
SH	14.01	12.47	12.28	11.62	14.23	9.97	17.61	19.26	-
Count	3,677	5,011	5,555	6,685	2,868	7,259	5,576	6,193	3,407

Table 3.4: Lexical similarities between the evaluation sets in Table 3.3 (in %). The count row shows the total number of lexical types in each corpus.

Table 3.5 shows Unix commands for listing the evaluation sets in Table 3.3; this table is added for those who own these corpora and want to replicate our experimental setup. Note that some of these commands assume the Bash shell with extended globing, which require running the following command first: ‘`bash; shopt -s extglob`’.

Source	Genre	Command
OntoNotes	BC	`ls bc/msnbc/00/msnbc_000[5-7]*`
	BN	`ls bn/cnn/03/*`
	MZ	`ls mz/sinorama/10/ectb_10+(6[5-9]|7)*`
	NW	`ls nw/wsj/23/*`
	TC	`ls tc/ch/00/ch_00[4-5]*`
	WB	`ls wb/eng/00/eng_001*`
Medical	MD	`ls mipacq2/0[1-2]/[4-9]*`
	MP	`ls medpedia/02/*`
	SH	`ls sharp-clinical/02/rec+(29|[3-4])*`

Table 3.5: Unix commands for listing the evaluation sets in Table 3.3.

3.2 Machine learning

For machine learning, Liblinear L2-regularization, L1-loss support vector classification is used (Fan et al., 2008). This algorithm shares the same underlying approach as support vector machines: given a set of instance-label pairs (x_i, y_i) where $x_i \in R^d, y_i \in \{-1, 1\}$, it finds a hyperplane w maximizing the margin between the two classes by solving the optimization problem in Equation (3.1) (C is a penalty parameter).

$$w = \min_w \frac{1}{2} w^T w + C \sum_{i=1}^{n} \max(1 - y_i w^T x_i, 0) \tag{3.1}$$

In our case, all features are binarized so $x_i \in \{0, 1\}^d$, where d is the dimensionality of the feature space. Furthermore, since our tasks require predictions of multiple classes, a one-vs-all approach is adapted, which generates M numbers of hyperplanes, $[w_m : m \in \{1, \ldots, M\}]$, where m is a class and w_m is a hyperplane separating the class m from the other classes. During decoding, the one-vs-all approach uses Equation (3.2) to predict the class m given the instance x_i.

$$m = \arg\max_m w_m^T x_i \tag{3.2}$$

The main difference between support vector machines and Liblinear is that support vector machines can handle non-linearly separable data using kernels (e.g., polynomial, gaussian) whereas Liblinear performs only linear classification without using any kernel. This aligns well with our data because our feature dimensionality is often very high (up to a few millions) so that classes essentially become linearly separable through different dimensions. As a result, Liblinear significantly reduces both the training and decoding times, yet performs very accurately (see Hsieh et al. (2008) for more details about how Liblinear computes weights using a dual coordinate descent method).

Table 3.6 shows the Liblinear learning parameters used for our experiments; c is a cost, e is a termination criterion, and B is a bias. These parameters are found by running cross-validations on the WSJ training set[2] with grid search on the hyper-parameters c and B; no parameter turning is performed on e. It is possible to improve performance of the OntoNotes models by running separate cross-validations on the OntoNotes training set, which we will explore in the future.

Task	**Model**	c	e	B
POS tagging	Model$_D$	0.1	0.1	0.9
	Model$_G$	0.2	0.1	0.4
Dependency parsing	-	0.1	0.1	0.0
Semantic role labeling	Model$^{\leftarrow}$	0.1	0.1	0.0
	Model$^{\rightarrow}$	0.1	0.1	0.1

Table 3.6: Liblinear learning parameters for each task; c: cost, e: termination criterion, B: bias. Model$_D$ and Model$_G$ are domain-specific and generalized models for POS tagging (Section 4.3.1). Model$^{\leftarrow}$ and Model$^{\rightarrow}$ are lefthand and righthand models for semantic role labeling (Section 6.4.2).

The *crossValidate*(CV, c, e, B) method in Algorithm 3.1 takes a list CV of cross-validation sets and Liblinear parameters (c, e, B), and returns the average accuracy of all cross-validation sets. The *getModel*(trn_i, c, e, B) method in line 4 takes a training set trn_i and the hyper-parameters (c, e, B), and returns a Liblinear model m_i. The *getAccuracy*(tst_i, m_i) method in line 5 takes an evaluation set tst_i and the Liblinear model m_i, and returns the accuracy of m_i on tst_i. This *crossValidate*(CV, c, e, B) method is performed several times with different combinations of the hyper-parameters and a combination with the highest average accuracy is used for the final model (ones in Table 3.6).

[2] Each section in the WSJ training set is considered a cross-validation set.

Algorithm 3.1 : $crossValidate(CV, c, e, B)$

Input: A list CV of cross-validation sets, and Liblinear parameters c, e, and B.
Output: The average accuracy of all cross-validation sets.

```
1: $acc \leftarrow 0$
2: for $tst_i$ in $CV$ do
3:     $trn_i \leftarrow \sum cv$, where $[\forall cv. (cv \in CV) \wedge (cv \neq tst_i)]$
4:     $m_i \leftarrow getModel(trn_i, c, e, B)$
5:     $acc \leftarrow acc + getAccuracy(tst_i, m_i)$
6: return $acc / |CV|$
```

It is worth mentioning that we had previously tried several supervised machine learning algorithms for dependency parsing and found out that Liblinear worked the best for our experiments. Compared to maximum entropy (Ratnaparkhi, 1998), Liblinear trained as fast yet performed significantly more accurately. Compared to robust risk minimization (Zhang et al., 2002), Liblinear trained slightly faster and performed slightly more accurately. Compared to support vector machines using a linear kernel (Cortes and Vapnik, 1995), Liblinear trained significantly faster and performed as accurately. Given this previous experience, we decided to use Liblinear for all our experiments although it would be interesting to see the impact of other machine learning algorithms on different tasks.

Model	**WSJ**			**OntoNotes**		
	Features	**Train**	**Decode**	**Features**	**Train**	**Decode**
POS_D	435,462	2.4GB	0.7GB	1,063,175	6.5GB	1.5GB
POS_G	412,405			1,034,895		
DEP	973,400	6.5GB	1.2GB	1,797,232	12GB	2.1GB
$\text{SRL}^{\leftarrow}$	239,321	3.2GB		444,265		
$\text{SRL}^{\rightarrow}$	247,944			462,532		

Table 3.7: The number of features and the amount of memory used for training and decoding the WSJ and OntoNotes models. The Features column shows the number of features, and the Train and Decode columns show the amount of RAM used for training and decoding, respectively.

Table 3.7 shows the number of features and the amount of memory used for training and decoding each of our final models, which are referred to 'ClearNLP' in the following sections. To run the WSJ models for all three tasks, 12.1GB and 2.4GB of RAM are required for training and decoding, respectively. To run the OntoNotes models for all three tasks, 25.3GB and 4.3GB of RAMs are required for training and decoding, respectively.

Chapter 4

Part-of-speech Tagging

4.1 Overview

Most state-of-the-art POS taggers take statistical learning approaches, which perform very well when their training and testing data are from the same source, achieving over 97% accuracy (Toutanova et al., 2003; Giménez and Màrquez, 2004; Shen et al., 2007). However, the performance drops when they are used for tagging data that varies from their training data. Several domain adaptation approaches have been proposed for the improvement of POS tagging in new domains (Daumé and Marcu, 2006; Daume III, 2007). Although these techniques work well, they are limited in three ways. First, they assume that target domains are already known, which is often not the case; especially when POS tagging is used for real-time systems such as online machine translators or search engines. Second, these approaches still require a relatively small but sufficient amount of manual annotation from target domains, which is not always available. Third, they build different models for handling new domains, which can be cumbersome when the number of target domains becomes large.

The dynamic model selection described in this chapter makes no assumption about the target domains. The objective is not to adapt a model to a specific domain different from existing training data, but to build a model that is general enough to work well for any domain. During training, two models are built; one is optimized for lexical items specific to the training data, called a domain-specific model, and the other is optimized for lexical items general to any domain, called a generalized model. Both models are derived from the same training data; no extra data is required for building either model. During decoding, one of these models is selected dynamically by measuring similarities between input sentences and lexical items used for the models. Our hypothesis is that the domain-specific and generalized models perform better for sentences similar and dissimilar to the training data, respectively.

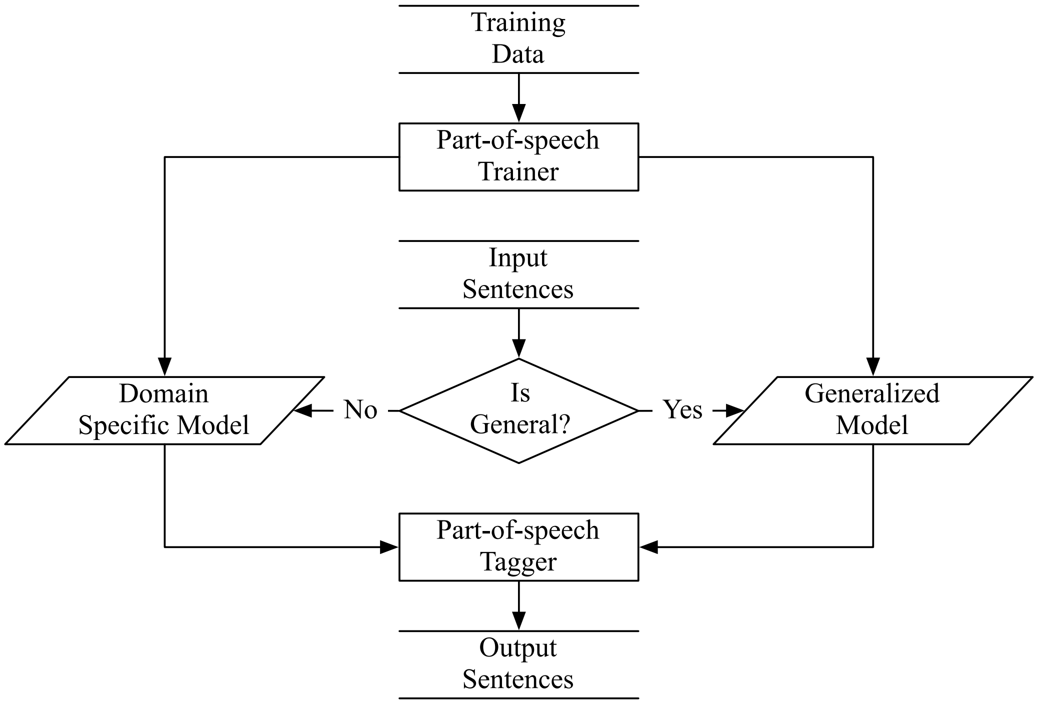


Figure 4.1: The overview of POS tagging using dynamic model selection.

Figure 4.1 shows the overview of POS tagging using our dynamic model selection approach. The following sections describe how to build both domain-specific and generalized models using the same training data (Section 4.3.1), and select an appropriate model for input sentences dynamically during decoding (Section 4.3.2). Each model uses a one-pass, left-to-right POS tagging algorithm (Section 4.4). Even with this simple tagging algorithm, our system shows tagging accuracy comparable to two other state-of-the-art POS taggers, the Stanford POS tagger and the SVMTool, when coupled with this dynamic model selection approach (Section 4.7.1). Furthermore, our system shows noticeably faster tagging speed compared to the other two systems (Section 4.7.2).

4.2 Background

The Penn Treebank project defined a set of POS tags (henceforth, the Penn POS tags) that has been used as the standard POS tagset for English (Marcus et al., 1993). The Penn POS tags were originally designed for annotating the Wall Street Journal corpus and later extended by the OntoNotes project to annotate corpora from several different sources (Weischedel et al., 2011). Table A.2 shows a list of the Penn POS tags.

Google	Penn	Description
.	`$|:|,|.|“|”|-LRB-|-RRB-|HYPH|NFP`	Punctuation
ADJ	`JJ|JJR|JJS`	Adjectives
ADP	`IN`	Prepositions, postpositions
ADV	`RB|RBR|RBS|WRB`	Adverbs
CONJ	`CC`	Conjunctions
DET	`DT|EX|PDT|WDT`	Determiners, articles
NOUN	`NN|NNP|NNPS|NNS`	Nouns
NUM	`CD`	Numerals
PRON	`PRP|PRP$\|WP\|WP$`	Pronouns
PRT	`AFX|POS|RP|TO`	Particles
VERB	`MD|VB|VBD|VBG|VBN|VBP|VBZ`	Verbs
X	`ADD|FW|GW|LS|SYM|UH|XX`	Other categories

Table 4.1: Mappings between Google's universal POS tags and the Penn POS tags. The Google column shows Google's universal POS tags and the Penn column shows lists of the Penn POS tags map to Google's POS tags (delimited by '`|`').

The Penn POS tags are fine-grained and reflect several linguistic phenomena such as pluralities of nouns, tenses of verbs, comparative forms of adjectives, etc. Recently, Google grouped the Penn POS tags into more coarse-grained ones that can be applied cross-linguistically (Petrov et al., 2012). Table 4.1 shows mappings between Google's universal POS tags and the Penn POS tags.

Sentence	He	bought	a	car	yesterday	that	is	red	.
Penn	PRP	VBD	DT	NN	NN	WDT	VBZ	JJ	.
Google	PRON	VERB	DET	NOUN	NOUN	DET	VERB	ADJ	.

Table 4.2: An example of the Penn POS tags and Google's universal POS tags. The Penn and Google rows show the Penn and Google's POS tags with respect to the word tokens.

4.3 Dynamic model selection

4.3.1 Training

Consider training data as a collection of documents where each document contains sentences focusing on similar topics. For the Wall Street Journal corpus, a document can be an individual file or all files within each section. Alternatively, a document may contain sentences covering many different topics but written in similar styles. For instance, the OntoNotes Treebank consists of corpora from various sources such that each corpus can be considered a document following similar writing styles (see Section 3.1 for more details about the OntoNotes Treebank).

To build a generalized model, lexical features (e.g., n-gram word forms) that are too specific to individual documents should be avoided; this way, a classifier can place more weights on features general to any document. To filter out document-specific features, a threshold is set to the document frequency of each *lowercase simplified word form* in the training data. A simplified word form is derived by applying the following regular expressions sequentially to a word form, w. `replaceAll` is a function that replaces all matches of the regular expression in w (the 1st parameter) with the specific string (the 2nd parameter). All numerical expressions are replaced with 0 in a simplified word form. A lowercase simplified word form is a decapitalized simplified word form.

(1) w.`replaceAll(\d%`, 0) (e.g., 1% → 0)

(2) w.`replaceAll(\$\d`, 0) (e.g., $1 → 0)

(3) w.`replaceAll(^\.\d`, 0) (e.g., .1 → 0)

(4) w.`replaceAll(\d(,|:|-|\/|\.)\d`, 0) (e.g., 1,2|1:2|1-2|1/2|1.2 → 0)

(5) w.`replaceAll(\d+`, 0) (e.g., 1234 → 0)

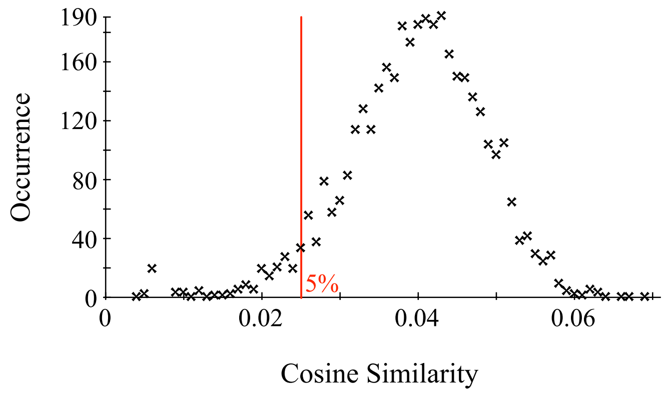
Given a set of lowercase simplified word forms whose document frequencies are greater than a certain threshold, a model is trained by using non-lexical features (e.g., POS tags of previously tagged word tokens, ambiguity classes) and lexical features only associated with these lowercase simplified word forms. For a generalized model, a threshold of 2 is used, meaning that only lexical features whose lowercase simplified word forms occur in at least 3 documents of the training data are used. For a domain-specific model, a threshold of 1 is used for our experiments. We have experimented with models using different thresholds (e.g., 0, 3), which did not improve tagging accuracy for our data.

The generalized and domain-specific models are trained separately. The domain-specific model is trained first; it is optimized by running n-fold cross-validation, where n is the number of documents in the training data, and grid search on the Liblinear parameters c and B (Hsieh et al. (2008); see Section 3.2 for more details about Liblinear). The generalized model is trained in a similar way, n-fold cross-validation with grid search, except that it uses both the domain-specific model that is previously built and the generalized model that is currently built during cross-validation and selects either model for the best results guided by gold-standard annotation. It may be possible to improve tagging accuracy by optimizing both models simultaneously, which will be explored in the future.

4.3.2 Decoding

Once domain-specific and generalized models are built, two approaches can be adapted for decoding. One is to run both models and merge their outputs. This approach produces potentially more accurate output than either model, but takes longer to decode because the merging process may take a long time and cannot be started until both models are finished. Instead, an alternative approach is taken; that is to select one of the models dynamically based on input sentences. With an efficient model selection scheme, this approach can run as fast as a single model approach, yet give more robust results.

The premise of this dynamic model selection is that the domain-specific model performs better for sentences that are similar to its training space, whereas the generalized model performs better for ones that are dissimilar. During training, a set of simplified word forms, say T, used for building the domain-specific model is collected. During decoding, another set of simplified word forms, say S, is collected from each input sentence. If the cosine similarity between T and S is greater than a certain threshold, the domain-specific model is selected; otherwise, the generalized model is selected for decoding. The threshold is derived by running another cross-validation on the training data. For each fold, both models are trained and evaluated simultaneously using the learning parameters found from the previous cross-validation (Section 4.3.1). During this cross-validation, cosine similarities are extracted for all testing sentences where the domain-specific model performs more accurately.



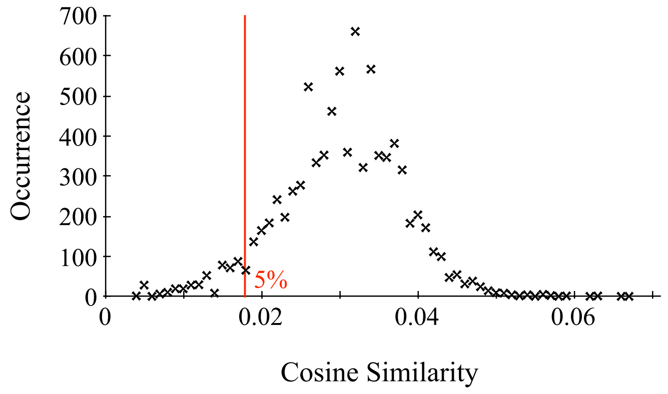


Figure 4.2: Cosine similarity distributions. The top and bottom distributions show cosine similarities extracted from the Wall Street Journal corpus and the OntoNotes corpora, respectively. The y-axis shows the number of occurrences of each cosine similarity during cross-validation.

Figure 4.2 shows the distributions of cosine similarities extracted from the Wall Street Journal corpus (top) and the OntoNotes corpora (bottom) during the cross-validation. From each distribution, the similarity at the first 5% area is chosen as the threshold; that is 0.025 for the Wall Street Journal corpus, and 0.018 for the OntoNotes corpora. This 5% area is not picked randomly but empirically derived during the cross-validation. It is possible to improve tagging accuracy by choosing different thresholds; we later found that picking the first 8% area, that is 0.022, gave the best results for the OntoNotes models. However, this threshold is not used for our experiments because it was not derived automatically, but rather ascertained by manual inspection.

4.4 Tagging algorithm

Each model uses a one-pass, left-to-right POS tagging algorithm. This algorithm tags the leftmost word token first and sequentially tags tokens on the right until all tokens are tagged. Each token is visited only once during tagging. The motivation is to analyze how our dynamic model selection works with a simple algorithm first then apply it to more sophisticated algorithms later (e.g., a bidirectional tagging algorithm). Even with this simple algorithm, our system shows comparable results against two other state-of-the-art POS taggers when coupled with the dynamic model selection.

4.5 Features

Lexical	f_{i-3}, f_{i-2}, f_{i-1}, f_i, f_{i+1}, f_{i+2}, f_{i+3}, (m_{i-2}, m_{i-1}), (m_{i-1}, m_i), (m_{i-1}, m_{i+1}), (m_i, m_{i+1}), (m_{i+1}, m_{i+2}), (m_{i-2}, m_{i-1}, m_i), (m_{i-1}, m_i, m_{i+1}), (m_i, m_{i+1}, m_{i+2}) $(m_{i-2}, m_{i-1}, m_{i+1})$, $(m_{i-1}, m_{i+1}, m_{i+2})$
POS	p_{i-3}, p_{i-2}, p_{i-1}, a_i, a_{i+1}, a_{i+2}, a_{i+3}, (p_{i-2}, p_{i-1}), (p_{i-1}, a_{i+1}), (a_{i+1}, a_{i+2}), (p_{i-2}, p_{i-1}, a_i), $(p_{i-2}, p_{i-1}, a_{i+1})$, (p_{i-1}, a_i, a_{i+1}), $(p_{i-1}, a_{i+1}, a_{i+2})$
Prefix	$c_{:1}$, $c_{:2}$, $c_{:3}$
Suffix	$c_{n:}$, $c_{n-1:}$, $c_{n-2:}$, $c_{n-3:}$
Binary	all uppercase, all lowercase, begins with a capital letter, contains a capital letter not at the beginning, contains two or more capital letters not at the beginning, contains a period, contains a number, contains a hyphen

Table 4.3: Feature templates for POS tagging. i: the index of the current word token, f: simplified word form (e.g., f_i is the simplified word form of the i'th word token), m: lowercase simplified word form, p: POS, a: ambiguity class, c_*: character sequence in w_i (e.g., $c_{:2}$: the 1st and 2nd characters of w_i, $c_{n-1:}$: the n-1'th and n'th characters of w_i).

Table 4.3 shows the feature templates used for our POS tagging experiments. Our feature templates are mostly inspired by Giménez and Màrquez (2004) except for a few changes. For lexical features, either simplified word forms or lowercase simplified word forms are used instead of the actual word forms, which provide more generalization of these features. Furthermore, ambiguity classes are derived selectively in our approach. Given a word form, we count how often each POS tag is used with this word form in training data and keep only POS tags above a certain threshold for the ambiguity class. For instance, if the threshold is 40% and a word form is used as a noun 50%, a verb 40%, and an adjective 10% in the training data, only the noun and verb POS tags are taken (e.g., `NN`, `VB`) and make up the ambiguity class of this word form, `NN_VB`. For the generalized model, a threshold of 90% is used, which favors precision more than recall. A threshold of 90% is sufficient to guarantee only one POS tag in the ambiguity class. Moreover, many forms end up not having ambiguity classes at all with this threshold. From our experiments, we find this to be more useful than expanding ambiguity classes with lower thresholds for the generalized model. For the domain-specific model, thresholds of 20% and 50% are used for the WSJ and the OntoNotes models, respectively.

4.6 Related work

Toutanova et al. (2003) introduced a tagging algorithm using bidirectional dependency networks, and showed improved results over other single directional model approaches. Giménez and Màrquez (2004) used a one-pass, left-to-right and right-to-left combined tagging algorithm and achieved near state-of-the-art performance. Shen et al. (2007) presented a tagging approach using guided learning for bidirectional sequence classification and showed current state-of-the-art performance against the other models trained and evaluated on the same data set. There are also semi-supervised learning approaches using external data that showed better performance than these supervised learning approaches (Spoustová et al., 2009; Søgaard, 2011).

Our individual models (generalized and domain-specific) are similar to Giménez and Màrquez (2004) in that we use a subset of their features and adapt a one-pass, left-to-right tagging algorithm, which is a simpler version of theirs. However, we use Liblinear for learning, which runs much faster than their classifier, Support Vector Machines, for both training and decoding. Furthermore, we use simplified word forms instead of the original word forms as features, which generalize more readily. Most importantly, we prune out lexical features using document frequencies for these individual models, which improves parsing accuracy (Section 4.7.1).

4.7 Experiments

4.7.1 Accuracy comparisons

Tables 4.4 and 4.5 show tagging accuracies of all tokens achieved by the WSJ and OntoNotes models. The Baseline and Baseline+ models are trained on features extracted from the original word forms and the lowercase simplified word forms in Section 4.5, respectively. These models take all lexical items as features regardless of their document frequencies. The Domain and General models are the domain-specific and generalized models in Section 4.3. The ClearNLP model is our dynamic model selection approach. The ClearNLP model is compared with two other state-of-the-art systems, the Stanford POS tagger (Toutanova et al., 2003) and the SVMTool (Giménez and Màrquez, 2004). Both systems are trained on the same training sets and use configurations optimized for their best results. The "G over D" row shows how often the generalized model is chosen over the domain-specific model during dynamic model selection.

Model	**BC**	**BN**	**MD**	**MP**	**MZ**	**SH**	**TC**	**WB**	**Avg$_i$**	**Avg$_o$**
Baseline	93.53	94.06	88.44	83.50	91.61	76.74	86.70	92.22	96.93	88.25
Baseline+	93.70	94.37	88.51	83.79	91.69	77.47	88.08	92.39	96.98	88.64
Domain	93.07	95.23	90.91	87.57	93.78	82.07	87.22	94.08	97.39	90.43
General	93.13	95.09	91.18	87.95	93.14	84.15	87.01	93.68	97.24	90.61
ClearNLP	**93.21**	95.37	**91.32**	88.03	**93.74**	**84.05**	**87.05**	**94.01**	97.40	**90.79**
Stanford	87.72	**95.50**	90.77	**88.45**	92.79	84.03	86.24	94.00	**97.41**	89.92
SVMTool	87.82	95.13	90.54	87.86	92.95	81.87	85.96	93.98	97.31	89.49
G over D	52.12	40.99	42.65	71.23	23.28	85.51	81.00	38.20	10.98	61.15

Table 4.4: Tagging accuracies of all tokens from the WSJ models (in %).

	OntoNotes						**Medical**			**Average**	
Model	**BC**	**BN**	**MZ**	**NW**	**TC**	**WB**	**MD**	**MP**	**SH**	**Avg$_i$**	**Avg$_o$**
Baseline	97.20	97.00	95.78	97.33	95.35	94.66	92.18	86.40	81.71	96.23	86.79
Baseline+	97.24	97.08	95.88	97.48	95.39	94.76	92.16	87.29	83.74	96.32	87.75
Domain	97.34	97.27	96.22	97.61	95.49	95.43	92.37	88.65	84.68	96.58	88.60
General	97.29	97.09	95.93	97.49	95.48	95.06	92.61	88.97	86.13	96.41	89.26
ClearNLP	**97.32**	97.26	**96.20**	97.60	95.43	**95.42**	**92.58**	89.05	**86.10**	**96.56**	**89.26**
Stanford	97.13	**97.31**	96.07	**97.61**	**95.44**	95.42	92.53	**90.03**	84.89	96.52	89.20
SVMTool	97.15	96.81	95.69	97.43	94.71	95.18	91.55	88.49	82.66	96.19	87.61
G over D	26.69	27.58	13.13	6.89	68.50	27.10	30.76	62.15	70.95	37.13	57.62

Table 4.5: Tagging accuracies of all tokens from the OntoNotes models (in %).

