Graph Transformations for Natural Language Processing

Printed and bound by Print Partners Ipskamp B.V.

ISBN-10: 90-9021296-5
ISBN-13: 978-90-9021296-8

Graph Transformations for Natural Language Processing

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof.mr. P.F. van der Heijden
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Aula der Universiteit
op dinsdag 28 november 2006, te 11.00 uur

door

Valentin Borisovich Zhizhkun

geboren te Sint-Petersburg, Rusland

Promotor: Prof.dr. M. de Rijke
Faculteit: Faculteit der Natuurwetenschappen, Wiskunde en Informatica

SIKS Dissertation Series No. 2006-18
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

The investigations were supported by the Netherlands Organization for Scientific Research (NWO) under project number 220-80-001.

Маме

Acknowledgments

I began my doctoral studies in Amsterdam in 2002, when I joined the *Language and Inference Technology (LIT)* group led by Maarten de Rijke, at that time a part of the *Institute for Logic, Language and Computation (ILLC)*. I started to work within the ambitious and exciting project *Computing with Meaning*, and computational logic and automated reasoning for description and hybrid logics were my main topics. Amsterdam was just the right place to do research in this area! It was impossible for me not to get thrilled by the beauty of modal and hybrid logics, with so many enthusiastic logicians right across the corridor: Carlos Areces, Juan Heguiabehere, Maarten Marx, Maarten de Rijke, Stefan Schlobach, and with the ILLC headquarters just across the canal. My affair with logic, however, turned out not to be a lasting one, which I blame on the amazing versatility of LIT and ILLC. I became interested in computational semantics of natural language, and soon found myself writing my first paper on a particular problem in natural language processing—the field that I discovered in Amsterdam with much help of my colleagues at LIT.

During my PhD years at LIT and, later, at the Information and Language Processing Systems group (ILPS), I had an opportunity to work on natural language processing tasks ranging from syntactic and semantic text analysis to Information Retrieval and Question Answering. I learned how to set up experiments and analyze results, how to write papers and give talks, how to find the right questions and how to search for answers. I am grateful to my colleagues at LIT and ILPS who taught me this and who continue to teach a lot more.

I'm grateful to my teachers in Russia, without whom I would never have started my PhD: to Elena Barashko for making me seek solutions, to Galina Frolova for always challenging me, to Lidia Volkhonskaya for teaching me to think for myself, to Tamara Prokofieva for teaching me to speak for myself, to Andrey Terekhov for turning programming into a science and a fun for me, and to Vladimir Orevkov for eliminating big cuts in my understanding of proofs.

I would not have been be able to keep trying and working on my thesis without my friends and colleagues in Amsterdam, Gatchina, Moscow, St.-Petersburg, The Hague: Andrea who designed the cover and made my life so much brighter, Irina whose warm home became mine, Iuliana who can magically share my ups and downs, Joris who can help with anything, even with the Dutch *Samenvatting*, Juan who welcomed me in Amsterdam, Khalil who introduced me to the world of language processing, Kseniya who makes me feel soaring and quiet, Lenka who should eat and write more, Maarten (Marx) whose picky remarks always hit the mark, Oksana and Lida who keep Russia a warm home for me, Tanya who is always near even when far away, Vlad with his incorrigible optimism.

I'm very grateful to the members of my reading committee: Pieter Adriaans, Johan Bos, Walter Daelemans, Eduard Hovy, Maarten Marx and Remko Scha. Their comments and suggestions helped me have a look at my work from several different angles and make my ideas clearer.

Finally, I would like to thank my supervisor Maarten de Rijke without whom my thesis would never have been finished. I'm very grateful for his patience and support, for letting me find my own way in science, but always being ready to help when it gets hard, for countless inspiring conversations and nice chats. All imperfections in the thesis are mine, but for many of the thoughts, ideas and remarks in the thesis I'm indebted to Maarten.

Valentin Jijkoun
Amsterdam, October 2006

Contents

Chapter 1

Introduction

There is a growing need for effective and robust, wide-coverage yet focused access to the constantly increasing volume of textual information available on the Web, in digital libraries and other text repositories. Complex information access tasks, such as Question Answering (Voorhees and Trang Dang, 2006), provide the Natural Language Processing (NLP) community with new and deep challenges. Such tasks require efficient language processing tools that are capable of recovering complex, deep structure in natural language texts, ranging from part-of-speech tags, syntactic chunks and named entities to syntactic, semantic and pragmatic analyses of text (Bouma *et al.*, 2005; Jijkoun *et al.*, 2004; Moldovan *et al.*, 2003a; Webber *et al.*, 2002).

The increased attention to deeper levels of text processing has led to an explosion of methods, each addressing a specific problem and each defined for specific syntactic and/or semantic formalisms. While many of the approaches are similar in their use of well-established statistical techniques, differences pertaining to the tasks addressed and to the language resources used often make it difficult to identify these similarities, to compare the techniques, and to re-use the ideas or the actual software for other, but related problems. This raises several important questions. Can approaches developed for particular tasks, operating with specific syntactic and semantic formalisms be transferred to other similar tasks? How important are the details of a particular syntactic representation (e.g., syntactic constituency trees or syntactic dependency structures) for the success of a particular semantically oriented task? How would using a different syntactic parser, based on a different set of syntactic labels, or even different syntactic formalism, affect the performance of a specific method? How do we combine several different types of linguistic analysis, making all of them easily available to applications?

In this thesis we bring together several NLP tasks related to the syntactic and semantic analysis of English sentences, namely, identification of predicate argument structure, extraction of semantic propositions and arguments, and converting between different syntactic formalisms. These tasks operate with different types of linguistic information about a given text, they aim at identifying different types of linguistic structures, and they are traditionally solved using fairly different methods. The aim of this thesis is to describe a uniform framework that allows us to represent and process different linguistic structures. More specifically, we will advocate a graph-based approach to language technology: various linguistic structures will be viewed as labeled directed graphs and language processing tasks will be cast as graph transformation (rewriting) problems. This uniform view will allow us, on the one hand, to elucidate similarities between different language processing tasks, and on the other hand, to identify and make explicit the parameters and biases needed for our general language processing method to perform well on the specific tasks: particular details of the graph-based representation, types of required graph transformations, types of features used to automatically learn these transformations.

1.1 Research questions

The questions listed in the previous section are often difficult to answer, since the NLP methods described in the literature assume a specific kind of syntactic input (e.g., constituency or dependency trees, with specific labels), and they often depend on the details of the input. Changing the syntactic analyzer for such methods can result in redesigning at least some parts of the employed processing models, whether statistical or symbolic. This motivates the following question, the main research question of this thesis:

Main Research Question 1
Can a general framework and a processing method be developed, that can be efficiently applied to a broad range of natural language processing problems, but are as independent as possible from the specifics of the representation of linguistic information and from the exact details of the language processing tasks being addressed?

In the thesis we will answer the question positively and propose such a single, unified approach to language processing. Specifically, we will propose to embed various types of linguistic structures and several language processing tasks into a framework based on graphs and graph transformations: linguistic structures are represented as directed labeled graphs, and NLP tasks are formulated as graph transformation problems: automatically transforming graphs of one kind into graphs of another, for example, graphs representing syntactic structure of sentences into graphs that also contain semantic information. Moreover, we will describe a general graph transformation method that allows us to automatically learn a sequence of graph rewrite rules for a given transformation problem.

Our answer to our main research question gives rise to a series of more specific questions, which we detail below.

Research Question 2
How well is our proposed graph-based framework suited for representing various types of linguistic structures?

We will address this question by presenting examples of encoding various types of linguistic information as labeled directed graphs, considering syntactic dependency, syntactic constituency, and semantic structures. We will also demonstrate how such encodings enable us to combine different types of linguistic information in one object, a single graph, making all of it accessible to the processing methods in a uniform way.

Research Question 3
How well and how natural can different language processing tasks be formulated as graph transformation problems, and what parameters and details of our graph processing methods need fine-tuning for specific tasks?

In the course of the thesis we will consider several NLP tasks related to syntactic and semantic analysis of text at different levels. We will demonstrate how a graph-based encoding of linguistic structures makes it possible to re-cast these tasks as instances of a general graph transformation problem. In particular, we will apply our graph transformation-based method to the identification of predicate argument structure, to the task of automatically converting between different syntactic formalisms, and to the identification of semantic arguments. For these experiments we will use the same general settings of the method, which will allow us to address the following research question.

Research Question 4
What are the limitations of the method?

We will analyze the performance of the method on several tasks, trying to determine weaknesses and possible workarounds. In particular, when defining our graph transformation-based method for solving NLP tasks we hardwire a small number of heuristics concerning the possible transformation rules it should consider. We will show that, while these heuristics are generally effective for the types of tasks that we consider, different types of tasks may call for different heuristics.

In addition, each of the case studies that we consider gives rise to specific contributions on the task addressed in the case study, as detailed in the corresponding chapters of the thesis.

1.2 Main contributions

The main contribution of the thesis is the introduction of a novel approach to NLP problems based on supervised learning of graph transformations. We describe the method in detail and demonstrate its applicability in a series of case studies:

- the automatic identification of predicate argument structures in the output of a syntactic parser;
- automatically converting syntactic structures from one dependency formalism to another;
- the automatic identification and labeling of semantic arguments.

For each of the case studies we formulate the task in terms of graph transformations. We show that our learning method is general enough to encompass all of them and shows competitive performance for some of them.

1.3 Organization of the thesis

The thesis is organized in 9 chapters, and starts with the present introduction. Next comes Chapter 2, which is a background chapter in which we give an overview of the various types of linguistic information that we will work with in the thesis, ranging from syntactic to shallow semantic, and we show how different linguistic structures can be represented as labeled graphs and how NLP tasks operating with these structures can be formulated as graph transformation problems. This chapter provides answers to our Research Questions 2 and 3.

Chapter 3 gives formal definitions of the concepts that will be important in the rest of the thesis: directed labeled graphs, graph patterns, graph rewrite rules, and graph pair comparisons. The chapter also outlines an informal description of the automatic graph transformation method that will be presented in detail in Chapter 5.

Chapter 4 gives a first example of an application of our graph-based approach to the task of identifying Penn Treebank-style predicate argument structure in English sentences analyzed by a syntactic parser. The purpose of this chapter is to show our graph-based ideas at work at an early stage in the thesis. We describe the task and our proposed method in detail, present the results and compare them to results in the literature. The method of Chapter 4 is based on learning a predefined sequence of simple graph transformations using an ad-hoc set of features extracted from dependency graphs. After introducing the general graph transformation method in Chapter 5, we will come back to this task in Chapter 6, removing the ad-hoc choices to compare both solutions.

Chapter 5 is the core chapter of the thesis, and details our answer to the main research question stated above. Here, we give a detailed description of our method for learning graph transformations from a training corpus of pairs of input and output graphs. The method is based on identifying local mismatches in the input and output graphs and thereby producing a sequence of graph rewrite rules that examine and changes local contexts. In the subsequent chapters we will apply the method to several NLP tasks, re-casting them as graph transformation problems.

As a first case study, in Chapter 6, we describe an application of our general method to the problem that was addressed by an ad-hoc method in Chapter 4: the identification of Penn Treebank-style predicate argument structures (Bies *et al.*, 1995). Apart from giving the first example of our graph transformation-based method at work, the purpose of this chapter is to show that the generality of the method does not compromise the performance and, in fact, is beneficial for the task in question, thus addressing Research Question 3. We also show that the method can be equally well applied both to consistuency- and dependency-based syntactic formalisms, which serves as another demonstration of its generality.

Chapter 7 presents a second case study. Here we apply our graph transformation method to another NLP task: converting between two different syntactic dependency formalisms. Specifically, we consider the task of converting dependency structures produced by Minipar, a syntactic parser of Lin (1994), to dependency structures directly derived from the Penn Treebank. We report on the results of our method for this task and examine the graph rewrite rules extracted by the method.

In Chapter 8 we show yet another application of our graph-based method to the task of identifying semantic arguments, using the data of PropBank (Palmer *et al.*, 2005) and FrameNet (Baker *et al.*, 1998). We will argue that graphs provide a natural way of describing the information in the corpora, and the application of our graph transformation method "out of the box" does demonstrate a reasonable performance for PropBank-based shallow semantic parsing. We will also describe a graph-based method for the task of FrameNet-based semantic role labeling. Our analysis of the application of the method to these tasks, however, will indicate several weak points: in particular, we will show that the default rule selection criteria used successfully for other applications in the thesis, do not work well for the identification of FrameNet roles. This negative result provides important insights into our graph transformation method and will direct further research into its applicability and effectiveness.

We conclude the thesis in Chapter 9 by re-examining our initial research questions, and reviewing our answers and contributions. We also identify a number of important open research questions that follow up on our findings.

Finally, in Appendix A we give a list of function tags and empty node types used in the annotations of the Penn Treebank II.

1.4 Paths through the thesis

We tried to make the chapters of the thesis as independent as possible and the thesis as a whole accessible and interesting for readers with different backgrounds. Therefore, there are different possible paths through the thesis. In particular, Chapter 2 can be skipped by readers familiar with graph-based representations of linguistic structures and with natural language processing. Chapter 3 is important in that it introduces notation used in the rest of the thesis, however the formal definitions of graph alignments and merges (Section 3.5) can be skimmed over. Chapter 4 can be skipped by readers with no special interest for the task of predicate argument structure identification. Chapter 5 is the core of the thesis and only relies on the notation introduced in Chapter 3. Finally, Chapters 6, 7 and 8 can be read in arbitrary order after Chapter 5.

1.5 Origins

The material in this thesis grew out of earlier publications and parallel research activities. The modeling efforts in Chapter 2 find their origins in (Jijkoun, 2003). Early versions of Chapter 4 were published in (Jijkoun, 2003; Jijkoun and de Rijke, 2004). Parts of the thesis (especially Chapters 6 and 8) grew out of (and feed back in to) research on Question Answering, Textual Entailment and Information Extraction (Ahn *et al.*, 2006a,b; Jijkoun and de Rijke, 2005a,b, 2006; Jijkoun *et al.*, 2004). And, finally, part of the work presented in Chapter 8 was first described in (Ahn *et al.*, 2004).

Chapter 2

Linguistic Analysis and Graphs

In this chapter we give several examples of different types of linguistic structures that are frequently in the focus of the NLP community. In later chapters of the thesis, we will use these structures to formulate and provide solutions for various NLP tasks. Now we will show how these types of structures can be easily and naturally represented using directed labeled graphs, and, more importantly, how *combinations* of these structures can be viewed as graphs as well.

In later chapters of the thesis we will heavily rely on the uniformity of our graph-based representations and on the ability to combine linguistic information of different nature and depth in a single graph. This type of uniformity will allow us to treat different NLP tasks simply as instances of a general graph transformation problem.

We start by considering two kinds of syntactic analysis of natural language sentences: dependency structures and phrase trees. We describe both the formalisms and the approaches to evaluating corresponding syntactic parsers. We believe that although these two classes correspond to different views on the nature of syntactic relations in text, they are often interchangeable and complementary. To illustrate the relation between the two formalisms, we will briefly describe a well-known method for transforming phrase structures into dependency graphs. In later chapters of the thesis we will consider NLP problems that involve either phrase or dependency structures, or sometimes both of them.

Then, we describe how other linguistic structures, such as word sequences and part-of-speech tags, predicate argument structures and semantic frames, can be naturally represented as directed labeled graphs. We also show that the choice of graphs as the underlying representation formalism allows for a straightforward combination of different linguistic structures in a single graph.

Finally, we briefly describe concrete NLP tasks, such as recovery of non-local dependencies and grammatical and function tags in bare syntactic parses, identification of predicate argument structure and shallow semantic parsing, and show how these different tasks can be viewed as specific types of graph transformation problems. This transition, the re-casting of NLP tasks as graph transformation problems, is one of the key elements of the method we are presenting in the thesis.

2.1 Syntactic structures as graphs

There are different ways to look at the syntax of a natural language. One may be interested in how words group together, how resulting word groups form yet bigger groups, eventually building up clauses and sentences. Or, one may be interested in what relations exist between the words, how they depend on each other and define each other's properties. Phrase structures or dependency structures? None of these two viewpoints is more "legitimate" that the other. In fact, they are far from being mutually exclusive, and sometimes complement each other, as illustrated, for example, by the success of head-lexicalized Probabilistic Context-Free Grammars for English.

In this thesis we will work with both these types of syntactic representations. Moreover, as we will see in Chapter 6, some of the language processing tasks can be naturally formulated and effectively addressed using either formalism. In particular, the core method of the present thesis, a method for learning graph transformations that we will present in Chapter 5, is designed is such a way that it can be easily parameterized for either phrase or dependency structures.

The main difference between the two formalisms is in the types of objects they operate with, and the types of relations between these objects. For phrase structure-based grammars the main objects are words and phrases and the main relation in containment, or the child-parent relation. Labels, assigned to phrases (constituents), distinguish various phrase types. Figure 2.1(b) shows an example of a phrase tree: the constituency tree for the sentence "*directors this month planned to seek seats*". We will typically depict phrase structures with parents on top. Sometimes we will also used arrows from parents to children.

Dependency-based grammars usually operate with objects of one type: words. To balance the absence of phrases, relations (dependencies) vary much more and are equipped with labels. Figure 2.1(b) shows an example of a labeled dependency structure for the same sentence. When depicting dependency graphs we will use arrows from heads to modifiers (dependents).

2.1.1 Evaluation of syntactic parsers

Since parsers, i.e., syntactic analyzers, exist for both formalisms, parser evaluation schemes have also been defined for each of them. Specifically, the PARSEVAL measure is typically used for comparison of sentence phrase trees produced by a parser to the gold standard phrase trees. In PARSEVAL, one counts precision and recall of correctly identified phrases:

$$\begin{aligned}\text{Precision} &= \frac{\text{\# correctly identified phrases}}{\text{\# phrases returned by the parser}}\\ \text{Recall} &= \frac{\text{\# correctly identified phrases}}{\text{\# phrases in the gold standard tree}}\\ F_1\text{-score} &= \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.\end{aligned}$$

Here, a phrase is identified by the parser if it spans the same set of words of the sentence and has the same label as some phrase in the gold standard parse of the sentence.

For dependency parsers, or sometimes for phrase structure parser as well (Lin, 1998), it is common to compute either precision and recall of labeled dependency relations (similarly to PARSEVAL, but now a dependency is correctly identified by a parser if a dependency with the same head and modifier words and the same label exist in the gold standard tree), or the error rate, the number of words that are assigned different heads.

2.1.2 From phrase structures to dependency structures

As mentioned above, although phrase structures and dependency structures present different views on similar linguistic information, it is possible to go from one to the other. In this section we describe a common method for converting phrase structures to dependency structures, described in detail in (Buchholz, 2002; Xia and Palmer, 2001) among others. The conversion routine, described below, can be applied both to the output of syntactic phrase structure parsers such as parsers of Collins (1999) and Charniak (2000), and to corpora manually annotated with syntactic phrase structures, such as the Penn Treebank. Note that in the latter case, input structures of the conversion routine may contain richer information than the structures obtained from the parsers. For example, the Penn Treebank, apart from syntactic phrases with bare labels, provides indication of non-local dependencies (subjects in control and raising, traces in WH-extraction, etc.). Our variant of the conversion routine preserves as much of this additional information as possible, whenever it is present.

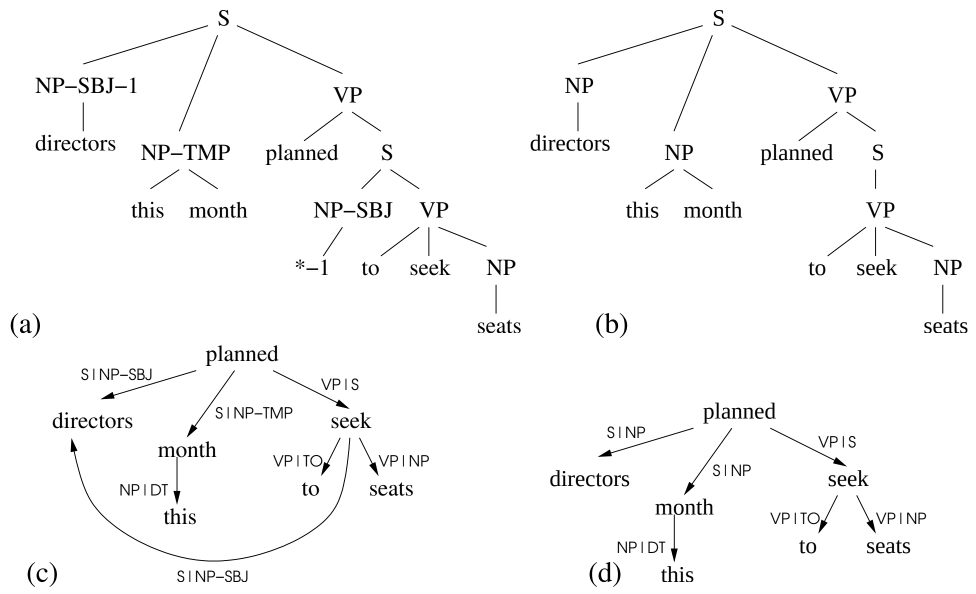


Figure 2.1: Example of (a) the Penn Treebank WSJ annotation, (b) the output of Charniak's parser, and the results of the conversion to dependency structures of (c) the Penn tree and of (d) the parser's output.