The Avg$_i$ and Avg$_o$ columns show the micro average accuracies for in-genre and out-of-genre experiments. For the WSJ models, the accuracy of NW is measured for Avg$_i$ and the micro average of all other corpora are measured for Avg$_o$. For the OntoNotes models, the micro averages of the OntoNotes and Medical corpora are measured for Avg$_i$ and Avg$_o$, respectively. The Baseline+ models show improvements over the Baseline models for all experiments except for MD in Table 4.5. The improvements are greater for out-of-genre experiments (e.g., 0.96% improvement for Avg$_o$ in Table 4.5), which indicates that the lowercase simplified word forms give features that generalize better for heterogeneous data. The domain-specific and generalized models perform better for in-genre and out-of-genre experiments, respectively (as expected). The ClearNLP models show the most robust results across genres.

Compared to the state-of-the-art systems, our dynamic model selection approach gives higher tagging accuracies for Avg$_o$ in Table 4.4, and for both Avg$_i$ and Avg$_o$ in Table 4.5. The differences between the ClearNLP and Stanford models in Table 4.5 are marginal. However, the difference for Avg$_o$ in Table 4.4 is statistically significant (McNemar, $p < .0001$), which implies that our approach is more effective when the training data is small and is used for tagging data with many varieties. Considering that our model uses a simple one-pass, left-to-right tagging algorithm (Section 4.4), these results are very encouraging.

Model	**BC**	**BN**	**MD**	**MP**	**MZ**	**SH**	**TC**	**WB**	**Avg$_i$**	**Avg$_o$**
ClearNLP	**69.78**	83.26	**66.90**	68.29	**74.80**	**68.21**	**26.18**	76.08	**89.11**	**66.80**
Stanford	19.24	**87.31**	64.25	**70.94**	66.32	67.35	20.64	**78.08**	88.30	61.16
SVMTool	19.08	78.35	62.94	66.53	65.23	61.45	19.62	76.43	86.88	57.70
Tokens	3,077	1,284	4,755	6,077	2,663	10,163	2,452	2,609	983	33,080

Table 4.6: Tagging accuracies of unknown tokens from the WSJ models (in %).

	OntoNotes						**Medical**			**Average**	
Model	**BC**	**BN**	**MZ**	**NW**	**TC**	**WB**	**MD**	**MP**	**SH**	**Avg$_i$**	**Avg$_o$**
ClearNLP	82.37	83.71	**87.39**	88.24	59.70	78.82	**72.32**	68.21	**73.15**	81.25	**71.48**
Stanford	**84.75**	**86.36**	87.28	**90.48**	**64.48**	**82.42**	72.11	**72.48**	69.22	**83.76**	70.79
SVMTool	80.34	80.87	84.27	85.85	55.42	75.34	65.95	64.62	63.92	78.12	64.54
Tokens	295	528	928	714	397	1,553	3,342	5,065	8,216	4,415	16,623

Table 4.7: Tagging accuracies of unknown tokens from the OntoNotes models (in %).

Tables 4.6 and 4.7 show tagging accuracies of unknown tokens achieved by the WSJ and OntoNotes models. The Tokens row shows the number of unknown tokens found in each genre. For both Avg$_i$ and Avg$_o$ in Table 4.6, the ClearNLP model shows higher accuracies than the other two systems; the difference for Avg$_o$ is statistically significant (McNemar, $p < .0001$). For Avg$_o$ in Table 4.7, the ClearNLP model also shows a higher accuracy that is statistically significant (McNemar, $p < .05$). However, the Stanford model shows an advantage over our model for Avg$_i$ in Table 4.7. We suspect that this is because the way we divide the OntoNotes training set into documents, treating each corpus as one document, does not work the best for our dynamic model selection approach such that a better document classification method is required to achieve better results. We will explore the possibility of using unsupervised clustering techniques that automatically group the training data into meaningful documents for dynamic model selection in the future.

For comparison, tagging accuracies of all tokens in MD, MP, and SH achieved by a model trained on a small amount of medical data are provided in Table 4.8. There is a total of 133,110 tokens in the training data. This model shows a micro average of 94.84% for the medical genres; the accuracy is higher for SH whose training data is larger. Notice that this model uses about $\frac{1}{15}$ the number of training instances used by the OntoNotes model, which shows an average of 89.26% on the same evaluation sets. This implies that even a small amount of training data in the same genre can significantly improve tagging accuracy.

	MD	**MP**	**SH**	**ALL**
Tokens	30,367	16,122	86,621	133,110
Accuracy	93.93	93.38	97.34	94.84

Table 4.8: Tagging accuracies achieved by the medical model. The Tokens row shows the number of tokens in the training data of each genre. The Accuracy row shows the tagging accuracy of each genre. The ALL column indicates the mixture of all three genres.

4.7.2 Speed comparisons

Tagging speeds are measured by running each system on a mixture of all data. The ClearNLP and Stanford taggers are written in Java; the Stanford POS tagger provides APIs that allow us to make fair comparisons between these two systems. The SVMTool is written in Perl. Table 4.9 shows speed comparisons between all three systems; the top and bottom three rows show results from the WSJ and OntoNotes models, respectively. The T# columns show how many milliseconds each system takes for tagging one sentence in 5 trials, the Avg column shows the average tagging speeds of the middle three trials, and the Tokens column shows how many tokens are tagged per second. Our system tags about 39.5 - 39.9K tokens per second, including the runtime for both POS tagging and dynamic model selection, which is noticeably faster than the other systems.[1]

Model		**T1**	**T2**	**T3**	**T4**	**T5**	**Avg**	**Tokens**
WSJ	ClearNLP	0.36	0.37	0.37	0.37	0.38	**0.37**	**39,491**
	Stanford	57.47	58.01	58.03	58.13	60.12	58.06	250
	SVMTool	13.10	13.40	13.83	13.91	14.07	13.71	1,058
ON	ClearNLP	0.36	0.36	0.37	0.37	0.37	**0.36**	**39,882**
	Stanford	105.19	105.52	106.55	106.95	107.09	106.34	136
	SVMTool	15.47	15.51	15.73	15.90	15.96	15.71	924

Table 4.9: Tagging speeds (in ms). The top and the bottom three rows show results from the WSJ and OntoNotes models, respectively.

[1] There is a POS tagger faster than our system, called the TnT POS tagger (Brants, 2000); however, it is written in C++ and its tagging accuracy is significantly lower than these state-of-the-art systems; thus, the TnT POS tagger is not included in our experiments.

Chapter 5

Dependency Parsing

5.1 Overview

Most current statistical dependency parsers take one of two parsing approaches. One is a transition-based approach that greedily searches for local optima (highest scoring transitions) and uses features based on parse history to predict the next transition (Nivre, 2008; Attardi and Dell'Orletta, 2009; Huang and Sagae, 2010; Goldberg and Elhadad, 2010). The other is a graph-based approach that searches for a global optimum (a maximum spanning tree) from a complete graph in which vertices represent word tokens and edges, directed and weighted, represent dependency relations (McDonald et al., 2005; Mcdonald and Pereira, 2006; Koo et al., 2010; Rush and Petrov, 2012). The transition-based approach searches for local optima, so it usually performs better on short-distance dependencies while the graph-based approach usually performs better on long-distance dependencies because it searches for a global optimum (Nivre and McDonald, 2008).[1] Some ensemble approaches using integrated models of both transition-based and graph-based approaches had also shown improved parsing results (Nivre and McDonald, 2008; Zhang and Clark, 2008; Surdeanu and Manning, 2010).

Lately, the usefulness of transition-based parsing has drawn more attention because it generally performs noticeably faster than graph-based parsing (Cer et al., 2010). Transition-based parsing has worst-case parsing complexities as low as $O(n)$ and $O(n^2)$ for the generation of projective and non-projective dependency trees, respectively, where n is the number of word tokens in

[1] The relation between far apart tokens considered in later parsing states and earlier mistakes make it harder for transition-based parsing to correctly establish long distance dependencies.

a sentence (Nivre, 2003; Covington, 2001). The parsing complexity is lower for the generation of projective dependency trees because it can deterministically drop tokens that can potentially violate the **projective** property from the search space (Section 2.1.2.1) whereas that is not advisable for the generation of non-projective dependency trees. Nevertheless, it is still possible to perform non-projective parsing in expected linear time because the amount of non-projective dependencies is notably smaller than the amount of projective dependencies (Nivre and Nilsson, 2005), so a parser can perform projective parsing for most cases and perform non-projective parsing only when it is needed (Choi and Nicolov, 2009; Nivre, 2009; Choi and Palmer, 2011a).

Another attractive aspect of transition-based parsing is that it can use features based on parse history for statistical learning, which helps make more accurate predictions (Nivre, 2006). Thus, many transition-based approaches incrementally build parsing states so that later parsing states can use parse history derived from earlier states as features. During training, this parse history is derived from gold-standard dependency trees. During decoding, however, it is derived from automatically generated dependency trees, which may not provide the same type of features as the ones used for training. By minimizing the gap between the features derived during training and decoding, it is possible to improve parsing accuracy; especially on new data where automatic parse output is poor.

This chapter focuses on the engineering of different aspects of transition-based, non-projective dependency parsing. To reduce the search space of non-projective dependency parsing, a new parsing algorithm is proposed, which combines transitions in both projective and non-projective dependency parsing algorithms (Section 5.2). To narrow down the gap between parse features derived from gold-standard and automatic dependency trees, a bootstrapping technique is introduced (Section 5.3). Additionally, a post-processing technique is suggested that guarantees an automatic parse output is a tree instead of a forest (Section 5.4). All of these approaches together show significant improvement for both in-domain and out-of-domain experiments. Our final model is evaluated on corpora from several different genres and shows comparable results in parsing accuracy and a clear advantage in parsing speed with respect to other state-of-the-art dependency parsers (Section 5.6). These results demonstrate the benefit of careful engineering that effectively leverages linguistic knowledge.

5.2 Transition-based dependency parsing

5.2.1 Transition decomposition

Table 5.1 shows functional decomposition of transitions used in Nivre's arc-eager and Covington's algorithms. Transition-based parsing can be viewed as a search problem with a very large branching factor. Nivre's arc-eager algorithm is a projective parsing algorithm that shows a worst-case parsing complexity of $O(n)$ (Nivre, 2003). Covington's algorithm is a non-projective parsing algorithm that shows a worst-case parsing complexity of $O(n^2)$ without backtracking (Covington, 2001). Covington's algorithm was later formulated as a transition-based parsing algorithm by Nivre (2008), called Nivre's list-based algorithm. Table 5.3 shows the relation between the decomposed transitions in Table 5.1 and the transitions from the original algorithms.

Operation	Transition	Current state	$\Rightarrow$	Resulting state
ARC	LEFT-$*_l$	$([\lambda_1\|i], \lambda_2, [j\|\beta], A)$	$\Rightarrow$	$([\lambda_1\|i], \lambda_2, [j\|\beta], A \cup \{i \xleftarrow{l} j\})$
	RIGHT-$*_l$	$([\lambda_1\|i], \lambda_2, [j\|\beta], A)$	$\Rightarrow$	$([\lambda_1\|i], \lambda_2, [j\|\beta], A \cup \{i \xrightarrow{l} j\})$
	NO-$*$	$([\lambda_1\|i], \lambda_2, [j\|\beta], A)$	$\Rightarrow$	$([\lambda_1\|i], \lambda_2, [j\|\beta], A)$
LIST	$*$-SHIFT$^{d\|n}$	$([\lambda_1\|i], \lambda_2, [j\|\beta], A)$	$\Rightarrow$	$([\lambda_1\|i\|\lambda_2\|j], [\ ], \beta, A)$
	$*$-REDUCE	$([\lambda_1\|i], \lambda_2, [j\|\beta], A)$	$\Rightarrow$	$(\lambda_1, \lambda_2, [j\|\beta], A)$
	$*$-PASS	$([\lambda_1\|i], \lambda_2, [j\|\beta], A)$	$\Rightarrow$	$(\lambda_1, [i\|\lambda_2], [j\|\beta], A)$

Table 5.1: Decomposed transitions grouped into the ARC and LIST operations.

Operation	Transition	Preconditions
ARC	LEFT-$*_l$	$[i \neq 0] \wedge \neg[\exists k. (i \leftarrow k) \in A] \wedge \neg[(i \rightarrow^* j) \in A]$
	RIGHT-$*_l$	$\neg[\exists k. (k \rightarrow j) \in A] \wedge \neg[(i \ {}^*\!\!\leftarrow j) \in A]$
	NO-$*$	$\neg[\exists l.$ LEFT-$*_l \vee$ RIGHT-$*_l]$
LIST	$*$-SHIFT$^{d\|n}$	$[\lambda_1 = [\ ]]^d \vee \neg[\exists k \in \lambda_1. (k \neq i) \wedge ((k \leftarrow j) \vee (k \rightarrow j))]^n$
	$*$-REDUCE	$[\exists h. (h \rightarrow i) \in A] \wedge \neg[\exists k \in \beta. (i \rightarrow k)]$
	$*$-PASS	$\neg[*$-SHIFT$^{d\|n} \vee *$-REDUCE$^*]$

Table 5.2: Preconditions of the decomposed transitions in Table 5.1.

Table 5.2 shows preconditions of the decomposed transitions in Table 5.1. Some preconditions need to be satisfied to ensure the properties of a well-formed dependency graph (Section 2.1.2.1). The parsing states are represented as tuples (λ_1, λ_2, β, A), where λ_1, λ_2 are lists of partially processed tokens and β is a list of the remaining unprocessed tokens. A is a set of labeled arcs representing previously identified dependencies. l is a dependency label, and i and j represent indices of their corresponding word tokens, w_i and w_j. $[\lambda_1|i]$ implies w_i is the last token in λ_1, and $[j|\beta]$ implies w_j is the first token in β. The initial state is ([0], [ ], $[1, \ldots, n]$, $\emptyset$). The 0 identifier corresponds to an initial token, w_0, introduced as the root of a dependency tree. The final state is (λ_1, λ_2, [ ], A), i.e., parsing terminates when all tokens in β are consumed (there is no remaining unprocessed token).

The decomposed transitions in Table 5.1 can be grouped into two operations, Arc and List. The Arc operation, consisting of the top three transitions, determines the dependency between the last token w_i in λ_1 and the first token w_j in β using an oracle O_d. In our case, O_d is a single multi-class classifier that predicts recomposed transitions in Section 5.2.2. It is possible to build two separate oracles for the Arc and List operations; we tried this approach, which did not lead to better parsing results than the single oracle approach. During training, O_d consults parse information derived from gold-standard dependency trees, whereas such information is provided by a machine learning algorithm to O_d during decoding. Left-$*_l$ is performed when O_d predicts w_j is the head of w_i with a dependency label l (notated as $i \xleftarrow{l} j$). Similarly, Right-$*_l$ is performed when O_d predicts w_i is the head of w_j with a dependency label l (notated as $i \xrightarrow{l} j$). No-$*$ is performed when O_d predicts there is no dependency between w_i and w_j. The Arc operation does not update a state of any list (λ_1, λ_2, or β); only A gets updated after these transitions are performed.

The List operation, consisting of the bottom three transitions, determines which pair of word tokens is compared next for a dependency using O_d. $*$-Shift$^{d|n}$ is performed when λ_1 is empty ($*$-Shift^{d}: deterministic shift), or O_d predicts there is no dependency between w_j and any token in λ_1 other than w_i ($*$-Shift^{n}: non-deterministic shift). After a $*$-Shift$^{d|n}$ transition, all tokens in λ_2 as well as w_j are moved to λ_1. $*$-Reduce is performed when w_i has already found its head, and O_d predicts w_i is not the head of any token in β. After a $*$-Reduce transition, w_i is removed from λ_1. $*$-Pass is performed when the conditions for both $*$-Shift$^{d|n}$ and $*$-Reduce fail. After a $*$-Pass transition, w_i is moved to the front of λ_2 so it can be compared to other tokens in β later. This transition is added for non-projective parsing where a long-distance dependency is present.

Transition decomposition as described here gives a clear distinction between the ARC and LIST operations as compared to transitions used in transition-based parsing algorithms. This decomposition makes it easier to integrate transitions from one algorithm into the other algorithm because all decomposed transitions now use the same data structures with uniform notation (e.g., Nivre's arc-eager algorithm does not use λ_2, which makes it harder to integrate its transitions into Nivre's list-based algorithm using λ_2). Section 5.2.2 shows how these decomposed transitions can be recomposed into transitions used in several different dependency parsing algorithms.

5.2.2 Transition recomposition

Any combination of two decomposed transitions in Table 5.1, one from each operation, can be recomposed into a new transition. For instance, the combination of LEFT-$*_l$ and $*$-REDUCE makes a transition, LEFT-REDUCE$_l$, which performs LEFT-$*_l$ and $*$-REDUCE sequentially; the ARC operation is always performed before the LIST operation. Table 5.3 shows how these decomposed transitions are recomposed into transitions used in different dependency parsing algorithms.

Transition	**Nivre'03**	**Covington'01**	**Nivre'08**	**C&P'11**	**This work**
LEFT-REDUCE$_l$	✓			✓	✓
LEFT-PASS$_l$		✓	✓	✓	✓
RIGHT-SHIFT$_l^n$	✓				✓
RIGHT-PASS$_l$		✓	✓	✓	✓
NO-SHIFT^d	✓	✓	✓	✓	✓
NO-SHIFT^n	✓		✓	✓	✓
NO-REDUCE	✓				✓
NO-PASS		✓	✓	✓	✓

Table 5.3: Transitions in different dependency parsing algorithms. The last column shows transitions used in our parsing algorithm. The other columns show transitions used in Nivre (2003), Covington (2001), Nivre (2008), and Choi and Palmer (2011a), respectively.

Nivre's arc-eager algorithm allows no combination of $*$-PASS, which removes or skips tokens that can violate the **projective** property (Nivre'03 in Table 5.3). As a result, this algorithm performs at most $2n-1$ transitions during paring, and can produce only projective dependency trees.[2] Covington's

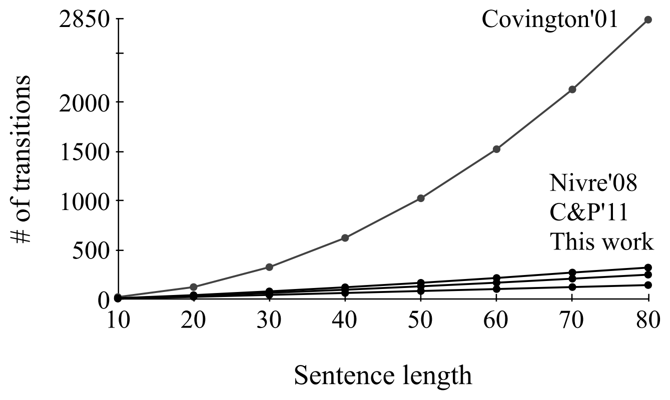
[2] The $*$-SHIFT^d transitions are not counted because they do not require comparison between word tokens.

algorithm allows no combination of $*$-SHIFT^n or $*$-REDUCE, which inevitably compares each token with all tokens prior to it (Covington'01). Thus, this algorithm performs $\frac{n(n+1)}{2}$ transitions during parsing, and can produce both projective and non-projective dependency trees.

The last three algorithms in Table 5.3 show cumulative updates to Covington's algorithm; they add one or two transitions from Nivre's arc-eager algorithm to Covington's algorithm. By adding these transitions, a fewer number of transitions needs to be performed during parsing, which reduces the expected running time of these approaches. Nivre (2008) introduced a concept of non-deterministic shift that could skip some tokens in λ_1 without comparing them to w_j (NO-SHIFT^n). Choi and Nicolov (2009) implemented this algorithm and showed that it was possible to achieve a linear time parsing speed in practice for non-projective parsing using non-deterministic shift. Choi and Palmer (2010a) added LEFT-REDUCE$_l$ to this approach and reduced the search space more.

Our parsing algorithm makes another incremental update to Choi and Palmer (2010a) by adding RIGHT-SHIFT$_l^n$ and NO-REDUCE. During training, it checks for the preconditions of all transitions and generates training instances with corresponding labels. During decoding, the oracle O_d predicts which transition to perform based on the current parsing state. With the addition of new transitions, O_d can choose either projective or non-projective parsing: by choosing the combinations with $*$-SHIFT^n or $*$-REDUCE, the algorithm performs projective parsing, whereas it performs non-projective parsing by choosing the combinations with $*$-PASS. Our experiments show that these additional transitions can reduce the expected running time as well as the feature space without compromising performance in parsing accuracy (Section 5.6.1). The advantage derives from improving the efficiency of the choice mechanism; it is now simply a transition choice and requires no additional processing.

Figure 5.1 shows the average number of transitions performed by each parsing algorithm in Table 5.3 with respect to sentence lengths. For comparison, gold-standard dependency trees in the OntoNotes Treebank are used. Covington's algorithm shows a quadratic growth in transitions as the sentence length increases while the other three approaches show more or less linear growths. The bottom figure in Figure 5.1 demonstrates reduction in the average number of transitions performed by our algorithm, in which the number of transitions grows linearly as the sentence length increases. This implies that our algorithm gives an expected linear time parsing speed for this data set although a worst-case parsing complexity is still $O(n^2)$.



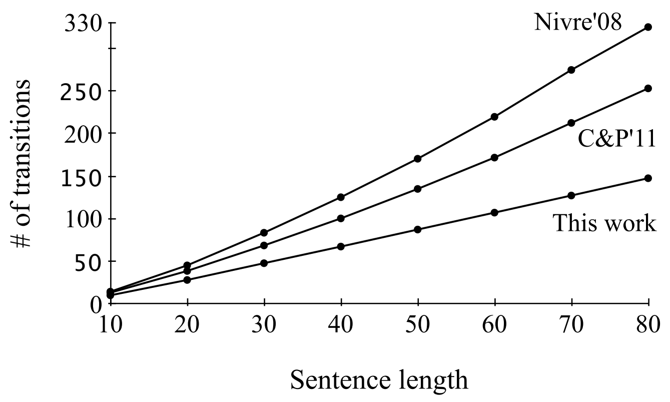


Figure 5.1: The average number of transitions performed by each algorithm in Table 5.3 with respect to sentence lengths. Each sentence length indicates the number of word tokens in a sentence. The top and the bottom figures show comparisons between the last four and three approaches in Table 5.3.

Note that Nivre (2009) introduced another transition-based parsing algorithm that could perform non-projective parsing in expected linear time. This algorithm tries to resolve non-projectivity by reordering tokens that are not within the same domain of locality. Transitions in this algorithm are not integrated into our algorithm because it deviates from a different dependency parsing algorithm (Nivre, 2004), which makes the integration difficult. We also tried to include two other transitions, LEFT-SHIFT$_l^n$ and RIGHT-REDUCE$_l$, which did not lead to better parsing results. Adding these transitions forces w_j to have at most one dependent, which is often not the case in our data.

5.2.3 Parsing algorithm

Algorithm 5.1 shows our non-projective dependency parsing algorithm using all recomposed transitions in Table 5.3.

Algorithm 5.1 : *getDependencyArcSet*(S)

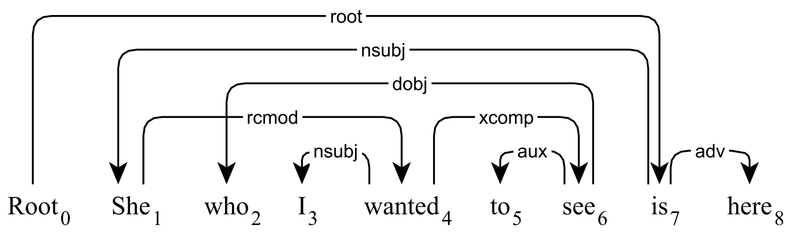
```
  Input:  A sentence $S = w_1, \ldots, w_n$.
 Output:  A set $A$ of labeled dependency arcs in $S$.
 1: $(\lambda_1, \lambda_2, \beta, A) := ([0], [\,], [1, 2, \ldots, n], \emptyset)$
 2: while $\beta \neq [\,]$ do
 3:   if $\lambda_1 = [\,]$ then
 4:     No-Shift$^d$
 5:   elif $O_d \Rightarrow (i \overset{l}{\leftarrow} j)$ then
 6:     if $i = 0$ then No-Shift$^d$
 7:     elif $(\exists k. (i \leftarrow k) \in A)$ or $((i \rightarrow^* j) \in A)$ then No-Pass
 8:     elif $O_d \Rightarrow \neg(\exists k \in \beta. (i \rightarrow k))$ then Left-Reduce$_l$
 9:     else Left-Pass$_l$
10:   elif $O_d \Rightarrow (i \overset{l}{\rightarrow} j)$ then
11:     if $(\exists k. (k \rightarrow j) \in A)$ or $((i \, {}^*\!\!\leftarrow j) \in A)$ then No-Pass
12:     elif $O_d \Rightarrow \neg(\exists k \in \lambda_1. (k \neq i) \wedge ((k \leftarrow j) \vee (k \rightarrow j)))$ then Right-Shift$^n_l$
13:     else Right-Pass$_l$
14:   else
15:     elif $O_d \Rightarrow \neg(\exists k \in \lambda_1. (k \leftarrow j) \vee (k \rightarrow j))$ then No-Shift$^n$
16:     elif $(\exists h. (h \rightarrow i) \in A)$ and $(O_d \Rightarrow \neg(\exists k \in \beta. (i \rightarrow k)))$ then No-Reduce
17:     else No-Pass
18: return $A$
```

The algorithm takes a sentence as input and produces a set of labeled dependency arcs in the sentence as output. It starts by initializing the parsing state (line 1) and terminates when β becomes empty (line 2). If λ_1 is empty, it performs No-Shift^d (lines 3-4). If O_d predicts that w_j is the head of w_i with a dependency label l (line 5), it performs either No-Shift^d if w_i is the root (line 6), No-Pass if w_i already has the head or is an ancestor of w_j (line 7), Left-Reduce$_l$ if O_d predicts that w_i is not the head of any token in β (line 8), or Left-Pass$_l$ (line 9). If O_d predicts that w_i is the head of w_j with a dependency label l (line 10), it performs either No-Pass if w_j already has the head or is an ancestor of w_i (line 11), Right-Shift^n_l if O_d predicts that there is no dependency between w_j and any token in λ_1 other than w_i (line 12), or Right-Pass$_l$ (line 13). If O_d predicts that there is no dependency between w_i and w_j (line 14), it performs No-Shift^n if O_d predicts that there is no dependency between w_j and any token in λ_1 (line 15), No-Reduce if w_i already has the head and is predicted not to be the head of any token in β by O_d (line 16), or No-Pass. Finally, the algorithm returns a set A containing all labeled dependency arcs in S.

The **root**, **single head**, and **acyclic** properties of a well-formed dependency graph in Section 2.1.2.1 are ensured by the conditions in lines 6, 7, and 11. The **connected** property, on the other hand, is ensured during post-processing (Section 5.4). Note that this algorithm performs only non-projective parsing without lines 8, 12, 15, and 16, which is equivalent to Covington's algorithm. It is worth mentioning the soundness and completeness of our parsing algorithm (Shieber et al., 1995). Let $T_k(s) = \{t_1, \ldots, t_k\}$ be a transition sequence generated by our algorithm for a sentence s, and $\mathbb{G}(T_k(s))$ be the dependency graph derived by $T_k(s)$. A transition-based dependency parsing algorithm is *sound* if and only if for every sentence s and every transition sequence $T_k(s)$, the **root**, **single head**, and **acyclic** properties are satisfied in $\mathbb{G}(T_k(s))$ (Nivre, 2008). The soundness of our algorithm can be proved by induction as follows:

- **Base**: $\mathbb{G}(T_0(s))$ satisfies all three properties for $T_1(s)$.
- **Inductive**: Assume that $\mathbb{G}(T_k(s))$ satisfies all three properties for $T_k(s)$, where $k > 1$. Let $T_{k+1}(s)$ be $\{t_1, \ldots, t_{k+1}\}$, where t_{k+1} is one of the transitions in Table 5.3. If t_{k+1} is either a Left-$*_l$ or Right-$*_l$ transition, a dependency arc δ generated by this transition preserves all three properties because of the preconditions in Table 5.2. If t_{k+1} is a No-$*_l$ transition, no dependency arc is generated. This makes $\mathbb{G}(T_{k+1}(s))$ either $\mathbb{G}(T_k(s))$ or $\mathbb{G}(T_k(s)) \cup \{\delta\}$; thus, $\mathbb{G}(T_{k+1}(s))$ satisfies all three properties.

A transition-based dependency parsing algorithm is *complete* if and only if for every sentence s and every well-formed dependency graph G for s, there is a transition sequence $T(s)$ generated by the algorithm such that $G = \mathbb{G}(T(s))$ (Nivre, 2008). The proof of completeness for our algorithm comes free from the proof of completeness for Nivre's list-based algorithm (Nivre, 2008, 5.1) because our algorithm can generate any transition sequence that Nivre's list-based algorithm generates.



	Transition	λ_1	λ_2	β	A
0	Initialization	[0]	[ ]	$[1\|\beta]$	$\emptyset$
1	NO-SHIFT^n	$[\lambda_1\|1]$	[ ]	$[2\|\beta]$	
2	NO-SHIFT^n	$[\lambda_1\|2]$	[ ]	$[3\|\beta]$	
3	NO-SHIFT^n	$[\lambda_1\|3]$	[ ]	$[4\|\beta]$	
4	LEFT-REDUCE	$[\lambda_1\|2]$	[ ]	$[4\|\beta]$	$A \cup \{3 \leftarrow\text{NSUBJ}- 4\}$
5	NO-PASS	$[\lambda_1\|1]$	[2]	$[4\|\beta]$	
6	RIGHT-SHIFT^n	$[\lambda_1\|4]$	[ ]	$[5\|\beta]$	$A \cup \{1 -\text{RCMOD}\rightarrow 4\}$
7	NO-SHIFT^n	$[\lambda_1\|5]$	[ ]	$[6\|\beta]$	
8	LEFT-REDUCE	$[\lambda_1\|4]$	[ ]	$[6\|\beta]$	$A \cup \{5 \leftarrow\text{AUX}- 6\}$
9	RIGHT-PASS	$[\lambda_1\|2]$	[4]	$[6\|\beta]$	$A \cup \{4 -\text{XCOMP}\rightarrow 6\}$
10	LEFT-REDUCE	$[\lambda_1\|1]$	[4]	$[6\|\beta]$	$A \cup \{2 \leftarrow\text{DOBJ}- 6\}$
11	NO-SHIFT^n	$[\lambda_1\|6]$	[ ]	$[7\|\beta]$	
12	NO-REDUCE	$[\lambda_1\|4]$	[ ]	$[7\|\beta]$	
13	NO-REDUCE	$[\lambda_1\|1]$	[ ]	$[7\|\beta]$	
14	LEFT-REDUCE	[0]	[ ]	$[7\|\beta]$	$A \cup \{1 \leftarrow\text{NSUBJ}- 7\}$
15	RIGHT-SHIFT^n	$[\lambda_1\|7]$	[ ]	[8]	$A \cup \{0 -\text{ROOT}\rightarrow 7\}$
16	RIGHT-SHIFT^n	$[\lambda_1\|8]$	[ ]	[ ]	$A \cup \{7 -\text{ADV}\rightarrow 8\}$

Table 5.4: Parsing states generated by Algorithm 5.1 for the example.

Table 5.4 shows a list of parsing states generated by Algorithm 5.1 for the example, assuming that the oracle O_d always makes correct decisions. After w_3 and w_4 are compared, w_3 is removed from λ_1 (state 4) so it is no longer compared to any other token in β (states 9 and 13). If w_3 were the head of any token in β, LEFT-PASS would be performed, in which case, w_3 would be moved to λ_2 instead. After w_2 and w_4 are compared, w_2 is moved to λ_2 (state 5) so it can be compared to other tokens in β (state 10). After w_1 and w_4 are compared, $*$-SHIFT^n is performed (state 6) because there is no dependency between w_4 and any other token in λ_1. On the other hand, $*$-PASS is performed after w_4 and w_6 are compared (state 9) because there is a dependency between w_6 and w_2 in λ_1 (state 10). After w_6 and w_7 are compared, w_6 is removed from λ_1 (state 12) because it is no longer needed for the later parsing states.

5.3 Bootstrapping

5.3.1 Bootstrapping parse history

Transition-based parsing, in comparison to graph-based parsing, has the advantage of using the parse history as features to make more accurate predictions without increasing parsing complexity. When w_i and w_j are compared in any parsing state (w_i is the last token in λ_1 and w_j in the first token in β; see Table 5.1), the subtree and head information of these tokens is partially provided by earlier parsing states. Figure 5.2 illustrates the range of subtree and head information given to w_i and w_j.

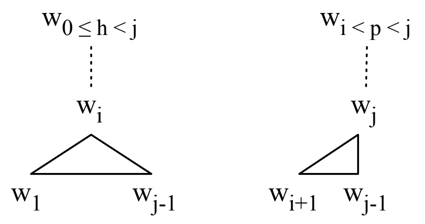


Figure 5.2: The range of subtree and head information, where $i < j$, and the dotted line indicates ancestors in the head relation.

Graph-based parsing can also take advantage of using the parse history. This is done by performing higher-order parsing, which improves parsing accuracy but also increases parsing complexity (Carreras, 2007; Koo and Collins, 2010; Rush and Petrov, 2012).[3] Transition-based parsing is attractive because it can use parse information from earlier parsing states without increasing parsing complexity. The qualification is that parse information provided by gold-standard dependency trees during training may not give the same type of features provided by automatically generated dependency trees during decoding. This can confuse a statistical model trained on features derived only from the gold-standard trees. The problem becomes more severe as the discrepancies between gold-standard and automatically generated trees get larger.

To reduce the gap between gold-standard and automatically generated trees, a bootstrapping technique is applied during training. First, a statistical parsing model is trained using gold-standard

[3] Higher-order, non-projective, graph-based dependency parsing is NP-hard without performing approximation.

trees. Next, the training data is parsed using this model. During parsing, new features are extracted for each parsing state, which consist of parse information derived from an automatically generated tree, then a training instance is created by joining these features with a gold-standard label. The gold-standard label is achieved by consulting the relation between w_i and w_j in the gold-standard tree. Figure 5.3 shows an automatically generated tree for the example in Table 5.4, where the `DOBJ` dependency is incorrectly predicted. Given a parsing state where w_i = *wanted* and w_j = *see*, the oracle O_d would predict a RIGHT-SHIFT^n transition. However, according to the gold-standard tree, this is not the correct transition to perform; a RIGHT-PASS transition should be performed given this parsing state (state 9 in Table 5.4). Thus, the gold-standard label, RIGHT-PASS, and features extracted from the tree in Figure 5.3 are joined to create a training instance for this state. The parsing state is then updated with the incorrectly predicted transition, RIGHT-SHIFT^n, so the algorithm can continue parsing (with the incorrect parse).

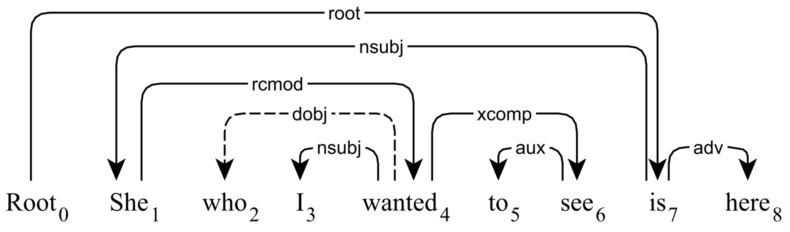


Figure 5.3: An automatically generated tree for the example in Table 5.4. The `DOBJ` dependency is incorrectly predicted compared to the gold-standard tree in Table 5.4.

When the parsing is done, a different parsing model is built using the training instances induced by the previous model. This procedure is repeated until a certain stopping criterion is met, which is determined by cross-validation. For each iteration, cross-validation is performed to check if the average parsing accuracy of the current cross-validation set is higher than the one from the previous iteration. The procedure is stopped when the parsing accuracy on the cross-validation set starts decreasing. During decoding, only the last model is used for parsing. Figure 5.4 shows a flowchart of our bootstrapping technique.

The concept of using automatically generated data for training is not new; it is common to use automatically generated POS tags for training, which is known to be more useful than using gold-standard POS tags for dependency parsing. Here, we go one step farther by bootstrapping parse information. Our experiments show that this bootstrapping technique gives a significant improvement to parsing accuracy (Section 5.6.1). This technique is not limited to transition-based dependency parsing but can be applied to other NLP components such as a POS tagger or an NP chunker that make incremental updates in predictions.

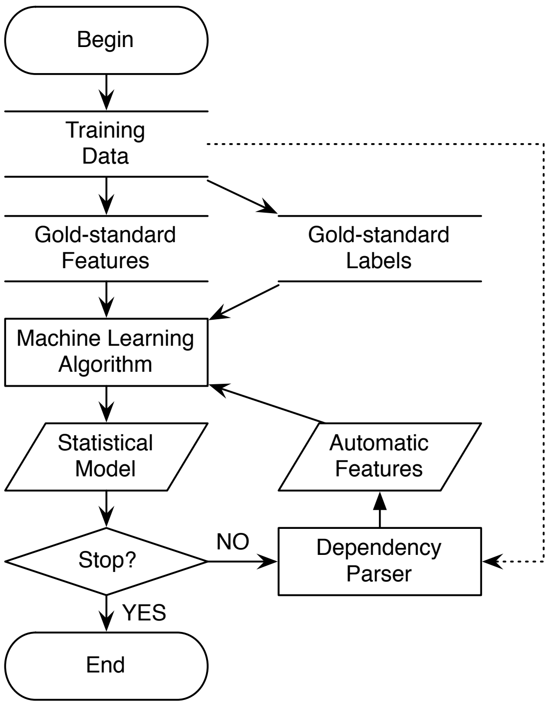


Figure 5.4: A flowchart of our bootstrapping technique.

5.3.2 Related work

Daumé et al. (2009) presented a learning algorithm, called SEARN, for integrating search and learning to solve complex structured prediction problems. Our bootstrapping technique can be viewed as a simplified version of SEARN. During training, SEARN iteratively creates a set of new cost-sensitive examples using a known policy. In our case, the new examples are training instances consisting of parse information derived from automatically generated trees induced by the previous model. Our technique is simplified because the new examples are not cost-sensitive. Furthermore, SEARN interpolates the current policy with the previous policy whereas we do not perform such interpolation. During decoding, SEARN generates a sequence of decisions and makes a final prediction. In our case, the decisions are predicted dependency arcs and the final prediction is a dependency tree. SEARN has been successfully adapted to several NLP tasks such as named entity recognition, syntactic chunking, and POS tagging. To the best of our knowledge, this is the first time that this idea has been applied to transition-based dependency parsing and shown improved results.

Brill (1995) introduced transformation-based learning that starts with simple rules for predicting certain values (e.g., POS tags, transitions) using baseline features, applies transformations to the features using a set of rule templates, reconstructs the rules using the transformed features, and selects transformations that derive the best rules for predicting the values. The transformations are applied repeatedly until they do not affect or hurt the performance. Transformation-based learning is similar to our bootstrapping approach in the sense that it iteratively transforms features to achieve better results and stops when no more benefit is found from the transformations.

Zhang and Clark (2008) suggested a transition-based projective dependency parsing algorithm that kept B different sequences of parsing states and chose the one with the highest score. They used beam search and showed a worst-case parsing complexity of $O(n)$ given a fixed size of beam. Their learning mechanism, using the structured perceptron algorithm, involves training on automatically derived parsing states that closely resemble potential states encountered during our decoding.

5.4 Post-processing

Just like many other transition-based dependency parsing algorithms, our parsing algorithm does not guarantee that the parse output is a connected tree. This implies that after the algorithm is finished, there can be headless tokens. For each token w_p for which the algorithm could not find the head during parsing, the same oracle O_d is used again to predict the head of w_p but this time, w_p is compared to all other tokens which may have been skipped during parsing. For each comparison, the oracle keeps track of the head candidate with the highest score and makes it the head of w_p. Any dependency which creates a cyclic relation is avoided during post-processing. Although this post-processing technique is simple, it not only ensures the **connected** property (Section 2.1.2.1), but also improves parsing accuracy (Section 5.6.1).

The *postProcess*(D) method in Algorithm 5.2 takes a dependency forest D and adds dependency arcs for all headless tokens in D. The head of each headless token is initialized to the artificial root, w_0 (line 3). If the oracle O_d predicts a dependency arc between w_i and w_j with a label l and a score s_i $(0 \leq i < j)$ and if w_j is not an ancestor of w_i, a dependency arc with the highest score is kept (lines 4-8). Similarly, if O_d predicts a dependency arc between w_j and w_k with a label l and a score s_k $(j < k \leq n)$ and if w_j is not an ancestor of w_k, a dependency arc with the highest score is kept (lines 9-13). The dependency arc with the overall highest score is added to A as the head of w_j. Notice that the *postProcess*(D) method process headless tokens left-to-right. The final parse output may vary if headless tokens are processed right-to-left, which we will explore in the future.

Algorithm 5.2 : *postProcess*(D)

```
Input:  A dependency forest $D$.
1:  $N \leftarrow$ an ordered list of headless tokens in $D$     # e.g,. $N \leftarrow [w_2, w_3, w_5]$
2:  for $w_j$ in $N$ do
3:      $(a_m, s_m) \leftarrow (0 \xrightarrow{l} j, 0)$, where $l \leftarrow$ ROOT
4:      $i \leftarrow (j-1)$
5:      while $i \geq 0$ do     # 0 is the ID of the artificial root
6:          if $(O_d \Rightarrow (i \xrightarrow{l} j, s_i))$ and $((i \overset{*}{\leftarrow} j) \notin A)$ and $(s_i > s_m)$ then
7:              $(a_m, s_m) \leftarrow (i \xrightarrow{l} j, s_i)$
8:          $i \leftarrow (i-1)$
9:      $k \leftarrow (j+1)$
10:     while $k \leq n$ do     # $n$ is the ID of the rightmost token in $D$
11:         if $(O_d \Rightarrow (j \xleftarrow{l} k, s_k))$ and $((j \overset{*}{\rightarrow} k) \notin A)$ and $(s_k > s_m)$ then
12:             $(a_m, s_m) \leftarrow (j \xleftarrow{l} k, s_k)$
13:         $k \leftarrow (k+1)$
14:     $A \leftarrow (A \cup a_m)$
```

5.5 Features

Table 5.5 shows the feature templates used for our dependency parsing experiments. $\lambda_1[0]$ and $\beta[0]$ are the last token in λ_1 and the first token in β, which are equivalent to w_i and w_j in Table 5.1, respectively. $\lambda_1[n]$ and $\beta[n]$ are the n'th to the last token in λ_1 and the n'th token in β, respectively. w_{x+n} is the token whose distance from w_x is n. For instance, w_{i-1} and w_{i+1} are the tokens prior and next to w_i, respectively. Note that w_{i-n} may or may not be equivalent to $\lambda_1[n]$. It is possible that w_{i-n} is already removed by the $*$-Reduce transition from a previous parsing state (Section 5.2.1), so it no longer exists in any list and can only be retrieved from the original sentence. $hd(x)$ stands for the head of x. $ld(x)$ and $rd(x)$ stand for the leftmost and rightmost dependents of x, respectively.

Token	**Form**	**Lemma**	**POS**	**Deprel**
$\lambda_1[0]$	✓	✓	✓	✓
$\lambda_1[1]$		✓	✓	
$\lambda_1[2]$		✓	✓	
$hd(\lambda_1[0])$		✓	✓	
$ld(\lambda_1[0])$			✓	✓
$rd(\lambda_1[0])$				✓
$\beta[0]$	✓	✓	✓	
$\beta[1]$	✓	✓	✓	
$\beta[2]$	✓		✓	
$\beta[3]$			✓	
$ld(\beta[0])$			✓	
w_{i-2}	✓		✓	
w_{i-1}	✓		✓	
w_{i+1}	✓		✓	
w_{i+2}	✓		✓	
w_{j-2}	✓		✓	
w_{j-1}	✓		✓	

Table 5.5: Feature templates for dependency parsing.

The Form, Lemma, POS, and Deprel columns indicate the word form, lemma, POS tag, and dependency label features of the corresponding token. These features are used either individually or jointly (e.g., POS tags of both $\lambda_1[0]$ and $\beta[0]$ make one feature). Additionally, three binary features are used, which check if w_i is the leftmost token in the original sentence, if w_j is the rightmost token in the original sentence, and if w_i and w_j are adjacent.

5.6 Experiments

5.6.1 Accuracy comparisons

Tables 5.6 and 5.7 show labeled attachment scores for all trees achieved by the WSJ and OntoNotes models, respectively. The Baseline model uses the parsing algorithm in Section 5.2.3. The Baseline+ model is the Baseline model with the post-processing in Section 5.4. The ClearNLP model is the Baseline+ model with the bootstrapping technique in Section 5.3. The C&N'09 and C&P'11 models are the transition-based dependency parsing approaches introduced by Choi and Nicolov (2009) and Choi and Palmer (2011a), respectively (see Section 5.2.2). These models use the same machine learning algorithm (Liblinear L2-L1 SVM, see Section 3.2), post-processing, and bootstrapping technique as the ClearNLP model. We use feature templates optimized for the best results achieved with the CoNLL'09 shared task data for English (Hajič et al., 2009).

The ClearNLP model is compared against two state-of-the-art dependency parsers, MaltParser and MSTParser. For MaltParser, a non-projective transition-based parsing algorithm using SWAP transitions is used for parsing (Nivre, 2009) and Liblinear multi-class support vector classification is used for learning (Crammer and Singer, 2002). We use feature templates provided by the MaltParser team instead of their default templates, which show improved parsing accuracy. For MSTParser, a non-projective graph-based parsing algorithm, Chu-Liu-Edmonds algorithm, is used for parsing (McDonald et al., 2005) and the margin infused relaxed algorithm (MIRA) is used for learning (Crammer and Singer, 2003). Only 1st-order features are used for MSTParser; 2nd-order features create a huge feature space for building the OntoNotes model, requiring too much memory for our machine. Thus, only the 1st-order features are used for our experiments; therefore, one must consider that parsing accuracy is expected to be higher but parsing speed is expected to be slower with the 2nd order features for MSTParser.[4] Note that neither MaltParser nor MSTParser have been optimized for our experiments. Their results are there to give a sense of how well our approach performs against these state-of-the-art parsers off-the-shelf but not for absolute comparisons.

[4] See Mcdonald and Pereira (2006) for more details about the 2nd order features.

Model	BC	BN	MD	MP	MZ	SH	TC	WB	Avg$_i$	Avg$_o$
Baseline	75.38	80.75	78.34	71.49	80.19	60.50	68.58	78.68	86.94	74.18
Baseline+	75.73	81.13	78.69	72.10	80.64	60.77	69.82	78.98	87.18	74.68
ClearNLP	**76.04**	**82.02**	**79.07**	**72.84**	**81.30**	63.44	70.19	**79.56**	**88.10**	**75.50**
C&N'09	75.85	81.54	78.76	**72.84**	80.86	63.61	69.64	79.13	87.79	75.23
C&P'11	75.96	81.55	79.00	72.80	81.10	63.55	69.70	79.45	88.03	75.34
MaltParser	73.53	80.39	77.54	72.46	79.77	63.26	68.28	77.86	86.49	74.10
MSTParser	74.99	79.84	77.34	72.39	79.31	**63.85**	**70.25**	78.09	86.03	74.46

Table 5.6: Labeled attachment scores for all trees from the WSJ models (in %).

	OntoNotes						**Medical**			**Average**	
Model	**BC**	**BN**	**MZ**	**NW**	**TC**	**WB**	**MD**	**MP**	**SH**	**Avg$_i$**	**Avg$_o$**
Baseline	85.54	85.00	85.26	87.56	81.36	81.89	80.81	74.10	61.88	84.51	72.37
Baseline+	85.67	85.18	85.48	87.77	81.89	82.13	81.15	74.56	62.17	84.76	72.73
ClearNLP	86.35	**85.96**	**86.33**	**89.00**	**83.39**	**82.58**	**81.72**	**75.57**	**64.98**	**85.68**	**74.18**
C&N'09	**86.40**	85.75	85.84	88.66	83.00	82.33	81.26	75.14	64.83	85.41	73.83
C&P'11	86.25	85.74	86.12	88.96	83.03	82.31	81.53	75.02	64.80	85.49	73.86
MaltParser	84.76	84.22	84.78	87.45	81.76	80.87	80.27	75.14	64.73	84.05	73.47
MSTParser	84.39	83.69	84.15	87.11	81.23	80.85	80.27	74.40	65.01	83.66	73.30

Table 5.7: Labeled attachment scores for all trees from the OntoNotes models (in %).

The Avg$_i$ and Avg$_o$ columns show the micro average labeled attachment scores for in-genre and out-of-genre experiments. For the WSJ models, the score of NW is measured for Avg$_i$ and the micro average of all other corpora are measured for Avg$_o$. For the OntoNotes models, the micro averages of the OntoNotes and Medical corpora are measured for Avg$_i$ and Avg$_o$, respectively. The Baseline+ models show improvements over the Baseline models for all experiments. The ClearNLP models show additional improvements over the Baseline+ models; the improvements are greater for out-of-genre experiments (e.g., 1.45% improvement for Avg$_o$ in Table 5.7), which indicates that our bootstrapping technique is more effective where automatic parse results are poor. The ClearNLP models show higher scores than the C&N'09 and C&P'11 models for both in-genre and out-of-genre experiments; all differences are statistically significant (McNemar, $p < 0.01$) except for the one between the C&P'11 and ClearNLP models for Avg$_i$ in Table 5.6. More importantly, the ClearNLP model uses about $\frac{1}{5}$ the number of features used by the C&N'09 and C&P'11 models (e.g., for the OntoNotes models, 5.43M, 4.84M, and 0.97M features are used by the C&N'09, C&P'11, and ClearNLP models, respectively), indicating that the ClearNLP model can perform as accurately as the other two models using a much smaller set of features. Compared to MaltParser and MSTParser, the ClearNLP models again show higher scores for both in-genre and out-of-genre experiments. We believe that it is possible to improve MaltParser's parsing accuracy by applying our bootstrapping and post-processing techniques, which can bring its performance closer to ours, whereas it is more difficult to apply these techniques to MSTParser, which is based on a graph-based parsing algorithm.

Model	**BC**	**BN**	**MD**	**MP**	**MZ**	**SH**	**TC**	**WB**	**Avg$_i$**	**Avg$_o$**
Baseline	79.20	83.34	81.88	75.88	83.42	65.14	74.36	81.46	88.57	78.04
Baseline+	79.60	83.80	82.24	76.57	83.90	65.44	75.82	81.83	88.81	78.60
ClearNLP	79.90	**84.60**	**82.53**	77.10	**84.41**	68.13	76.12	**82.46**	89.68	**79.36**
C&N'09	79.79	84.32	82.07	77.36	84.04	68.01	75.43	81.97	89.50	79.08
C&P'11	**79.91**	84.27	82.36	77.16	84.34	67.91	75.55	82.31	**89.74**	79.18
MaltParser	77.89	83.29	81.17	77.33	83.23	67.88	74.69	81.14	88.23	78.29
MSTParser	79.84	83.55	81.84	**77.91**	83.44	**68.74**	**77.15**	81.92	88.36	79.26

Table 5.8: Unlabeled attachment scores for all trees from the WSJ models (in %).

	OntoNotes						**Medical**			**Average**	
Model	**BC**	**BN**	**MZ**	**NW**	**TC**	**WB**	**MD**	**MP**	**SH**	**Avg$_i$**	**Avg$_o$**
Baseline	87.72	86.89	87.40	88.87	83.95	84.10	83.48	78.38	66.60	86.54	76.26
Baseline+	87.87	87.12	87.68	89.11	84.55	84.40	83.86	78.87	66.90	86.83	76.65
ClearNLP	88.47	**87.86**	**88.49**	90.32	**86.17**	**84.87**	**84.32**	**79.89**	69.67	**87.75**	**78.05**
C&N'09	**88.58**	87.71	87.90	90.06	85.61	84.67	83.75	78.96	69.32	87.48	77.43
C&P'11	88.45	87.70	88.20	**90.36**	85.62	84.67	84.02	78.69	69.26	87.57	77.40
MaltParser	87.27	86.44	87.06	89.06	84.73	83.47	83.10	79.59	69.67	86.40	77.54
MSTParser	87.49	86.67	87.10	89.23	85.29	84.08	83.72	79.49	**70.36**	86.70	77.94

Table 5.9: Unlabeled attachment scores for all trees from the OntoNotes models (in %).

Tables 5.8 and 5.9 show unlabeled attachment scores for all trees achieved by the WSJ and OntoNotes models, respectively. The ClearNLP model shows the most robust results across genres. For all out-of-genre experiments, the ClearNLP models show higher scores than the other models. For in-genre experiments, the ClearNLP model shows a higher score than all other models in Table 5.9; however, the C&P'11 model shows a higher score than the ClearNLP model in Table 5.8. We suspect that this is because the additional features used by the C&P'11 model help make more accurate predictions for Avg$_i$ in Table 5.8, which is trained and evaluated on the same corpus (WSJ), whereas they give less impact on other corpora. Notice that MSTParser consistently performs more accurately than MaltParser except for MP in Table 5.9, which is not the case for labeled attachment scores in Tables 5.6 and 5.7. This implies that graph-based parsing has an advantage in finding arcs whereas transition-based parsing has an advantage in finding labels.

Tables 5.10 and 5.11 show labeled attachment scores and Tables 5.12 and 5.13 show unlabeled attachment scores for non-projective trees achieved by the WSJ and OntoNotes models, respectively. A non-projective tree is a dependency tree consisting of at least one non-projective dependency. The Trees row shows the number of non-projective trees in each genre. There are not many non-projective trees in our corpora; English is a rigid word-order language that does not contain many non-projective dependencies. In the future, we will explore the possibility of comparing parse results for non-projective trees with languages such as Czech (Hajič et al., 2000), Danish (Kromann, 2003), and Sloven (Džeroski et al., 2006) that contain more non-projective dependencies. Note that our previous approach, C&P'11, was evaluated on the CoNLL'09 shared task data for Czech and showed the second highest score against other parsers (Choi and Palmer, 2011a). Considering that our current approach, ClearNLP, gives comparative results to our previous approach, it is expected to perform well for Czech and other languages as well.