First, for the Penn Treebank data (Bies *et al.*, 1995), all non-local dependencies, indicated using co-indexing of tree nodes, are resolved and corresponding empty nodes are replaced with links to target constituents, so that syntactic trees become directed graphs. For *ICH*, *RNR* and *EXP* traces (pseudo-attachments: moved constituents, right node raising and it-extraposition, respectively) the target constituent is removed from its original location. Second, for each constituent we detect its head daughters (first conjunct in the case of conjunction) and recursively identify lexical heads (headwords) of all constituents. Then, for each constituent we output new dependencies between its lexical head and the lexical heads of its non-head daughters. The label of every new dependency is the constituent's phrase label, stripped of all functional tags and co-indexing marks of the Treebank (if present), conjoined with the label of the non-head daughter, with its functional tags but without co-indexing marks.

Figure 2.1 shows an example of the original Penn annotation (a), the output of Charniak's parser (b) and the results of our conversion of these trees to dependency structures (c and d). The interpretation of the dependency labels is straightforward: e.g., the label S|NP-TMP appears when a sentence (phrase label S) is modified by a temporal noun phrase (NP-TMP), as shown in the example.

Evaluation	Parser	unlabeled			bare labels			with func. tags		
		P	R	f	P	R	f	P	R	f
PARSEVAL	Charniak	90.4	90.1	90.2	89.1	88.9	89.0		–	
	Collins	89.8	89.3	89.5	88.5	88.0	88.2		–	
Dependencies	Charniak	89.9	83.9	86.8	85.9	80.1	82.9	68.0	63.5	65.7
	Collins	90.4	83.7	87.0	86.7	80.3	83.4	68.4	63.4	65.8

Table 2.1: PARSEVAL scores and dependency-based evaluation of the parsers of Collins (1999) and Charniak (2000) (section 23 of the Penn Treebank). The dependency evaluation includes non-local dependencies.

The core of the conversion routine is the selection of head daughters of the constituents. Following (Buchholz, 2002; Collins, 1999), we identified constituent heads using a head percolation table, for every constituent label listing possible constituent labels or part-of-speech tags of its head daughters. Our most notable extension of the tables of Collins (1999) is the handling of conjunctions, which are often left relatively flat in the Penn Treebank and, as a result, in the output of parsers trained on the Penn Treebank data. Specifically, we used simple pattern-based heuristics to detect conjuncts and mark the first conjunct as heads of the conjunction and other conjuncts as dependents of the first conjunct, labeling the new dependency with CONJ.

Moreover, after converting phrase trees using this method, we modify every resulting dependency structure as follows:

- Auxiliary verbs (*be, do, have*) are made dependents of corresponding main verbs (similar to modal verbs, which are handled by the head percolation table);
- In order to fix a Penn Treebank inconsistency, we move the -LGS tag (indicating logical subject of passive in a *by*-phrase) from the PP to its daughter NP.

Our converter is available at `http://www.science.uva.nl/~jijkoun/software`.

2.1.3 Dependency-based evaluation of phrase structure parsers

To give a practical example of parser evaluation based on dependency relations, we present such an evaluation for the phrase structure parsers of Collins (1999) and Charniak (2000). We refer to (Bies *et al.*, 1995) for a detailed description of the Penn Treebank annotation.

The phrase structures of the test section of the Penn Treebank Wall Street Journal corpus (section 23) and the parsers' outputs for the sentences of section 23 are converted to dependency structures as described in Section 2.1.2. Then we evaluate the performance of the parsers by comparing the resulting dependency graphs with the graphs derived from the corpus, using three different settings:

- on unlabeled dependencies;
- on dependencies, discarding all Penn function tags;
- on labeled dependencies with function tags.

Note that since neither Collins' nor Charniak's parser produce Penn function labels, all dependencies with function labels in the gold corpus will be judged incorrect in the third evaluation setting. The evaluation results on section 23 are shown in Table 2.1, were we give the precision, the recall and the F_1-score for the three settings. For reference we also give the usual labeled and unlabeled PARSEVAL scores for both parsers.

The dependency-based evaluation is more stringent than PARSEVAL in the sense that the gold dependency corpus contains non-local dependencies (not directly identified by the parsers), whereas empty nodes and non-local dependencies are ignored in the PARSEVAL evaluation. This accounts for the fact that the recall of dependency relations is substantially lower than the recall for constituents. Moreover, as explained above, the low numbers for the dependency evaluation with functional tags are expected, because the two parsers do not produce (and were not intended to produce) function labels.

Interestingly, the relative ranking of the two parsers is different for the dependency-based evaluation than for PARSEVAL: Charniak's parser obtains a slightly higher PARSEVAL score than Collins' (89.0% vs. 88.2%), but slightly lower F_1-score on dependencies without functional tags (82.9% vs. 83.4%).

As the evaluation results show, both parsers perform with F_1-score around 87% on unlabeled dependencies. When evaluating on bare dependency labels (i.e., disregarding functional tags) the performance drops to 83%. The new errors that appear when taking labels into account come from different sources: incorrect POS tags (NN vs. VBG), different degrees of flatness of analyses in gold and test parses (JJ vs. ADJP, or CD vs. QP) and inconsistencies in the Penn annotation (VP vs. RRC). Finally, the performance goes down to around 66% when taking into account functional tags, which, again, are not produced by the parsers.

Having described two of the possible ways of representing syntactic structures, phrase trees and dependency graphs, we now turn to graph-based representations of other types of linguistic information.

2.2 Encoding other linguistic information

In the previous section we described two different approaches to representing natural language syntax: phrase and dependency structures. Both formalisms can naturally be represented using directed node- and edge-labeled graphs. Now we consider other possible types of linguistic analysis. Although each type provides a different view on the linguistic data, we argue that different types of linguistic information are essentially relational and thus allow a straightforward graph-based representation.

To illustrate this, Figure 2.2 shows representations of different annotations of the same sentence: part-of-speech tagging, syntactic parsing and two different types of shallow semantic analysis (PropBank and FrameNet frames). The linguistic structures in all cases are different, but it is fairly straightforward to combine all of them in one graph.

Figure 2.3 gives an example of such a combination. Here we (arbitrarily) chose to represent PropBank and FrameNet semantic arguments by largest corresponding constituents and the target by the target word itself. Other combination schemes are possible, and moreover, even different schemes can be put into the same graph. Other useful linguistic analyses on various levels (morphology, chunking, named entities, non-local syntactic relations, constituent heads, syntactic dependencies, discourse information etc.) can also be added to such graph representation.

As we will show in the following chapters of the thesis, combining different linguistic analyses in a single labeled graph facilitates straightforward simultaneous access to all available information and simple and uniform definitions of different NLP tasks. In fact, this uniformity will allow us to consider various NLP tasks as instances of a generic graph transformation problem.

2.3 Linguistic analysis and graph transformations

As mentioned above, with the graph-centered view on linguistic data, many language processing tasks can be defined in terms of graph transformations. In this section we will briefly look at several possible tasks and the types of transformations they require.

Figure 2.4 gives an example of two similar but different linguistic analyses for the sentence "*asbestos was used in making paper in the 1950s*" (this is a simplified sentence from the Penn Treebank). Although the manual annotation (left) and the parser's output (right) both provide the basic phrase structure of the sentence, the manual annotation also indicates a non-local dependency between *used* and *asbestos* (the curve on the figure), empty nodes (PRO NP, the empty subject of

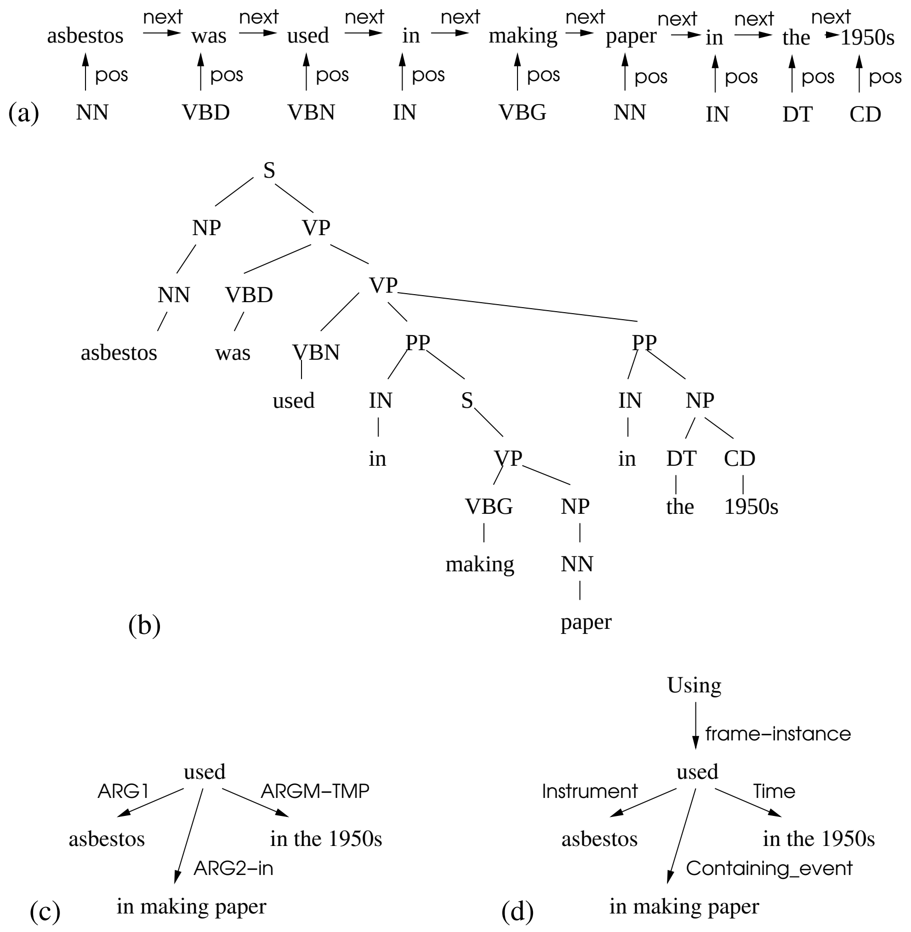


Figure 2.2: Different linguistic annotations of the sentence *asbestos was used in making paper in the 1950s*: (a) word sequence and part-of-speech tagging; (b) syntactic parse tree; (c) semantic propositions from PropBank; (d) FrameNet annotation of frame semantics.

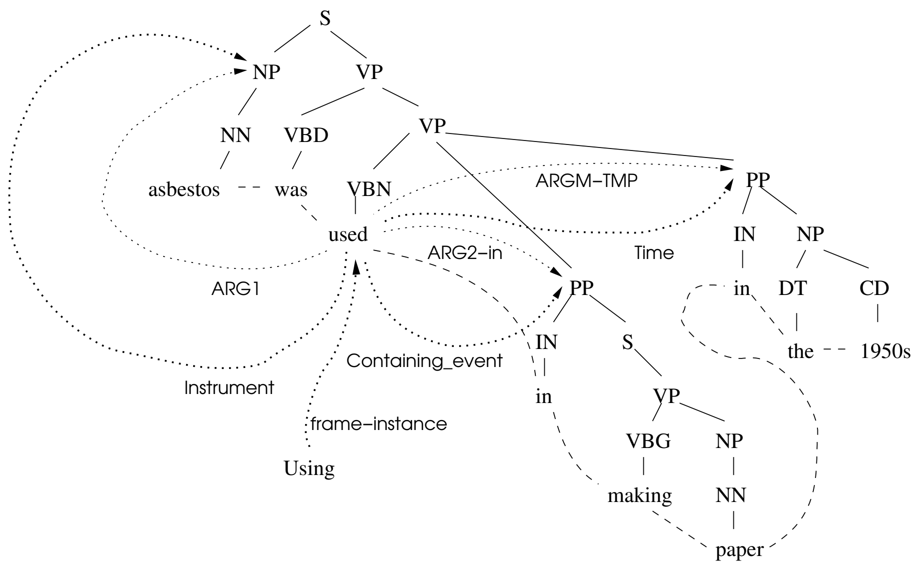


Figure 2.3: Different linguistic annotations combined in one graph. For clarity some arc labels and arc arrows are omitted.

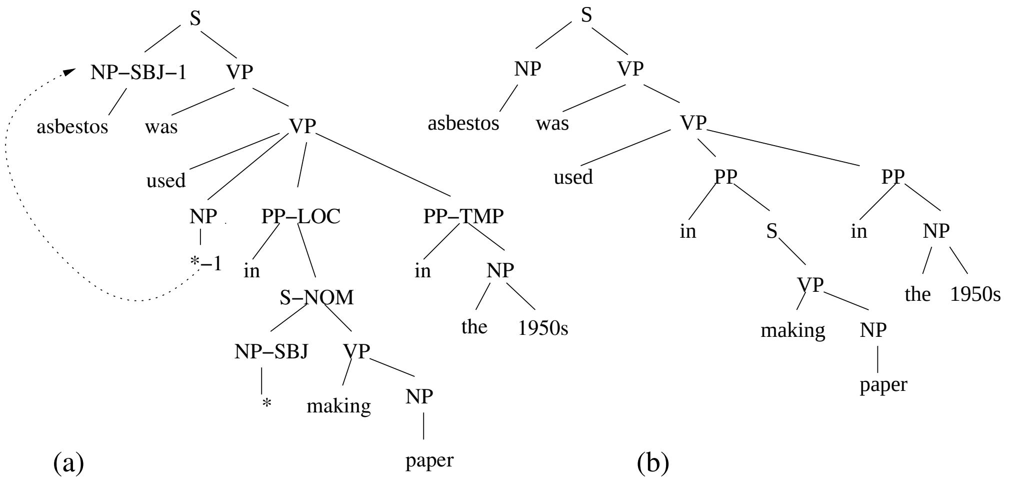


Figure 2.4: (a) the Penn Treebank annotation, (b) the output of Charniak's parser for the sentence *asbestos was used in making paper in the 1950s.*

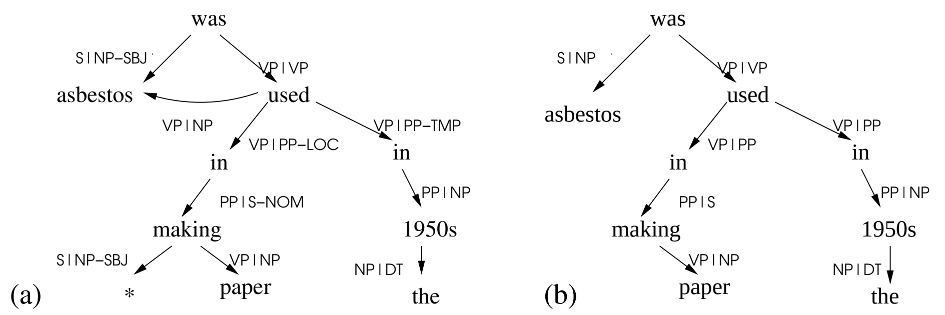


Figure 2.5: Dependency graphs from (a) WSJ annotation, (b) the output of Charniak's parser for the sentence *asbestos was used in making paper in the 1950s.*

making) and some additional functional and semantic information: phrase label tags -TMP (temporal), -LOC (locative), -SBJ (subject), -NOM (nominative).

When we look at the structures as labeled directed graphs (nodes being words and phrases and edges connecting parent and child phrases), we can view these differences as mismatches between the two graphs: different node labels, missing edges, missing nodes, etc. The natural and useful task of adding the missing treebank information to the output of the parser can be seen as the task of transforming graph (b) in Figure 2.4 into graph (a). Moreover, we see that in this case the transformation can be viewed as a sequence of simple graph transformation steps, involving only single nodes and edges.

Similarly, Figure 2.5 displays two dependency graphs for the same sentence: the graphs derived from Penn Treebank (left) and from the parser's output (right). Again, the mismatches in analysis between manual and automatic annotation of the sentence correspond to "atomic" differences between the two graphs: edge labels (PP|S-NOM and PP|S), one missing node (*, on the right graph), two missing edges (between *asbestos* and *used*, and between *making* and *). As before, transforming the parser's output into the annotation from the corpus can be seen as a chain of simple atomic graph transformations.

The transition from the structure in Figure 2.5(b) to the structure in Figure 2.5(a) is a useful and non-trivial graph transformation. Essentially, it recovers information not provided by a parser (empty nodes, non-local relations, functional tags), adding it to the parser's graph.

2.4 Conclusions

In this chapter we have given an overview of various types of linguistic structures that we will come across later in the thesis and we have demonstrated how these structures can easily be represented as graphs. We focused on syntactic phrase structures, syntactic dependency graphs, predicate argument structure and frame semantics. We showed that graphs provide a natural way to uniformly and simultaneously represent various linguistic analyses of the same natural language data, and some NLP tasks can be viewed as graph transformations. In the following chapters of the thesis we will use these representations to address concrete NLP tasks. But before we go into linguistic applications, in the next chapter we make a detour into Computer Science in order to make our notions of *graphs* and *graph transformations* precise.

Chapter 3

Graphs and Graph Transformations

In the previous chapter, we have already seen that various linguistic structures can naturally be thought of and represented as directed labeled graphs, with nodes corresponding to words, syntactic phrases and semantic units, and directed arcs corresponding to linguistic relations. This view has a number of methodological advantages. First, it provides us with unified simultaneous access to different types of linguistic analysis. Second, it allows us to view NLP tasks as instances of graph transformation problems, i.e., transforming graphs of one type to graphs of another type, usually richer and more complex.

In this chapter we lay the basis for our subsequent work on developing such NLP tools within our graph-based paradigm. We will formally introduce the notions of directed labeled graphs, patterns, graph comparison and merging, graph rewrite rules, that we will use throughout the thesis. Finally, we provide a first sketch of our method for learning graph transformations, that will be fully described in its more general form in Chapter 5.

In our presentation of the concepts related to graphs and graph transformations we will generally follow the logic-based approach to structure replacement systems, described in detail in (Schürr, 1997). Of course, other approaches to graph rewrite systems and graph grammars exist, most notably, algebraic, where graphs are represented as sets of nodes and edges accompanied with labeling functions, and graph rewrite rules are defined as node or edge replacements (Drewes *et al.*, 1997; Engelfriet and Rozenberg, 1997). The advantage of the logic-based approach is that it allows us to formulate complex graph patterns, constraints and rewrite rules using the familiar language of first-order predicate logic. Moreover, representing graphs and patterns as sets of atomic predicates will allow us to define graph rewrite rules as simple functions operating on these sets.

We start with basic definitions of graphs.

3.1 Graphs, patterns, occurrences

The main objects used throughout the thesis will be directed labeled graphs. Informally speaking, these are structures consisting of nodes and edges connecting pairs of nodes, with each node and edge possibly assigned any number of named attributes.

We will represent directed attributed graphs as sets of atomic formulas (*atoms*) of first-order predicate logic with equality. Symbols $a, b, c, n, e, \ldots$ (which we will call *objects* or *object identifiers*) refer to nodes and edges of graphs. Symbols $\mathsf{was}, \mathsf{S|NP}, \mathsf{pos}, \ldots$ (called *domain constants* or simply *constants*) refer to labels, attribute names and values. Both objects and domain constants are considered 0-ary function symbols. The predicate symbols node, edge and attr are used to describe nodes, edges and their attributes. More precisely, the predicates are interpreted as follows:

- $\mathsf{node}(n)$: defines n as a graph node;
- $\mathsf{edge}(e, n_1, n_2)$: defines e as a directed edge from n_1 to n_2;
- $\mathsf{attr}(b, \mathsf{attr}, \mathsf{val})$: declares that the object b has an attribute with name attr and value val; and
- $b = c$: declares the equality between objects b and c.

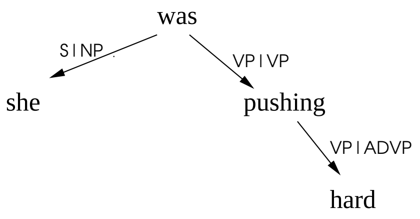
Thus, a *graph* is a set of such predicates, such that

- every object is either a node or edge, but not both;
- endpoints of every edge are declared as nodes;
- the object of an attribute declaration is either a node or an edge.

For a given graph G, we will use N_G to denote the set of nodes of G and E_G its set of edges. Furthermore, $\mathcal{O}_G = N_G \cup E_G$ is the set of all objects of G.

We will interpret sets of formulas as conjunctions, and will use sets (with commas between elements) and conjunctions ($\&$) interchangeably. We will also use the following shorthand to declare that e is an edge from n to m labeled with l:

$$\mathsf{edge}(e, l, n, m) \stackrel{\text{def}}{=} \mathsf{edge}(e, n, m) \ \& \ \mathsf{attr}(e, \mathsf{label}, l).$$



$$\begin{aligned}F = \{ &\mathsf{node}(n_1), \mathsf{node}(n_2), \mathsf{node}(n_3), \mathsf{node}(n_4),\\ &\mathsf{attr}(n_1, \mathsf{label}, \mathsf{was}), \mathsf{attr}(n_2, \mathsf{label}, \mathsf{she}),\\ &\mathsf{attr}(n_3, \mathsf{label}, \mathsf{pushing}), \mathsf{attr}(n_4, \mathsf{label}, \mathsf{hard}),\\ &\mathsf{edge}(e_1, \mathsf{S|NP}, n_1, n_2),\\ &\mathsf{edge}(e_2, \mathsf{VP|VP}, n_1, n_3),\\ &\mathsf{edge}(e_3, \mathsf{VP|ADVP}, n_3, n_4)\}\end{aligned}$$

Figure 3.1: Example of encoding a graph as a list of atomic formulas.

Figure 3.1 shows an example of a labeled graph and the corresponding set (conjunction) of atoms. In the following we will use the term ***graph*** to refer both to the actual structure with nodes and edges, to the set of atoms, and to the corresponding logic formula (the conjunctions of atoms). Accordingly, the same statements can be expressed in different ways. For example, for a graph F, the following are equivalent:

- e is an edge of F;
- $\mathsf{edge}(e, n_1, n_2) \in F$ for some n_1 and n_2; and
- $F \vdash \exists x, y : \mathsf{edge}(e, x, y)$.

For a graph F, an edge e and nodes n and m such that $\mathsf{edge}(e, n, m) \in F$, we will say that e is ***incident*** with both n and m, and e is an outgoing edge for n and an incoming edge for m. Moreover, we will call n and m ***neighbor*** nodes.

Arbitrary attributes can be assigned to edges and nodes. For example, for graph nodes that correspond to words of the text, we will indicate the part of speech tags of the words using attributes of the type $\mathsf{attr}(n, \mathsf{pos}, \mathsf{VBD})$. As another example, in the graphs representing various kinds of linguistic structures used later in the thesis, for each object (node or edge) we will use the attribute type to indicate the type of the object: word nodes, constituent nodes, dependency edges, child edges or antecedents in phrase trees, etc.

We will call a graph G a ***subgraph*** of a graph F iff $G \subset F$.

Now we introduce the main concepts needed to describe pattern matching on graphs. For the sake of space and readability we omit some details and formal definitions and refer to the in-depth description of Schürr (1997).

Patterns are simply arbitrary graphs, i.e., sets of atomic predicates. Consider, for instance, the following graph pattern:

$$G = \big\{ \mathsf{node}(a), \mathsf{node}(b), \mathsf{edge}(d, \mathsf{VP|VP}, b, a) \big\}.$$

This pattern contains two nodes, a and b, connected by an edge d with label VP|VP. We can use the mapping $u = \big\{a \mapsto n_1, b \mapsto n_3, d \mapsto e_2\big\}$ to embed G into the graph F in Figure 3.1. The mapping specifies an *occurrence* of the pattern G in the graph F. An *embedding* u associates objects of G with objects of F in such a way that nodes are mapped to nodes and edges to edges. In this case we will also say that u is an *occurrence* of a pattern G in F.

We will characterize u as an occurrence of G in the graph F by requiring that $u(G) \subset F$. For our example this is true, since

$$u(G) = \big\{ \mathsf{node}(n_1), \mathsf{node}(n_3), \mathsf{edge}(e_2, \mathsf{VP|VP}, n_3, n_1) \big\} \subset F.$$

Because we view graphs both as sets of atoms and as logic formulas, yet another way to check that u is an embedding of G into F is to *prove* $F \vdash u(G)$ using some complete first-order predicate calculus. This view opens an opportunity to generalize the notion of patterns and embeddings to arbitrary sets of formulas, not only sets of atoms (i.e., graphs). Allowing such more complex patterns, however, would require that we introduce explicit closed-world assumptions and additional constraints on graph representation, as discussed in (Schürr, 1997). Throughout the thesis we will only use simple patterns: sets of atoms.

We are ready now to define graph rewrite rules, which will be our main instrument for transforming graphs.

3.2 Graph rewrite rules

While patterns allow us to locate certain subgraphs by finding occurrences of patterns, graph rewrite rules will allow us not only to query graphs, but also to transform them. Basically, a rewrite rule defines a basic, atomic graph transformation: locating a certain subgraph and replacing it with another subgraph.

Formally, a *graph rewrite rule* is a triple $p = (L, C, R)$, where:

- L, the left-hand side of the rule (LHS for short) is a pattern;
- C, the constraint of the rule, is an arbitrary formula (or even an arbitrary boolean function) that only uses the objects of L; and
- R, the right-hand side (RHS for short) is a graph that might use some objects of L and some new objects.

For a rule p we will also use $LHS(p)$, $C(p)$ and $RHS(p)$ to denote its LHS, constraint and RHS, respectively.

When applying a rewrite rule $p = (L, C, R)$ to a graph F, we need to find all occurrences of the pattern L such that the constraint C is satisfied (i.e., the logic formula or the boolean function evaluates to *true*). For such an occurrence u, we remove from F all atoms (i.e., nodes, edges, attributes) that are in L but not in R and, moreover, add all atoms that are in R but not in L. In other words, we replace an occurrence of L with an occurrence of R, provided that a constraint C holds. As a result of such a replacement, some nodes and edges may be deleted or added, and those common to L and R remain.

More formally, a graph F' is *derivable* from graph F by applying rule p (we write $F \xrightarrow{p} F'$) iff:

(1) there is an embedding u of $L \cup C$ into F;

(2) there is an embedding w of R into F';

(3) u and w map all common objects of L and R into common objects; and

(4) F' is obtained from F by

 (a) removing all atoms that are in $u(L)$;
 (b) adding all atoms that are in $w(R)$; and
 (c) removing atoms that use objects occurring in $u(L)$ but not occurring in $w(R)$.

As an example, consider the following rewrite rule r_{intr}:

$$\begin{aligned}
LHS(r_{intr}) &= \big\{\, \mathsf{edge}(e, \mathsf{S|NP}, a, b) \big\} \\
C(r_{intr}) &= \big\{ \neg\exists x, y : \mathsf{edge}(x, \mathsf{VP|NP}, y, b) \big\} \\
RHS(r_{intr}) &= \big\{\, \mathsf{edge}(e, \mathsf{S|NP}, a, b), \mathsf{node}(c), \mathsf{attr}(c, \mathsf{label}, \mathsf{intr}), \\
&\qquad \mathsf{edge}(g, \mathsf{subj}, c, a), \mathsf{edge}(h, \mathsf{verb}, c, b) \big\}.
\end{aligned}$$

An occurrence u of the rule's LHS and constraint in a graph G locates a node $u(b)$ with incoming edge labeled S|NP and no incoming edges labeled VP|NP, i.e., a word having a subject dependent but no object dependents. For every such occurrence, the rule keeps the original edge and adds a new node $u(c)$ (labeled intr) with two outgoing edges: one pointing to $u(b)$ and labeled verb and another pointing to the subject dependent of b and labeled subj. In a way, the rule finds and marks intransitive uses of verbs (in practice, we would also add to $C(r_{intr})$ a requirement that b is indeed a verb, e.g., $\mathsf{attr}(b, \mathsf{pos}, \mathsf{VBD})$). An example of applying the rewrite rule r_{intr} to a graph is shown in Figure 3.2.

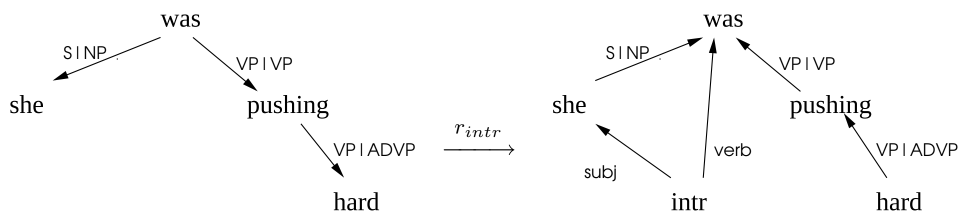


Figure 3.2: The result of applying the rewrite rule r_{intr}.

In the definition above we assumed that constraints of graph rewrite rules are logical formulas. However, in later chapters of the thesis we will often use rewrite rules with constraints defined as arbitrary boolean functions that map an occurrence of an LHS to either *true* or *false*, where these boolean functions are implemented using machine learning classifiers. The modification of the definition for this special case is straightforward.

3.3 Graphs with partial node ordering

Throughout the thesis, graphs representing linguistic information will be accompanied by some natural partial order on the set of nodes, e.g., when the nodes correspond to words or constituents of a text. Consider, for instance, the examples in Figure 2.2 on page 26, where:

- the set of all word nodes of the sentence is linearly ordered from left to right; and moreover,
- sets of children for every constituent are ordered in a natural left-to-right order, thus, defining a partial order on the set of all word and constituent nodes.

Let's recall that a partial order (we will denote it with "$\prec$") is an irreflexive, asymmetric, transitive relation (it is *partial* because there may exist a and b with neither $a \prec b$ nor $b \prec a$). For each partial order, there exists a (generally not unique) minimal relation $\prec_0$ such that $\prec$ is its transitive closure. For example, given a sentence $S = w_1 \dots w_k$, and a graph with nodes $n_1, \dots, n_k, n_{k+1}, \dots$ with node n_i corresponding to word w_i for $i = 1, \dots k$, the relations $\prec_0$ and $\prec$ can be defined as

$$n_i \prec n_j \iff i < j \leq k$$
$$n_i \prec_0 n_j \iff i + 1 = j \leq k.$$

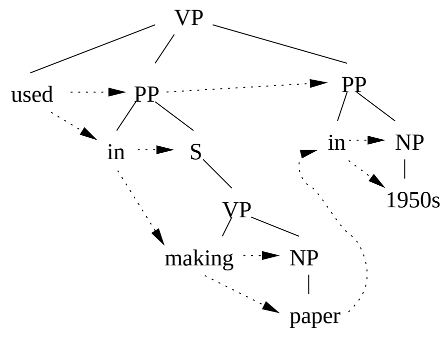
A *graph with a partial order* is a graph F containing edges that define a partial order $\prec$ and its corresponding Hasse diagram $\prec_0$ as follows:

- whenever $n_1 \prec_0 n_2$, graph F contains an edge from n_1 to n_2 with the label next and the type attribute equal to ord; and
- whenever $n_1 \prec n_2$, graph F contains an edge from n_1 to n_2 with label after and the type attribute equal to ord.

We will also use "$\prec$" and "$\prec_0$" as shorthands for the corresponding formulas:

$$n_1 \prec_0 n_2 \overset{\text{def}}{=} \mathsf{edge}(e', n_1, n_2) \ \& \ \mathsf{attr}(e', \mathsf{type}, \mathsf{order}) \ \& \ \mathsf{label}(e', \mathsf{next})$$
$$n_1 \prec n_2 \overset{\text{def}}{=} \mathsf{edge}(e'', n_1, n_2) \ \& \ \mathsf{attr}(e'', \mathsf{type}, \mathsf{order}) \ \& \ \mathsf{label}(e'', \mathsf{after}),$$

where e' and e'' are new object identifiers. Below is an example of a syntactic phrase tree represented as a graph with a partial order using the new types of edges:



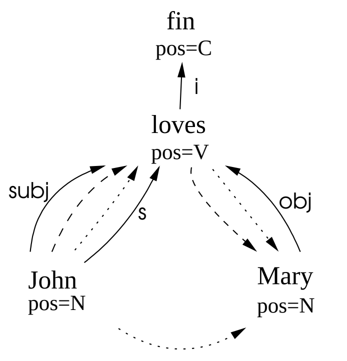
The partial order here is induced by the natural word order and the order of the constituents' children. We only show next edges as dotted arrows.

In later chapters we will mainly encounter graphs of two types, both with a partially ordered set of nodes: syntactic dependency graphs and syntactic phrase structure graphs. In the next section we formally describe these two types of graphs.

3.4 Dependency and phrase structure graphs

In Chapter 2 we have informally introduced the two types of syntactic structures that we will use for addressing several NLP tasks later in the thesis. Now, having formally defined directed labeled graphs with partial node ordering, we can formally introduce these two notions: dependency graphs and phrase structure graphs.

A *dependency graph* is a directed graph with nodes labeled with words they represent and their part of speech tags (e.g., $\mathsf{node}(n)$, $\mathsf{attr}(n, \mathsf{word}, \mathsf{was})$, $\mathsf{attr}(n, \mathsf{label}, \mathsf{VBD})$), edges going from dependents to heads and labeled with names of corresponding dependency relations. To distinguish these edges from those representing



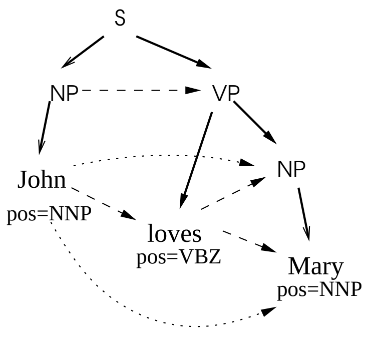
$$
\begin{aligned}
F = \{ & \mathsf{node}(n_1), \mathsf{attr}(n_1, \mathsf{word}, \mathsf{fin}), \mathsf{attr}(n_1, \mathsf{label}, \mathsf{C}), \\
& \mathsf{node}(n_2), \mathsf{attr}(n_2, \mathsf{word}, \mathsf{loves}), \mathsf{attr}(n_2, \mathsf{label}, \mathsf{V}), \\
& \mathsf{node}(n_3), \mathsf{attr}(n_3, \mathsf{word}, \mathsf{John}), \mathsf{attr}(n_3, \mathsf{label}, \mathsf{N}), \\
& \mathsf{node}(n_4), \mathsf{attr}(n_4, \mathsf{label}, \mathsf{Mary}), \mathsf{attr}(n_4, \mathsf{label}, \mathsf{N}), \\
& \mathsf{edge}(e_1, \mathsf{i}, n_2, n_1), \mathsf{attr}(e_1, \mathsf{type}, \mathsf{dep}), \\
& \mathsf{edge}(e_2, \mathsf{subj}, n_3, n_2), \mathsf{attr}(e_2, \mathsf{type}, \mathsf{dep}), \\
& \mathsf{edge}(e_3, \mathsf{s}, n_3, n_2), \mathsf{attr}(e_3, \mathsf{type}, \mathsf{dep}), \\
& \mathsf{edge}(e_4, \mathsf{obj}, n_4, n_2), \mathsf{attr}(e_4, \mathsf{type}, \mathsf{dep}), \\
& \mathsf{edge}(e_5, \mathsf{next}, n_3, n_2), \mathsf{attr}(e_5, \mathsf{type}, \mathsf{ord}), \\
& \mathsf{edge}(e_6, \mathsf{after}, n_3, n_2), \mathsf{attr}(e_6, \mathsf{type}, \mathsf{ord}), \\
& \mathsf{edge}(e_7, \mathsf{next}, n_2, n_4), \mathsf{attr}(e_7, \mathsf{type}, \mathsf{ord}), \\
& \mathsf{edge}(e_8, \mathsf{after}, n_2, n_4), \mathsf{attr}(e_8, \mathsf{type}, \mathsf{ord}), \\
& \mathsf{edge}(e_9, \mathsf{after}, n_3, n_4), \mathsf{attr}(e_9, \mathsf{type}, \mathsf{ord}) \}
\end{aligned}
$$

Figure 3.3: Dependency graph, the output of Minipar for the sentence "*John loves Mary*" and its representation as a first-order logic formula.

the partial order, we will add the attribute type with value dep and use the term *dependency edges*. For uniformity with phrase structure graphs (see below), we will also add type attributes to all nodes of dependency graphs (e.g., $\mathsf{attr}(n, \mathsf{type}, \mathsf{word})$).

Figure 3.3 gives an example of a dependency graph, the output of Minipar (Lin, 1994), for a simple sentence. Note that the parser produces an additional node fin, marking a finite clause, that is not included in the partial order. Minipar outputs two dependency relations between the words John and loves: subj for subject of a verb and s for surface subject (see Lin (1994) for details). In the figure, $\prec_0$-edges (i.e., those labeled next) are shown as dashed arrows, $\prec$-edges (labeled after) as dotted arrows, and some edge attributes are omitted.

A *phrase structure graph* is a node-ordered graph, with nodes representing either constituents (e.g., $\mathsf{node}(n)$, $\mathsf{attr}(n, \mathsf{label}, \mathsf{NP})$, $\mathsf{attr}(n, \mathsf{type}, \mathsf{phrase})$) or individual words (e.g., $\mathsf{node}(n)$, $\mathsf{attr}(n, \mathsf{word}, \mathsf{loves})$, $\mathsf{attr}(n, \mathsf{type}, \mathsf{word})$). Word nodes are also labeled with part of speech tags (e.g., $\mathsf{attr}(n, \mathsf{pos}, \mathsf{VBZ})$). Unlabeled edges go from constituent nodes to their children; we will use the term *constituent edges* and mark these edges using attribute type with value child. The partial order ($\prec$ and $\prec_0$) reflects the natural left-to-right ordering of words and ordering of children of every constituent.



$$
\begin{aligned}
F = \{\ & \mathsf{node}(n_1), \mathsf{attr}(n_1, \mathsf{label}, \mathsf{S}),\\
& \mathsf{node}(n_2), \mathsf{attr}(n_2, \mathsf{label}, \mathsf{NP}),\\
& \mathsf{node}(n_3), \mathsf{attr}(n_3, \mathsf{label}, \mathsf{VP}),\\
& \mathsf{node}(n_4), \mathsf{attr}(n_4, \mathsf{label}, \mathsf{NP}),\\
& \mathsf{attr}(n_1, \mathsf{type}, \mathsf{phrase}), \mathsf{attr}(n_2, \mathsf{type}, \mathsf{phrase}),\\
& \mathsf{attr}(n_3, \mathsf{type}, \mathsf{phrase}), \mathsf{attr}(n_4, \mathsf{type}, \mathsf{phrase}),\\
& \mathsf{node}(n_5), \mathsf{attr}(n_5, \mathsf{word}, \mathsf{John}),\\
& \mathsf{attr}(n_5, \mathsf{label}, \mathsf{NNP}), \mathsf{attr}(n_5, \mathsf{type}, \mathsf{word}),\\
& \mathsf{node}(n_6), \mathsf{attr}(n_6, \mathsf{word}, \mathsf{loves}),\\
& \mathsf{attr}(n_6, \mathsf{label}, \mathsf{VBZ}), \mathsf{attr}(n_6, \mathsf{type}, \mathsf{word}),\\
& \mathsf{node}(n_7), \mathsf{attr}(n_7, \mathsf{word}, \mathsf{Mary}),\\
& \mathsf{attr}(n_7, \mathsf{label}, \mathsf{NNP}), \mathsf{attr}(n_7, \mathsf{type}, \mathsf{word}),\\
& \mathsf{edge}(e_1, n_1, n_2), \mathsf{attr}(e_1, \mathsf{type}, \mathsf{child}),\\
& \mathsf{edge}(e_2, n_1, n_3), \mathsf{attr}(e_2, \mathsf{type}, \mathsf{child}),\\
& \mathsf{edge}(e_3, n_2, n_5), \mathsf{attr}(e_3, \mathsf{type}, \mathsf{child}),\\
& \ldots \}
\end{aligned}
$$

Figure 3.4: Phrase structure, the output of Charniak's parser for the sentence "*John loves Mary*" and its representation as a first-order logic formula.