Model	**BC**	**BN**	**MD**	**MP**	**MZ**	**SH**	**TC**	**WB**	**Avg**$_i$	**Avg**$_o$
ClearNLP	75.43	**76.02**	75.89	74.23	**75.56**	72.35	65.46	**73.78**	82.25	**73.87**
C&N'09	**75.53**	75.28	**76.32**	**74.50**	73.85	**72.97**	64.80	72.95	82.83	73.41
C&P'11	75.24	75.42	76.00	72.42	75.23	72.56	**66.45**	**73.78**	**83.07**	73.67
MaltParser	72.64	74.68	75.16	73.06	75.03	70.27	61.90	71.80	80.89	71.94
MSTParser	73.52	73.14	74.32	69.17	73.72	71.31	63.43	72.63	80.99	71.67
Trees	109	94	28	45	51	22	96	65	65	510

Table 5.10: Labeled attachment scores for non-projective trees from the WSJ models (in %).

	OntoNotes						**Medical**			**Average**	
Model	**BC**	**BN**	**MZ**	**NW**	**TC**	**WB**	**MD**	**MP**	**SH**	**Avg**$_i$	**Avg**$_o$
ClearNLP	82.28	79.77	**80.11**	**86.47**	76.37	**77.69**	78.74	**76.13**	76.09	**80.61**	77.10
C&N'09	**82.97**	**80.23**	79.64	85.26	76.54	76.36	80.11	75.59	**76.30**	80.45	77.41
C&P'11	82.38	79.60	79.51	85.55	**76.92**	77.37	**80.32**	76.04	75.68	80.42	**77.57**
MaltParser	79.67	77.63	78.33	84.68	73.74	76.77	77.37	75.77	72.14	78.58	75.68
MSTParser	79.20	76.96	76.15	83.07	73.41	77.23	77.26	72.88	75.26	77.88	74.97
Trees	109	94	51	65	96	65	28	45	22	480	95

Table 5.11: Labeled attachment scores for non-projective trees from the OntoNotes models (in %).

Model	BC	BN	MD	MP	MZ	SH	TC	WB	Avg$_i$	Avg$_o$
ClearNLP	78.79	**79.06**	77.58	76.67	78.52	75.47	71.05	76.68	84.38	**77.15**
C&N'09	**78.89**	78.43	**77.79**	**77.31**	77.40	**76.09**	69.79	75.85	84.92	76.71
C&P'11	78.42	78.70	77.58	75.23	**79.12**	75.68	**71.71**	76.45	**85.50**	76.99
MaltParser	76.53	78.29	**77.79**	76.67	78.79	74.01	68.26	75.16	83.32	75.88
MSTParser	77.16	77.49	77.26	73.78	78.46	75.26	69.85	**77.14**	83.51	76.11

Table 5.12: Unlabeled attachment scores for non-projective trees from the WSJ models (in %).

	OntoNotes						**Medical**			**Average**	
Model	**BC**	**BN**	**MZ**	**NW**	**TC**	**WB**	**MD**	**MP**	**SH**	**Avg$_i$**	**Avg$_o$**
ClearNLP	84.13	82.47	**82.48**	**88.12**	78.95	79.67	80.74	**79.11**	79.00	**82.79**	79.70
C&N'09	**84.98**	**82.98**	82.35	86.81	78.67	78.61	81.89	78.12	**79.63**	82.68	79.82
C&P'11	84.39	82.01	82.15	87.15	**79.44**	79.67	**82.11**	78.93	78.79	82.63	**80.09**
MaltParser	82.09	80.47	81.95	86.71	76.70	79.21	79.58	78.48	76.09	81.25	78.44
MSTParser	82.09	80.57	79.64	85.26	77.30	**81.05**	80.11	76.22	**79.63**	81.17	78.32

Table 5.13: Unlabeled attachment scores for non-projective trees from the OntoNotes models (in %).

The ClearNLP model shows the highest score for Avg$_o$ whereas the C&P'11 model shows the highest score for Avg$_i$ in Tables 5.10 and 5.12. However, their results are reversed in Tables 5.11 and 5.13; the ClearNLP model shows the highest score for Avg$_i$ whereas the C&P'11 model shows the highest score for Avg$_o$. This is probably because the ClearNLP model makes early $*$-Shift and $*$-Reduce transitions that hurt performance in finding some non-projective dependencies, but more thorough study needs to be done to make sure which transitions help or hurt performance in finding what kind of non-projective dependencies.

5.6.2 Speed comparisons

Parsing speeds are measured by running each system on a mixture of all data. All systems are written in Java; C&N'09 and C&P'11 are implemented in the same project called ClearParser,[5] and the other systems are implemented in their own projects. Table 5.14 shows speed comparisons between all five systems; the top and bottom five rows show results from the WSJ and OntoNotes models, respectively. The T# columns show how many milliseconds each system takes for parsing one sentence in 5 trials, and the Avg column shows the average parsing speeds of the middle three

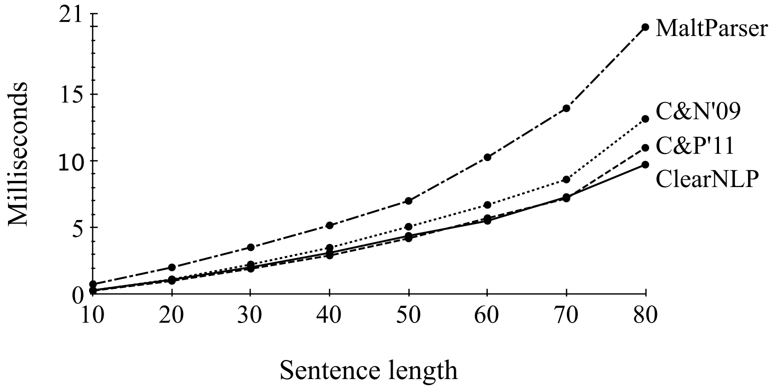
[5] ClearParser: `clearparser.googlecode.com`

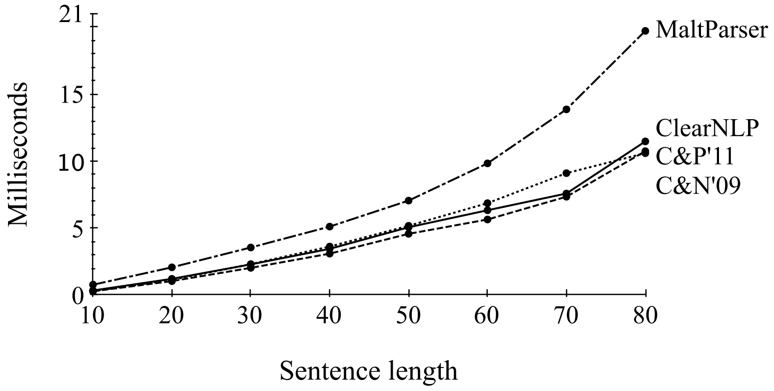
trials, and the Trees column shows how many trees are parsed per second. The transition-based parsing approaches, the top four systems, take about 1 - 2 milliseconds per sentence, whereas the graph-based parsing approach, MSTParser, shows over 30 times slower parsing speeds than the transition-based parsing approaches. Our system takes about 1.16 - 1.28 milliseconds per sentence, which is fast but not as fast as C&P'11. This contradicts our transition comparison in Figure 5.1; it is caused by an implementation difference between ClearParser and ClearNLP. ClearParser assumes input and output to be a tree so it is optimized for generating a tree structure whereas ClearNLP does not make such an assumption so that it is designed for generating a graph structure, which is more complicated and gives higher overheads during runtime.

Model		**T1**	**T2**	**T3**	**T4**	**T5**	**Avg**	**Trees**
WSJ	ClearNLP	1.15	1.15	1.16	1.16	1.20	**1.16**	**865**
	C&N'09	1.23	1.24	1.25	1.25	1.39	1.25	801
	C&P'11	1.05	1.06	1.09	1.11	1.22	1.08	922
	MaltParser	2.13	2.14	2.14	2.15	2.19	2.14	467
	MSTParser	61.71	62.44	62.55	62.62	62.79	62.38	16
ON	ClearNLP	1.28	1.28	1.28	1.29	1.30	**1.28**	**780**
	C&N'09	1.23	1.23	1.27	1.27	1.46	1.26	795
	C&P'11	1.06	1.07	1.08	1.21	1.22	1.12	892
	MaltParser	2.12	2.13	2.14	2.16	2.17	2.14	467
	MSTParser	65.60	66.36	66.77	67.73	67.79	66.62	15

Table 5.14: Parsing speeds (in ms.). The top and the bottom five rows show results from the WSJ and OntoNotes models, respectively.

Figure 5.5 shows average parsing speeds with respect to sentence lengths achieved by the WSJ (the top figure) and OntoNotes (the bottom figure) models. MSTParser is excluded from this comparison because its parsing speed is noticeably slower than the other four systems. All four systems show similar growth rates that are close to linear although some systems begin to show curves towards the end (e.g., sentence groups 70 and 80). Considering there are not many sentences in these groups, all four systems show linear parsing speeds in practice. Notice that ClearNLP and MaltParser show very similar growth rates with the OntoNotes models whereas that is not really the case with the WSJ models. This is probably because they start performing a similar number of non-projective transitions (∗-Pass for ClearNLP and Swap for MaltParser) with the OnteNotes models where non-projective dependencies are classified more accurately.





Group	≤ 10	≤ 20	≤ 30	≤ 40	≤ 50	≤ 60	≤ 70	≤ 80
Sentences	9,789	6,159	2,966	1,313	479	178	75	30

Figure 5.5: Average parsing speeds with respect to sentence lengths. The top and the bottom figures show parsing speeds achieved by the WSJ and OntoNotes models, respectively. The bottom table shows the number of sentences in each group. –·–·–: MaltParser, -------: C&N'09, ·········: C&P'11, ———: ClearNLP.

Chapter 6

Semantic Role Labeling

6.1 Overview

Recently, dependency-based semantic role labeling has demonstrated two advantages over constituent-based semantic role labeling. First, semantic role labeling is often performed after syntactic parsing, either dependency parsing or constituent parsing, because it uses syntactic parses as its input. Syntactic parsing is usually considered a bottleneck to semantic role labeling in terms of execution time; however, since dependency parsing is much faster than constituent parsing (Cer et al., 2010), this becomes less of an issue. Second, dependency structures are more similar to predicate argument structures than constituent structures because they can directly define relations between predicates and arguments with labeled arcs (see Figures 6.2 and 6.3). Unlike constituent-based SRL that maps phrases to semantic roles, dependency-based SRL maps head-tokens to semantic roles and considers the subtrees of the head-tokens as arguments. This may lead to a concern about getting the actual semantic chunks back, but previous studies have shown that it is possible to recover the original chunks from the head-tokens with minimal loss (Ekeklint and Nivre, 2007; Choi and Palmer, 2010a).

We use predicate argument structures in PropBank for our experiments (Palmer et al., 2005). For both constituent-based and dependency-based SRL, it is possible to reduce the search space by pruning argument candidates. This is because the PropBank guidelines generally restrict argument candidates to constituents within the same domain of locality as their predicates (e.g., siblings of the predicates in constituent trees, direct dependents of the predicates in dependency trees); thus, a semantic role labeler does not need to search beyond these candidates for argument identification.[1] This pruning strategy works well when semantic role labeling is performed on gold-standard trees; however, it does not work as well when automatically generated trees are used. This motivates the development of an enhanced pruning algorithm that gives better recall for argument identification, yet effectively prunes out argument candidates.

Traditionally, semantic role labeling is done in two steps, argument identification and argument classification (Gildea and Jurafsky, 2002; Johansson and Nugues, 2008). Argument identification is the task of finding arguments of each predicate, and argument classification is the task of assigning a semantic role to each argument with respect to the predicate. This is from a general belief that each task requires a different set of features (Xue and Palmer, 2004), and training these tasks in a pipeline takes less time than training them as a joint-inference task. However, recent machine learning algorithms can train a large scale feature space quickly (Hsieh et al., 2008). Furthermore, Choi and Palmer (2011b) have shown that this joint-inference approach performs as accurately as the pipeline approach without using two separate sets of features. Thus, argument identification and classification is performed jointly in our approach.

This chapter first illustrates how PropBank predicate argument structures, annotated on constituent trees, can be mapped to dependency trees (Section 6.2). An enhanced argument pruning algorithm, called a conditional higher-order argument pruning algorithm, is introduced next (Section 6.3). Our experiments show that this new pruning algorithm improves both argument identification and classification, yet keeps the average number of argument candidates close to the original pruning algorithm (Section 6.5).

[1] There are exceptional cases such as verb-particle constructions, prepositional phrases, coordinations, etc.

6.2 Semantic roles in dependency structure

6.2.1 Predicate argument structures in PropBank

PropBank is a corpus in which arguments are annotated with the semantic roles they play with respect to their predicates (Palmer et al., 2005). Predicates can be verbs, including light verbs (Hwang et al., 2010), nouns, or even adjectives (Hawwari et al., 2011). Each predicate encompasses one or more senses defining their own predicate argument structures in PropBank. For example, the verb predicate *open* contains three senses, {`open.01`, `open.02`, `open.03`}, and each sense defines its own argument structure (Figure 6.1). Thus, the verb *open* in "He *opened* the door with his foot at ten" takes the first sense, `open.01`, and forms the argument structure described in Figure 6.2.

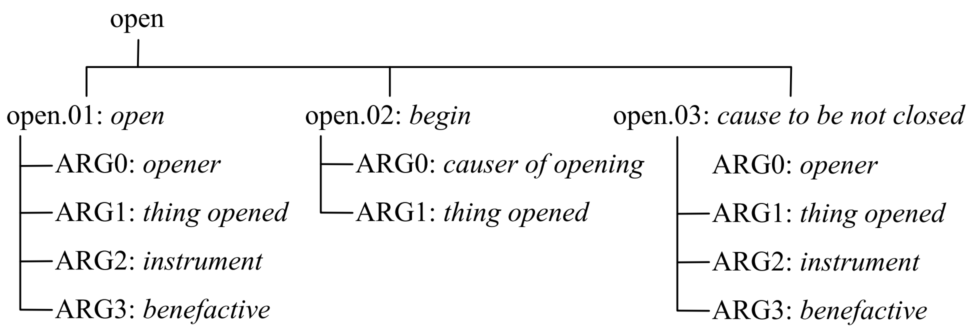


Figure 6.1: Three senses of the verb *open* and their argument structures in PropBank. The sense `open.03` is for a verb-particle construction, *open up*.

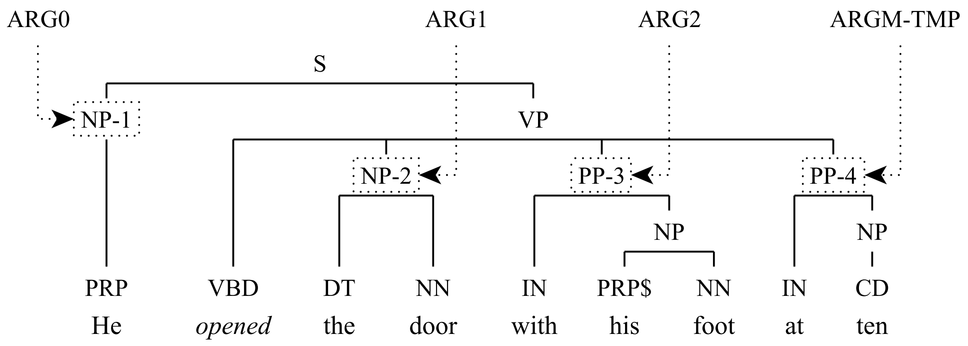


Figure 6.2: An example of a PropBank predicate argument structure. The verb predicate *open* is annotated with the sense `open.01` and the argument structure containing three numbered arguments, `ARG0`, `ARG1`, and `ARG2`, and one modifier, `ARGM-TMP`.

PropBanks in most languages are annotated on top of constituent Treebanks except for Hindi where annotation is done on a dependency Treebank (Vaidya et al., 2011). When a constituent Treebank is used, each argument is annotated on one or more phrasal nodes (indicated by dotted boxes in Figure 6.2). `ARG0` and `ARG1` represent an agent and a theme, respectively. `ARG[0-4]`, called numbered arguments, are elements that frequently co-occur with their predicates. These numbered arguments do not always correspond to core arguments. For instance, PropBank annotates *with his foot* as a numbered argument, `ARG2` (an instrument), which is considered an oblique argument in some other Linguistic theories (Van Valin, 1997, Chap. 3.2). Additionally, PropBank annotates general event modifiers such as a temporal adjunct with `ARGM-*` labels (e.g., `ARGM-TMP`). Section B.1 provides more details about the semantic role labels used in PropBank.

6.2.2 Syntactic and semantic dependencies

When a dependency Treebank is used for PropBanking, arguments are labeled on the head-tokens of the phrases instead. Head-tokens are retrieved by the headrules and heuristics described in Chapter 2. Figure 6.3 shows an example of a dependency tree annotated with the semantic arguments in Figure 6.2, where the top arcs show syntactic dependencies from the dependency conversion (Chapter 2) and the bottom arcs show semantic dependencies extracted from PropBank. The head-token of each argument phrase (indicated by a dotted box) has an incoming arc from its predicate with a PropBank label. For instance, *door* is the head-token of the `NP`, *the door*, so it becomes a semantic dependent of *opened* with the label, `ARG1`. This implies that the syntactic subtree of this head-token, *the door*, is a semantic argument of the predicate *open* with the semantic role, `ARG1`.

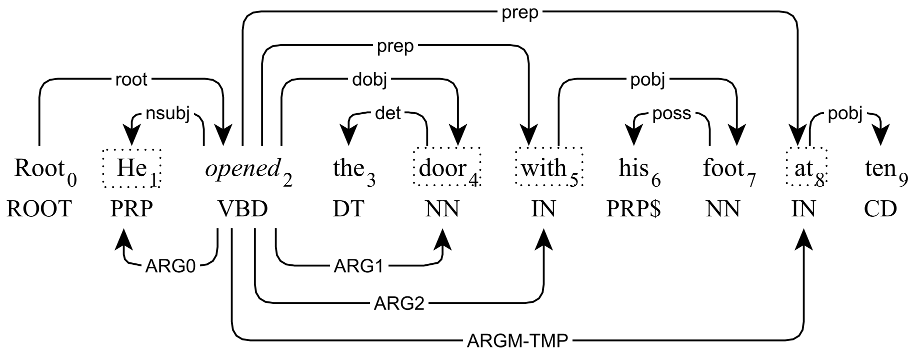


Figure 6.3: A dependency tree annotated with the semantic arguments in Figure 6.2. The top arcs show syntactic dependencies from the dependency conversion in Chapter 2, and the bottom arcs show semantic dependencies extracted from PropBank.

Semantic and syntactic dependencies are often aligned together. In Figure 6.3, *door* is a syntactic dependent, DOBJ (direct object), as well as a semantic dependent, ARG1 (patient), of *opened*. Table 6.1 shows mappings between the semantic and syntactic dependencies in the OntoNotes Treebank. The R-* and C-* labels represent referent and continuous arguments, respectively (see Section 6.2.3 for more details about these labels).

SRL	**DEP**	**Diverge**	**Count**
ARG0	NSUBJ:66.13	30.31	158,931
ARG1	DOBJ: 34.88, NSUBJ: 21.71	12.69	245,924
ARG2	PREP:24.59, ACOMP:20.06	3.22	84,368
ARG3	PREP:41.78	6.07	5,572
ARG4	DIR:57.43, PREP:24.77	2.92	4,898
ARGA	NSUBJ:56.52	30.43	23
R-ARG0	NSUBJ:82.56	17.04	9,989
R-ARG1	NSUBJ:42.6, DOBJ:33.37	7.38	11,454
R-ARG2	ATTR:33.53	3.34	1,736
R-ARG3	NSUBJ:24.05	12.66	79
R-ARG4	DIR:78.46	0.00	65
C-ARG0	NSUBJ:34.91	23.08	169
C-ARG1	XCOMP:59.71	2.07	2,807
C-ARG2	PREP:38.14	10.31	97
C-ARG3	PREP:42.86	7.14	14
C-ARG4	DIR:50, PREP:25	0.00	8
C-V	PRT:70.31	17.44	8,825
ARGM-ADJ	AMOD:73.45	3.54	339
ARGM-ADV	ADVMOD:41.34, ADVCL:29.54	5.74	28,780
ARGM-CAU	PRP:80.94	6.27	4,990
ARGM-COM	PREP:91.84	3.15	539
ARGM-DIR	DIR:45.98, PRT:22.47	1.36	4,785
ARGM-DIS	CC:34.24, INTJ:23.71, ADVMOD:22.42	6.05	28,316
ARGM-EXT	ADVMOD:43.45	3.27	1,802
ARGM-GOL	PREP:61.73	3.23	1,147
ARGM-LOC	LOC:78.38	10.42	18,352
ARGM-MNR	MNR:56.42	7.65	17,509
ARGM-MOD	AUX:92.3	7.41	28,357
ARGM-NEG	NEG:91.35	3.15	14,776
ARGM-PNC	PRP:62.46, PREP:23.38	4.85	1,031
ARGM-PRD	ADVCL:42.68, PREP:28.74	5.20	3,845
ARGM-PRP	PRP:80.22	5.02	5,304
ARGM-REC	NPADVMOD:51.95	3.90	154
ARGM-TMP	TMP:85.61	7.93	50,528

Table 6.1: Mapping between semantic and syntactic dependencies. The SRL column shows semantic labels, the DEP column shows syntactic labels associated with the semantic labels with probabilities (in %), the Diverge column shows the probabilities of semantic dependents whose heads are not their syntactic heads (in %), and the Count column shows the count of each semantic label. The R-* and C-* labels represent referent and continuous arguments, respectively.

Note that the syntactic dependencies in Table 6.1 use the semantic function tags from constituent trees for dependency labels, as described in Figure 2.37 (page 43). This gives a clearer idea about how the semantic function tags in constituent trees correspond to the semantic role labels in PropBank. Our experiments show that the semantic and syntactic dependents are aligned about 86.44%.

6.2.3 Adding semantic roles to dependency trees

In principle, the head-token of each argument phrase becomes a semantic dependent of its predicate with the same PropBank label in a dependency tree (Figures 6.2 and 6.3). However, for numbered arguments containing discontinuous constituents, the same label is used for the leftmost constituent whereas the `C-*` labels, indicating continuous arguments, are used for the remaining pieces of the constituent in the parse tree. These continuous arguments are caused by linguistic phenomena such as subject raising or right node raising, represented by secondary dependencies like `XSUBJ` or `RNR` (Sections 2.5.4 and 2.5.3) in a dependency tree. In Figure 6.4, PropBank annotates `S`-2 as an `ARG1` of the verb predicate *allow*, which is linked to `NP`-1 by the empty category `*`-1, representing subject raising (see the constituent tree on the left). As a result, the numbered argument `ARG1` consists of the discontinuous constituents, `NP`-1 and `S`.

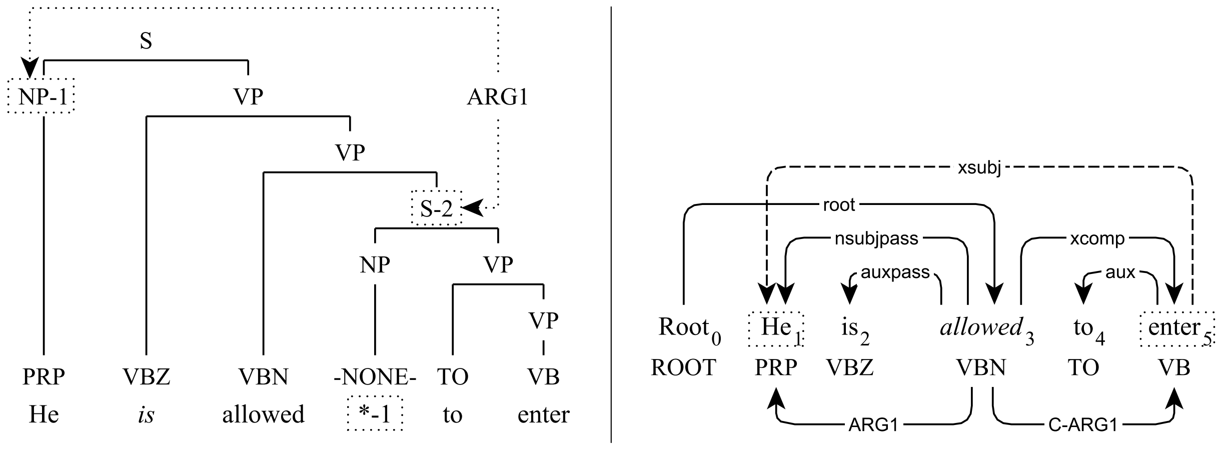


Figure 6.4: An example of a continuous argument. The dependency tree on the right is converted from the constituent tree on the left, and the top and the bottom arcs show syntactic and semantic dependencies, respectively. The numbered argument `ARG1` consists of discontinuous constituents, `NP`-1 and `S`. The secondary dependency `XSUBJ` is indicated by a dotted line.

When this constituent tree is converted into a dependency tree (the rightmost tree in Figure 6.4), there is no token whose subtree includes all tokens in both `NP-1` and `S`; that is, *He*, *to*, and *enter*. At least two tokens, *He* and *enter*, are required to represent the `ARG1` of *allowed* in this dependency tree. Note that the subtree of *enter* would have covered all of these tokens if the secondary dependency `XSUBJ` were assumed; however, this assumption cannot be generally made because most dependency parsers do not produce such secondary dependencies (Section 2.5). Labeling both *He* and *enter* as `ARG1`s of *allowed* creates the ambiguity of them being either two separate arguments or one argument with discontinuous constituents. Thus, only *He* is labeled as an `ARG1` whereas *enter* is labeled as a `C-ARG1`, meaning that it is still an `ARG1` of *allowed* but is a continuation from another `ARG1`, *He*.