Further, for phrase structure graphs we will sometimes also indicate head children by adding the attribute $\mathsf{attr}(e, \mathsf{head}, 1)$ to the child edge from a constituent to its head child. We will also mark lexical heads by adding an edge from a constituent node to the word node corresponding to the lexical head, labeling the edge with the attribute $\mathsf{attr}(e, \mathsf{type}, \mathsf{lexhead})$. Antecedent constituents of Penn Treebank empty nodes will be indicated using edges from the empty node to its antecedent with the attribute $\mathsf{attr}(e, \mathsf{type}, \mathsf{antecedent})$.

Figure 3.4 presents an example of a phrase structure produced by the parser of Charniak (2000), represented as a graph. Again, in the figure we omit some node attributes, and use dashed and dotted arrows for edges labeled next and after, respectively.

3.5 Comparing graphs

Throughout the thesis we will need to compare pairs of graphs, identifying what two given graphs have in common and where they differ, and sometimes characterize the differences numerically. One obvious example is comparing the output of a syntactic parser to syntactic structures annotated manually, e.g., as in Figure 2.4 on page 27. For this and other examples we may be interested both in a quantitative comparison of these structures (i.e., how many of their nodes, edges, labels are shared by both graphs) and a qualitative comparison (i.e., what these differences actually are). Throughout the thesis we will use quantitative graph comparisons for the evaluation of systems that operate on graphs, and qualitative comparisons for identifying frequent errors or mismatches in collections of graph pairs, as well as for error analysis. In this section we introduce the apparatus that will allow us to make such comparisons in our graph-based setting.

3.5.1 Graph alignment

An *alignment* of two disjoint graphs F and G is an injective partial mapping $u : \mathcal{O}_F \to \mathcal{O}_G$ that maps nodes to nodes and edges to edges. We will sometimes represent an alignment of two graphs as a set of atoms and write $u = \big\{ \mathsf{align}(x_1, y_1), \ldots, \mathsf{align}(x_n, y_n)\big\}$, where $u(x_i) = y_i$ and align is a new predicate symbol. In other words, an alignment is a one-to-one correspondence between some (not necessarily all) nodes and edges of two graphs.

For u, an alignment of F and G, we will use $u(F)$ to denote the graph obtained from F by changing x to $u(x)$ for all objects x such that $u(x)$ is defined.

In general, we will only be interested in alignments of pairs of graphs that preserve certain graph properties. An *alignment schema* is a constraint on the possible alignments of two graphs. Formally, an alignment schema is a closed first-order logic formula not containing object identifiers (i.e., containing only bound variables) that only uses predicates node, edge, attr and align. We will use schemata to assert certain well-formedness criteria. An alignment u is *consistent* with alignment schema S iff $F \cup G \cup u \vdash S$. We will use $S(F, G)$ to denote the set of all alignments of F and G that are consistent with the schema S.

Throughout the thesis we will only consider alignments of graphs such that:

- if two edges are aligned, their respective end-points are aligned as well;
- there are no conflicts in the ordering of aligned nodes in the two graphs;
- alignments preserve types of the aligned objects (i.e., values of type attributes;

- alignments preserve words (i.e., values of word attributes) for graph nodes that actually correspond to surface words on sentences (i.e., nodes with the attribute $\mathsf{type} = \mathsf{word}$.

More specifically, we will only consider alignments satisfying the alignments schema $S = S_0 \mathbin{\&} S_\prec \mathbin{\&} S_w$, with S_0, $S_\prec$ and S_w defined below.

Definition 3.1 (Basic alignment schema S_0)
The schema S_0 states that whenever two edges are aligned, their incident nodes are aligned as well:

$$S_0 \overset{\text{def}}{=} \forall x_1, x_2, s, y_1, y_2, t : \Big(\mathsf{edge}(s, x_1, x_2) \mathbin{\&} \mathsf{edge}(t, y_1, y_2) \mathbin{\&} \mathsf{align}(s, t) \\ \Rightarrow \mathsf{align}(x_1, y_1) \mathbin{\&} \mathsf{align}(x_2, y_2) \Big).$$

Definition 3.2 (Order-preserving alignment schema $S_\prec$)
The schema $S_\prec$ states that an alignment preserves a partial order on nodes (if defined):

$$S_\prec \overset{\text{def}}{=} \forall x_1, x_2, y_1, y_2 : \Big(\mathsf{align}(x_1, y_1) \mathbin{\&} \mathsf{align}(x_2, y_2) \\ \Rightarrow x_1 \prec x_2 \Leftrightarrow \neg(y_2 \prec y_1) \Big)$$

In other words, alignments consistent with $S_\prec$ should not introduce any conflicts between the partial orders of the two aligned graphs.

In the course of the thesis we will only be comparing pairs of graphs that refer to the same natural language sentence. Moreover, we will assume that tokenization (i.e., splitting sentences into words) is similar in the two analyses that the graphs represent. A natural additional constraint for the alignment of graphs annotating the same sequence of words is that their word nodes are aligned. We introduce this condition formally:

Definition 3.3 (Type and word-preserving alignment schema S_w)
The schema S_w states that the types of aligned objects are identical, as well as the words of aligned word nodes:

$$S_w \overset{\text{def}}{=} \forall x, y : \Big(\mathsf{align}(x, y) \Rightarrow \exists t : \mathsf{attr}(x, \mathsf{type}, t) \mathbin{\&} \mathsf{attr}(y, \mathsf{type}, t) \Big) \mathbin{\&} \\ \forall x, y : \Big(\big(\mathsf{align}(x, y) \vee \mathsf{align}(y, x) \big) \\ \Rightarrow \forall w : \big(\mathsf{attr}(x, \mathsf{word}, w) \Rightarrow \mathsf{attr}(y, \mathsf{word}, w) \big) \Big)$$

3.5.2 Graph merges

Graph alignments allow us to identify pairs of nodes or edges in two graphs, e.g., two possibly different linguistic analyses of the same sentence: the output of a parser and a treebank annotation, or even outputs of two different parsers. *Graph merges*, structures introduced in this section, will allow us to view a pair of graphs together with their alignment *as a single object*, in fact, as a single graph such that each of its elements (nodes, edges, attributes) "knows" whether it belongs to both aligned graphs or to only one of them. This view on graph pairs as graphs will enable us to reuse the whole apparatus of graph pattern matching and graph rewrite rules for aligned pairs of graphs.

Essentially, a graph merge of two aligned graphs F and G is a graph, with all nodes, edges and attributes corresponding to nodes, edges and attributes of F and G and split into three disjoint set: those pertaining to F only, to G only, and to both graphs. We now give a formal definition.

Let F and G be two graphs with disjoint sets of objects, i.e., $\mathcal{O}_F \cap \mathcal{O}_G = \emptyset$. Let u be an alignment of F and G. A *merge* of F and G with respect to u is defined as a disjoint union of three sets $M = M(F, G, u) = M_{\mathit{left}} \cup M_{\mathit{both}} \cup M_{\mathit{right}}$, such that $M = u(F) \cup G$ and for any atom $A \in M$:

1. $A \in M_{\mathit{left}}$ iff $A \in u(F)$ and $A \notin G$;
2. $A \in M_{\mathit{both}}$ iff $A \in u(F)$ and $A \in G$; and
3. $A \in M_{\mathit{right}}$ iff $A \notin u(F)$ and $A \in G$.

In words, $M(F, G, u)$ is obtained from G by adding to G all elements of F that were not mapped into G using u. Intuitively, we merge F and G, identifying aligned objects.

For a merge M, we will use the terms *left* (*common* and *right*) objects to refer to objects from M_{left}, M_{both} and M_{right}, respectively.

Note that for any graphs F and G, alignment u and merge $M = M(F, G, u)$, both $M_{\mathit{left}} \cup M_{\mathit{both}}$ and $M_{\mathit{both}} \cup M_{\mathit{right}}$ are always graphs, since by the definition of a merge:

$$\begin{aligned} M_{\mathit{left}} \cup M_{\mathit{both}} &= u(F), \text{and} \\ M_{\mathit{both}} \cup M_{\mathit{right}} &= G \end{aligned}$$

On the other hand, the full merge, M_{left}, M_{both} and M_{right} is not necessarily a graph, because F and G may have conflicting attributes for aligned objects.

Note that if the alignment u is empty (i.e., F and G are not aligned) the merge $M(F, G, u)$ is still defined and is simply the disjoint union of F and G, such that $M_{\mathit{left}} = F$, $M_{\mathit{right}} = G$ and $M_{\mathit{both}} = \emptyset$.

3.5.3 Finding graph alignments

As mentioned above, graph alignments and merges will be our main device for comparing graphs, both numerically and qualitatively: we will use them for evaluation purposes and for detecting and describing systematic differences between graphs. In this section we address an important practical question: how do we find an alignment for a pair of graphs? For a given pair of graphs F and G and a given alignment schema S, there may exist many possible alignments of F and G consistent with S. For example, for any graph pair and any alignment schema, an empty alignment (i.e., the one that does not align any nodes or edges) is always a valid alignment. Moreover, for any alignment, any subset (i.e., the result of excluding some edge pairs and/or some node pairs with incident edge pairs) is a valid alignment as well. On the other hand, some of the alignments for a given pair of graphs may be truly different, e.g., assigning some node of F to different nodes of G. Obviously, we are more interested in those alignments that make the two graphs as similar as they can be made, i.e., that identify as many nodes and edges as possible, without violating the validity criteria (the alignment schemata).

In principle, for two graphs F and G, the number of all possible alignments is finite though super-exponential in the size of the graphs. Indeed, for any subset of objects of F, $K \subset \mathcal{O}_F$, there are $\binom{|\mathcal{O}_G|}{|K|}$ many ways to align them to objects in G. Since the number of all possible $K \subset \mathcal{O}_F$ is exponential, we see that the number of all possible alignments is super-exponential (note, though, that many of these alignments will not satisfy the alignment schemata we defined above).

For the applications of graph methods to NLP problems, described in the thesis, the graphs we deal with are generated for English sentences, with nodes corresponding to words, phrases, semantic arguments etc., which implies that the graphs we are working with are not extremely large. For example, for our encoding of the syntactic phrase structures in the Penn Treebank, the graphs have 45 nodes and 113 edges on average. Nevertheless, these sizes prohibit graph alignment methods based on full enumeration of all possible alignments.

In this subsection we describe our heuristic method for aligning pairs of graphs, that will be used for all NLP problems described in later chapters. Our method uses the node ordering of our graphs and the properties of valid alignments (as defined by the alignment schemata, Section 3.5.1).

In order to find an alignment for a given pair of graphs F and G, we use a simple dynamic programming algorithm (Thomas H. Cormen and Stein, 2001) that tries to find the best possible alignment of the *nodes* of the two graphs, by maximizing a gain function:

$$gain_{align}(F, G) \overset{\text{def}}{=} \sum_{n,m:\text{aligned}} gain_{align}(n, m),$$

where the gain in aligning a pair of nodes, $gain_{align}(n, m)$, is defined as follows:

- $-\infty$ if n and m differ in the value of the type attribute. This allows us to guarantee that words will only be aligned to words, syntactic phrases to syntactic phrases, semantic elements to semantic elements, etc.

- if n and m are both words (i.e., the value of the type attribute is word), then the gain is defined as 1 if the corresponding words are identical (i.e., $\mathsf{attr}(n, \mathsf{word}, w) \in F$ and $\mathsf{attr}(m, \mathsf{word}, w) \in G$ for some string w), and $-\infty$ otherwise. Thus, we ensure that only word nodes with identical words are aligned.

- if n and m are both phrases (i.e., the value of the type attribute is phrase), the gain is defined as the degree of word overlap. More precisely, we compute yields of both phrases in both graphs (the values of the word attribute of all word-descendants when following child-edges) and compute the alignment gain as the ratio of the number of words in common and the total number of words in the two yields.

- for other nodes types, e.g., semantic elements, we define the alignment gain in the same was as for phrases (i.e., based on the word yield), as in the representations we consider in the thesis, semantic nodes always link to phrase or word nodes.

We use the ordering of nodes (word and phrases) to run a dynamic programming algorithm similar to the algorithm for the calculation of edit distance for strings (Levenshtein, 1966).

Now, with all technicalities in place, we will give a brief high-level description of our method to automatically learn a sequence of graph rewrite rules given a corpus of aligned graphs.

3.6 Learning graph transformations: an outline

For some NLP tasks formulated as graph transformation problems, rather than manually creating a list of transformation rules, we might prefer a system that can *learn* such rules automatically. As we will show later in the thesis, this is often possible if the system has access to a corpus annotated with the information relevant to the transformations. Here we outline a general method for automatically learning graph transformations. We postpone a fully detailed description of the method until Chapter 5.

In order to learn how to transform graphs, our method uses a corpus of aligned graph pairs $\{(F_i, G_i)\}$, where F_i exemplifies an input graph and G_i the ideal output of the graph transformation engine to be learned. By comparing graphs in the aligned pairs, our method identifies a sequence of useful graph rewrite rules that can then be applied to previously unseen graphs.

More specifically, our method learns a sequence of rewrite rules of the form (*LHS,C,RHS*): the left-hand side, the constraint and the right-hand side (see Section 3.2). The rule specifies that all occurrences of the *LHS* should be found, for which the constraint C is satisfied, and these occurrences should be replaced with the *RHS*.

Our method is similar to Transformation-Based Learning (TBL) (Brill, 1995) and is largely inspired by the TBL paradigm. In fact, the method can be considered an extension of TBL, as we will show below.

Both for our own method and for TBL, the crucial ingredients are:

- the types of basic rewrite transformations considered by the system, i.e., types of possible *LHS*'s considered by the system;
- the mechanism for selecting the most promising transformations of a chosen type;
- the realization of the constraint C. While in classic TBL all constraints are typically included in the *LHS* (and thus, TBL uses straightforward pattern matching and replacement: $LHS \to RHS$), we will often use a more flexible and more sophisticated constraint implementation, where constraints are realized by an off-the-shelf machine learning engine.

As in the classic TBL paradigm, the process of learning graph transformations is an iteration of learning steps. At each step, a corpus of graph pairs is used to identify possible rewrite rules of specific types and to select the most promising one(s) (i.e., those that reduce the total number of mismatches in the graphs most, according to some target metric). Then, the selected rules are applied to the corpus of graph pairs itself, thus making a corpus that will be used at the next iteration of the method to learn subsequent transformations. Thus, the structure of our transformation method follows the structure of the Transformation Based Learning paradigm (see Figure 3.5). As mentioned above, more details on the method and its comparison to the TBL paradigm are presented in Chapter 5.

3.7 Conclusions

In this chapter we made precise the notions of graph, graph transformation and graph comparison. In our formalization we followed the logic-based approach of

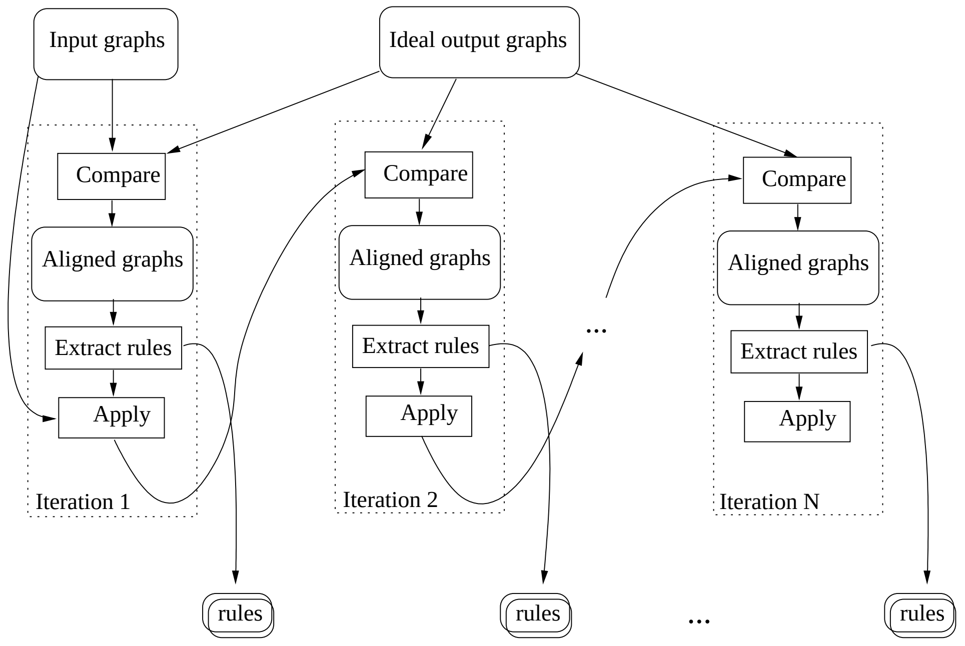


Figure 3.5: Outline of the graph transformation method. Arrows indicate data flow. The method takes two corpora (input and output graphs, top) and produces a list of rewrite rules (bottom).

Schürr (1997), where graphs are represented as first-order logic formulas, or, more precisely, as sets (conjunctions) of atomic formulas defining nodes, edges and their attributes, such as labels or part of speech tags. We also defined graph merges and graph rewrite rules, which will be the main devices in the automatic graph transformation method we have briefly outlined here and will describe in full in Chapter 5. But before we present this detailed description, it is time to put our formalization to work. In the next chapter we give an example of how the idea of graph transformations can be used for addressing a concrete NLP problem: recovering non-local dependencies and function tags in syntactic parses of sentences, or, putting it differently, identifying Penn Treebank-style predicate argument structure.

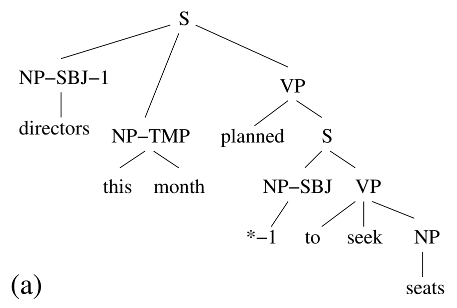
Chapter 4

Identifying Predicate Argument Structure

In this chapter we present the first example of a practical application of our graph-based approach to NLP problems: namely, to the problem of identifying the predicate argument structure of English sentences. There are different views on what constitutes the predicate argument structure (PAS) of a sentence. In this chapter we focus on one of them, taking the Penn Treebank II (Marcus *et al.*, 1994), the largest available syntactically annotated corpus of English, as our source of information about PAS.

In particular, we formulate the predicate argument structure identification problem as a problem of transforming the output of a syntactic parser. Such a formulation will allow us to embed the task into our general graph-based framework and to demonstrate the applicability of our graph-based view on the linguistic information. We present our solution for a specific representation of parsers' outputs (namely, as dependency graphs) and a specific choice of types of graph transformations and graph features used to implement the transformations. Later in the thesis, after presenting a general method for learning graph transformation in Chapter 5, we will come back to the task of PAS identification in Chapter 6 and describe a more systematic solution, applying the general method for this task with both dependency graphs and phrase structures, and comparing the results to the system of the present chapter.

The rest of the chapter is organized as follows. After providing some background and motivation for the PAS identification task in Section 4.1, we give a general overview of our graph-based method in Section 4.4. The method consists of three sequential steps, described in detail in Sections 4.5, 4.6 and 4.7. We discuss the results in Section 4.8.



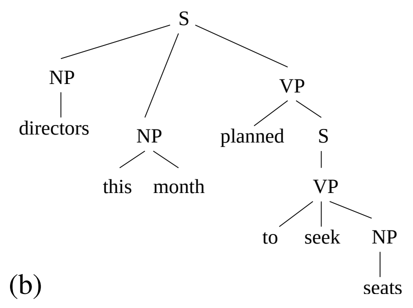


Figure 4.1: Example of (a) the Penn Treebank WSJ annotation and (b) the output of Charniak's parser for the sentence "*directors this month planned to seek more seats.*"

4.1 Motivation

State-of-the-art statistical phrase structure parsers, e.g., parsers trained on the Penn Treebank, produce syntactic parse trees with bare phrase labels, such as NP, PP, S (noun phrase, prepositional phrase, clause, resp., see Figure 4.1(b)), i.e., providing classical surface grammatical analysis of sentences, even though the training corpus is richer and contains additional grammatical and semantic information distinguishing various types of modifiers, complements, subjects, objects and specifying non-local dependencies, i.e., relations between phrases not adjacent in the parse tree (see Figure 4.1(a) which gives an example of the Penn Treebank annotation). This information constitutes an essential part of the predicate argument structure, which can be seen as a step up from syntactic constituent structure of text towards semantic representation. For example, by only looking at bare phrase labels of a parse tree, it is often not possible to distinguish complements and adjuncts as they might be realized as phrases of the same type. In the example in Figure 4.1(b), both noun phrases (NPs) "*directors*" and "*this month*" are in a similar syntactic position in the sentence: they are children of an S constituent, and are not distinguishable based only on the provided syntactic information (constituent labels). At the same time, these NPs play very different roles in the PAS of the sentence: a subject and a temporal adjunct.