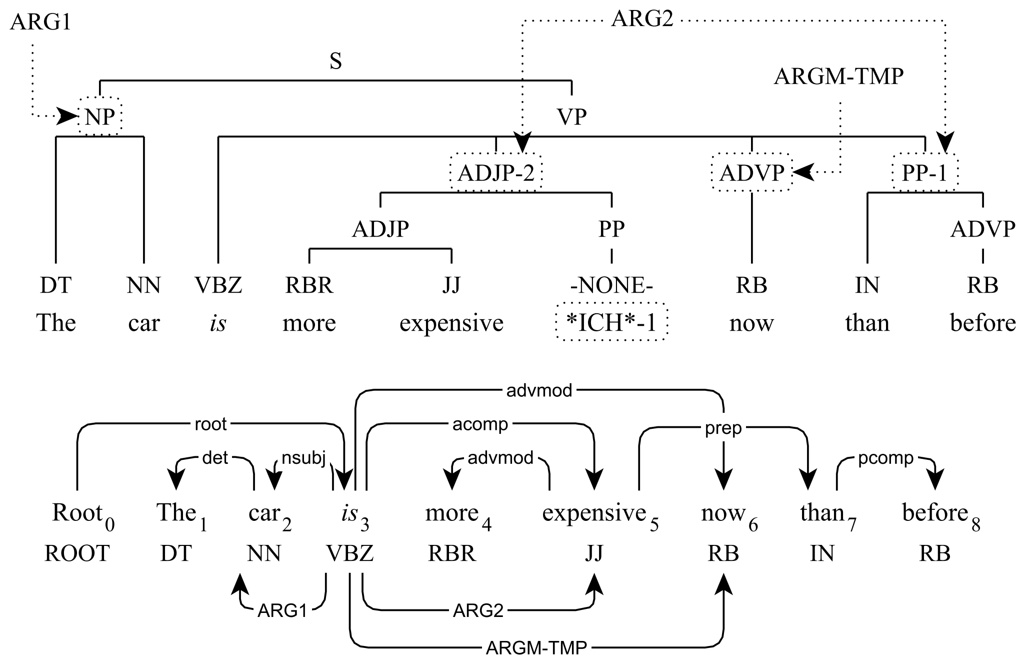

ARG1
ARG2
S
ARGM-TMP
NP
VP
ADJP-2
ADVP
PP-1
ADJP
PP
ADVP
DT NN VBZ RBR JJ -NONE- RB IN RB
The car is more expensive *ICH*-1 now than before
advmod
root
acomp
prep
det
nsubj
advmod
pcomp
Root0 The1 car2 is3 more4 expensive5 now6 than7 before8
ROOT DT NN VBZ RBR JJ RB IN RB
ARG1
ARG2
ARGM-TMP

Figure 6.5: An example of discontinuous constituents, repeated from Figures 2.16 and 2.17. The dependency tree at the bottom is converted from the constituent tree at the top with semantic roles.

Figure 6.5 shows the constituent and dependency trees in Figures 2.16 and 2.17 labeled with semantic roles. PropBank annotates the `ADJP-2` as an `ARG2` of the verb predicate *be*, which is linked to `PP-1` by the empty category, `*ICH*-1`. Unlike the previous case, this dependency tree has the head-token, *expensive*, whose subtree contains all the tokens in both `ADJP-2` and `PP-1`; that is, *more*, *expensive*, *than*, and *before*. Thus, only *expensive* is labeled as an `ARG2` of *is*, and the head-token of `PP-1`, *than*, is not labeled as a `C-ARG2`. Referent arguments are labeled as `R-*` when they are annotated in a dependency tree. In Figure 6.6, PropBank annotates `NP-2` as an `ARG1` of the verb predicate *head*, which is linked to `WHNP-1` by the empty category `*T*-1` representing *wh*-movement, then linked to `NP-3` by the *linkReferent*(C) method in Algorithm 2.11 (Section 2.5.2). Although there is a head-token, *highway*, whose subtree contains all tokens in `NP-3` and `WHNP-1`, this subtree also contains other tokens including the predicate, which creates a cyclic relation. Thus, only the subtree of *highway* excluding the subtree of *heads* is considered an `ARG0` in the dependency tree; that is, *The* and *highway*. This leaves out *that* as an argument, so it is labeled as a `R-ARG0`, meaning that it is an `ARG0` of *heads*, and its referent, *highway*, is also an `ARG0` of the same predicate.

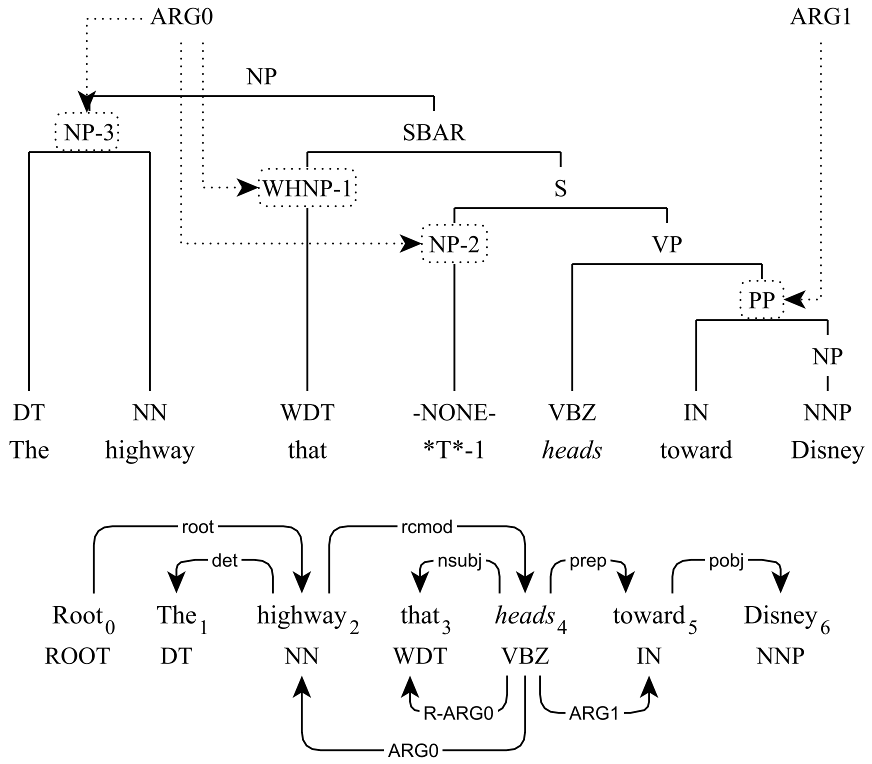

ARG0
ARG1
NP
NP-3
SBAR
WHNP-1
S
NP-2
VP
PP
NP
DT NN WDT -NONE- VBZ IN NNP
The highway that *T*-1 heads toward Disney
root
det
rcmod
nsubj
prep
pobj
Root0 The1 highway2 that3 heads4 toward5 Disney6
ROOT DT NN WDT VBZ IN NNP
R-ARG0
ARG1
ARG0

Figure 6.6: An example of a referent argument. The dependency tree at the bottom is converted from the constituent tree at the top with semantic roles.

Note that referent arguments can be applied to modifiers as well. For instance, *wh*-complementizers such as *when* or *where* are often labeled as a `R-ARGM-TMP` or a `R-ARGM-LOC`, meaning that it is a temporal or a locative modifier with a referent, respectively.

6.3 Argument pruning

6.3.1 First-order argument pruning

Xue and Palmer (2004) introduced a pruning algorithm for constituent-based semantic role labeling that reduces the size of argument candidates by visiting only phrases that are siblings of either the predicate or predicate's ancestors. In Figure 6.2, `NP`-2, `PP`-3, and `PP`-4 are siblings of the predicate, *open*, and `NP`-1 is a sibling of the predicate's parent, `VP`; thus, all of these four phrases would be considered argument candidates by this algorithm. Additionally, their algorithm considered direct children of prepositional phrases argument candidates as well; thus, `IN`s and `NP`s under `PP`-3 and `PP`-4 would also be considered argument candidates.

Johansson and Nugues (2008) adapted this algorithm for dependency-based semantic role labeling by visiting only direct dependents of either the predicate or predicate's ancestors. Let us term this *first-order* argument pruning since it takes only direct dependents into account. The term 'n'th-order' is borrowed from dependency parsing where only direct dependents are used for first-order parsing whereas indirect dependents such as grand-dependents are also used for higher-order parsing. In Figure 6.3, *He*, *door*, *with*, and *at* are direct dependents of *opened*, so these tokens are considered argument candidates by this algorithm. Coupled with their predicate-argument reranker, Johansson and Nugues (2008) achieved state-of-the-art performance for dependency-based semantic role labeling in the CoNLL'08 shared task (Surdeanu et al., 2008) using this pruning algorithm.

6.3.2 Higher-order argument pruning

Although first-order argument pruning works well and effectively reduces the number of argument candidates, it is based on a Linguistic theory called "domain of locality" (Joshi, 2004), which is most effectively applied to gold-standard dependency trees, but does not apply as well to automatically generated ones. Our experiments show that this first-order pruning algorithm visits over 99% of all arguments using gold-standard dependency trees in our evaluation data sets, but visits less than 93% using automatically generated trees (see Figure 6.7). Argument coverage is particularly low for corpora whose automatic parse results are poor. This implies that the upper bound for recall for semantic role labeling using this pruning algorithm is lower than 93% when automatically generated dependency trees are used. Such a low upper bound in recall affects the overall F1-score; thus, a higher-order pruning algorithm is needed for better argument coverage.

Algorithm 6.1 shows a method that takes a dependency tree T as input and produces a set of semantic dependency arcs in T as output. This algorithm uses higher-order argument pruning, which significantly improves the argument coverage when automatically generated dependency trees are used for input (Figure 6.7). Note that all predicates are assumed to be identified in T; predicate identification is often not considered a part of semantic role labeling (Carreras and Màrques, 2004; Carreras and Màrquez, 2005; Hajič et al., 2009) although it is a necessary step. The difficulty of building a predicate identification model comes from incomplete annotation in PropBank. Currently, PropBank annotation in the OntoNotes v4.99 misses about 5% of verb instances, which brings enough noise to negatively impact training and evaluation. Thus, gold-standard predicate identification is used for our experiments.[2]

Algorithm 6.1 : *getArgumentArcSet*(T)

```
Input:  A dependency tree $T$, where predicates are already identified.
Output: A set $A$ containing semantic dependency arcs in $T$.
1: $A \leftarrow \emptyset$
2: for $p$ in all predicates in $T$ do
3:     getArgumentArcSetDown$(p, p, A)$                          # Algorithm 6.2
4:     if $p$ has the head $h$ then getArgumentArcSetUp$(p, h, A)$   # Algorithm 6.3
5: return $A$
```

Algorithm 6.2 : *getArgumentArcSetDown*(p, n, A)

```
Input: A predicate $p$, a dependency node $n$, and a set of dependency arcs $A$,
       where $n$ is either $p$ or a descendant of $p$.
1: for $d$ in all dependents of $n$ do
2:     if $d$ is an argument of $p$ with a label $l$ then $A \leftarrow A \cup \{d \xleftarrow{l} p\}$
3:     getArgumentArcSetDown$(p, d, A)$
```

[2] The PropBank annotation is expected to be complete in v5.0. Once the OntoNotes v5.0 is released, we will train and evaluate models for predicate identification.

Algorithm 6.3 : *getArgumentArcSetUp*(p, h, A)

```
Input:  A predicate $p$, a dependency node $h$, and a set of dependency arc $A$,
        where $h$ is an ancestor of $p$.
1: if $h$ is an argument of $p$ with a label $l$ then $A \leftarrow A \cup \{h \overset{l}{\leftarrow} p\}$
2: for $d$ in all dependents of $h$ do
3:     if $d$ is an argument of $p$ with a label $l$ then $A \leftarrow A \cup \{d \overset{l}{\leftarrow} p\}$
4: if $h$ has the head $h'$ then getArgumentArcSetUp$(p, h', A)$
```

For each predicate in T, the *getArgumentArcSetDown*(p, d, A) method in Algorithm 6.2 is called first, which finds semantic dependencies between the predicate and all descendants of the predicate. This is where our algorithm is distinguished from the previous algorithm; only direct dependents of the predicate are considered argument candidates in the first-order pruning algorithm. Then, the *getArgumentArcSetUp*(p, h, A) method in Algorithm 6.3 is called, which finds semantic dependencies between the predicate and predicate's ancestors as well as their direct dependents. This time, only direct dependents of the predicate's ancestors are considered argument candidates; our experiments show that visiting all descendants of the ancestors significantly increases the number of argument candidates (Figure 6.10) without giving much improvement in labeling accuracy (Section 6.5.1).

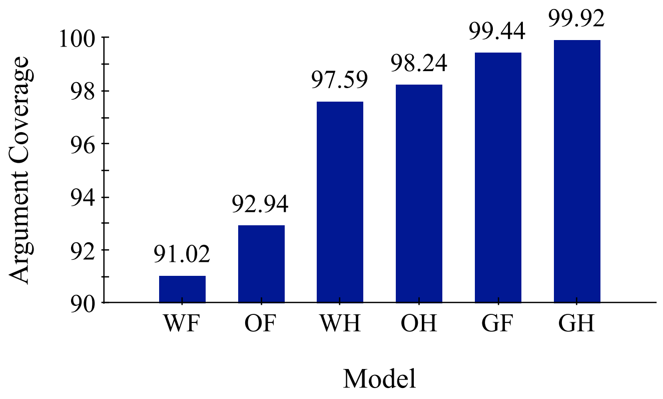


Figure 6.7: Argument coverage rates achieved by different pruning algorithms and input trees (in %). `W*` and `O*` stand for pruning algorithms using trees automatically generated by the Wall Street Journal and the OntoNotes models, and `G*` stands for algorithms using gold-standard trees, respectively. `*-F` and `*-H` stands for the first-order and the higher-order pruning algorithms.

Figure 6.7 shows argument coverage rates achieved by different pruning algorithms and input trees (in %). Given gold-standard dependency trees, both first-order and higher-order pruning algorithms show above 99% argument coverage rates. However, given automatically generated trees, the first-order pruning algorithm shows around 91-93% whereas the higher-order pruning algorithm shows around 97-98% argument coverage rates; the differences are statistically significant (McNemar's test, $p < .0001$). The argument coverage rates are higher for the OntoNotes model because it generates higher quality dependency trees than the WSJ model (Section 5.6.1).

6.3.3 Conditional higher-order argument pruning

Higher-order argument pruning gives better argument coverage to semantic role labeling; however, it also increases the number of argument candidates. To reduce the average number of argument candidates, conditional higher-order argument pruning is proposed, which brings the complexity close to the one achieved by the first-order argument pruning, yet still shows noticeably better argument coverage. During training, dependency paths between predicates and dependency nodes whose subtrees contain arguments of the predicates are collected. These paths are used to prune out argument candidates that are not likely to form predicate argument structures during decoding.

Algorithm 6.4 : *collectPathSets*(T, c_d, c_u)

```
Input:  A dependency tree $T$, where predicates are already identified.
        Cutoffs $c_d$ and $c_u$ for down-paths and up-paths, respectively.
Output: Two sets $S_d$ and $S_u$ containing down-paths and up-paths in $T$, respectively.
1: $(D_d, D_u) \leftarrow (\{\ \}, \{\ \})$                                  # initialize empty dictionaries
2: for $p$ in all predicates in $T$ do
3:   for $d$ in all grand-dependents of $p$ do
4:     $collectPathsDown(p, d, D_d)$                                       # Algorithm 6.5
5:   if $p$ has the grand-head $h$ then $collectPathsUp(p, h, D_u)$         # Algorithm 6.6
6: $S_d \leftarrow D_d.getKeys(c_d)$
7: $S_u \leftarrow D_u.getKeys(c_u)$
8: return $(S_d, S_u)$
```

Algorithm 6.4 shows a method that takes a dependency tree T and two cutoffs, c_d and c_u, as input and produces two sets, S_d and S_u, as output. The algorithm begins by initializing two dictionaries, D_d and D_u, in which keys and values are dependency paths and their occurrences in T, respectively (line 1). For each predicate, dependency paths between the predicate and its descendants whose subtrees contain arguments of the predicate, called down-paths, are added to D_d (lines 3-4). Note that dependency paths between the predicate and its direct dependents are not collected because they are always considered argument candidates regardless of their paths to the predicate. Similarly, dependency paths between the predicate and its ancestors whose dependents are arguments of the predicate, called up-paths, are added to D_u (line 5). Again, dependency paths between the predicate and its head are not collected because the head and head's dependents are always considered argument candidates. Once this procedure is done, two sets S_d and S_u are created by selecting only paths in D_d and D_u whose occurrences are greater than the cutoffs c_d and c_u, respectively (lines 6-7). For our experiments, cutoffs of 5 and 0 are used for c_d and c_u, respectively.

Algorithm 6.5 : *collectPathsDown*(p, n, D_d)

```
Input:  A predicate $p$, a dependency node $n$, and a dictionary $D_d$, where $n$ is a descendant of $p$,
        and keys and values in $D_d$ are down-paths and their occurrences, respectively.
1: if $n$ is an argument of $p$ then
2:   for $a$ in getPathList($p$, $n$.head) do $D_d[a] \leftarrow D_d[a] + 1$   # Algorithm 6.7
3: for $d$ in all dependents of $n$ do
4:   collectPathsDown($p, d, D_d$)
```

Algorithm 6.5 shows a method that collects down-paths between a predicate p and p's descendant n, where n is an argument of p (lines 1-2). In Figure 6.8, *what* is an `R-ARG1` of the predicate *see*; thus, dependency paths are collected between *saw* and *what*, ↓`CONJ` and ↓`CONJ`↓`PREP`, indicating that these are paths from p to p's descendants whose subtrees contain arguments of p. This procedure is repeated recursively until all descendants of n are visited (lines 3-4).

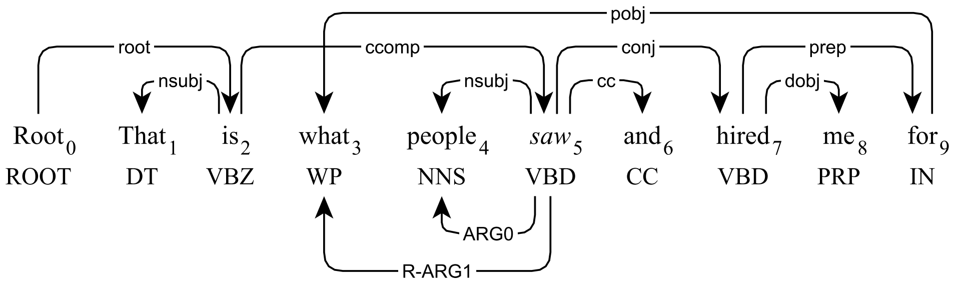

pobj
root
ccomp
conj
prep
nsubj
nsubj
cc
dobj
Root0 That1 is2 what3 people4 saw5 and6 hired7 me8 for9
ROOT DT VBZ WP NNS VBD CC VBD PRP IN
ARG0
R-ARG1

Figure 6.8: An example of a down-path. For the predicate *see* and its argument *what*, two dependency paths are collected, ↓`CONJ` and ↓`CONJ`↓`PREP`.

Algorithm 6.6 : *collectPathsUp*(p, h, D_u)

```
Input:  A predicate $p$, a dependency node $h$, and a dictionary $D_u$, where $h$ is an ancestor of $p$,
        and keys and values in $D_u$ are up-paths and their occurrences, respectively.

1: if $h$ has a dependent that is an argument of $p$ then
2:     for $a$ in getPathList($h$, $p$) do $D_u[a] \leftarrow D_u[a] + 1$    # Algorithm 6.7
3: if $h$ has the head $h'$ then
4:     collectPathsUp($p, h', D_u$)
```

Algorithm 6.6 shows a method that collects up-paths between a predicate p and p's ancestor h whose dependent is an argument of p (lines 1-2). In Figure 6.9, *He* is an `ARG0` of the predicate *join*; thus, dependency paths are collected between *saw* and *He*, ↑`XCOMP` and ↑`XCOMP`↑`CONJ`, indicating that these are either intermediate or full paths from p to p's ancestor whose dependent is an argument of p. This procedure is repeated recursively until all ancestors of h are visited (lines 3-4).

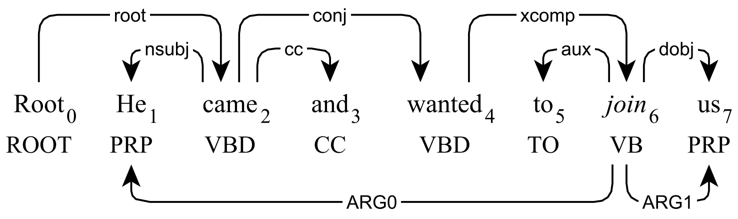


Figure 6.9: An example of an up-path. For the predicate *join* and its argument *He*, two dependency paths are collected, ↑`XCOMP` and ↑`XCOMP`↑`CONJ`.

Algorithm 6.7 shows a method that takes dependency nodes t and b as input, where t is an ancestor of b, and a list L of dependency paths between t and b as output. A dependency path can have a height of one (e.g., ↓`CONJ`) or many (e.g., ↓`CONJ`↓`PREP`), depending on the hierarchy between t and b.

Algorithm 6.7 : *getPathList*(t, b)

```
Input:  Dependency nodes $t$ and $b$, where $t$ is an ancestor of $b$.
Output: A list $L$ containing dependency paths between $t$ and $b$.

1: $L \leftarrow [\,]$
2: while $t \neq b$ do
3:     $L.add$(the dependency path between $t$ and $b$)    # e.g., ↓CONJ, ↑XCOMP
4:     $b \leftarrow b$.head
5: return $L$
```

Given the sets of dependency paths collected by Algorithm 6.4, the higher-order pruning algorithm in Section 6.3.2 can be rewritten as a conditional higher-order pruning algorithm. Algorithm 6.8 shows a method that uses conditional higher-order argument pruning; it is similar to the method in Algorithm 6.1, except that it calls Algorithms 6.9 and 6.10, which prune out argument candidates more rigorously than Algorithms 6.2 and 6.3 called in Algorithms 6.1.

Algorithm 6.8 : *getArgumentArcSetCon*(T, S_d, S_u)

```
 Input:  A dependency tree $T$, a set of down-paths $S_d$, and a set of up-paths $S_u$,
         where predicates are already identified in $T$.
Output:  A set $A$ containing semantic dependency arcs in $T$.
1: $A \leftarrow \emptyset$
2: for $p$ in all predicates in $T$ do
3:     getArgumentArcSetDownCon$(p, p, A, S_d)$                        # Algorithm 6.9
4:     if $p$ has the head $h$ then getArgumentArcSetUpCon$(p, h, A, S_u)$   # Algorithm 6.10
5: return $A$
```

Algorithm 6.9 shows a method that is similar to the method in Algorithm 6.2, except that it checks if the down-path from predicate p to its descendant d exists in S_d, which contains down-paths collected by Algorithm 6.5 (line 3). If S_d does not contain the down-path between p and d, all descendants of d are discarded from being argument candidates of p.

Algorithm 6.9 : *getArgumentArcSetDownCon*(p, n, A, S_d)

```
Input:  A predicate $p$, a dependency node $n$, a set of dependency arcs $A$, and
        a set of down-paths $S_d$, where $n$ is either $p$ or a descendant of $p$.
1: for $d$ in all dependents of $n$ do
2:     if $d$ is an argument of $p$ with a label $l$ then $A \leftarrow A \cup \{d \xleftarrow{l} p\}$
3:     if $S_d$ contains the dependency path between $p$ and $d$ then    # conditional
4:         getArgumentArcSetDownCon$(p, d, A, S_d)$
```

Algorithm 6.10 shows a method that is similar to the method in Algorithm 6.3, except that it checks if the up-path from predicate p to its ancestor h' exists in S_u, which contains up-paths collected by Algorithm 6.6 (line 5). If S_u does not contain the up-path between p and h', all dependents and ancestors of h' as well as h' itself are discarded from being argument candidates of p.

Algorithm 6.10 : *getArgumentArcSetUpCon*(p, h, A, S_u)

```
Input:  A predicate $p$, a dependency node $h$, a set of dependency arc $A$, and
        a set of up-paths $S_u$, where $h$ is an ancestor of $p$.

1: if $h$ is an argument of $p$ with a label $l$ then $A \leftarrow A \cup \{h \overset{l}{\leftarrow} p\}$
2: for $d$ in all dependents of $h$ do
3:     if $d$ is an argument of $p$ with a label $l$ then $A \leftarrow A \cup \{d \overset{l}{\leftarrow} p\}$
4: if ($h$ has the head $h'$) and
5:     ($S_u$ contains the dependency path between $h'$ and $p$) then        # conditional
6:     getArgumentArcSetUpCon$(p, h', A, S_u)$
```

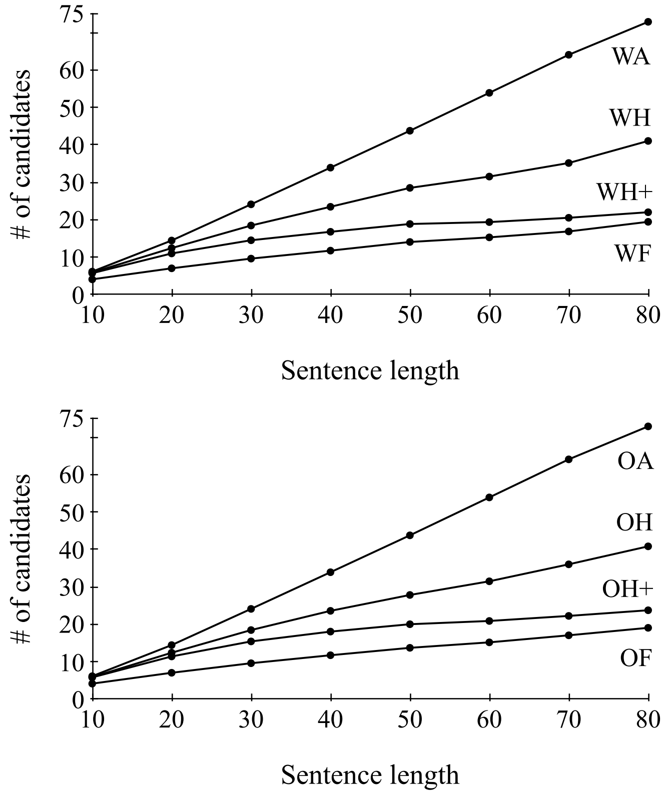


Figure 6.10: The average number of argument candidates for different pruning algorithms with respect to sentence lengths. The top and the bottom figures show results achieved by the Wall Street Journal and OntoNotes models, respectively. `*A` stands for an algorithm considering all dependency nodes as argument candidates. `*F`, `*H`, and `*H+` stand for the first-order, the higher-order, and the conditional higher-order pruning algorithms, respectively.

Figure 6.10 shows the average numbers of argument candidates compared by different pruning algorithms with respect to sentence lengths. For both the Wall Street Journal (`W+`) and OntoNotes (`O+`) models, the higher-order pruning algorithm (`*H`) consistently compares about twice as many argument candidates as the first-order pruning algorithm (`*F`), although it compares significantly less than the algorithm comparing all dependency nodes (`*A`). The conditional higher-order pruning algorithm (`*H+`) gives an average number of argument candidates close to the one achieved by the first-order pruning algorithm. The gap between these two algorithms becomes smaller as the sentence length increases, which implies that the conditional higher-order pruning algorithm is even more effective for longer sentences.

6.4 Features

6.4.1 Feature templates

Table 6.2 shows the feature templates used for our semantic role labeling experiments. Our feature templates are inspired by Johansson and Nugues (2008) although we combine their feature sets, one for argument identification and the other for argument classification, into one.[3] w_p and w_a stand for the current predicate and the argument candidate, respectively. w_{x+n} stands for the word token whose distance from w_x is n. $hd(x)$ stands for the head of x. $ld(x)$ and $rd(x)$ stand for the leftmost and rightmost dependents of x. $ls(x)$ and $rs(x)$ stand for the left-nearest and right-nearest siblings of x. The Form, Lemma, POS, Deprel, Depset, Subcat$_L$, and Subcat$_R$ columns indicate the word form, lemma, POS tag, dependency label, sub-dependency label set, left subcategorization, and right subcategorization features of the corresponding token, respectively. These features are used either individually or jointly (e.g., pos tags of both w_p and w_a make one feature).