As another example in the same sentence, consider the relation between the verb "*seek*" and the noun phrase "*directors*". Whereas in the Penn Treebank annotation (Figure 4.1(a)) the fact that the NP is the subject of the verb (this is an example of subject control) is explicitly indicated using an empty subject NP containing a trace co-indexed with "*directors*," this information is missing in the parser's output. Such *non-local* dependencies obviously constitute a part of the predicate argument structure.

Whereas the information about the entire predicate argument structure of the sentences is explicitly annotated in the Penn Treebank, it is ignored by many state-of-the-art parsers. One of the exceptions is the Combinatory Categorial Grammar parser of Hockenmaier (2003), which incorporates non-local dependencies into the parser's statistical model, and thus not only produces these dependencies in the output, but moreover, uses non-local relations during parsing. As another exception, the parser of Collins (1999) apart from bare constituency trees recovers WH-extraction traces and provides a simple complement/adjunct distinction (by adding the -A tag to labels of constituents that function as complements).

None of the known syntactic parsers, however, is capable of providing all information that is available in the Penn Treebank. The reason is that bringing in more information, though helpful in principle, in practice often makes the underlying statistical parsing models more complicated: more parameters need to be estimated and some independence assumptions (in particular, those used in probabilistic context-free grammars (PCFGs)) may no longer hold. Klein and Manning (2003), for example, report that using function tags of the Penn Treebank (temporal, location, subject, predicate, etc.) with a simple unlexicalized PCFG generally had a negative effect on the parser's performance. Similarly, Dienes (2004) report that a combination of trace tagger and a trace-aware parser that extends one of the models of Collins (1997), shows good results for the identification of non-local dependencies, but a drop of accuracy on the constituent-based PARSEVAL measure. Currently, there are no parsers trained on the Penn Treebank that use the structure of the treebank in full and that are capable of producing syntactic or, rather, predicate argument analysis that include all types of information information annotated in the corpus.

In this chapter, we address the problem of recovering *all* types of information that is annotated in the Penn Treebank, including semantic and grammatical tags, empty nodes and non-local dependencies. We will describe a method that views the output of a parser as a directed labeled graph and learns to add the missing information by comparing graphs derived from the parser and graphs derived from the Penn Treebank. In this graph-based setting, adding different types of information corresponds to different simple graph modifications: identifying function tags to edge labels correction, identifying empty nodes to adding nodes to graphs, identifying non-local dependencies to adding new graph edges.

We will focus on the version of PAS based on the dependency syntax derived from the Penn Treebank as described in Section 2.1.2. Later, in Chapter 6, we will present the method for identification of predicate argument structure based on phrase trees and on the original Penn Treebank.

4.2 Related work

In recent years there has been a substantial interest in getting more information from parsers than just bare syntactic phrase trees. The approaches to this problem can be divided into two groups: those aiming at identifying Penn Treebank function tags and those addressing the recovery of empty nodes and traces.

4.2.1 Recovery of function tags

Blaheta and Charniak (2000) presented the first method for assigning Penn Treebank function tags to constituents identified by a syntactic parser. The method, further studied in (Blaheta, 2004), consists in formulating the function tag identification task as a classification problem: given a constituent produced by a parser, assign a (possibly empty) function tag. Blaheta described experiments with three classifiers: Maximum Entropy, Decision Trees and Perceptrons. All three learners were provided with the set of features describing the constituent in question and its neighborhood in the parse tree:

- labels of the constituent itself, its parent, grandparent and left and right siblings in the parse tree;
- lexical heads of the constituent, its parent and its grandparent;
- part of speech tags of these heads;
- in case the constituent is a prepositional phrase (i.e., its label is PP), the head of its object and its part of speech;
- for each of the labels and part of speech tags, its cluster: one of the three manually created groups of labels;
- for each of the words, its cluster: word clusters were created automatically; the number of clusters was not reported.

Experiments with the three classifiers and different subsets of features demonstrated that the maximum entropy performed best for identification of the semantic Penn function tags (manner, temporal, locative, etc.), while perceptron was best for the identification of syntactic tags (subject, predicate, topicalization, etc.).

4.2.2 Recovery of non-local dependencies

Several methods have been applied to the task of identification of Penn Treebank empty nodes and non-local dependencies (traces).

The first approach to the task was presented by Johnson (2002), who used a simple pattern matching algorithm. From the training corpus, he extracted the smallest fragments of the phrase trees that contained traces, i.e., empty nodes together with their antecedents, and used these fragments as tree patterns: whenever such a pattern occurs in a phrase tree, an empty node is inserted in the tree and linked to the antecedent as defined by the pattern.

We used a similar pattern-matching approach in (Jijkoun, 2003), applying it to dependency rather than phrase trees, and extending the pattern-matcher with a trained classifier that predicted whether for a given match an empty node and a trace should be inserted. We used a Memory-Based Learner (Daelemans *et al.*, 2003) that had access to a number of features describing a match in a dependency graph: words and their part of speech tags, presence of subject and object dependents, finiteness of verb clusters. Machine learning was demonstrated to be useful for the task.

Experiments described in (Dienes and Dubey, 2003a,b) show that the task of recovering non-local dependencies can be successfully addressed using a combination of a preprocessing method and trace-aware syntactic parser. Their system predicts positions of empty elements in sentences, before parsing, by training a Maximum Entropy tagger with a set of features designed to capture aspects of the context important for the task: surrounding words and their part of speech tags, presence of passive, to-infinitive, complementizers, WH-pronouns, etc. Empty elements detected by the tagger are reported to a lexicalized parser, that uses a gap-threading mechanism similar to Model 3 of Collins (1997), to incorporate this information into the parsing model. Furthermore, combining a preprocessor and a parser in a single probabilistic architecture, Dienes (2004) shows an improvement in the recovery of non-local dependencies. The experiments of Dienes show that including non-local information in a PCFG parsing model allows one to achieve high accuracy on the task of recovering non-local dependencies. However, this seems to come at a price of a drop in the parsing accuracy: the PARSEVAL score of the parser decreases from 88.0 to 87.1 when using non-locals in the parsing model.

Levy and Manning (2004) describe another post-processing approach to the task. They describe a linguistically-motivated algorithm that takes into account specifics of the Penn Treebank annotation of non-local dependencies. Two classifiers based on log-linear models are used to identify extracted constituents, i.e., antecedents of empty nodes. This is different from the system of Dienes and Dubey (2003a) where a classifier was used to detect empty elements themselves in sentences before parsing. The results of the comparison of the two systems are interesting: while Dienes and Dubey (2003a) seem to perform better specifically for traces, Levy and Manning (2004) report better results for the dependency-based

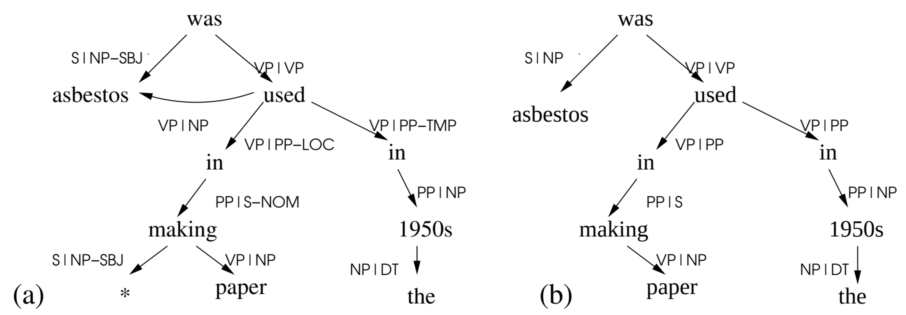


Figure 4.2: Dependency graphs derived from (a) the Penn Treebank annotation and (b) the output of Charniak's parser for the sentence "*asbestos was used in making paper in the 1950s.*"

evaluation that considers *both* local and non-local dependencies.

4.3 PAS identification and graph transformations

As discussed above, in the present chapter we address the task of identifying predicate argument structure in dependency graphs. More precisely, the task is to transform dependency graphs derived from a parser in such a way that they contain all information annotated in the Penn Treebank trees. This corresponds to adding Penn function tags (e.g., -SBJ, -TMP, -LOC), empty nodes (e.g., empty complementizers) and non-local dependencies (control traces, WH-extraction, etc.). In terms of dependency graphs these transformations can be seen as local modifications of the graph structure: changing edge labels, adding new nodes and new edges.

For example, consider two dependency graphs in Figure 4.2, the result of the conversion of phrase structures from (a) the Penn Treebank and (b) Charniak's parser (repeated here from Figure 2.5). We can identify the following mismatches between the two graphs:

- different dependency labels: S|NP vs. S|NP-SBJ, VP|PP vs. VP|PP-TMP, VP|PP vs. VP|PP-LOC and PP|S vs. PP|S-NOM;
- an empty node "*", a dependent of *making* in the Penn Treebank graph; and
- an edge between *asbestos* and *used*, an object trace in the passive.

Our method for adding predicate argument information to the output of a parser, described in the next section, is based on identifying such small local transformations.

4.4 Transforming graphs

In this section we give a high-level overview of our method for transforming the output of a parser. The steps of the method will be described in detail in the following sections.

We experimented with dependency structures automatically derived from the output of two phrase structure parsers of Charniak (2000) and Collins (1999). For Collins' parser the text was tagged for part of speech before the parsing, using the maximum entropy tagger of Ratnaparkhi (1996).

Our graph transformation method consists of two phases:

- training phase, which induces the necessary knowledge from existing data sources; and
- application phase, which uses the induced knowledge to process previously unseen data.

The training phase of the method consists in learning which transformations need to be applied to the output of a parser to make it as similar to the data from the gold standard as possible.

As a preliminary step, as mentioned above, we convert the constituency-based Penn Treebank annotation of PAS to a dependency corpus without losing the annotated information (constituent labels, function tags, empty nodes, non-local dependencies). The same conversion is applied to the output of the parsers we consider (we refer to Section 2.1.2 on page 21 for details).

Training proceeds by comparing graphs derived from a parser's output with the graphs from the dependency corpus (the corpus of predicate-argument structures), detecting local graph mismatches, such as differing edge labels and missing or incorrect nodes or edges.

We considered the following types of local transformations (rewrite rules), defining the three steps of method:

1. changing edge labels, such as S|NP to S|NP-SBJ;
2. adding new nodes (e.g., empty nodes of the treebank annotation), together with dependency edges that connect these nodes to a dependency graph; and
3. adding new edges, e.g., dependency edges corresponding to the Penn Treebank annotation of non-local dependencies.

At the application phrase, our method transforms input dependency graphs, sequentially applying graph rewrite rules that implement transformations of these three types. Obviously, other types of rewrite rules are possible and make sense for our task, such as deleting edges or performing rewrites involving more that one node. In Chapter 6, after the introduction of a general transformation learning method, we will generalize the method by allowing arbitrary rewrites in arbitrary sequence.

For the experiments, the corpus of gold standard dependency graphs (Penn Treebank automatically converted to dependencies) was split into training (sections 02–21), development (sections 00–01) and test (section 23) corpora. During the training phase of the method, for each of the steps 1, 2 and 3 we proceed as follows:

1. compare the training corpus to the output of the parser on the strings of the corpus, after applying the transformations of the previous steps;

2. identify left-hand sides of possible beneficial transformations (which edge labels need to be changed or at which positions in the graphs new nodes or edges need to be added);

3. train a classifier to realize constraints and predict right-hand sides of transformation rules, given an occurrence of a left-hand side and its context (i.e., information about the local structure of the dependency graph in the neighborhood of the occurrence).

While the forms of left-hand sides, contexts and right-hand sides of the rules are different for the three steps, the general structure of the method remains the same at each stage. Sections 4.5, 4.6 and 4.7 describe the steps in detail.

For the application of the method to new, unseen data, we proceed similarly. First, the output of the parser is converted to dependency graphs (Section 2.1.2), then the learners trained during steps 1, 2 and 3 are applied consecutively to detect graph transformations and the detected graph modifications are performed.

Apart from the conversion from phrase structures to dependency graphs and the extraction of some linguistic features for the learning described in the following sections, our method does not use any information about the details of the corpus annotation or the parser's output: it simply works with labeled directed graphs. Thus, a similar method can potentially be applied to any graph transformation problem as long as the output of the parser and the training corpus are represented as edge- and node-labeled graphs. We will support this claim after describing such a more general method in Chapter 5.

In Table 4.1 the row labeled "*initially*" gives the evaluation results for the two parsers we considered immediately after the conversion to dependency graphs, repeated here from Table 2.1 (see the discussion of these results in Section 2.1.1 on page 21).

Evaluation	Parser	unlabeled P	unlabeled R	unlabeled f	base labels P	base labels R	base labels f	with func. tags P	with func. tags R	with func. tags f
initially	Charniak	89.9	83.9	86.8	85.9	80.1	82.9	68.0	63.5	65.7
	Collins	90.4	83.7	87.0	86.7	80.3	83.4	68.4	63.4	65.8
relabeling	Charniak	89.9	83.9	86.8	86.3	80.5	83.3	83.8	78.2	80.9
(Section 4.5)	Collins	90.4	83.7	87.0	87.0	80.6	83.7	84.6	78.4	81.4
adding nodes	Charniak	90.1	85.4	87.7	86.5	82.0	84.2	84.1	79.8	81.9
(Section 4.6)	Collins	90.6	85.3	87.9	87.2	82.1	84.6	84.9	79.9	82.3
adding edges	Charniak	90.0	89.7	89.8	86.5	86.2	86.4	84.2	83.9	84.0
(Section 4.7)	Collins	90.4	89.4	89.9	87.1	86.2	86.6	84.9	83.9	84.4

Table 4.1: Dependency-based evaluation of the transformation method for the output of the parsers of Charniak (2000) and Collins (1999).

In the following sections we describe the three transformation steps, discuss evaluation results and where appropriate compare our method to previously published approaches.

4.5 Step 1: Changing dependency labels

Comparing dependencies produced by a parser to dependencies derived from the Penn Treebank, we automatically identify possible relabelings. For Collins' parser, the following were found to be the most frequent relabelings in the training data:

- S|NP→S|NP-SBJ,
- PP|NP-A→PP|NP,
- VP|NP-A→VP|NP,
- S|NP-A→S|NP-SBJ, and
- VP|PP→VP|PP-CLR.

In total, around 30% of all the parser's dependencies had different labels in the Penn Treebank.

We learned simple rewrite rules that modify edge labels of dependency graphs, implementing the constraints of the rules using TiMBL, a memory-based multi-class classifier (Daelemans *et al.*, 2003). The task of the learner was, given a dependency edge in a graph, predict how its label should be changed. We provided the learner with the following symbolic features for each dependency edge in a dependency graph:

- the head (h), dependent (d) and their part of speech tags;
- the leftmost dependent of d, with respect to the surface word order, and its part of speech tag;
- the head of h (h'), its part of speech and the label of the dependency $h' \to h$;
- the closest left and right siblings of d (dependents of h) and their part of speech tags;
- the label of the dependency ($h \to d$) as derived from the parser's output.

When included in feature vectors, all dependency labels were split at '|', e.g., the label S|NP-A resulted in two symbolic features: S and NP-A.

Testing was done as follows. The test corpus (section 23) was also parsed, and for each dependency, the features were extracted and the trained classifier used to assign the corrected dependency label. After this transformation the outputs of the parsers were evaluated, as before, on dependencies in the three settings.

The results of the evaluation at this step are shown in Table 4.1 (the row marked "relabeling"). Obviously, relabeling does not change the unlabeled scores. The 1% improvement for evaluation on bare labels suggests that our approach is capable not only of adding function tags, but can also correct the parser's phrase labels and part of speech tags: for Collins' parser the most frequent correct changes not involving function labels were NP|NN→NP|JJ and NP|JJ→NP|VBN, fixing part of speech tagging errors: NN (common singular noun) in place of JJ (adjective) and JJ in place of VBN (past participle of a verb). A substantial increase of the labeled score (from 66% to 81%), which is only 6% lower than unlabeled score, clearly indicates that, although the parsers do not produce function labels, this information is to a large extent implicitly present in trees and can be recovered.

4.5.1 Comparison to earlier work

The main effect of the relabeling procedure described above is the recovery of Penn function tags. Thus, it is informative to compare our results with those reported in (Blaheta and Charniak, 2000) and (Blaheta, 2004) for this task. Blaheta measured the tagging accuracy and the precision/recall for function tag identification only for constituents *correctly identified* by the parser (i.e., having the correct span and constituent label). Since our method uses a dependency formalism, to make a meaningful comparison we need to model the notion of a constituent being correctly found by a parser. For a word w we say that the constituent corresponding to its maximal projection is *correctly identified* if there exists h, the head of w, and for the dependency $w \to h$ the right part of its label (e.g., NP-SBJ for S|NP-SBJ) is a nonterminal (i.e., not a part of speech tag) and matches the right part of the label in the gold standard dependency structure, after stripping function tags. In other words, we require that the constituent's label and headword are correct, but not necessarily the span. Moreover, 2.5% of all constituents with function labels (246 out of 9928 in section 23) are not maximal projections. Since our method ignores function tags of such constituents (these tags disappear after the conversion of phrase structures to dependency graphs), we consider them as errors, i.e., reducing our recall value.

Below, the tagging accuracy, precision and recall are evaluated on constituents correctly identified by Charniak's parser for section 23. The results of Blaheta (2004) are shown only for one of their models that uses a Maximum Entropy classifier with a manually constructed feature tree. Their other MaxEnt models and perceptrons with automatically selected feature sets show better results, but only the scores for a subset of function tags are reported (namely, syntactic and semantic tags). We will present a more detailed and straightforward comparison to the best model of Blaheta (2004) in Chapter 6.

Method	Accuracy	P	R	F_1
Blaheta and Charniak (2000)	98.6	87.2	87.4	87.3
Blaheta (2004)	**98.9**	87.9	**88.6**	**88.6**
Here	94.7	**90.2**	86.9	88.5

Our system shows better precision for the identification of Penn Treebank tags and a comparable F_1 score. The large difference in accuracy scores is due to two reasons. First, because of the different definition of a *correctly identified constituent* in the parser's output, we apply our method to a larger portion of all labels produced by the parser (95% vs. 89% reported in (Blaheta and Charniak, 2000)). This might make the task for our system more difficult. And second, whereas 22% of *all* constituents in section 23 have a function tag, 36% of the *maximal projections* have one. Since we apply our method only to labels of maximal projections, this means that our accuracy baseline (i.e., never assign any tag) is lower.

In Chapter 6 we will present a system that is capable (among other) of recovering function tags in phrase trees, which will allow us to make a more direct comparison with the results of Blaheta (2004).

4.6 Step 2: Adding missing nodes

As the "relabeling" row in Table 4.1 indicates, the recall is relatively low for both parsers (6% lower than precision): while the Penn Treebank trees, and hence the derived dependency structures, contain non-local dependencies and empty nodes, the parsers simply do not provide this information. To make up for this, we considered two further transformations of the output of the parsers: adding new nodes (corresponding to empty nodes in the Penn Treebank), and adding new labeled edges. This section describes the first of these transformations.

As described in Section 2.1.2 on page 21, when converting Penn Treebank trees to dependency structures, traces are resolved, their empty nodes removed and new dependencies introduced. Of the remaining empty nodes (i.e., non-traces), the most frequent in the Penn Treebank are: NP PRO, empty units, empty complementizers, empty relative pronouns (see Appendix A for details). In order to add missing empty nodes to dependency graphs, we compare the output of the parsers on the strings of the training corpus after converting it to dependency structures and automatic relabeling of the dependency labels (Section 4.5). We trained a classifier which, for every word in the parser's output, had to decide whether an empty node should be added as a new dependent of the word, and what its symbol ('*', '*T*', '*U*' or '0' in the Penn Treebank), part of speech tag (always -NONE- in the Penn Treebank) and the label of the new dependency connecting the new node to the word (e.g., 'S|NP-SBJ' for NP PRO and 'VP|SBAR' for empty complementizers) should be. This decision is conditioned on the word itself and its context. We used the following features to describe the context of a word:

- the word and its part of speech tag,
- binary features indicating whether the word has any subject and object dependents, and whether it is the head of a finite verb group;
- the same information for the word's head (if any) and also the label of the corresponding dependency;
- the same information for the rightmost and leftmost dependents of the word (if exist) along with the corresponding dependency labels.

In total, we extracted 23 symbolic features for every word in the corpus. TiMBL was trained on sections 02–21 and applied to the output of the parsers (after steps 0 and 1) on the test corpus (section 23), producing a list of empty nodes to be inserted in the dependency graphs. After insertion of the empty nodes, the resulting graphs were evaluated against section 23 of the gold dependency treebank. The results are shown in Table 4.1 (the row "adding nodes"). For both parsers the insertion of empty nodes improves the recall by 1.5%, resulting in a 1% increase of the F_1-score.

4.6.1 Comparison to earlier work

A procedure for empty node recovery was first described in (Johnson, 2002), along with an evaluation criterion: an empty node is correct if its category and position in the sentence are correct. Since our method works with dependency structures, not phrase trees, we adopt a different but comparable criterion: an empty node should be attached as a dependent to the correct word, and with the correct dependency label. Unlike the first metric, our correctness criterion also requires that possible attachment ambiguities are resolved correctly (e.g., as in *the number of reports 0 they sent*, where the empty relative pronoun may be attached either to *number* or to *reports*).

Below we compare our results for the recovery of empty elements with the system of Dienes and Dubey (2003a) and the best model of Dienes (2004). Note that this evaluation does not include traces (i.e., empty elements with antecedents): recovery of traces is described in Section 4.7.

Type	Dienes and Dubey (2003a)			Dienes (2004)			Here		
	P	R	F_1	P	R	F_1	P	R	F_1
PRO-NP	68.7	**70.4**	69.5	**74.7**	68.1	**71.3**	73.1	63.9	68.1
COMP-SBAR	**93.8**	78.6	**85.5**	78.6	**83.7**	81.7	82.6	83.1	82.8
COMP-WHNP	67.2	38.3	48.8	**70.4**	35.5	47.2	65.3	**40.0**	**49.6**
UNIT	99.1	92.5	**95.7**	94.4	91.2	92.8	95.4	91.8	93.6

For comparison we use the notation of Dienes and Dubey: PRO-NP for uncontrolled PROs (nodes '*' in the Penn Treebank), COMP-SBAR for empty complementizers (nodes '0' with dependency label VP|SBAR), COMP-WHNP for empty relative pronouns (nodes '0' with dependency label X|SBAR, where X≠VP) and UNIT for empty units (nodes '*U*').

Surprisingly, none of the methods outperform others on all empty node categories and the differences between the three are not large. Below, in Section 4.7 and in Chapter 6 we will present an overall evaluation of the methods, on all empty elements, including traces.

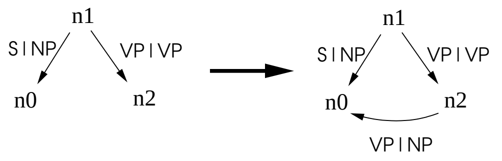
4.7 Step 3: Adding missing dependencies

We now get to the third and final step of our transformation method: adding missing edges to dependency graphs. The parsers we considered do not explicitly provide information about non-local dependencies (control, WH-extraction) present in the treebank. In this section we describe the insertion of missing dependencies.

Similar to the previous stages of our transformation method, we compare the output of the parsers on the strings of the training corpus, converted to dependencies and transformed using systems of steps 1 and 2 (edge label relabeling of Section 4.5 and empty nodes insertion of Section 4.6), to the dependency structures in the training corpus. For every Penn Treebank dependency that is missing in the parser's output, we find the shortest undirected path in the dependency graph connecting its head and dependent. These paths, connected sequences of labeled dependencies, define the set of possible left-hand sides of rewrite rules. For our experiments we only considered patterns occurring more than 100 times in the training corpus. E.g., for Collins' parser, 67 different patterns were found.

Consider the example in Figure 4.2, page 52. With the dependency VP|NP between the words *used* and *asbestos* missing in the parser's output, the shortest path between *used* and *asbestos* consists of two edges: from *was* to *used* and from *was* to *asbestos*. The following rewrite rule will therefore be extracted:

$$\begin{aligned}
LHS = \{\; & \mathsf{node}(n_0), \mathsf{node}(n_1), \mathsf{node}(n_2),\\
& \mathsf{edge}(e_0, n_1, n_0), \mathsf{edge}(e_1, n_1, n_2),\\
& \mathsf{attr}(e_0, \mathsf{label}, \mathsf{S|NP}), \mathsf{attr}(e_1, \mathsf{label}, \mathsf{VP|VP})\}\\
RHS = \{\; & \mathsf{node}(n_0), \mathsf{node}(n_1), \mathsf{node}(n_2),\\
& \mathsf{edge}(e_0, n_1, n_0), \mathsf{edge}(e_1, n_1, n_2),\\
& \mathsf{attr}(e_0, \mathsf{label}, \mathsf{S|NP}), \mathsf{attr}(e_1, \mathsf{label}, \mathsf{VP|VP})\\
& \mathsf{edge}(e_2, n_2, n_0), \mathsf{attr}(e_2, \mathsf{label}, \mathsf{VP|NP})\}
\end{aligned}$$



Next, from the parsers' output on the strings of the training corpus, we extracted all occurrences of the left-hand sides of the identified rewrite rules, along with information about the nodes involved. For every node in an occurrence of the left-hand side of a rule we extracted symbolic features describing:

- the word and its part of speech tag;
- whether the word has subject and object dependents (i.e., outgoing edges with labels S|NP-SBJ and VP|NP, respectively);
- whether the word is the head of a finite verb cluster.

We then trained TiMBL to predict the label of the missing dependency (or 'none'), given an occurrence of a pattern and the features of all the nodes of the occurrence. We trained a separate classifier for each of the 67 identified pattern.

For application of the learned transformations to new dependency graphs, we extract all occurrences of the left-hand sides of all patterns identified at the training stage, and use TiMBL to predict whether for a given occurrence a new edge should be added and if so, what its dependency label is.

We applied the learned transformations to the output of two parsers on the strings of the test corpus, after modifying the dependency graphs with the rewrite rules learned at steps 1 and 2 , i.e., after correcting dependency labels and adding empty nodes (Sections 4.5 and 4.6). Then we compared the resulting dependency graphs to the graphs of the test corpus. The results are shown in Table 4.1 (the row "step 3"). As expected, adding missing dependencies substantially improves the recall (by 4% for both parsers) and allows both parsers to achieve an 84% F_1-score on dependencies with function tags (90% on unlabeled dependencies). The unlabeled F_1-score 89.9% for Collins' parser is close to the 90.9% reported in (Collins, 1999) for the evaluation on unlabeled *local* dependencies only (without empty nodes and traces). Since as many as 5% of all dependencies in the Penn Treebank involve traces or empty nodes, the results in Table 4.1 are encouraging.

4.7.1 Comparison to earlier work

We evaluated the performance of the entire dependency graph transformation system (i.e., executing steps 1, 2 and 3) on empty nodes and non-local dependencies. As usual, a dependency is correctly found if its head, dependent, and label are correct. The evaluation based on this criterion directly corresponds to the evaluation using the head-based antecedent representation described in (Johnson, 2002); for empty nodes without antecedents this is exactly the measure we used in Section 4.6.1. To make our results comparable to other methods, we strip function tags from the dependency labels before label comparison. Below are the overall precision, recall, and F_1-score for our method and the scores of the systems of Dienes and Dubey (2003a) and Dienes (2004) for the antecedent recovery with an lexicalized parser.

Method	P	R	F_1
Dienes and Dubey	81.5	68.7	74.6
Dienes (2004)	82.5	**70.1**	**75.8**
Here	**82.8**	67.8	74.6

The performance of the three methods is comparable, and the system of Dienes (2004) based on a probabilistic combination of an empty element tagger and a lexicalized parser outperforms both other systems by over 1%. In Chapter 6 we will present another system for transforming a parser's output, and come back to this evaluation.

4.8 Analysis and conclusions

In this chapter we described an application of the graph-based approach to an NLP task. In particular, we presented a method that allows us to automatically enrich the output of a parser with information that is not provided by the parser itself, but is available in a treebank. Using the method with two state-of-the-art statistical parsers and the Penn Treebank allowed us to recover function tags, empty nodes and non-local dependencies—essential aspects of the predicate argument structure of sentences. The method is capable of providing all types of information available in the corpus, without modifying the parser, viewing it as a black box. The evaluation against a dependency corpus derived from the Penn Treebank showed that, after our post-processing, two state-of-the-art statistical parsers achieve 84% accuracy on a fine-grained set of dependency labels.

We viewed the task of the PAS identification as a graph transformation problem: given a dependency graph derived from the output of a syntactic parser, transform the graph to include information that defines the predicate argument structure. Analyzing the resulting graph transformation problem, we found that the task can be split into three simpler subtasks, each involving graph rewrite rules of only one of the three type: changing edge labels, adding new nodes and adding new edges between existing nodes. For each of the three subtasks, we trained memory-based classifiers to predict whether a specific transformation should be applied at a specific position in an input dependency graph.

While the method of this chapter demonstrates that a particular NLP problem, the identification of predicate argument structure, can be formulated as a problem of transforming graphs, the method has a number of shortcomings. First, the types of graph transformation we defined are determined based on our intuitions about the task, and other transformation types could also be considered, such as removing nodes or edges, or manipulating larger and more complex subgraphs. Second, the order of the three select transformation types is fixed: we first correct edge labels, then add new nodes with connecting edges, and finally, add new edges between existing nodes. This order was also defined in an ad-hoc way and is not necessarily optimal. Finally, the features we extracted from dependency graphs to provide classifiers with information about contexts of occurrences of rules' left-hand sides, were defined based on our knowledge of the specifics of both input and output graph corpora (the structures produced by a parser and the annotations in the

Penn Treebank) and the task. Applying a similar method to graphs of a different type (e.g., phrase trees, rather than dependency structures) or to different graph transformation problems is, therefore, likely to require defining new transformation types, choosing the order of the transformations and defining new features.

In order to address these shortcomings, we will propose a general method that will allow us to automatically learn sequences of graph rewrite rules for a given graph transformation task. We present the method in the next chapter, and later, in Chapter 6 apply it to the same problem as in the present chapter, predicate argument structure identification, and compare the resulting graph transformation system to the system described above.

Chapter 5

Learning Graph Transformations

In the previous chapter we described a first application of our graph-based approach to the task of identifying predicate argument structure in syntactic dependency structures. We viewed dependency structures as directed labeled graphs and re-formulated the task as a graph transformation problem: given the output of a parser, transform the graphs to include missing information about the predicate argument structure. Moreover, we described a method that allowed us to automatically learn this transformation. This method was, in fact, a task-specific instantiation of the general approach to transforming graphs, outlined at the end of Chapter 3 (in Section 5.2).

The purpose of the present chapter is to give a detailed description of a general graph transformation method, that generalizes the approach that we have already seen at work in Chapter 4. The method will be more general in that it allows more complicated graph rewrite rules and does not require defining a task-specific sequence of such rules, as well as defining a task-specific set of descriptive features used by the method internally. Instead, our general method will allow us to learn the sequence of rules that suits the task at hand, automatically.

In later chapters we will show how this general method works for specific NLP tasks. In particular, in Chapter 6 we will apply the method introduced in this chapter to the same problem of PAS identification, showing that the generality and flexibility of the method do result in an improved performance for the task.

The chapter is organized as follows. We describe our motivation in detail in Section 5.1 and give a high-level overview of the method in Section 5.2. We go through the steps of the method in Sections 5.3 to 5.7, consider technical issues in Section 5.8, discuss related work in Section 5.9 and conclude in 5.10.

5.1 Motivation

In Chapter 4 we already re-cast one NLP problem, the identification of predicate-argument structure, as a graph transformation task: automatically transforming input graphs (the results of the syntactic analysis) into output graphs (syntactic graphs with PAS information added). Our solution involved representing all structures as directed labeled graphs and splitting the full transformation problem into three basic transformation steps, namely:

- changing labels of graph edges, which corresponds to adding function and semantic tags to parser's labels;
- adding new graph nodes, which corresponds to adding the Penn Treebank empty nodes; and
- adding new graph edges, which corresponds to adding the Penn Treebank non-local dependencies.

Each step involved the extraction of possible application points of the rule, classifying such potential application points (to determine whether a transformation should be applied or not) using an off-the-shelf machine learning engine, and inserting the decisions of the learner back into the graphs.

Whereas this approach falls nicely into our general graph transformation framework (i.e., using an existing corpus of input/output graph pairs to learn which transformation should be applied), the realization proposed in the previous chapter is rather task specific: we defined a sequence of three particular transformation types and the set of features for each machine learning subtask, based only on our intuition and understanding of the NLP problem under consideration. Obviously, there is no guarantee that the selected sequence and the selected features are optimal or even well-suited for the task.

Let us also try to abstract away from the particular task of PAS identification. In order to apply our method to other tasks (e.g., identification of semantic roles or automatic conversion between syntactic formalisms), we would need substantial investments of time and resources to come up with appropriate definitions of transformations and features for each of the new tasks. It is likely that the method described in Chapter 4 does not work out of the box and requires a substantial amount of fine-tuning to be applicable for other NLP problems.

The main purpose of this chapter is to present a more general method of learning to transform graphs. The method we propose is not specific to any particular task and uses very little knowledge indeed about the linguistic nature of NLP problems, apart from what has been *explicitly* annotated using the structure of the graph (nodes, edges and attributes). To emphasize this, we will use example graphs with abstract labels throughout this chapter.

5.2 Outline of the graph transformation method

Before going into the details of our method for learning graph transformations, let us once again give a high-level overview.

The method starts with a corpus of input graphs and a corpus of corresponding ideal output graphs. Our task is to create a system that is capable of converting input to output graphs. Our method learns a sequence of graph rewrite rules, by iterating the following steps:

1. compare pairs of graphs in the input and output corpora (Section 5.3),
2. identify possible graph rewrite rules (Section 5.4),
3. select best rewrite rules (Section 5.5),
4. train machine learners that implement constraints of the rules (Section 5.6),
5. apply the learned rules to the corpus and start again (Section 5.7).

Figure 5.1 shows the overall structure of the graph transformation method (it is repeated here, for convenience, from Figure 3.5 in Chapter 3).

In the following sections we give a detailed description of the five steps that, together, constitute each iteration of the method.

5.3 Comparing pairs of graphs in the corpora

To compare a pair of graphs from the input and output corpora, we build a merge of the pair as described in Section 3.5.2 on page 42. Recall that a merge of two graphs is a union of the graphs that tries to unify as many of their nodes and edges as possible. The exact method for aligning a pair of graphs and constructing a merge is heuristic and puts validity requirements on graph alignments that depend on the linguistic nature of the graphs. Specifically,

- for dependency graphs, nodes are only aligned if they refer to the same word of the sentence;
- for phrase structure graphs, aligned word nodes are required to refer to the same word, and phrase nodes are aligned in such a way as to maximize the overlap of their respective yields.

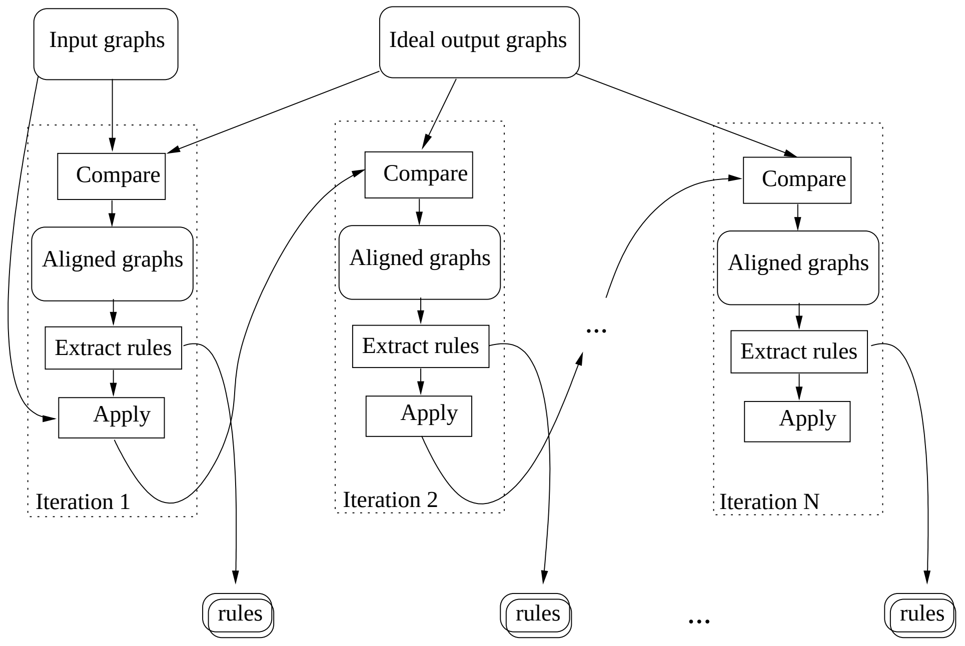


Figure 5.1: Outline of the graph transformation method. Arrows indicate data flow. The method takes two corpora (input and output graphs, top) and produces a list of rewrite rules (bottom).

Moreover, merges are only created using alignments that preserve the order of nodes in each graph (see Section 3.5.3 on page 43 for more details).

Let us also recall that a merge M of a graph pair (F, G) is a union of three disjoint sets of graph elements, $M = M_{left} \cup M_{both} \cup M_{right}$, such that $M_{left} \cup M_{both}$ is isomorphic to F and $M_{both} \cup M_{right}$ is isomorphic to G. We interpret the three sets as follows:

- M_{left} (the set of *left-only objects* of M) consists of the elements that exist in F but not in G; these elements need to be removed from F in order to make it similar to G;

- M_{right} (the set of *right-only objects* of M) consists of the elements that exist in G but not in F; these elements need to be added to F in order to make it similar to G; and

- finally, M_{both} (the set of *common objects* of M) consists of graph elements common to F and G (those aligned when creating the merge).

Figure 5.2 on the next page gives an example of F and G (left) and their merge M (right). We show the graphs as well as their representations using our logic-based graph formalism.

The first step of an iteration of our method for learning graph transformations, i.e., comparing input and output graphs in the training corpus, results in a corpus of graph pair merges. This corpus is used at the second step as detailed in the following section.

5.4 Identifying possible rewrite rules

The second step of an iteration of our method consists in identifying which rewrite rules are potentially applicable and useful for the task of transforming input graphs to output graphs. We implement this by considering all graph merges in the corpus and extracting all rewrite rules for each merge, as will be described in this section. For a merge $M = M(F, G)$ from the corpus we will assume that F is an input graph of the transformation and G is an output graph, i.e., our method will need to transform left graphs of merges into right graphs of merges.

In a given graph pair merge, we will identify possible graph rewrite rules of three different types:

1. changing attributes of a single object (node or edge);
2. adding connected subgraphs with possible change of graph neighborhood;
3. removing a connected subgraph.

Below we describe the extraction of possible rules of these three types given a graph merge.

5.4.1 Rewrite rules: changing object attributes

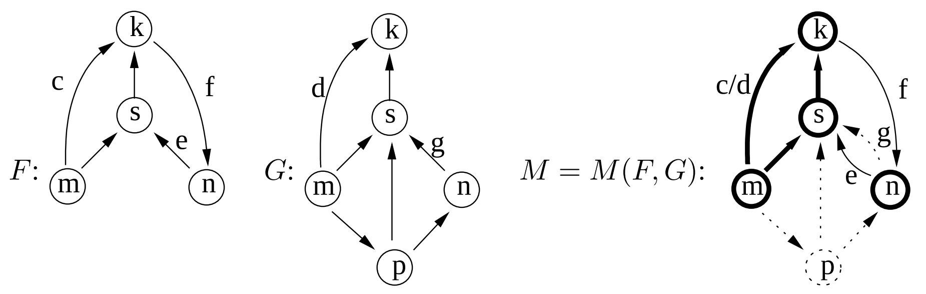
We start with the simplest possible graph rewrite rule: changing attributes of an node or an edge. This rule will be extracted from a graph merge for objects that belong to both left and right parts of the merge, but whose attributes in the two parts are not identical.