The dependency label set is derived by collecting all dependency labels of w_{pred}'s direct dependents. For the predicate *open* in Figure 6.3, the dependency label set is {`NSUBJ`,`DOBJ`, `PREP`}. Unlike Johansson and Nugues (2008), we decompose subcategorization features into two parts: one representing the left-hand side and the other representing the right-hand side dependencies of w_{pred}. For the predicate *open* in Figure 6.3, the left-hand side subcategorization is $\overleftarrow{\texttt{NSUBJ}}$ and the right-hand side subcategorization is $\overrightarrow{\texttt{DOBJ-PREP-PREP}}$. Our experiments show that with this separation, the subcategorization features generalize better. In addition to the features in Table 6.2, three

[3] Johansson and Nugues (2008)'s feature templates are also inspired by previous work on constituent-based semantic role labeling (Gildea and Jurafsky, 2002; Pradhan et al., 2008).

kinds of path features are added: POS paths, dependency paths, and height paths. In Figure 6.9, the POS and dependency paths between the predicate *join* and its argument *He* are ↑`VBD`↑`VBD`↓`PRP` and ↑`XCOMP`↑`CONJ`↓`NSUBJ`, respectively. The height path, introduced by Choi and Palmer (2011b), indicates the height between a predicate and its argument. Let LCA be the lowest common ancestor of *join* and *He* in Figure 6.9; that is *came*. The height path between these two tokens is ↑2↓1, implying that the height between *join* and LCA is 2 and the height between *He* and LCA is 1.

Token	**Form**	**Lemma**	**POS**	**Deprel**	**Subset**	**Subcat**$_L$	**Subcat**$_R$
w_p	✓	✓	✓	✓	✓	✓	✓
w_{p-1}		✓	✓				
w_{p+1}		✓	✓				
$hd(w_p)$		✓	✓			✓	✓
$ld(w_p)$		✓	✓				
$rd(w_p)$		✓	✓				
w_a	✓	✓	✓	✓			
w_{a-1}		✓	✓				
w_{a+1}		✓					
$hd(w_a)$		✓					
$ld(w_a)$		✓	✓				
$rd(w_a)$		✓	✓				
$ls(w_a)$		✓					
$rs(w_a)$		✓	✓				

Table 6.2: Feature templates for semantic role labeling.

We also use the last two predicted numbered argument labels of the current predicate as features. Finally, two kinds of binary features are used: if w_{arg} is a syntactic head of w_{pred} and if w_{pred} is a syntactic head of w_{arg}.

6.4.2 Positional feature separation

Features in Section 6.4.1 are divided into two sets and trained separately: one contains features for arguments on the left-hand side and the other contains features for arguments on the right-hand side of the predicate. From our experiments, we found that separating these features improves the overall F1-score of semantic role labeling, evaluated for both argument identification and classification, whereas having one combined set of features shows an advantage when semantic role labeling is evaluated for only argument identification.

6.5 Experiments

6.5.1 Accuracy comparisons

Tables 6.3 and 6.4 show F1-scores for semantic role labeling ($F_{h \wedge l}$ in Table 3.1), achieved by the WSJ and OntoNotes models, respectively. The Baseline model uses the first-order pruning algorithm in Section 6.3.1. The Baseline+ model is the Baseline model with positional feature separation from Section 6.4.2; positional feature separation is also applied to all the other models. The High-order model uses the higher-order pruning algorithm in Section 6.3.2. The No-pruning model does not use any pruning algorithm. The ClearNLP model uses the conditional higher-order pruning algorithm in Section 6.3.3. The ClearNLP model is compared to another dependency-based semantic role labeler, ClearParser, which uses a transition-based semantic role labeling algorithm showing state-of-the-art performance on the CoNLL'09 shared task data for English (Choi and Palmer, 2011b). ClearParser uses the same positional feature separation and does not use any pruning algorithms.

Model	BC	BN	MD	MP	MZ	SH	TC	WB	Avg$_i$	Avg$_o$
Baseline	72.79	75.84	69.40	68.68	73.21	61.67	69.82	71.53	81.88	71.07
Baseline+	73.20	76.51	69.93	69.66	73.76	62.26	70.29	72.02	82.28	71.64
High-order	73.56	76.86	70.19	70.00	73.93	62.15	70.38	**72.48**	**82.52**	71.90
No-pruning	**73.63**	**76.87**	**70.47**	**70.08**	73.99	62.18	70.46	72.25	82.48	**71.95**
ClearNLP	73.45	76.80	70.07	69.97	**74.11**	62.21	70.39	72.21	82.42	71.85
ClearParser	73.37	76.40	68.52	68.36	73.57	**62.76**	**71.03**	72.53	82.26	71.52

Table 6.3: F1-scores of semantic role labeling from the WSJ models (in %).

	OntoNotes						Medical			Average	
Model	BC	BN	MZ	NW	TC	WB	MD	MP	SH	Avg$_i$	Avg$_o$
Baseline	81.76	80.62	78.99	82.90	82.57	76.17	71.84	71.63	62.67	80.73	70.02
Baseline+	82.00	81.27	79.43	83.43	83.35	77.14	72.42	72.03	63.26	81.33	70.54
High-order	82.32	81.47	79.36	**83.68**	83.54	77.28	72.56	**72.32**	63.11	81.51	70.68
No-pruning	82.33	**81.50**	79.37	83.62	83.50	77.12	**72.79**	72.26	63.43	81.48	**70.81**
ClearNLP	82.33	81.46	79.37	83.66	83.57	77.33	77.04	72.23	63.26	81.52	70.68
ClearParser	**82.53**	81.36	**79.66**	83.66	**83.96**	**77.50**	71.32	71.57	**63.88**	**81.69**	70.01

Table 6.4: F1-scores of semantic role labeling from the OntoNotes models (in %).

The Avg$_i$ and Avg$_o$ columns show the micro average F1-scores for in-genre and out-of-genre experiments. For the WSJ models, the F1-score of NW is measured for Avg$_i$ and the micro average of all other corpora are measured for Avg$_o$. For the OntoNotes models, the micro averages of the OntoNotes and Medical corpora are measured for Avg$_i$ and Avg$_o$, respectively. The Baseline+ model shows higher scores than the Baseline model and the High-order model shows higher scores than the Baseline+ model for all experiments, which indicates that the positional feature separation and the higher-order pruning algorithm improve labeling accuracy for both in-genre and out-of-genre experiments. Although the High-order model performs better than the No-pruning model for all in-genre experiments, the No-pruning model performs better for all out-of-genre experiments; this is expected because the higher-order pruning algorithm performs less accurately when syntactic parse input is poor, which is the case for out-of-genre experiments.

The ClearNLP model shows lower scores than the High-order model for both Avg$_i$ and Avg$_o$ in Table 6.3; however, it shows similar or slightly higher scores than the High-order model for both Avg$_i$ and Avg$_o$ in Table 6.4. This is because the ClearNLP model could not learn enough conditions from the WSJ corpus whereas enough conditions are learned from the OntoNotes coprora so the ClearNLP model can perform as accurately as the High-order model. For both Avg$_i$ and Avg$_o$ in Table 6.3, the ClearNLP model shows higher scores than ClearParser. In Table 6.4, ClearParser performs better for Avg$_i$ whereas the ClearNLP model performs significantly better for Avg$_o$. This implies that our approach works more effectively for out-of-genre experiments.

	WSJ: Avg$_i$		**WSJ: Avg$_o$**		**Onto: Avg$_i$**		**Onto: Avg$_o$**	
Model	**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**
Baseline	84.74	79.20	72.39	69.79	83.20	78.41	66.63	73.77
Baseline+	86.52	78.44	74.07	69.36	85.24	77.77	68.12	73.14
High-order	86.65	78.78	74.29	69.65	85.12	78.19	68.21	73.33
No-pruning	**86.94**	78.45	**74.70**	69.40	**85.30**	77.99	**68.64**	73.13
ClearNLP	86.47	78.73	74.13	69.70	85.05	78.28	68.08	73.48
ClearParser	84.45	**80.17**	71.74	**71.29**	83.50	**79.96**	65.92	**74.64**

Table 6.5: Precisions (P) and recalls (R) for in-genre (Avg$_i$) and out-of-genre (Avg$_o$) experiments (in %). The WSJ and Onto columns show scores achieved by the WSJ and OntoNotes models.

Table 6.5 shows precisions and recalls for in-genre and out-of-genre experiments. The No-pruning model shows the highest precisions whereas ClearParser shows the highest recalls for all experiments. The High-order model shows higher recalls than the Baseline+ model for all experiments, which is expected. Interestingly, the ClearNLP model shows higher recalls than the No-pruning models for all experiments; the differences are statistically significant (McNemar, $p < .01$). Furthermore, the ClearNLP models show higher recalls than the High-order models for most experiments even with the more aggressive pruning algorithm; the differences for the OntoNotes experiments are statistically significant (McNemar, $p < .01$). We suspect that this is because the conditional higher-order pruning algorithm reduces more noise for recall than the naive higher-order pruning algorithm so it performs better even with fewer argument candidates. On the other hand, the ClearNLP models do not show high precision for all experiments. We will explore the possibility of improving precision for the ClearNLP models through error analysis in the future.

	WSJ: Avg$_i$		**WSJ: Avg$_o$**		**Onto: Avg$_i$**		**Onto: Avg$_o$**	
Model	**ARGN**	**ARGM**	**ARGN**	**ARGM**	**ARGN**	**ARGM**	**ARGN**	**ARGM**
Baseline	85.81	71.14	75.91	59.07	84.69	71.76	73.49	60.81
Baseline+	86.27	71.18	76.52	59.35	85.40	71.93	74.06	60.94
High-order	86.49	**71.43**	76.76	**59.58**	85.51	72.28	74.08	**61.39**
No-pruning	86.54	71.08	**76.83**	59.56	**85.53**	72.10	**74.25**	61.38
ClearNLP	86.48	71.07	76.72	59.56	**85.53**	**72.29**	74.14	61.22
ClearParser	**86.66**	70.21	76.58	58.98	85.90	72.15	73.90	59.70

Table 6.6: F1-scores of numbered arguments (ARGN) and modifiers (ARGM) for in-genre and out-of-genre experiments (in %).

Table 6.6 shows F1-scores of numbered arguments and modifiers for in-genre and out-of-genre experiments. The ClearNLP model shows the highest scores for predicting both numbered arguments and modifiers for the OntoNotes in-genre experiment (Onto: Avg$_i$), whereas the other models show advantages for other experiments.

6.5.2 Speed comparisons

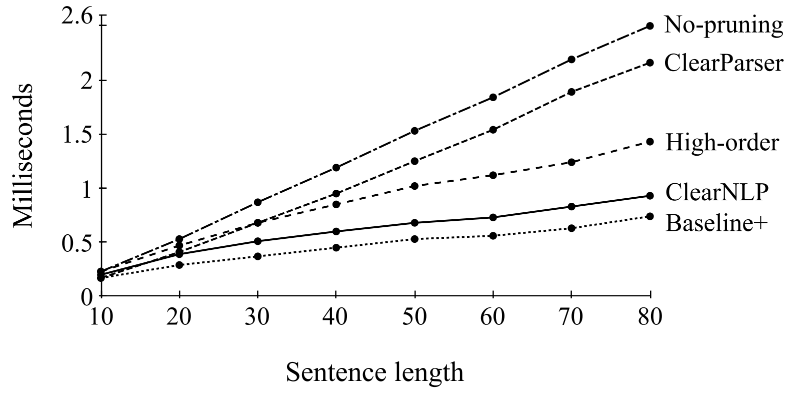
Labeling speeds are measured by running each system on a mixture of all data. All systems are written in Java; ClearParser has its own implementation and the other models are implemented in the same system. Table 6.7 shows speed comparisons between five models; the top and bottom five rows show results from the WSJ and OntoNotes models, respectively. The T# columns show how many milliseconds each model takes for labeling all arguments of each predicate in 5 trials, and the

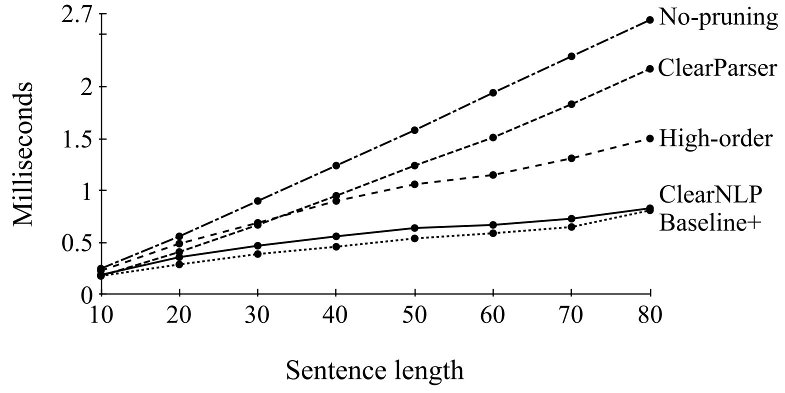
Avg column shows the average labeling speeds of the middle three trials, and the Arguments column shows how many arguments are labeled per second. The ClearNLP models take about 0.42 - 0.45 milliseconds per predicate, which is slower than the Baseline+ model but faster than all the other models. The ClearNLP model labels over 1,500 more arguments per second than the high-order model, which is noticeably faster. Notice that ClearParser shows faster speeds than the No-pruning model even though neither of them uses any pruning algorithm. This comes from implementation differences between these two systems; ClearParser walks through an array for searching arguments whereas the No-pruning model traverses a tree. Since accessing an array is quicker than traversing a tree, the No-pruning model shows slower labeling speeds than ClearParser although they compare an equal number of argument candidates.

Model		**T1**	**T2**	**T3**	**T4**	**T5**	**Avg**	**Arguments**
WSJ	Baseline+	0.34	0.34	0.34	0.35	0.39	0.34	7,111
	High-order	0.58	0.60	0.60	0.60	0.61	0.60	4,062
	No-pruning	0.77	0.78	0.80	0.81	0.88	0.80	3,081
	ClearNLP	0.41	0.42	0.42	0.42	0.42	**0.42**	**5,856**
	ClearParser	0.60	0.61	0.63	0.66	0.69	0.63	3,877
ON	Baseline+	0.35	0.35	0.35	0.35	0.39	0.35	6,934
	High-order	0.62	0.62	0.62	0.64	0.64	0.63	3,904
	No-pruning	0.80	0.82	0.83	0.84	0.86	0.83	2,947
	ClearNLP	0.45	0.45	0.45	0.45	0.46	**0.45**	**5,410**
	ClearParser	0.61	0.63	0.63	0.63	0.66	0.63	3,882

Table 6.7: Labeling speeds (in ms.). The top and the bottom five rows show results from the WSJ and OntoNotes models, respectively.

Figure 6.11 shows labeling speeds with respect to sentence lengths achieved by the WSJ (the top figure) and OntoNotes (the bottom figure) models. ClearParser and the No-pruning models show almost complete linear growth whereas the other models show somewhat similar logarithm growth, implying that the pruning algorithms work more effectively for longer sentences. The ClearNLP model shows labeling speeds close to the Baseline+ model; especially for longer sentences. This implies that the ClearNLP model, using the conditional higher-order pruning algorithm, runs almost as fast as the Baseline+ model, using the first-order pruning algorithm, yet performs more accurately.





Group	≤ 10	≤ 20	≤ 30	≤ 40	≤ 50	≤ 60	≤ 70	≤ 80
Sentences	9,789	6,159	2,966	1,313	479	178	75	30

Figure 6.11: Average labeling speeds with respect to sentence lengths. The top and the bottom figures show parsing speeds achieved by the WSJ and OntoNotes models, respectively. The bottom table shows the number of sentences in each group. –·–·–: No-pruning, -------: ClearParser, - - - - -: High-order, ———: ClearNLP, ·········: Baseline+.

Chapter 7

Conclusion

This thesis focuses on the optimization of NLP components for robustness and scalability. All trees in constituent Treebanks are converted into dependency trees in the CLEAR dependency format and semantic roles in PropBanks are mapped accordingly to these dependency trees. For dependency conversion, new head-finding rules and heuristics are introduced to handle format changes in recent English Treebanks, allowing us to generate richer and more robust dependency representations than previous approaches. For POS tagging, dynamic model selection is introduced, which shows more robust tagging accuracy across different genres and runs noticeably faster than two other state-of-the-art POS taggers. Our selection approach is more effective when training data is small and is used for tagging data with many varieties. For dependency parsing, a new transition-based parsing algorithm and a bootstrapping technique are introduced. Our parsing algorithm shows linear time parsing speed for generating both projective and non-projective dependency trees and shows a lower average number of transitions performed compared to other transition-based parsing algorithms. Our bootstrapping technique gives significant improvement on parsing accuracy showing higher performance against two other state-of-the-art dependency parsers. For semantic role labeling, a conditional higher-order argument pruning algorithm is introduced. A higher-order argument pruning algorithm improves the coverage of argument candidates and shows improvement on the overall F1-score. The conditional higher-order argument pruning algorithm noticeably reduces the average number of argument candidates with little compromise of the F1-score.

To the best of our knowledge, this is the first time that these three components have been evaluated on such a wide variety of English data using the same experimental settings. Our results for in-genre and out-of-genre experiments give helpful feedback for many NLP tasks such as machine translation or information extraction that depend significantly on the outputs of these components and need to process large-scale heterogeneous data. Processing all three components takes about 2.49-2.69 milliseconds per sentence using our approaches and implementations (0.36-0.37 milliseconds for POS tagging, 1.16-1.28 milliseconds for dependency parsing, and 0.97-1.04 milliseconds for semantic role labeling), which handles over 370 sentences per second. Additionally, our dependency conversion is useful for those who want to generate gold-standard dependency trees for training. All components are implemented in an open source project called ClearNLP and are publicly available.[1] It is worth mentioning that the main advantage of ClearNLP over ClearParser comes from modular programming. ClearParser is designed specifically for dependency parsing and semantic role labeling and it runs fast for those tasks, but it is hard to extend its APIs to other components. ClearNLP is designed to be more modular and to interface easily with other NLP components in general i.e., it is easier to extend its APIs to other NLP components such as a POS tagger or a DAG (directed acyclic graph) parser.[2]

There is still much room for improvement. For all three tasks, no extensive feature engineering is performed, which needs to be done for optimum results. We will try to apply automatic feature selection techniques such as Chi-square (Liu and Setiono, 1995) that may improve the speed and accuracy of our NLP components. For POS tagging, we will try to integrate our dynamic model selection approach with more sophisticated tagging algorithms such as bidirectional dependency networks (Toutanova et al., 2003) in hopes of improve accuracy further. Moreover, we will explore the possibility of using unsupervised clustering techniques to automatically group training data into meaningful document sets for dynamic model selection. For dependency parsing, we will try to compare parse results for non-projective trees with languages containing more non-projective

[1] ClearNLP: `clearnlp.googlecode.com`, ClearParser: `clearparser.googlecode.com`

[2] ClearNLP includes a DAG parser that is still experimental.

dependencies such as Czech (Hajič et al., 2000), Danish (Kromann, 2003), Sloven (Džeroski et al., 2006), etc. and see which transitions help or hurt performance in non-projective parsing. We will also try to apply more exhaustive search techniques such as k-best parsing or beam search to improve our parsing accuracy further. For semantic role labeling, we plan to implement a predicate identification module in ClearNLP so the entire SRL process can be automated. We also plan to implement an argument spanning module in ClearNLP so it can produce the actual spans of semantic arguments instead of just head-tokens. Most importantly, we will analyze what types of errors made by our dependency parser hurt the most for semantic role labeling and try to improve labeling accuracy for these error cases.

Bibliography

Giuseppe Attardi and Felice Dell'Orletta. Reverse revision and linear tree combination for dependency parsing. In Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics - Human Language Technologies (NAACL-HLT'09), 2009.

Ann Bies, Mark Ferguson, Karen Katz, and Robert MacIntyre. Bracketing Guidelines for Treebank II Style Penn Treebank Penn Treebank Project. Technical report, University of Pennsylvania, 1995.

Claire Bonial, Olga Babko-Malaya, Jinho D. Choi, Jena Hwang, and Martha Palmer. PropBank Annotation Guidelines. Technical report, University of Colorado at Boulder, 2010.

Claire Bonial, William Corvey, Martha Palmer, Volha V. Petukhova, and Harry Bunt. A Hierarchical Unification of LIRICS and VerbNet Semantic Roles. In Proceedings of the 2011 IEEE 5th International Conference on Semantic Computing, ICSC'11, pages 483–489, 2011.

Thorsten Brants. TnT: a statistical part-of-speech tagger. In Proceedings of the 6th Conference on Applied Natural Language Processing, ANLC'00, pages 224–231, 2000.

Eric Brill. Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part-of-Speech Tagging. Computational Linguistics, 21(23):543–565, 1995.

Sabine Buchholz and Erwin Marsi. CoNLL-X shared task on multilingual dependency parsing. In Proceedings of the Tenth Conference on Computational Natural Language Learning, CoNLL'06, pages 149–164, 2006.

Xavier Carreras. Experiments with a Higher-Order Projective Dependency Parser. In Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL'07 (CoNLL'07), pages 957–961, 2007.

Xavier Carreras and Lluís Màrques. Introduction to the CoNLL-2004 Shared Task: Semantic Role Labeling. In Proceedings of the 8th Conference on Computational Natural Language Learning: Shared Task (CoNLL'04), pages 89–97, 2004.

Xavier Carreras and Lluís Màrquez. Introduction to the CoNLL-2005 shared task: semantic role labeling. In Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL'05), pages 152–164, 2005.

Daniel Cer, Marie-Catherine de Marneffe, Daniel Jurafsky, and Christopher D. Manning. Parsing to Stanford Dependencies: Trade-offs between speed and accuracy. In Proceedings of the 7th International Conference on Language Resources and Evaluation, LREC'10, 2010.

Jinho D. Choi and Nicolas Nicolov. K-best, Locally Pruned, Transition-based Dependency Parsing Using Robust Risk Minimization. In Recent Advances in Natural Language Processing V, pages 205–216. John Benjamins, 2009.

Jinho D. Choi and Martha Palmer. Retrieving correct semantic boundaries in dependency structure. In Proceedings of ACL workshop on Linguistic Annotation (LAW'10), pages 91–99, 2010a.

Jinho D. Choi and Martha Palmer. Robust Constituent-to-Dependency Conversion for Multiple Corpora in English. In Proceedings of the 9th International Workshop on Treebanks and Linguistic Theories, TLT'9, 2010b.

Jinho D. Choi and Martha Palmer. Getting the Most out of Transition-based Dependency Parsing. In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, ACL:HLT'11, pages 687–692, 2011a.

Jinho D. Choi and Martha Palmer. Transition-based Semantic Role Labeling Using Predicate Argument Clustering. In Proceedings of ACL workshop on Relational Models of Semantics, RELMS'11, pages 37–45, 2011b.

Noam Chomsky. Lectures in Government and Binding. Dordrecht, Foris, 1981.

Noam Chomsky. The Minimalist Program. The MIT Press, 1995.

Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. Machine Learning, 20(3):273–297, 1995.

Isaac G. Councill, Ryan McDonald, and Leonid Velikovich. What's Great and What's Not: Learning to Classify the Scope of Negation for Improved Sentiment Analysis. In Proceedings of the Workshop on Negation and Speculation in Natural Language Processing, NeSp-NLP'10, pages 51–59, 2010.

Michael A. Covington. A Fundamental Algorithm for Dependency Parsing. In Proceedings of the 39th Annual ACM Southeast Conference, pages 95–102, 2001.

Elizabeth A. Cowper. A Concise Introduction to Syntactic Theory. The University of Chicago Press, 1992.

Koby Crammer and Yoram Singer. On the Algorithmic Implementation of Multiclass Kernel-based Vector Machines. Journal of Machine Learning Research, 2:265–292, 2002.

Koby Crammer and Yoram Singer. Ultraconservative Online Algorithms for Multiclass Problems. Journal of Machine Learning Research, 3:951–991, 2003.

Hang Cui, Renxu Sun, Keya Li, Min-Yen Kan, and Tat-Seng Chua. Question Answering Passage Retrieval Using Dependency Relations. In Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval, ACM-SIGIR'05, pages 400–407, 2005.

Hal Daumé, III and Daniel Marcu. Domain Adaptation for Statistical Classifiers. Journal of Artificial Intelligence Research, 26(1):101–126, 2006.

Hal Daumé, Iii, John Langford, and Daniel Marcu. Search-based structured prediction. Machine Learning, 75(3):297–325, 2009.

Hal Daume III. Frustratingly Easy Domain Adaptation. In Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics, ACL'07, pages 256–263, 2007.

Marie-Catherine de Marneffe and Christopher D. Manning. The Stanford typed dependencies representation. In Proceedings of the COLING workshop on Cross-Framework and Cross-Domain Parser Evaluation, 2008a.

Marie-Catherine de Marneffe and Christopher D. Manning. Stanford Dependencies manual. `http://nlp.stanford.edu/software/dependencies_manual.pdf`, 2008b.

Saš Džeroski, Tomaž Erjavec, Nina Ledinek, Petr Pajas, Zdenek Žabokrtsky, and Andreja Žele. Towards a Slovene dependency treebank. In Proceedings of 5th International Conference on Language Resources and Evaluation, LREC'06, pages 24–26, 2006.

Susanne Ekeklint and Joakim Nivre. A Dependency-Based Conversion of PropBank. In Proceedings of FRAME: Building Frame Semantics Resources for Scandinavian and Baltic Languages, FRAME'07, pages 19–25, 2007.

Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification Journal of Machine Learning Research. The Journal of Machine Learning Research, 9:1871–1874, 2008.

Daniel Gildea and Daniel Jurafsky. Automatic Labeling of Semantic Roles. Computational Linguistics, 28(3), 2002.

Jesús Giménez and Lluís Màrquez. SVMTool: A general POS tagger generator based on Support Vector Machines. In Proceedings of the 4th International Conference on Language Resources and Evaluation, LREC'04, 2004.

Yoav Goldberg and Michael Elhadad. An Efficient Algorithm for Easy-First Non-Directional Dependency Parsing. In Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, HLT:NAACL'10, pages 742–750, 2010.

Jan Hajič, Alena Böhmová, Eva Hajičová, and Barbora Vidová-Hladká. The Prague Dependency Treebank: A Three-Level Annotation Scenario. In A. Abeillé, editor, Treebanks: Building and Using Parsed Corpora, pages 103–127. Amsterdam:Kluwer, 2000.

Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štěpánek, Pavel Straňák, Mihai Surdeanu, Nianwen Xue, and Yi Zhang. The CoNLL-2009 Shared Task: Syntactic and Semantic Dependencies in Multiple Languages. In Proceedings of the 13th Conference on Computational Natural Language Learning: Shared Task, CoNLL'09, pages 1–18, 2009.

Abdelati Hawwari, Jena D. Hwang, Aous Mansouri, and Martha Palmer. Classification and Deterministic PropBank Annotation of Predicative Adjectives in Arabic. In Proceedings of the 6th Joint ISO - ACL SIGSEM Workshop on Interoperable Semantic Annotation, ISA'11, pages 44–48, 2011.

Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi, and S. Sundararajan. A Dual Coordinate Descent Method for Large-scale Linear SVM. In Proceedings of the 25th international conference on Machine learning, ICML'08, pages 408–415, 2008.

Liang Huang and Kinji Sagae. Dynamic Programming for Linear-Time Incremental Parsing. In Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL'10), 2010.

Jena D. Hwang, Archna Bhatia, Clare Bonial, Aous Mansouri, Ashwini Vaidya, Nianwen Xue, and Martha Palmer. PropBank Annotation of Multilingual Light Verb Constructions. In Proceedings of ACL workshop on Linguistic Annotation, LAW'10, pages 82–90, 2010.

Ray Jackendoff. Gapping and Related Rules. Linguistic Inquiry, 2:21–35, 1971.

Richard Johansson. Dependency-based Semantic Analysis of Natural-language Text. PhD thesis, Lund University, 2008.

Richard Johansson and Pierre Nugues. Extended Constituent-to-dependency Conversion for English. In Proceedings of the 16th Nordic Conference of Computational Linguistics, NODALIDA'07, 2007.

Richard Johansson and Pierre Nugues. Dependency-based Semantic Role Labeling of PropBank. In Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing (EMNLP'08), pages 69–78, 2008.

Aravind K. Joshi. Domains of locality. Data & Knowledge Engineering, 50(3):277–289, 2004.

Terry Koo and Michael Collins. Efficient Third-order Dependency Parsers. In Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL'10), 2010.

Terry Koo, Alexander M. Rush, Michael Collins, Tommi Jaakkola, and David Sontag. Dual Decomposition for Parsing with Non-Projective Head Automata. In Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, EMNLP'10, pages 1288–1298, 2010.

Matthias T Kromann. The Danish Dependency Treebank and the underlying linguistic theory. In Proceedings of the 2nd Workshop on Treebanks and Linguistic Theories, TLT'03, 2003.

Marco Kuhlmann and Joakim Nivre. Transition-Based Techniques for Non-Projective Dependency Parsing. Northern European Journal of Language Technology, 2(1):1–19, 2010.

Robert D. Levine. Right Node (non)-Raising. Linguistic Inquiry, 16(3):492–497, 1985.

Ding Liu and Daniel Gildea. Semantic Role Features for Machine Translation. In Proceedings of the 23rd International Conference on Computational Linguistics, COLING'10, pages 716–724, 2010.

Huan Liu and Rudy Setiono. Chi2: Feature Selection and Discretization of Numeric Attributes. In Proceedings of the 7th International Conference on Tools with Artificial Intelligence, TAI'95, pages 88–91, 1995.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a Large Annotated Corpus of English: The Penn Treebank. Computational Linguistics, 19(2):313–330, 1993.

Ryan Mcdonald and Fernando Pereira. Online Learning of Approximate Dependency Parsing Algorithms. In Proceedings of the Annual Meeting of the European American Chapter of the Association for Computational Linguistics, EACL'06, pages 81–88, 2006.

Ryan McDonald and Giorgio Satta. On the Complexity of Non-Projective Data-Driven Dependency Parsing. In Proceedings of the 10th International Conference on Parsing Technologies, IWPT'07, pages 121–132, 2007.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. Non-projective Dependency Parsing using Spanning Tree Algorithms. In Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing (HLT-EMNLP'05), pages 523–530, 2005.

Rodney D. Nielsen, James Masanz, Philip Ogren, Wayne Ward, James H. Martin, Guergana Savova, and Martha Palmer. An architecture for complex clinical question answering. In Proceedings of the 1st ACM International Health Informatics Symposium, IHI'10, pages 395–399, 2010.

Jens Nilsson, Joakim Nivre, and Johan Hall. Graph Transformations in Data-Driven Dependency Parsing. In Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics, COLING:ACL'06, pages 257–264, 2006.

Joakim Nivre. An Efficient Algorithm for Projective Dependency Parsing. In Proceedings of the 8th International Workshop on Parsing Technologies, IWPT'03, pages 149–160, 2003.

Joakim Nivre. Incrementality in Deterministic Dependency Parsing. In Proceedings of the ACL'04 Workshop on Incremental Parsing: Bringing Engineering and Cognition Together, pages 50–57, 2004.

Joakim Nivre. Inductive Dependency Parsing. Springer, 2006.

Joakim Nivre. Algorithms for deterministic incremental dependency parsing. Computational Linguistics, 34(4):513–553, 2008.

Joakim Nivre. Non-Projective Dependency Parsing in Expected Linear Time. In Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP (ACL-IJCNLP'09), pages 351–359, 2009.

Joakim Nivre and Ryan McDonald. Integrating Graph-based and Transition-based Dependency Parsers. In Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL:HLT'08), pages 950–958, 2008.

Joakim Nivre and Jens Nilsson. Pseudo-Projective Dependency Parsing. In Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics, ACL'05, pages 99–106, 2005.

Joakim Nivre and Mario Scholz. Deterministic Dependency Parsing of English Text. In Proceedings of the 20th International Conference on Computational Linguistics, COLING'04, pages 64–70, 2004.

Joakim Nivre, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. The CoNLL 2007 Shared Task on Dependency Parsing. In Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007, pages 915–932, 2007.

Martha Palmer, Daniel Gildea, and Paul Kingsbury. The Proposition Bank: An Annotated Corpus of Semantic Roles. Computational Linguistics, 31(1):71–106, 2005.

Slav Petrov and Ryan McDonald. Overview of the 2012 Shared Task on Parsing the Web. In Proceedings of the 1st Workshop on Syntactic Analysis of Non-Canonical Language: shared task, SANCL'12, 2012.

Slav Petrov, Dipanjan Das, and Ryan McDonald. A Universal Part-of-Speech Tagset. In Proceedings of the 8th International Conference on Language Resources and Evaluation, LREC'12, pages 2089–2096, 2012.

Sameer Pradhan, Wayne Ward, and James H. Martin. Towards Robust Semantic Role Labeling. Computational Linguistics: Special Issue on Semantic Role Labeling, 34(2):289–310, 2008.

Owen Rambow, Cassandre Creswell, Rachel Szekely, Harriet Taber, and Marilyn Walker. A Dependency Treebank for English. In Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC'02), 2002.

Adwait Ratnaparkhi. Maximum entropy models for natural language ambiguity resolution. PhD thesis, University of Pennsylvania, 1998.

Alexander M. Rush and Slav Petrov. Vine Pruning for Efficient Multi-Pass Dependency Parsing. In The 12th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL:HLT'12, 2012.

Dan Shen and Mirella Lapata. Using Semantic Roles to Improve Question Answering. In Proceedings of the Conference on Empirical Methods in Natural Language Processing and on Computational Natural Language Learning, EMNLP:CoNLL'07, pages 12–21, 2007.

Libin Shen, Giorgio Satta, and Aravind Joshi. Guided Learning for Bidirectional Sequence Classification. In Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics, ACL'07, pages 760–767, 2007.

Libin Shen, Jinxi Xu, and Ralph Weischedel. A New String-to-Dependency Machine Translation Algorithm with a Target Dependency Language Model. In Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, ACL:HLT'08, pages 577–585, 2008.

Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and Implementation of Deductive Parsing. In Journal of Logic Programming, number 24 in JLP'95, pages 3–36, 1995.

Anders Søgaard. Semi-supervised condensed nearest neighbor for part-of-speech tagging. In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, ACL'11, pages 48–52, 2011.

Drahomíra "johanka" Spoustová, Jan Hajič, Jan Raab, and Miroslav Spousta. Semi-supervised Training for the Averaged Perceptron POS Tagger. In Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics, EACL'09, pages 763–771, 2009.

Mihai Surdeanu and Christopher D. Manning. Ensemble models for dependency parsing: Cheap and good? In Proceedings of the North American Chapter of the Association for Computational Linguistics Conference, NAACL'10, 2010.

Mihai Surdeanu, Richard Johansson, Adam Meyers, Lluís Màrquez, and Joakim Nivre. The CoNLL-2008 Shared Task on Joint Parsing of Syntactic and Semantic Dependencies. In Proceedings of the 12th Conference on Computational Natural Language Learning: Shared Task, CoNLL'08, pages 59–177, 2008.

Ann Taylor. Treebank 2a Guidelines. `http://www-users.york.ac.uk/~lang22/TB2a_Guidelines.htm`, 2006.

Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, NAACL'03, pages 173–180, 2003.

Ashwini Vaidya, Jinho D. Choi, Martha Palmer, and Bhuvana Narasimhan. Analysis of the Hindi Proposition Bank using Dependency Structure. In roceedings of ACL workshop on Linguistic Annotation, pages 21–29, 2011.

Robert D. Van Valin. Syntax: Structure, Meaning, and Function. Cambridge University Press, 1997.

M. Čmejrek, J. Cuřín, and J. Havelka. Prague Czech-English Dependency Treebank: Any Hopes for a Common Annotation Scheme? In HLT-NAACL'04 workshop on Frontiers in Corpus Annotation, pages 47–54, 2004.

Karin Verspoor, Kevin B. Cohen, Arrick Lanfranchi, Colin Warner, Helen L. Johnson, Christophe Roeder, Jinho D. Choi, Christopher Funk, Yuriy Malenkiy, Miriam Eckert, Nianwen Xue, William A. Baumgartner Jr., Mike Bada, Martha Palmer, and Hunter Larry E. A corpus of full-text journal articles is a robust evaluation tool for revealing differences in performance of biomedical natural language processing tools. BMC Bioinformatics, 2012. in press.

Ralph Weischedel, Eduard Hovy, Martha Palmer, Mitch Marcus, Robert Belvin, Sameer Pradhan, Lance Ramshaw, and Nianwen Xue. OntoNotes: A Large Training Corpus for Enhanced Processing. In Joseph Olive, Caitlin Christianson, and John McCary, editors, Handbook of Natural Language Processing and Machine Translation. Springer, 2011.

Nianwen Xue and Martha Palmer. Calibrating Features for Semantic Role Labeling. In Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2004.

Tong Zhang, Fred Damerau, and David Johnson. Text Chunking based on a Generalization of Winnow. Journal of Machine Learning Research, 2(23):615–637, 2002.

Yue Zhang and Stephen Clark. A Tale of Two Parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP'08), pages 562–571, 2008.

Appendix A

Constituent Treebank Tags

This appendix shows tags used in various constituent Treebanks for English (Marcus et al., 1993; Nielsen et al., 2010; Weischedel et al., 2011; Verspoor et al., 2012). Tags followed by * are not the typical Penn Treebank tags but used in some other Treebanks.

A.1 Function tags

Syntactic roles			
ADV	Adverbial	PUT	Locative complement of *put*
CLF	*It*-cleft	PRD	Non-VP predicate
CLR	Closely related constituent	RED*	Reduced auxiliary
DTV	Dative	SBJ	Surface subject
LGS	Logical subject in passive	TPC	Topicalization
NOM	Nominalization		
Semantic roles			
BNF	Benefactive	MNR	Manner
DIR	Direction	PRP	Purpose or reason
EXT	Extent	TMP	Temporal
LOC	Locative	VOC	Vocative
Text and speech categories			
ETC	Et cetera	SEZ	Direct speech
FRM*	Formula	TTL	Title
HLN	Headline	UNF	Unfinished constituent
IMP	Imperative		

Table A.1: A list of function tags for English.

A.2 Part-of-speech tags

Word level tags			
ADD	Email	POS	Possessive ending
AFX	Affix	PRP	Personal pronoun
CC	Coordinating conjunction	PRP$	Possessive pronoun
CD	Cardinal number	RB	Adverb
CODE	Code ID	RBR	Adverb, comparative
DT	Determiner	RBS	Adverb, superlative
EX	Existential there	RP	Particle
FW	Foreign word	TO	To
GW	Go with	UH	Interjection
IN	Preposition or subordinating conjunction	VB	Verb, base form
JJ	Adjective	VBD	Verb, past tense
JJR	Adjective, comparative	VBG	Verb, gerund or present participle
JJS	Adjective, superlative	VBN	Verb, past participle
LS	List item marker	VBP	Verb, non-3rd person singular present
MD	Modal	VBZ	Verb, 3rd person singular present
NN	Noun, singular or mass	WDT	*Wh*-determiner
NNS	Noun, plural	WP	*Wh*-pronoun
NNP	Proper noun, singular	WP$	Possessive *wh*-pronoun
NNPS	Proper noun, plural	WRB	*Wh*-adverb
PDT	Predeterminer	XX	Unknown

Punctuation like tags			
$	Dollar	-LRB-	Left bracket
:	Colon	-RRB-	Right bracket
,	Comma	HYPH	Hyphen
.	Period	NFP	Superfluous punctuation
‘‘	Left quote	SYM	Symbol
’’	Right quote	PUNC	General punctuation

Table A.2: A list of part-of-speech tags for English.

A.3 Clause and phrase level tags

Clause level tags	
S	Simple declarative clause
SBAR	Clause introduced by a subordinating conjunction
SBARQ	Direct question introduced by a *wh*-word or a *wh*-phrase
SINV	Inverted declarative sentence
SQ	Inverted yes/no question, or main clause of a *wh*-question

Phrase level tags			
ADJP	Adjective phrase	NX	N-bar level phrase
ADVP	Adverb phrase	PP	Prepositional phrase
CAPTION*	Caption	PRN	Parenthetical phrase
CIT*	Citation	PRT	Particle
CONJP	Conjunction phrase	QP	Quantifier Phrase
EDITED	Edited phrase	RRC	Reduced relative clause
EMBED	Embedded phrase	TITLE*	Title
FRAG	Fragment	TYPO	Typo
HEADING*	Heading	UCP	Unlike coordinated phrase
INTJ	Interjection	VP	Verb phrase
LST	List marker	WHADJP	*Wh*-adjective phrase
META	Meta data	WHADVP	*Wh*-adverb phrase
NAC	Not a constituent	WHNP	*Wh*-noun phrase
NML	Nominal phrase	WHPP	*Wh*-prepositional phrase
NP	Noun phrase	X	Unknown

Table A.3: A list of clause and phrase level tags for English.

Appendix B

Semantic Role Labels

B.1 PropBank semantic role labels

This appendix shows a list of the PropBank semantic role labels. See Bonial et al. (2010) for more details about the PropBank semantic role labels.

Label	**Description**
`ARG0`	Agent
`ARG1`	Patient, theme
`ARG2`	Instrument, benefactive, attribute
`ARG3`	Staring point
`ARG4`	Ending point
`ARGA`	External causer
`ARGM-ADJ`	Adjectival
`ARGM-ADV`	Adverbial
`ARGM-CAU`	Cause
`ARGM-COM`	Comitative
`ARGM-DIR`	Direction
`ARGM-DIS`	Discourse
`ARGM-GOL`	Goal
`ARGM-EXT`	Extent
`ARGM-LOC`	Location
`ARGM-MNR`	Manner
`ARGM-MOD`	Modal
`ARGM-NEG`	Negation
`ARGM-PRD`	Secondary predication
`ARGM-PRP`	Purpose (previously, `ARGM-PNC`)
`ARGM-REC`	Recipricol
`ARGM-TMP`	Temporal

Table B.1: A list of the PropBank semantic role labels.

B.2 VerbNet thematic role labels

This appendix shows a list of the VerbNet thematic role labels. See Bonial et al. (2011) for more details about the VerbNet thematic role labels.

Label	**Description**
`actor1,2`	pseudo-agents, used for some communication classes
`agent`	animate subject, volitional or internally controlled
`asset`	currency, used for Build/Get/Obtain Classes
`attribute`	changed quality of patient/theme
`beneficiary`	Entity benefiting from action
`cause`	entity causing an action, used for psychological/body verbs
`destination`	End point/target of motion
`experiencer`	Participant that is aware of experiencing something
`extent`	Range or degree of change
`instrument`	Objects/forces that come into contact and cause change in another object
`location`	Underspecified destination/source/place
`material`	Starting point of transformation
`patient1,2`	Affected participants, used for some combining/attaching verbs
`predicate`	Predicative complement
`product`	End result of transformation
`recipient`	Target of transfer
`source`	Spatial location, starting point
`stimulus`	Events/objects that elicit a response from an experiencer
`theme`	Participants in/undergoing a change of location
`theme1,2`	Indistinct themes, used for differ/exchange classes
`patient`	Affected participants undergoing a process
`time`	Class-specific, express temporal relations
`topic`	Topic of conversation, message, used for communication verbs
`proposition`	Complement clause indicating desired/requested action, used for order class

Table B.2: A list of the VerbNet thematic role labels.

Appendix C

Dependency Labels

C.1 CoNLL dependency labels

This appendix shows a list of the CoNLL dependency labels. See Johansson (2008, Chap. 4) for more details about the CoNLL dependency labels.

Labels retained from function tags			
`ADV`	Unclassified adverbial	`MNR`	Manner
`BNF`	Benefactor	`PRD`	Predicative complement
`DIR`	Direction	`PRP`	Purpose or reason
`DTV`	Dative	`PUT`	Locative complement of *put*
`EXT`	Extent	`SBJ`	Subject
`LGS`	Logical subject	`TMP`	Temporal
`LOC`	Locative	`VOC`	Vocative
Labels inferred from constituent relations			
`AMOD`	Modifier of adjective or adverb	`OPRD`	Object predicate
`CONJ`	Conjunct	`P`	Punctuation
`COORD`	Coordination	`PMOD`	Modifier of preposition
`DEP`	Unclassified dependency	`PRN`	Parenthetical
`EXTR`	Extraposed element	`PRT`	Particle
`GAP`	Gapping	`QMOD`	Modifier of quantifier
`IM`	Infinitive marker	`ROOT`	Root
`NMOD`	Modifier of nominal	`SUB`	Subordinating conjunction
`OBJ`	Object or clausal complement	`VC`	Verb chain

Table C.1: A list of the CoNLL dependency labels.

C.2 Stanford dependency labels

This appendix shows a list of the Stanford dependency labels. See de Marneffe and Manning (2008b) for more details about Stanford dependency labels.

Label	Description	Label	Description
ABBREV	Abbreviation modifier	NPADVMOD	Noun phrase as ADVMOD
ACOMP	Adjectival complement	NSUBJ	Nominal subject
ADVCL	Adverbial clause modifier	NSUBJPASS	Nominal subject (passive)
ADVMOD	Adverbial modifier	NUM	Numeric modifier
AGENT	Agent	NUMBER	Element of compound number
AMOD	Adjectival modifier	PARATAXIS	Parataxis
APPOS	Appositional modifier	PARTMOD	Participial modifier
ATTR	Attribute	PCOMP	Prepositional complement
AUX	Auxiliary	POBJ	Object of a preposition
AUXPASS	Auxiliary (passive)	POSS	Possession modifier
CC	Coordination	POSSESSIVE	Possessive modifier
CCOMP	Clausal complement	PRECONJ	Preconjunct
COMPLM	Complementizer	PREDET	Predeterminer
CONJ	Conjunct	PREP	Prepositional modifier
COP	Copula	PREPC	Prepositional clausal modifier
CSUBJ	Clausal subject	PRT	Phrasal verb particle
CSUBJPASS	Clausal subject (passive)	PUNCT	Punctuation
DEP	Dependent	PURPCL	Purpose clause modifier
DET	Determiner	QUANTMOD	Quantifier phrase modifier
DOBJ	Direct object	RCMOD	Relative clause modifier
EXPL	Expletive	REF	Referent
INFMOD	Infinitival modifier	REL	Relative
IOBJ	Indirect object	ROOT	Root
MARK	Marker	TMOD	Temporal modifier
MWE	Multi-word expression	XCOMP	Open clausal complement
NEG	Negation modifier	XSUBJ	Controlling subject
NN	Noun compound modifier		

Table C.2: A list of the Stanford dependency labels.

Appendix D

The CLEAR Dependency Labels

This appendix gives descriptions of the CLEAR dependency labels. Algorithms introduced here are called by the *getDependencyLabel*(C, P, p) method in Algorithm 2.10. Algorithms followed by * (e.g., *setPassiveSubject*$(D, H)^{*}$ in Algorithm D.2) are called after the *getDependencyLabel*(C, P, p) method and applied to all dependency nodes.

D.1 Arguments: subject related

Subject-related labels consist of agents (`AGENT`), clausal subjects (`CSUBJ`), clausal passive subjects (`CSUBJPASS`), expletives (`EXPL`), nominal subjects (`NSUBJ`), and nominal passive subjects (`NSUBJPASS`).

Algorithm D.1 : *getSubjectLabel*(C, d)

Input: Constituents C and d, where d is the head dependent of C.
Output: `CSUBJ`, `NSUBJ`, `EXPL`, or `AGENT` if C is a subject-related argument; otherwise, `null`.

```
1: if $C$ has SBJ then
2:   if $C$ is S* return CSUBJ      # Section D.1.2
3:   if $d$ is EX return EXPL       # Section D.1.4
4:   return NSUBJ                   # Section D.1.5
5: if $C$ has LGS return AGENT      # Section D.1.1
6: return null
```

Algorithm D.2 : *setPassiveSubject*(D, H)*

```
Input:  Dependents $D$ and $H$, where $H$ is the head of $D$.
Output: If $D$ is a passive subject, append PASS to its label.

1: if $H$ contains AUXPASS then
2:     if   $D$ is CSUBJ then $D$.label ← CSUBJPASS    # Section D.1.3
3:     elif $D$ is NSUBJ then $D$.label ← NSUBJPASS    # Section D.1.6
```

D.1.1 AGENT: agent

An agent is the complement of a passive verb that is the surface subject of its active form. In our approach, the preposition *by* is included as a part of AGENT.

(1)	The car was bought [by John]	AGENT(bought, by), POBJ(by, John)
(2)	The car bought [by John] is red	AGENT(bought, by), POBJ(by, John)

D.1.2 CSUBJ: clausal subject

A clausal subject is a clause in the subject position of an active verb. A clause with a SBJ function tag is considered a CSUBJ.

(1)	[Whether she liked me] doesn't matter	CSUBJ(matter, liked)
(2)	[What I said] was true	CSUBJ(was, said)
(3)	[Who I liked] was you	CCOMP(was, liked), NSUBJ(was, you)

In (3), *Who I liked* is topicalized such that it is considered a clausal complement (CCOMP) of *was*; *you* is considered a nominal subject (NSUBJ) of *was*.

D.1.3 CSUBJPASS: clausal passive subject

A clausal passive subject is a clause in the subject position of a passive verb. A clause with the SBJ function tag that depends on a passive verb is considered a CSUBJPASS.

(1)	[Whoever misbehaves] will be dismissed	CSUBJPASS(dismissed, misbehaves)

D.1.4 EXPL: expletive

An expletive is an existential *there* in the subject position.

(1)	There was an explosion	EXPL(was, There)

D.1.5 NSUBJ: nominal subject

A nominal subject is a non-clausal constituent in the subject position of an active verb. A non-clausal constituent with the SBJ function tag is considered a NSUBJ.

(1)	[She and I] came home together	NSUBJ(came, She)
(2)	[Earlier] was better	NSUBJ(was, Earlier)

D.1.6 NSUBJPASS: nominal passive subject

A nominal passive subject is a non-clausal constituent in the subject position of a passive verb. A non-clausal constituent with the SBJ function tag that depends on a passive verb is considered a NSUBJPASS.

(1)	I [am] drawn to her	NSUBJPASS(drawn, I)
(2)	We will [get] married	NSUBJPASS(married, We)
(3)	She will [become] nationalized	NSUBJPASS(nationalized, She)

D.2 Arguments: object related

Object-related labels consist of attributes (ATTR), direct objects (DOBJ), indirect objects (IOBJ), and object predicates (OPRD).

Algorithm D.3 : *getObjectLabel*(C)

```
 Input:  A constituent $C$ whose parent is VP|SINV|SQ.
Output:  DOBJ or ATTR if $C$ is in an object or an attribute; otherwise, null.

1: if $C$ is NP|NML then
2:    if $C$ has PRD return ATTR   # Section D.2.1
3:    return DOBJ                  # Section D.2.2
4: return null
```

D.2.1 ATTR: attribute

An attribute is a noun phrase that is a non-VP predicate usually following a copula verb.

(1)	This product is [a global brand]	ATTR(is, brand)
(2)	This area became [a prohibited zone]	ATTR(became, zone)

D.2.2 DOBJ: direct object

A direct object is a noun phrase that is the accusative object of a (di)transitive verb.

(1)	She bought me [these books]	DOBJ(bought, books)
(2)	She bought [these books] for me	DOBJ(bought, books)

D.2.3 IOBJ: indirect object

An indirect object is a noun phrase that is the dative object of a ditransitive verb.

(1)	She bought [me] these books	IOBJ(bought, me)
(2)	She bought these books [for me]	PREP(bought, for)
(3)	[What] she bought [me] were these books	DOBJ(bought, What), IOBJ(bought, me)
(4)	I read [them] [one by one]	DOBJ(read, them), NPADVMOD(read, one)

In (2), *for me* is considered a prepositional modifier although it is the dative object in an unshifted form. This information is preserved with a function tag DTV as additional information in our representation (Section 2.6.2). In (3), *What* and *me* are considered direct and indirect objects of *bought*, respectively. In (4), the noun phrase *one by one* is not considered an IOBJ, but an adverbial noun phrase modifier (NPADVMOD) because it carries an adverbial function tag, MNR. This kind of information is also preserved with semantic function tags in our representation (Section 2.6.1).

Algorithm D.4 : *setIndirectObject*$(C)^{*}$

```
 Input:  A dependent $D$.
Output:  If $D$ is an indirect object, set its label to IOBJ.

1: if ($D$ is DOBJ) and ($D$ is followed by another DOBJ) then $D$.label ← IOBJ
```

D.2.4 OPRD: object predicate

An object predicate is a non-VP predicate in a small clause that functions like the predicate of an object. Section 2.3.4 describes how object predicates are derived.

(1)	She calls [me] [her friend]	DOBJ(calls, me), OPRD(calls, friend)
(2)	She considers [[me] her friend]	CCOMP(considers, friend), NSUBJ(me, friend)
(3)	I am considered [her friend]	OPRD(considered, friend)
(4)	I persuaded [her] [to come]	DOBJ(persuaded, her), XCOMP(persuaded, come)

In (2), the small clause *me her friend* is considered a clausal complement (CCOMP) because we treat *me* as the subject of the non-VP predicate, *her friend.* In (4), the open clause *to come* does indeed predicate over *her* but is not labeled as an OPRD but rather an open clausal complement (XCOMP). This is because the dependency between *her* and *come* is already shown in our representation as an open clausal subject (XSUBJ) whereas such information is not available for the non-VP predicates in (1) and (3); thus, without labeling them as object predicates, it can be difficult to infer the relation between the objects and object predicates.

Algorithm D.5 : *isObjectPredicate*(C)

```
 Input:  A constituent $C$.
Output:  True if $C$ is an object predicate; otherwise, False.

1: if ($C$ is S) and ($C$ contains no VP) and ($C$ contains both SBJ and PRD) then
2:   if the subject of $C$ is an empty category return True
3: return False
```

D.3 Auxiliaries

Auxiliary labels consist of auxiliaries (AUX) and passive auxiliaries (AUXPASS). The *getAuxiliaryLabel*(C) method in Algorithm D.6 shows how auxiliary labels are distinguished. Note that a passive auxiliary is supposed to modify only a past participle (VBN), which is sometimes annotated as a past tense verb (VBD). The condition in lines 5 and 8 resolves such an erroneous case. Lines 6-7 are added to handle the case of coordination where vp_1 is just an umbrella constituent that groups `VP` conjuncts together.