We start with simple definitions. For a merge M and an object $x \in M_{both}$, we define ***left-attributes*** of x as follows:

$$Attr_{left}(x) \overset{\text{def}}{=} \{ \mathsf{attr}(x, a, v) \in M_{left} \}.$$

The definition of ***right-attributes*** of x is similar:

$$Attr_{right}(x) \overset{\text{def}}{=} \{ \mathsf{attr}(x, a, v) \in M_{right} \}.$$



$F = \{\, \text{node}(x_1),$
$\text{node}(x_2), \text{node}(x_3),$
$\text{node}(x_4),$
$\text{attr}(x_1, \text{label}, \text{k}),$
$\text{attr}(x_2, \text{label}, \text{s}),$
$\text{attr}(x_3, \text{label}, \text{m}),$
$\text{attr}(x_4, \text{label}, \text{n}),$
$\text{edge}(y_1, x_3, x_1),$
$\text{attr}(y_1, \text{label}, \text{c}),$
$\text{edge}(y_2, x_2, x_1),$
$\text{edge}(y_3, x_1, x_4),$
$\text{attr}(y_3, \text{label}, \text{f}),$
$\text{edge}(y_4, x_3, x_2),$
$\text{edge}(y_5, x_4, x_2),$
$\text{attr}(y_5, \text{label}, \text{e})\,\}$

$G = \{\, \text{node}(z_1),$
$\text{node}(z_2), \text{node}(z_3),$
$\text{node}(z_4), \text{node}(z_5),$
$\text{attr}(z_1, \text{label}, \text{k}),$
$\text{attr}(z_2, \text{label}, \text{s}),$
$\text{attr}(z_3, \text{label}, \text{m}),$
$\text{attr}(z_4, \text{label}, \text{n}),$
$\text{attr}(z_5, \text{label}, \text{p}),$
$\text{edge}(w_1, z_3, z_1),$
$\text{attr}(w_1, \text{label}, \text{d}),$
$\text{edge}(w_2, z_2, z_1),$
$\text{edge}(w_4, z_3, z_2),$
$\text{edge}(w_6, z_4, z_2),$
$\text{attr}(w_6, \text{label}, \text{g}),$
$\text{edge}(w_7, z_3, z_5),$
$\text{edge}(w_8, z_5, z_2),$
$\text{edge}(w_9, z_5, z_4),\,\}$

$M_{both} = \{\, \text{node}(u_1),$
$\text{node}(u_2), \text{node}(u_3),$
$\text{node}(u_4),$
$\text{attr}(u_1, \text{label}, \text{k}),$
$\text{attr}(u_2, \text{label}, \text{s}),$
$\text{attr}(u_3, \text{label}, \text{m}),$
$\text{attr}(u_4, \text{label}, \text{n}),$
$\text{edge}(v_1, u_3, u_1),$
$\text{edge}(v_2, u_2, u_1),$
$\text{edge}(v_4, u_3, u_2)\,\}$
$M_{left} = \{$
$\text{attr}(v_1, \text{label}, \text{c}),$
$\text{edge}(v_3, u_1, u_4),$
$\text{attr}(v_3, \text{label}, \text{f}),$
$\text{edge}(v_5, u_4, u_2),$
$\text{attr}(v_5, \text{label}, \text{e})\,\}$
$M_{right} = \{\, \text{node}(u_5),$
$\text{attr}(u_5, \text{label}, \text{p}),$
$\text{attr}(v_1, \text{label}, \text{d}),$
$\text{edge}(v_6, u_4, u_2),$
$\text{attr}(v_6, \text{label}, \text{g}),$
$\text{edge}(v_7, u_3, u_5),$
$\text{edge}(v_8, u_5, u_2),$
$\text{edge}(v_9, u_5, u_4)\,\}$

Figure 5.2: Two graphs (left) and their possible merge (right). In the diagram of the merge, common objects are in bold, right-only objects are shown with dotted lines.

We define $Label_{left}(x)$ to be a set containing the attribute $\mathsf{attr}(x, \mathsf{label}, l)$ if $\mathsf{attr}(x, \mathsf{label}, l) \in M_{left} \cup M_{both}$, and containing the attribute $\mathsf{attr}(x, \mathsf{type}, t)$ if $\mathsf{attr}(x, \mathsf{type}, t) \in M_{left} \cup M_{both}$. In other words, $Label_{left}(x)$ is the label and the type of the object x in the left part of the merge (if those are defined). Similarly, we define $Label_{right}(x)$ to be the label and the type of x in the right part of the merge.

Whenever for an object x of a merge we have that $Attr_{left}(x) \neq Attr_{right}(x)$, i.e., attributes of x in the left and in the right parts of the merge differ, we consider a possible rewrite rule: replacing the left-attributes of x with its right-attributes. More precisely, we consider a potential rewrite rule r_x:

$$\begin{aligned} LHS(r_x) &= Attr_{left}(x) \cup Label_{left}(x) \\ RHS(r_x) &= Attr_{right}(x) \cup Label_{right}(x). \end{aligned}$$

Note that either of $Attr_{left}(x)$ and $Attr_{right}(x)$ can be empty: in this case the rewrite rule simply adds or removes some of the attributes. Note also that the label of the object is *always* included in the left-hand side of the rewrite rule. This is important for pattern selectivity: it avoids unlabeled patterns.

Consider the example in Figure 5.2. The only object in the merge that belongs to both parts of the merge but has different left- and right-attributes is the edge v_1 that goes from m to k. Indeed,

$$\begin{aligned} Attr_{left}(v_1) &= \mathsf{attr}(v_1, \mathsf{label}, \mathsf{c}), \text{whereas} \\ Attr_{right}(v_1) &= \mathsf{attr}(v_1, \mathsf{label}, \mathsf{d}). \end{aligned}$$

Then according to our definitions above, the following rewrite rule r_{v_1} will be extracted from the merge in Figure 5.2:

$$\begin{aligned} LHS(r_x) &= \mathsf{attr}(x, \mathsf{label}, \mathsf{c}), \\ RHS(r_x) &= \mathsf{attr}(x, \mathsf{label}, \mathsf{d}). \end{aligned}$$

This rule tells us to locate nodes with label c and change their labels to d.

We have now described the extraction of simple rewrite rules: changing object attributes. Now we turn to the second, more complex type of graph rewrite rules.

5.4.2 Rewrite rules: adding subgraphs

We start with several technical definitions. We will first define *right-subgraphs* of a merge as parts of its right graph that can be added to the left graph as one connected chunk. These subgraphs will be used as candidates for right-hand sides of graph rewrite rules. Then, we will define *supports* of a right-subgraph as connected chunks of the left graph of the merge, defining the attachment points of the right-subgraph. Supports will be used as the candidates for left-hand sides of rewrite rules.

More specifically, given a graph merge $M = M(F, G)$, a connected subgraph $S \subset M$ is a ***right-subgraph*** iff:

1. S contains at least one right-only object, i.e., $S \cap M_{right} \neq \emptyset$;
2. for each right-only node, S contains all its incident edges;
3. for every object in S, all its right-only attributes also belong to S, i.e., for every $x \in \mathcal{O}_S$: $\mathsf{attr}(x, l, v) \in M_{right} \Rightarrow \mathsf{attr}(x, l, v) \in S$;
4. for every object x in S, elements of $Label_{right}$ also belong to S;
5. S is minimal, i.e., no objects can be removed from S without violating the above conditions.

It is useful to think of right-subgraphs of a merge as being created in the following way. We start from a subgraph consisting of a single right-only object (node or edge) and add to it all right-only attributes, all incident nodes or edges, labels and types of all objects, and all their right-only attributes. If there are right-only objects among the added ones, we continue the graph extension in the same way, until no more right-only objects are added. It is easy to see that all conditions for the resulting subgraph are satisfied, and moreover, each right-only subgraph can be constructed in this way, starting from any of its right-only objects.

Consider the example of a graph merge in Figure 5.2(right). Starting from the right-only node u_5 (p), we expand the right-subgraph to include its incident edges, i.e., v_7 (m $\rightarrow$ p), v_8 (p $\rightarrow$ s) and v_9 (p $\rightarrow$ n), together with their label and type attributes (not present in this example). To make the resulting set a proper graph, we also add all incident nodes of the newly added edges, i.e., the nodes u_3 (m), u_2 (s) and u_4 (n), as well as their labels. Since no new right-only nodes have been added, we stop the subgraph extension. The resulting right-subgraph of the merge consists of the nodes labeled p, m, s and n and edges connecting p to the other three nodes. This graph is shown in Figure 5.3 (graph S_1). Note that the merge M contains another right-subgraph, consisting of two nodes and one edge. This right-subgraph is also shown in Figure 5.3 (graph S_2)

For a right-subgraph S of a merge M, we define its ***border*** as the set of nodes of S which are common objects of M, and its ***interior*** as all other nodes, edges and attributes of S:

$$Brd(S) \stackrel{\text{def}}{=} \{\mathsf{node}(n) \in S \cap M_{Both}\},$$
$$Int(S) \stackrel{\text{def}}{=} S \setminus Brd(S).$$

For the example in Figure 5.2, considered above, the border of the right-subgraph containing the node p (graph S_1 in Figure 5.3) consists of three nodes: m, s and n. In a sense, these are the nodes that "connect" the right-subgraph to the rest of the graph. The interior of the subgraph consists of the node p and its three incident edges.

A *support* of a right-subgraph S of a merge M is a subgraph of M such that

1. it is a connected graph;
2. it contains $Brd(S)$ and does not contain any right-only objects;
3. it contains $Label^{left}(x)$ for all its objects x;
4. it is minimal, i.e., no elements can be removed from it without violating the above conditions.

In words, a support of a right-subgraph S is a minimal connected subgraph of the left graph of the merge, containing the border of S and containing labels and types of all its objects (if they exist). A support of S can easily be computed by finding a minimal connected extension of the $Brd(S)$ not containing any right-only objects.

Intuitively, for a merge $M(F, G)$, a right-subgraph S can be added to F "in one move" by attaching its interior to the border nodes (which are already present in F, by definition), while border nodes of S can be identified in F by finding an occurrence of a support graph. Figure 5.3 shows two right-subgraphs for the example in Figure 5.2 (S_1 and S_2) and their respective supports (S_1^* and S_2^*).

We will consider right-subgraphs of a merge and their supports as defining possible right- and left-hand sides of graph rewrite rules rules. More specifically, for a merge M, a right-subgraph S and its support S^* we define a rewrite rule r_S with left-hand side and right-hand side as follows:

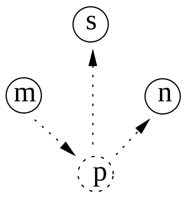
$$LHS(r_S) = S^*,$$
$$RHS(r_S) = S \cup (S^* \setminus M_{left}).$$

The rule r_S says that we should find occurrences of the support S^* and replace each such occurrence with the right-subgraph S, leaving also parts of the support that belong to the right graph of the merge. The effect of the rule is that the left-only elements of each occurrence of the support S^* are removed, and the interior of the right-subgraph S is added to a graph.

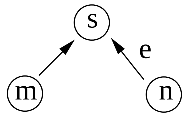
Figure 5.4 shows the two rewrite rules generated in this way for the two right-subgraphs and their supports in Figure 5.3.

Now, having described two types of rewrite rules that we will extract from graph merges (changing attributes and adding subgraphs), we turn to the third and last type.

Right-subgraph S_1:



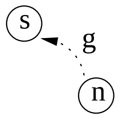
Support S_1^*:



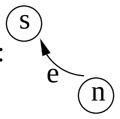
$$S_1 = \{\,\mathsf{node}(u_2), \mathsf{node}(u_3),\\ \mathsf{node}(u_4), \mathsf{node}(u_5),\\ \mathsf{attr}(u_2, \mathsf{label}, \mathsf{s}),\\ \mathsf{attr}(u_3, \mathsf{label}, \mathsf{m}),\\ \mathsf{attr}(u_4, \mathsf{label}, \mathsf{n}),\\ \mathsf{attr}(u_5, \mathsf{label}, \mathsf{p}),\\ \mathsf{edge}(v_7, u_3, u_5), \mathsf{edge}(v_8, u_5, u_2),\\ \mathsf{edge}(v_9, u_5, u_4)\}$$

$$S_1^* = \{\,\mathsf{node}(u_2), \mathsf{node}(u_3),\\ \mathsf{node}(u_4),\\ \mathsf{attr}(u_2, \mathsf{label}, \mathsf{s}),\\ \mathsf{attr}(u_3, \mathsf{label}, \mathsf{m}),\\ \mathsf{attr}(u_4, \mathsf{label}, \mathsf{n}),\\ \mathsf{edge}(v_4, u_3, u_2),\\ \mathsf{edge}(v_5, u_4, u_2),\\ \mathsf{attr}(v_5, \mathsf{label}, \mathsf{e})\}$$

Right subgraph S_2:



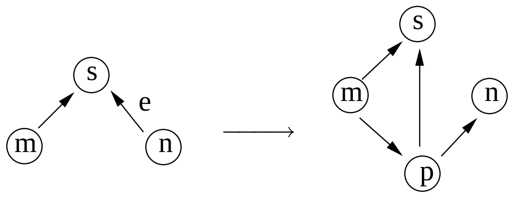
Support S_2^*:



$$S_2 = \{\,\mathsf{node}(u_2), \mathsf{node}(u_4),\\ \mathsf{attr}(u_2, \mathsf{label}, \mathsf{s}),\\ \mathsf{attr}(u_4, \mathsf{label}, \mathsf{n}),\\ \mathsf{edge}(v_6, u_4, u_2),\\ \mathsf{attr}(v_6, \mathsf{label}, \mathsf{g})\}$$

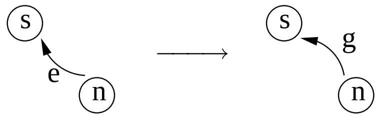
$$S_2^* = \{\,\mathsf{node}(u_2), \mathsf{node}(u_4),\\ \mathsf{attr}(u_2, \mathsf{label}, \mathsf{s}),\\ \mathsf{attr}(u_4, \mathsf{label}, \mathsf{n}),\\ \mathsf{edge}(v_5, u_4, u_2),\\ \mathsf{attr}(v_5, \mathsf{label}, \mathsf{e})\}$$

Figure 5.3: Two add-subgraphs of the merge M in Figure 5.2 (S_1 and S_2) and examples of their supports (S_1^* and S_2^*, respectively).



$$LHS = \{\, \mathsf{node}(u_2), \mathsf{node}(u_3), \mathsf{node}(u_4), \mathsf{attr}(u_2, \mathsf{label}, \mathsf{s}), \mathsf{attr}(u_3, \mathsf{label}, \mathsf{m}), \mathsf{attr}(u_4, \mathsf{label}, \mathsf{n}), \mathsf{edge}(v_4, u_3, u_2), \mathsf{edge}(v_5, u_4, u_2), \mathsf{attr}(v_5, \mathsf{label}, \mathsf{e}) \}$$

$$RHS = \{\, \mathsf{node}(u_2), \mathsf{node}(u_3), \mathsf{node}(u_4), \mathsf{node}(u_5), \mathsf{attr}(u_2, \mathsf{label}, \mathsf{s}), \mathsf{attr}(u_3, \mathsf{label}, \mathsf{m}), \mathsf{attr}(u_4, \mathsf{label}, \mathsf{n}), \mathsf{attr}(u_5, \mathsf{label}, \mathsf{p}), \mathsf{edge}(v_4, u_3, u_2), \mathsf{edge}(v_7, u_3, u_5), \mathsf{edge}(v_8, u_5, u_2), \mathsf{edge}(v_9, u_5, u_4) \}$$



$$LHS = \{\, \mathsf{node}(u_2), \mathsf{node}(u_4), \mathsf{attr}(u_2, \mathsf{label}, \mathsf{s}), \mathsf{attr}(u_4, \mathsf{label}, \mathsf{n}), \mathsf{edge}(v_5, u_4, u_2), \mathsf{attr}(v_5, \mathsf{label}, \mathsf{e}) \}$$

$$RHS = \{\, \mathsf{node}(u_2), \mathsf{node}(u_4), \mathsf{attr}(u_2, \mathsf{label}, \mathsf{s}), \mathsf{attr}(u_4, \mathsf{label}, \mathsf{n}), \mathsf{edge}(v_6, u_4, u_2), \mathsf{attr}(v_6, \mathsf{label}, \mathsf{g}) \}$$

Figure 5.4: Two rewrite rules extracted from the merge M in Figure 5.2 using right-subgraphs and supports shown in Figure 5.3.

5.4.3 Rewrite rules: removing subgraphs

In order to identify possible graph rewrite rules that involve removing whole subgraphs (but possibly without adding new graph elements), we first introduce the notion of ***left-subgraphs*** in a way similar to the right-subgraphs in the previous section. Left-subgraphs are the parts of graphs that can be removed as one chunk.

More precisely, for a graph merge $M = M(F, G)$, a connected subgraph $S \subset M$ is a ***left-subgraph*** if:

1. S contains at least one left-only object;
2. for each left-only node S contains all its incident edges;
3. for every object in S, all its left-only attributes also belong to S, i.e., for every $x \in \mathcal{O}_S$: $\mathsf{attr}(x, l, v) \in M_{right} \Rightarrow \mathsf{attr}(x, l, v) \in S$;
4. for every object x in S, elements of $Label_{left}(x)$ also belong to S;
5. S is minimal, i.e., no objects can be removed from S without violating the above conditions.

For a left-subgraph S of a merge M, we also define its ***border***, $Brd(S)$ as the set of nodes of S which are common objects of M.

For example, the graph merge in Figure 5.2, considered above, contains two left-subgraphs: R_1, consisting of the edge labeled f going from k to n, and R_2, consisting of the edge e going from n to s:

$$\begin{aligned} R_1 &= \big\{ \mathsf{node}(u_1), \mathsf{attr}(u_1, \mathsf{label}, \mathsf{k}), \mathsf{node}(u_4), \mathsf{attr}(u_4, \mathsf{label}, \mathsf{n}), \\ &\qquad \mathsf{edge}(v_3, u_1, u_4), \mathsf{attr}(v_3, \mathsf{label}, \mathsf{f}) \big\}, \\ Brd(R_1) &= \big\{ \mathsf{node}(u_1), \mathsf{node}(u_4) \big\}, \\ \\ R_2 &= \big\{ \mathsf{node}(u_4), \mathsf{attr}(u_4, \mathsf{label}, \mathsf{n}), \mathsf{node}(u_2), \mathsf{attr}(u_2, \mathsf{label}, \mathsf{s}), \\ &\qquad \mathsf{edge}(v_5, u_4, u_2), \mathsf{attr}(v_5, \mathsf{label}, \mathsf{e}) \big\}, \\ Brd(R_2) &= \big\{ \mathsf{node}(u_4), \mathsf{node}(u_2) \big\} \end{aligned}$$

For every S, a left-subgraph of a merge M, we define the following possible rewrite rule:

$$\begin{aligned} LHS(r_S) &= S, \\ RHS(r_S) &= (S \setminus M_{left}) \cup Attr_{right}\big(Brd(S)\big). \end{aligned}$$

Effectively, the extracted rule removes the left-only subgraph S, leaving only its border nodes and adjusting their attributes. For example, the following rewrite rules will be extracted for the left-subgraphs R_1 and R_2:

$$\begin{aligned}
LHS(r_{R_1}) = \{ & \mathsf{node}(u_1), \mathsf{attr}(u_1, \mathsf{label}, \mathsf{k}), \mathsf{node}(u_4), \mathsf{attr}(u_4, \mathsf{label}, \mathsf{n}), \\
& \mathsf{edge}(v_3, u_1, u_4), \mathsf{attr}(v_3, \mathsf{label}, \mathsf{f}) \}, \\
RHS(r_{R_1}) = \{ & \mathsf{node}(u_1), \mathsf{node}(u_4) \},
\end{aligned}$$

$$\begin{aligned}
LHS(r_{R_1}) = \{ & \mathsf{node}(u_4), \mathsf{attr}(u_4, \mathsf{label}, \mathsf{n}), \mathsf{node}(u_2), \mathsf{attr}(u_2, \mathsf{label}, \mathsf{s}), \\
& \mathsf{edge}(v_5, u_4, u_2), \mathsf{attr}(v_5, \mathsf{label}, \mathsf{e}) \}, \\
RHS(r_{R_2}) = \{ & \mathsf{node}(u_4), \mathsf{node}(u_2) \}
\end{aligned}$$

Note that for this example the right-hand sides of the rules coincide with the borders of the respective left-subgraphs.

Let us briefly summarize Section 5.4. We have described three general types of possible rewrite rules that can be extracted from a corpus of graph merges: changing arbitrary attributes of graph objects, adding connected subgraphs, and removing connected subgraphs. These types are defined for any graph merges and they do not use any knowledge about the structure of the graphs, except for the special treatment of the attributes type and label, which are always included in the left-hand sides of the extracted rewrite rules.