Algorithm D.6 : *getAuxiliaryLabel*(C)

Input: A constituent C whose parent is `VP|SINV|SQ`.
Output: `AUX` or `AUXPASS` if C is an auxiliary or a passive auxiliary; otherwise, `null`.

```
1:  if C is MD|TO return AUX                                   # Section D.3.1
2:  if (C is VB*) and (C contains VP) then
3:    if C is be|become|get then
4:      let vp_1 be the first VP in C
5:      if vp_1 contains VBN|VBD return AUXPASS                # Section D.3.2
6:      if (vp_1 contains no VB*) and (vp_1 contains VP) then  # for coordination
7:        let vp_2 be the first VP in vp_1
8:        if vp_2 contains VBN|VBD return AUXPASS
9:    return AUX
10: return null
```

D.3.1 AUX: auxiliary

An auxiliary is an auxiliary or modal verb that gives further information about the main verb (e.g., tense, aspect). The preposition *to*, used for infinitive, is also considered an `AUX`. Auxiliary verbs for passive verbs are assigned with a separate dependency label `AUXPASS` (Section D.3.2).

(1)	I [have] [been] seeing her	`AUX`(seeing, have), `AUX`(seeing, been)
(2)	I [will] meet her tomorrow	`AUX`(meet, will)
(3)	I [am] [going] [to] meet her tomorrow	`AUX`(meet, am), `AUX`(meet, going), `AUX`(meet, to)

D.3.2 AUXPASS: passive auxiliary

A passive auxiliary is an auxiliary verb, *be*, *become*, or *get*, that modifies a passive verb.

(1)	I [am] drawn to her	`AUXPASS`(drawn, am)
(2)	We will [get] married	`AUXPASS`(married, get)
(3)	She will [become] nationalized	`AUXPASS`(nationalized, become)

D.4 Complements

Complement labels consists of adjectival complements (`ACOMP`), clausal complements (`CCOMP`), and open clausal complements (`XCOMP`). Additionally, complementizers (`COMPLM`) are included to indicate the beginnings of clausal complements.

D.4.1 `ACOMP`: adjectival complement

An adjectival complement is an adjective phrase that modifies the head of a `VP|SINV|SQ`, that is usually a verb.

(1)	She looks [so beautiful]	`ACOMP`(looks, beautiful)
(2)	Please make [sure to invite her]	`ACOMP`(make, sure)
(3)	Are you [worried]	`ACOMP`(Are, worried)
(4)	[Most important] is your heart	`ACOMP`(is, important), `NSUBJ`(is, heart)

In (4), *Most important* is topicalized such that it is considered an `ACOMP` of *is* although it is in the subject position; *your heart* is considered a nominal subject (`NSUBJ`) of *is*.

D.4.2 `CCOMP`: clausal complement

A clausal complement is a clause with an internal subject that modifies the head of an `ADJP|ADVP|NML|NP|WHNP|VP|SINV|SQ`. For `NML|NP|WHNP`, a clause is considered a `CCOMP` if it is neither a infinitival modifier (Section D.7.3), a participial modifier (Section D.7.7), nor a relative clause modifier (Section D.7.10).

(1)	She said [(that) she wanted to go]	`CCOMP`(said, wanted)
(2)	I am not sure [what she wanted]	`CCOMP`(sure, wanted)
(3)	She left no matter [how I felt]	`CCOMP`(matter, felt)
(4)	I don't know [where she is]	`CCOMP`(know, is)
(5)	She asked [should we meet again]	`CCOMP`(asked, meet)

(6)	I asked [why did you leave]	`CCOMP`(asked, leave)
(7)	I said [may God bless you]	`CCOMP`(said, bless)
(8)	The fact [(that) she came back] made me happy	`CCOMP`(fact, came)

In (4), *where she is* is considered a `CCOMP` although it carries arbitrary locative information. Clauses such as polar questions (5), *wh*-questions (6), or inverted declarative sentences (7) are also considered `CCOMP`. A clause with an adverbial function tag is not considered a `CCOMP`, but an adverbial clause modifier (Section D.5.1).

Algorithm D.7 : *isClausalComplement*(C)

```
 Input:  A constituent C whose parent is ADJP|ADVP|NML|NP|WHNP|VP|SINV|SQ.
Output:  True if C is a clausal complement; otherwise, False.

1: if C is S|SQ|SINV|SBARQ return True
2: if C is SBAR then
3:   if C contains a wh-complementizer return True
4:   if C contains a null complementizer, 0 return True
5:   if C contains a complementizer, if, that, or whether then
6:     set the dependency label of the complementizer to COMPLM   # Section D.4.4
7:     return True
8: return False
```

D.4.3 `XCOMP`: open clausal complement

An open clausal complement is a clause without an internal subject that modifies the head of an `ADJP|ADVP|VP|SINV|SQ`.

(1)	I want [to go]	`XCOMP`(want, go)
(2)	I am ready [to go]	`XCOMP`(ready, go)
(3)	It is too soon [to go]	`XCOMP`(soon, go)
(4)	He knows [how to go]	`XCOMP`(knows, go)
(5)	What do you think [happend]	`XCOMP`(think, happened)
(6)	He forced [me] [to go]	`DOBJ`(forced, me), `XCOMP`(forced, go)
(7)	He expected [[me] to go]	`CCOMP`(expected, go), `NSUBJ`(me, go)

In (7), *me to go* is not considered an `XCOMP` but a clausal complement (`CCOMP`) because *me* is considered a nominal subject (`NSUBJ`) of *go* (see Section 2.5.4 for more examples of open clauses).

Algorithm D.8 : *isOpenClausalComplement*(C)

Input: A constituent C whose parent is `ADJP|ADVP|VP`.
Output: `True` if C is an open clausal complement; otherwise, `False`.

```
1: if $C$ is S then
2:   return ($C$ contains VP) and (the subject of $C$ is an empty category)
3: if ($C$ is SBAR) and ($C$ contains a null complementizer) then
4:   let $c$ be $S$ in $C$
5:   return isOpenClausalComplement($c$)
6: return False
```

D.4.4 `COMPLM`: complementizer

A complementizer is a subordinating conjunction, *if*, *that*, or *whether*, that introduces a clausal complement (Section D.4.2). A `COMPLM` is assigned when a clausal complement is found (see the line 6 of *isClausalComplement*(C) in Section D.4.2).

(1)	She said [that] she wanted to go	`COMPLM`(wanted, that)
(2)	I wasn't sure [if] she liked me	`COMPLM`(liked, if)
(3)	I wasn't sure [whether] she liked me	`COMPLM`(liked, whether)

D.5 Modifiers: adverbial related

Adverbial related modifiers consist of adverbial clause modifiers (`ADVCL`), adverbial modifiers (`ADVMOD`), markers (`MARK`), negation modifiers (`NEG`), and noun phrases as adverbial modifiers (`NPADVMOD`).

Algorithm D.9 : *hasAdverbialTag*(C)

```
  Input:  A constituent $C$.
 Output:  True if $C$ has an adverbial function tag; otherwise, False.

1: if $C$ has ADV|BNF|DIR|EXT|LOC|MNR|PRP|TMP|VOC return True
2: return False
```

D.5.1 `ADVCL`: adverbial clause modifier

An adverbial clause modifier is a clause that acts like an adverbial modifier. A clause with an adverbial function tag (see *hasAdverbialTag*(C)) is considered an `ADVCL`. Additionally, a subordinate clause or an open clause is considered an `ADVCL` if it does not satisfy any other dependency relation (see Appendices D.4.2 and D.4.3 for more details about clausal complements).

(1)	She came [as she promised]	`ADVCL`(came, promised)
(2)	She came [to see me]	`ADVCL`(came, see)
(3)	[Now that she is here] everything seems fine	`ADVCL`(seems, is)
(4)	She would have come [if she liked me]	`ADVCL`(come, liked)
(5)	I wasn't sure [if she liked me]	`CCOMP`(sure, liked)

In (2), *to see me* is an `ADVCL` (with a semantic role, purpose) although it may appear to be an open clausal complement of *came* (Section D.4.3). In (4) and (5), *if she liked me* is considered an `ADVCL` and a clausal complement (`CCOMP`), respectively. This is because *if* in (3) creates a causal relation between the matrix and subordinate clauses whereas it does not serve any purpose other than introducing the subordinate clause in (4), just like a complementizer *that* or *whether*.

D.5.2 ADVMOD: adverbial modifier

An adverbial modifier is an adverb or an adverb phrase that modifies the meaning of another word. Other grammatical categories can also be ADVMOD if they modify adjectives.

(1)	I did [not] know her	ADVMOD(know, not)
(2)	I invited her [[as] well]	ADVMOD(invited, well), ADVMOD(well, as)
(3)	She is [already] [here]	ADVMOD(is, already), ADVMOD(is, here)
(4)	She is [so] beautiful	ADVMOD(beautiful, so)
(5)	I'm not sure [any] more	ADVMOD(more, any)

In (5), *any* is a determiner but considered an ADVMOD because it modifies the adjective, *more*.

Algorithm D.10 : *isAdverbialModifier*(C)

```
  Input:  A constituent $C$.
 Output:  True if $C$ is an adverbial function tag; otherwise, False.

1: if $C$ is ADVP|RB*|WRB then
2:    let $P$ be the parent of $C$
3:    if ($P$ is PP) and ($C$'s previous sibling is IN|TO) and ($C$ is the last child of $P$) return False
4: return True
```

D.5.3 MARK: maker

A marker is a subordinating conjunction (e.g., *although*, *because*, *while*) that introduces an adverbial clause modifier (Section D.5.1).

(1)	She came [as she promised]	MARK(promised, as)
(2)	She came [because she liked me]	MARK(liked, because)

Algorithm D.11 : *setMarker*(C, P)

```
  Input:  Constituents $C$ and $P$, where $P$ is the parent of $C$.
 Output:  If $C$ is a marker, set its label to MARK.

1: if ($P$ is SBAR) and ($P$ is ADVCL) and ($C$ is IN|DT|TO) then $C$.label $\leftarrow$ MARK
```

The *setMarker*(C, P) method is called after P is identified as an adverbial modifier.

D.5.4 NEG: negation modifier

A negation modifier is an adverb that gives negative meaning to its head.

(1)	She decided not to come	NEG(come, not)
(2)	She didn't come	NEG(come, n't)
(3)	She never came	NEG(came, never)
(4)	This cookie is no good	NEG(is, no)

Algorithm D.12 : *setNegationModifier*(D)*

```
Input:  A dependent $D$.
Output: If $D$ is a negation modifier, set its label to NEG.

1: if ($D$ is NEG) and ($D$ is never|not|n't|'nt|no) then $D$.label ← NEG
```

D.5.5 NPADVMOD: noun phrase as adverbial modifier

An adverbial noun phrase modifier is a noun phrase that acts like an adverbial modifier. A noun phrase with an adverbial function tag (see *hasAdverbialTag*(C)) is considered an NPADVMOD. Moreover, a noun phrase modifying either an adjective or an adverb is also considered an NPADVMOD.

(1)	Three times [a week]	NPADVMOD(times, week)
(2)	It is [a bit] surprising	NPADVMOD(surprising, bit)
(3)	[Two days] ago	NPADVMOD(ago, days)
(4)	It [all] feels right	NPADVMOD(feels, all)
(5)	I wrote the letter [myself]	NPADVMOD(wrote, myself)
(6)	I met her [last week]	NPADVMOD(met, week)
(7)	She lives [next door]	NPADVMOD(lives, door)

In (6) and (7), both *last week* and *next door* are considered NPADVMOD although they have different semantic roles, temporal and locative, respectively. These semantic roles can be retrieved from function tags and preserved as additional information (Section 2.6.1).

D.6 Modifiers: coordination related

Coordination related modifiers consist of conjuncts (`CONJ`), coordinating conjunctions (`CC`), and pre-correlative conjunctions (`PRECONJ`).

D.6.1 `CONJ`: conjunct

A conjunct is a dependent of the leftmost conjunct in coordination. The leftmost conjunct becomes the head of a coordinated phrase. Section 2.3.3 describes how conjuncts are derived.

(1)	John, [Mary], and [Sam]	`CONJ`(John, Mary), `CONJ`(John, Sam)
(2)	John, [Mary], and [so on]	`CONJ`(John, Mary), `CONJ`(John, on)
(3)	John, [Mary], [Sam], [etc.]	`CONJ`(John, Mary), `CONJ`(John, Sam), `CONJ`(John, etc.)

Although there is no coordinating conjunction in (3), the phrase is considered coordinated because of the presence of *etc.*

D.6.2 `CC`: coordinating conjunction

A coordinating conjunction is a dependent of the leftmost conjunct in coordination.

(1)	John, Mary, [and] Sam	`CC`(John, and)
(2)	I know John [[as] [well] as] Mary	`CC`(John, as), `ADVMOD`(as, as), `ADVMOD`(as, well)
(3)	[And], I know you	`CC`(know, And)

In (1), *and* becomes a `CC` of *John*, which is the leftmost conjunct. In (2), *as well as* is a multi-word expression so the dependencies between *as* and the others are not so meaningful but there to keep the tree connected. In (3), *And* is supposed to join the following clause with its preceding clause; however, since we do not derive dependencies across sentences, it becomes a dependent of the head of this clause, *know*.

Algorithm D.13 : *isCoordinatingConjunction*(C)

Input: A constituent C.
Output: `True` if C is a coordinating conjunction; otherwise, `False`.

```
1: return $C$ is CC|CONJP
```

D.6.3 `PRECONJ`: pre-correlative conjunction

A pre-correlative conjunction is the first part of a correlative conjunction that becomes a dependent of the first conjunct in coordination.

(1)	[Either] John [or] Mary	`PRECONJ`(John, Either), `CC`(John, or), `CONJ`(John, Mary)
(2)	[Not only] John [but also] Mary	`PRECONJ`(John, Not), `CC`(John, but), `CONJ`(John, Mary)

Algorithm D.14 : *isPreCorrelativeConjunction*(C)

Input: A constituent C.
Output: `True` if C is a pre-correlative conjunction; otherwise, `False`.

```
1: if ($C$ is CC) and ($C$ is both|either|neither|whether) return True
2: if ($C$ is CONJP) and ($C$ is not only) return True
3: return False
```

D.7 Modifiers: noun phrase related

Noun phrase related modifiers consist of appositional modifiers (`APPOS`), determiners (`DET`), infinitival modifiers (`INFMOD`), modifiers of nominals (`NMOD`), noun compound modifiers (`NN`), numeric modifiers (`NUM`), participial modifiers (`PARTMOD`), possessive modifiers (`POSSESSIVE`), predeterminers (`PREDET`), and relative clause modifiers (`RCMOD`).

Algorithm D.15 : *getNonFiniteModifierLabel*(C)

```
  Input:  A constituent C whose parent is NML|NP|WHNP.
 Output:  INFMOD or PARTMOD.

 1: if isOpenClausalComplement(C) or (C is VP) then   # Section D.4.3
 2:   if isInfinitivalModifier(C) return INFMOD        # Section D.7.3
 3: return PARTMOD                                    # Section D.7.7
```

Algorithm D.16 : *getNounModifierLabel*(C)

```
  Input:  A constituent C whose parent is NML|NP|NX|WHNP.
 Output:  AMOD, DET, NN, NUM, POSSESSIVE, PREDET, or NMOD.

 1: if C is VBG|VBN return AMOD        # Section D.10.1
 2: if C is DT|WDT|WP return DET       # Section D.7.2
 3: if C is PDT return PREDET          # Section D.7.9
 4: if C is NML|NP|FW|NN* return NN    # Section D.7.5
 5: if C is CD|QP return NUM           # Section D.7.6
 6: if C is POS return POSSESSIVE      # Section D.7.8
 7: return NMOD                        # Section D.7.4
```

D.7.1 `APPOS`: appositional modifier

An appositional modifier of an `NML|NP` is a noun phrase immediately preceded by another noun phrase, which gives additional information to its preceding noun phrase. A noun phrase with an adverbial function tag (Section D.5.1) is not considered an `APPOS`. Section 2.3.2 describes how appositional modifiers are derived.

(1)	John, [my brother]	`APPOS`(John, bother)
(2)	The year [2012]	`APPOS`(year, 2012)
(3)	He [himself] bought the car	`APPOS`(He, himself)
(4)	Computational Linguistics [(CL)]	`APPOS`(Linguistics, CL)
(5)	The book, Between You and Me	`APPOS`(book, Between)
(6)	MacGraw-Hill Inc., New York	`NPADVMOD`(Inc., York)

D.7.2 `DET`: determiner

A determiner is a word token whose POS tag is `DT|WDT|WP` that modifies the head of a noun phrase.

(1) [The] US military	`DET`(military, The)
(2) [What] kind of movie is this	`DET`(movie, What)

D.7.3 `INFMOD`: infinitival modifier

An infinitival modifier is an infinitive clause or phrase that modifies the head of a noun phrase.

(1)	I have too much homework [to do]	`INFMOD`(homework, do)
(2)	I made an effort [to come]	`INFMOD`(effort, come)

Algorithm D.17 : *isInfinitivalModifier*(C)

```
  Input:  A constituent $C$ whose parent is NML|NP|WHNP.
 Output:  True if $C$ is an infinitival modifier; otherwise, False.

 1: if $C$ is VP then $vp \leftarrow C$
 2: else
 3:    let $t$ be the first descendant of $C$ that is VP
 4:    $vp \leftarrow$ ($t$ exists) ? $t$ : null
 5: if $vp \neq$ null then
 6:    let $t$ be the first child of $vp$ that is VP
 7:    while $t$ exists do
 8:       $vp \leftarrow t$
 9:       if $vp$'s previous sibling is TO return True
10:       let $t$ be the first child of $vp$ that is VP
11:    if $vp$ contains TO return True
12: return False
```

D.7.4 NMOD: modifier of nominal

A modifier of nominal is any unclassified dependent that modifies the head of a noun phrase.

D.7.5 NN: noun compound modifier

A noun compound modifier is any noun that modifies the head of a noun phrase.

(1) The [US] military	PREDET(military, US)
(2) The [video] camera	PREDET(camera, video)

D.7.6 NUM: numeric modifier

A numeric modifier is any number or quantifier phrase that modifies the head of a noun phrase.

(1) [14] degrees	NUM(degrees, 14)
(2) [One] nation, [fifty] states	NUM(nation, One), NUM(states, fifty)

D.7.7 PARTMOD: participial modifier

A participial modifier is a clause or phrase whose head is a verb in a participial form (e.g., gerund, past participle) that modifies the head of a noun phrase.

(1)	I went to the party [hosted by her]	PARTMOD(party, hosted)
(2)	I met people [coming to this party]	PARTMOD(people, coming)

D.7.8 POSSESSIVE: possessive modifier

A possessive modifier is a word token whose POS tag is POS that modifies the head of a noun phrase.

(1) John['s] car	NMOD(John, 's)

D.7.9 PREDET: predeterminer

A predeterminer is a word token whose POS tag is PDT that modifies the head of a noun phrase.

(1) [Such] a beautiful woman	PREDET(woman, Such)
(2) [All] the books we read	PREDET(books, All)

D.7.10 RCMOD: relative clause modifier

A relative clause modifier is a either relative clause or a reduced relative clause that modifies the head of an NML|NP|WHNP.

(1)	I bought the car [(that) I wanted]	RCMOD(car, wanted)
(2)	I was the first person [to buy this car]	INFMOD(person, buy)
(3)	This is the car [for which I've waited]	RCMOD(car, waited)
(4)	It is a car [(that is) worth buying]	RCMOD(car, worth)

In (2), *to buy this car* is considered an infinitival modifier (INFMOD) although it contains an empty *wh*-complementizer in the constituent tree. (4) shows an example of a reduced relative clause.

Algorithm D.18 : *isRelativeClauseModifier*(C)

```
 Input:   A constituent $C$ whose parent is NML|NP|WHNP.
Output:   True if $C$ is a relative clause modifier; otherwise, False.

1: if $C$ is RRC return True
2: if ($C$ is SBAR) and ($C$ contains a wh-complementizer) return True
3: return False
```

D.8 Modifiers: prepositional phrase related

Prepositional phrase related modifiers consist of complements of prepositions, objects of prepositions, and prepositional modifiers.

Algorithm D.19 : *getPrepositionModifierLabel*(C)

```
 Input:   A constituent $C$ whose parent is NP|WHPP.
Output:   POBJor PCOMP.

1: if $C$ is NP|NML return POBJ   # Section D.8.2
2: return PCOMP                   # Section D.8.1
```

D.8.1 PCOMP: complement of a preposition

A complement of a preposition is any dependent that is not a POBJ but modifies the head of a prepositional phrase.

(1)	I agree with [what you said]	PCOMP(with, said)

D.8.2 POBJ: object of a preposition

An object of a preposition is a noun phrase that modifies the head of a prepositional phrase, which is usually a preposition but can be a verb in a participial form such as VBG.

(1)	On [the table]	POBJ(On, table)
(2)	Including us	POBJ(Including, us)
(3)	Given us	POBJ(Given, us)

D.8.3 PREP: prepositional modifier

A prepositional modifier is any prepositional phrase that modifies the meaning of its head.

(1)	Thank you [for coming [to my house]]	PREP(Thank, for), PREP(coming, to)
(2)	Please put your coat [on the table]	PREP(put, on)
(3)	Or just give it [to me]	PREP(give, to)

In (1), *to my house* is a PREP carrying a semantic role, direction. These semantic roles are preserved as additional information in our representation (Section 2.6.1). In (2), *on the table* is a PREP, which is considered the locative complement of *put* in some linguistic theories. Furthermore, in (3), *to me* is the dative object of *give* in the unshifted form, which is also considered a PREP in our analysis. This kind of information is also preserved with syntactic function tags in our representation (Section 2.6.2).

D.9 Modifiers: quantifier phrase related

Quantifier phrase related modifiers consist of number compound modifiers (`NUMBER`) and quantifier phrase modifiers (`QUANTMOD`).

D.9.1 `NUMBER`: number compound modifier

A number compound modifier is a cardinal number that modifies the head of a quantifier phrase.

(1)	[Seven] million dollars	`NUMBER`(million, Seven), `NUM`(dollars, million)
(2)	[Two] to [three] hundred	`NUMBER`(hundred, Two), `NUMBER`(hundred, three)

D.9.2 `QUANTMOD`: quantifier phrase modifier

A quantifier phrase modifier is a dependent of the head of a quantifier phrase.

(1)	[More] [than] five	`AMOD`(five, More), `QUANTMOD`(five, than)
(2)	[Five] [to] six	`QUANTMOD`(six, Five), `QUANTMOD`(six, to)

Quantifier phrases often form a very flat hierarchy, which makes it hard to derive correct dependencies for them. In (1), *More than* is a multi-word expression that should be grouped into a separate constituent (e.g., [More than] one); however, this kind of analysis is not used in our constituent trees. Thus, *More* and *than* become an `AMOD` and a `QUANTMOD` of *five*, respectively. In (2), *to* is more like a conjunction connecting *Five* to *six*, which is not explicitly represented. Thus, *Five* and *to* become `QUANTMOD`s of *six* individually. More analysis needs to be done to derive correct dependencies for quantifier phrases, which will be explored in future work.

D.10 Modifiers: miscellaneous

Miscellaneous modifiers consists of adjectival modifiers (`AMOD`), unclassified dependents (`DEP`), interjections (`INTJ`), meta modifiers (`META`), parenthetical modifiers (`PARATAXIS`), possession modifiers (`POSS`), particles (`PRT`), punctuation (`PUNCT`), and roots (`ROOT`).

D.10.1 `AMOD`: adjectival modifier

An adjectival modifier is an adjective or an adjective phrase that modifies the meaning of another word, usually a noun.

(1)	A [beautiful] girl	`AMOD`(girl, beautiful)
(2)	A [five year old] girl	`AMOD`(girl, old)
(3)	[How many] people came	`AMOD`(people, many)

D.10.2 `DEP`: unclassified dependent

An unclassified dependent is a dependent that does not satisfy conditions for any other dependency.

D.10.3 `INTJ`: interjection

An interjection is an expression made by the speaker of an utterance.

(1)	[Well], it is my birthday	`INTJ`(is, Well)
(2)	I [um] will throw a party	`INTJ`(throw, um)

Algorithm D.20 : *isInterjection*(C)

```
Input:  A constituent $C$.
Output: True if $C$ is an interjection; otherwise, False.

1: return $C$ is INTJ|UH
```

D.10.4 `META`: meta modifier

A meta modifier is code (1), embedded (2), or meta (3) information that is randomly inserted in a phrase or clause.

(1)	[choijd] My first visit	`META`(visit, choijd)
(2)	I visited Boulder and {others} [other cities]	`META`(Boulder, others), `CONJ`(Boulder, cities)
(3)	[Applause] Thank you	`META`(Thank, Applause)

Algorithm D.21 : *isMetaModifier*(C)

```
Input:  A constituent $C$.
Output: True if $C$ is a meta modifier; otherwise, False.

1: return $C$ is CODE|EDITED|EMBED|LST|META
```

D.10.5 PARATAXIS: parenthetical modifier

A parenthetical modifier is an embedded chunk, often but not necessarily surrounded by parenthetical notations (e.g,. brackets, quotes, commas, etc.), which gives side information to its head.

(1)	She[, I mean,] Mary was here	PARATAXIS(was, mean)
(2)	[That is to say,] John was also here	PARATAXIS(was, is)

Algorithm D.22 : *isParentheticalModifier*(C)

```
Input:  A constituent $C$.
Output: True if $C$ is a parenthetical modifier; otherwise, False.

1: return $C$ is PRN
```

D.10.6 POSS: possession modifier

A possession modifier is either a possessive determiner (PRP$) or a NML|NP|WHNP containing a possessive ending that modifies the head of a ADJP|NML|NP|QP|WHNP.

(1)	I bought [his] car	POSS(car, his)
(2)	I bought [John's] car	POSS(car, John)
(3)	This building is [Asia's] largest	POSS(largest, Asia)

Note that *Asia's* in (3) is a POSS of *largest*, which is an adjective. Such an expression does not occur often but we anticipate it to appear more when dealing with informal texts (e.g., text-messages, conversations, web-texts).

Algorithm D.23 : *isPossessionModifier*(C)

```
Input:  Constituents $C$ and $P$, where $P$ is the parent of $C$.
Output: True if $C$ is a possession modifier; otherwise, False.

1: if $C$ is PRP$ return True
2: if $P$ is ADJP|NML|NP|QP|WHNP then
3:     return $C$ contains POS
4: return False
```

D.10.7 PRT: particle

A particle is a preposition in a phrasal verb that forms a verb-particle construction.

(1) Shut [down] the machine PRT(Shut, down)

(2) Shut the machine [down] PRT(Shut, down)

D.10.8 PUNCT: punctuation

Any punctuation is assigned the dependency label PUNCT.

Algorithm D.24 : *isPunctuation*(C)

```
Input:  A constituent $C$.
Output: True if $C$ is punctuation; otherwise, False.

1: return ($C$ is :|,|.|“|”|-LRB-|-RRB-|HYPH|NFP|SYM|PUNC)
```

D.10.9 ROOT: root

A root is the root of a tree that does not depend on any node in the tree but the artificial root node whose ID is 0. A tree can have multiple roots only if the top constituent contains more than one child in the original constituent tree (this does not happen with the OntoNotes Treebank but happens quite often with medical corpora).