Now, having described the extraction of possible rewrite patterns from graph merges, we turn to the next step in an iteration of our method for learning graph transformations.

5.5 Selecting rewrite rules

Given a set of all possible rewrite rules extracted from a corpus of graph pair merges as described above, how do we select which of these rewrite rules should be used at the current iteration of the transformation method?

In Transformation Based Learning, which uses the same iterative extract-apply scheme, each extracted transformation rule is evaluated on the training corpus to estimate its effectiveness in reducing mismatches between input and output taggings. Then the rewrite rule with the highest error reduction is selected (or, for efficiency reasons, several best rules) at each iteration of the method, which corresponds to hill-climbing search for the optimal sequence of transformation rules.

The advantage of this rule selection criterion is that the measure for estimating a rule's efficiency (error reduction) can be dependent on the task at hand and its evaluation metric. In typical applications of Transformation Based Learning (e.g., part-of-speech tagging), the estimate of the performance of a set of rules is often based on reduction of accuracy errors (e.g., tagging accuracy).

In our graph-transformation setting such a criterion based on task-specific evaluation measures is difficult to apply for efficiency reasons. Applying a specific rewrite rule to the entire training corpus and then calculating the error reduction is very time-consuming as it involves an NP-complete subgraph isomorphism problem (graph pattern matching). Moreover, the candidate rewrite rules extracted from the training corpus are incomplete: as described in Section 5.4, we only extract possible left-hand sides and right-hand sides, while constraints are left to be implemented later, using a machine learning engine. In principle, it would be possible for each of the extracted candidate rewrite rules to train a machine learner that implements its constraint, apply the resulting rules with constraint to the training corpus and calculate the error reduction for each rule. However, the total number of patterns extracted at one iteration is rather prohibitive: for different tasks described later in the thesis it ranges from 3,000 to 80,000. Obviously, we need to back-off to heuristic estimation of the effectiveness of the extracted rewrite rules.

For a heuristic, we take a simple frequency-based estimation of a rule's effectiveness. More precisely, in the set of all candidate rewrite rules extracted from the training corpus, $\mathcal{W} = \big\{(LHS_i, RHS_i)\big\}$, we identify the top N_L most frequent (up to graph isomorphism) left-hand sides: $\mathcal{L} = \big\{LHS_j\big\}$, where $j = 1, \ldots, N_L$. Then, for each left-hand side $LHS_j \in \mathcal{L}$, we find the top N_R most frequent right-hand sides that that occur with LHS_j in $\mathcal{W}$: $RHS_j^1, \ldots, RHS_j^{N_R}$. This leaves us with the set $\mathcal{W}_{best} = \big\{(LHS_j, RHS_j^k)\big\}$, where $j = 1, \ldots, N_L$ and $k = 1, \ldots, N_R$. The rules in $\mathcal{W}_{best}$ constitute the result of our rule selection procedure.

In all experiments reported in the following chapters of the thesis, we arbitrarily took $N_L = N_R = 20$, which gave us at most $N_L \times N_R = 400$ rewrite rules per iteration of the graph transformation method.

Once the set of rewrite rules has been determined, for each rule, we train a separate machine learner that implements the constraint of the rule. This training is described in the next section.

5.6 Training the constraints for rewrite rules

At this stage in every iteration of our graph transformation method we have extracted from the training corpus a list of left- and right-hand sides of potentially useful rewrite rules (no more than 400, see the previous section). In this section we describe how the constraints for these rewrite rules are implemented by training machine learning classifiers. For each rule, the trained classifier (the constraint) will determine whether or not the rewrite rule should be applied at a given occurrence of its left-hand side.

Training proceeds as follows. For every rule $r = (LHS, RHS)$, we extract all occurrences of LHS in all graphs of the training corpus. For each such occurrence, we encode the information about the occurrence and its immediate graph neighborhood, as a vector of named scalar features (see below). For each occurrence we also check whether it actually corresponds to an instance of r, i.e., whether r was extracted from this position in the training corpus, in which case the occurrence is a positive example for classifier being trained. This information, collected for all occurrences of r, is used as a training base for a machine learning classifier. The trained learner can then be used to classify feature vectors describing occurrences of LHS into positive (the rule r should be applied) and negative (the occurrence should be left intact).

The output of this step of an iteration of our graph transformation method is, thus, a set of graph rewrite rules with constraints: $\{(LHS, C, RHS)\}$.

We describe the encoding of occurrences as feature vectors and the details of training the classifier below in Sections 5.6.1 and 5.6.2.

5.6.1 Encoding pattern occurrences

As described above, for every unconstrained graph rewrite rule $r = (LHS, RHS)$ we extract from the training corpus all occurrences of LHS that correspond both to positive and negative instances of possible applications of r. The task of a classifier will be to separate these cases.

Traditionally, in machine learning a classification task is defined as automatically assigning instances to one class out of several possible classes, with instances represented as vectors of scalar features. Moreover, some classifiers put additional restrictions on the types of instance features (e.g., allowing only numerical or binary features). We will address the latter issue in Section 5.6.2.

How do we encode an occurrence of a graph pattern as a flat vector of scalar features that can be given to a machine learning classifier?

In Chapter 4, when describing paths in graphs using feature vectors, we used a pre-defined task-specific set of features: labels of nodes and edges, existence of incident edges with certain labels, labels of their end points, etc. Obviously, a feature set defined in this way depends not only on the exact graph transformation task we are addressing, but also on the details of the graph representation. For example, for the predicate argument structure identification task considered in Chapter 4, changing the syntactic parser would require re-defining at least some of the features. Here we describe a general method for encoding an occurrence of a pattern in a graph as a set of named scalar features, that is independent on the task being addressed and on the exact nature of structures represented using graphs.

Let L be a pattern, and u its occurrence in a graph G, i.e., a one-to-one mapping between the objects of L and a subset of the objects of G: $u : \mathcal{O}_L \rightarrow \mathcal{O}_G$. We will use $u(x)$ to denote the object of G assigned to the object x of the pattern L, and $\overline{u}(y)$ to denote the object of L which is mapped to y in the graph G (if such an object exists). Moreover, $u(L)$ is the entire image of L in G.

For an object $x \in G$ we define $dsc(x)$, the *descriptor* of x, as the string obtained by concatenating values of its type and label attributes (using empty strings in place of undefined attributes). For example, if

$$\{\mathsf{node}(x), \mathsf{attr}(x, \mathsf{type}, \mathsf{phrase}), \mathsf{attr}(x, \mathsf{type}, \mathsf{NP})\} \subset G,$$

then $dsc(x) = \mathsf{phrase{:}NP}$.

We define the *neighborhood* of an occurrence u (notation: $Nb(u)$) as a set of objects of G such that:

1. it contains all objects of the occurrence of the pattern: $u(L) \subset Nb(u)$;
2. for every node of $u(L)$ it contains all its incident edges and their end points;
3. it is the smallest set satisfying these conditions.

In other words, the neighborhood of an occurrence of a pattern contains only nodes and edges of the occurrence, plus those *one step away*. When describing a pattern occurrence as a set of features, we will only include information about the objects in its neighborhood. Note that there is no general reason to include in the neighborhood only objects one step away. We can also consider larger neighborhoods, though it will dramatically increase the number of scalar features that are generated for an occurrence.

When encoding a neighborhood using a set of scalar features, we will preserve the information about how exactly a node or an edge of the neighborhood is related to the objects of the pattern occurrence. We implement this using the set $Paths(u)$ that for each object of $Nb(u)$ stores its possible relations (graph paths) to the nodes and edges of the pattern occurrence. The set $Paths(u)$ will consist of pairs (x, d), where d will be used as the scalar (string) description of the object x with respect to the occurrence u of the pattern L. More specifically, $Paths(u)$ is a set such that:

1. for each $x \in u(L)$, $(x, \overline{u}(x)) \in Paths(u)$; in other words, nodes and edges of the pattern occurrence are described using their own preimages in the pattern;

2. for an edge $e \in Nb(u) \setminus u(L)$ from node x to node y, if $(x, d) \in Paths(u)$ for some string d, then $(e, \textit{d}\text{:out:}dsc(e)) \in Paths(u)$; in other words, all neighboring edges are described by concatenating the description of one of their end points with the direction of the edge (i.e., out) and the descriptor of the edge;

3. similarly, if $(y, d) \in Paths(u)$ for some string d, then $(e, \textit{d}\text{:in:}dsc(e)) \in Paths(u)$;

4. for a node $x \in Nb(u) \setminus u(L)$, if x is an end point of some edge e such that $(e, d) \in Paths(u)$, then $(x, \textit{d}\text{:node}) \in Paths(u)$; in other words, neighboring nodes are described by concatenating the description of its connecting edge with the fixed string node.

Consider the example in Figure 5.5: a graph G, a pattern L and its occurrence u in G. For this example all objects of G fall into the neighborhood of the occurrence as all objects are at most one step away from it:

$$Nb(u) = \{k, m, n, p, d, a, b\}.$$

Starting from the nodes and the edge of the occurrence, the following description of the neighborhood is generated:

$$\begin{aligned} Paths(u) = \big\{ & (k, \text{x}), (m, \text{y}), (d, \text{z}), \\ & (a, \text{x:in::A}), (b, \text{x:out::B}), (c, \text{y:out:T:C}), \\ & (n, \text{x:in::A:node}), (p, \text{x:out::B:node}), (p, \text{y:out:T:C:node}) \big\}. \end{aligned}$$

Note that for this example $Paths(u)$ contains two different descriptions for the node p, because the node is accessible from the occurrence of the pattern by following either edge b or edge c.

Let us now assume that for a given occurrence u of a pattern L in a graph G we have computed the sets $Nb(u)$ and $Paths(u)$. The feature set $Feat(u)$, consisting of pairs (feature name, feature value) is defined so that for every $(x, d) \in Paths(u)$:

$$\begin{aligned} & (d, 1) \in Feat(u), \text{ and} \\ & (\textit{d:name}, val) \in Feat(u) \text{ for every } \text{attr}(x, name, val) \in G, \\ & \qquad\qquad\qquad\qquad \text{excluding label and type for edges.} \end{aligned}$$

In other words, for every description d of an object x in the neighborhood of the occurrence, we generate one feature (with name d and value 1) that indicates the presence of the object, and then one more feature for every attribute of x, excluding label and type attributes of edges, which are already implicitly indicated in feature names.

Continuing our running example (see Figure 5.5), the following features will be generated for $Paths(u)$:

$$\begin{aligned}
Feat(u) = \big\{ & (\mathsf{x}, 1), (\mathsf{y}, 1), (\mathsf{z}, 1), \\
& (\mathsf{x{:}label}, \mathsf{K}), (\mathsf{y{:}label}, \mathsf{M}), (\mathsf{z{:}label}, \mathsf{D}), \\
& (\mathsf{x{:}in{::}A}, 1), (\mathsf{x{:}out{::}B}, 1), (\mathsf{y{:}out{:}T{:}C}, 1), \\
& (n, \mathsf{x{:}in{::}A{:}node}), (p, \mathsf{x{:}out{::}B{:}node}), (p, \mathsf{y{:}out{:}T{:}C{:}node}), \\
& (\mathsf{x{:}in{::}A{:}node{:}label}, \mathsf{N}), (\mathsf{x{:}out{::}B{:}node{:}label}, \mathsf{P}), (\mathsf{y{:}out{:}T{:}C{:}node{:}label}, \mathsf{P}) \big\}
\end{aligned}$$

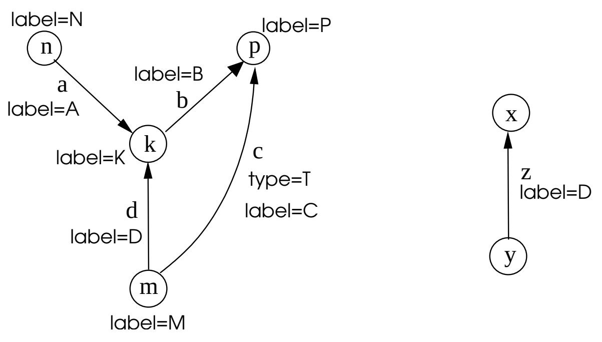
For a given occurrence of a pattern L in a graph G, the number of computed named features depends on the size of the pattern (i.e., the number of nodes and edges) and on the topology of the graph G, in particular, on the degrees (the number of in- and out-edges) of the nodes in the occurrence, as well as on the number of attributes that are associated with nodes and edges in the neighborhood. In practical examples that we consider in later chapters, the number of named symbolic features extracted for a single occurrence is typically not very large, between 20 and 30, but for some patterns it can be as much as 130.

5.6.2 Classification task

Having extracted a set of named features for every occurrence of a pattern, i.e., the left-hand side of a rewrite rule $r = (LHS, RHS)$, in a corpus, we first convert these occurrences to feature vectors. We collect all feature names used in all feature sets extracted from the training corpus and enumerate them sequentially. Since different occurrences of the same pattern often correspond to differing sets of features (due to the varying neighborhoods of the occurrences), the total number of features extracted from the entire corpus can be much larger than for individual occurrences.

In order to avoid extremely large sets of features, we only considered those that occur at least $N_f = 20$ times in the feature sets extracted from the training corpus. In the applications of our transformation method considered in later chapters, for a single pattern the number of features left after this thresholding was typically in the range 100–3,000, though in some cases as large as 6,000.

After enumerating and thresholding the features for occurrences of a given pattern, we convert each occurrence to a flat feature vector with symbolic features. For statistical classifiers (e.g., SVM-Light, used in our applications later in the thesis) that can only process numerical and binary features, we need to take an additional step: binarization of feature values. Since some of the features used



$$G = \big\{ \mathsf{node}(n), \mathsf{attr}(n, \mathsf{label}, \mathsf{N}), \mathsf{node}(p), \mathsf{attr}(p, \mathsf{label}, \mathsf{P}), \mathsf{node}(k), \mathsf{attr}(k, \mathsf{label}, \mathsf{K}), \mathsf{node}(m), \mathsf{attr}(m, \mathsf{label}, \mathsf{M}), \mathsf{edge}(a, n, k), \mathsf{attr}(a, \mathsf{label}, \mathsf{A}), \mathsf{edge}(b, k, p), \mathsf{attr}(a, \mathsf{label}, \mathsf{B}), \mathsf{edge}(c, m, p), \mathsf{attr}(a, \mathsf{label}, \mathsf{C}), \mathsf{attr}(c, \mathsf{type}, \mathsf{T}), \mathsf{edge}(d, m, k), \mathsf{attr}(a, \mathsf{label}, \mathsf{D}) \big\}$$

$$L = \big\{ \mathsf{node}(x), \mathsf{node}(y), \mathsf{edge}(z, y, x), \mathsf{attr}(z, \mathsf{label}, \mathsf{D}) \big\}$$

$$u = \big\{ x \mapsto k, y \mapsto m, z \mapsto d \big\}$$

Figure 5.5: Graph G (left), pattern L (right) and occurrence u of L in G.

in our NLP applications allow thousands of different values (e.g., lexical features, such as words and lemmas), the binarization explodes the number of features even further, making it as large as 460,000 in some cases.

Once all occurrences of LHS are described using feature vectors suitable for the machine learner, we train a separate classifier for each rewrite rule $r = (LHS, RHS)$, to predict, for an occurrence of LHS, whether the rewrite should be applied or not.

5.7 Applying rewrite rules

As described above, at each iteration of the graph transformation method, a set of rewrite rules is selected and classifiers that implement their constraints are trained. The result of each iteration is a set of constrained rewrite rules $\mathcal{W} = \big\{(LHS_j, C_j^k, RHS_j^k)\big\}$ (note that in general, there can be several possible right-hand sides for a single left-hand side, each with its own constraint).

We apply these rewrite rules to a corpus of graphs (whether to the training corpus, as a preparation for the next iteration of the method, or to a previously unseen test/application corpus) as follows. For each pattern LHS_j, we extract all its occurrences in the corpus and use the trained classifiers to evaluate the constraints $\{C_j^k\}$. We then select the right-hand side RHS_j^k for which the classifier of C_j^k produces the highest confidence score. Finally, we apply the rewrite rule $LHS_j \to RHS_j^k$ to the occurrence, i.e., the occurrence of LHS_j is removed from a corresponding graph and an occurrence of RHS_j^k is added instead (see Section 3.2 for the definition of the application of a rewrite rule).

For each LHS_j, we select a corresponding RHS_j^k and apply the rewrite rule to the graphs independently.

5.8 Practical considerations and refi nements

While describing the method we intentionally omitted or avoided discussing some important but rather technical aspects. We summarize those below.

5.8.1 Magic constants

Our method is parameterized with several constants. Although formally they are parameters it is not clear how to set their value. We selected the values once based on our preliminary experiments and did not systematically experiment with different ones.

- $N_L = 20$ is the number of left-hand sides or rewrite rules considered at each iteration of the learning cycle;

- $N_R = 20$ is the maximum number of right-hand sides per one LHS. When extracting rewrite rules, for each LHS we consider only at most N_R most frequent RHS's;

- $N_f = 20$ is the minimum number of times a feature should occur in the training data in order to be used for the training the classifier;

- $\delta = 0.1$ is used in the application in later chapters for the learning termination criterion: the learning cycle is stopped as soon as the improvement of some task-specific measure, as compared to the previous iteration, is less than δ.

5.8.2 Rule selection

Graph rewrite rules are selected as described above, based on frequencies of their left-hand sides among all extracted possible rewrites. In all applications later in the thesis, for each learning task we disallow repetitions of left-hand sides at different iterations of the learning. This was done in order to allow less frequent rules to be selected. We did not systematically experiment with this feature to see its effect on the performance.

5.8.3 Defi nition of patterns

Patterns are extracted from actual graphs. When including an object in a pattern, we always include two of its attributes (if present): type and label. The former defines the types of the object (word, phrase, dependency relation, sequential order edge, etc.) and the latter determines the selectivity of the pattern: for a phrasal node this is constituency label, for dependency relations this is the label of the dependency.

Edges with type = ord, i.e., those defining the order of words or siblings in a phrase tree, are never included in left-hand sides (patterns).

5.9 Related work

Our approach to learning graph transformations is largely inspired by Transformation-Based Error-Driven Learning (Brill, 1995; Curran and Wong, 2000), and can be seen as a variant and extension of it.

Transformation-Based Learning (TBL) is a supervised learning method that identifies a sequence of simple transformation rules to correct some initial annotation of data, e.g., the results of some coarse-grained analyzer. The method starts by comparing this initial annotation with the *truth*, e.g., a manually annotated gold standard corpus. TBL considers a list of candidate transformations, usually specified using templates. Each transformation consists of a rewrite rule and a triggering environment. Below is an example transformation from (Brill, 1995) for the task of part of speech tagging:

Rewrite: *Change the tag from modal to noun.*
Trigger: *The preceding word is a determiner.*

A TBL application is defined by (1) an initial annotator, (2) a space of allowable transformations, and (3) an objective function for comparing the corpus to the *truth*. The method proceeds iteratively, at each iteration using the objective function to select the most promising transformation, appending the selected transformation to the list, the result of the learning, and moreover, updating the training corpus using this transformation. Iteration stops when no transformation is found that improves the objective function of the training corpus.

Various refinements and improvements of the TBL learning process have been suggested and studied. To give a few examples, Brill (1995) mentions learning with a look-ahead window, that is, when several rules are evaluated in sequence when selecting a rule at a given iteration; the rule that starts the best sequence is selected, rather than the best rule, thereby making the greedy search more informed. Williams *et al.* (2004) propose a different technique, a look-behind searcher: after selecting a next transformation the system is allowed to try and reorder the last n learned transformations so as to improve the objective function. Curran and Wong (2000) describe learning with transformation template evolution: at later iterations the system is allowed to consider more and more complex transformations, which are less frequent and more time consuming.

Transformation-Based Learning has been successfully applied to a broad variety of language processing tasks, from part of speech tagging (Brill, 1995), to prepositional phrase attachment disambiguation (Brill and Resnik, 1994), grammatical relation finding (Ferro *et al.*, 1999), document format processing (Curran and Wong, 1999), grammar induction (Brill and Ngai, 2000), semantic role labeling (Williams *et al.*, 2004) and parsing (Brill, 1996; Satta and Brill, 1996). The main reasons for the popularity of TBL among computational linguists are:

- simplicity: the learning model is indeed very simple and clear;
- directness: the learning method works in a hill-climbing manner, optimizing any given objective function, whether it is the number of tagging errors, F_1-score of precision and recall or some possibly exotic task-specific measure;
- efficiency: extracted rules are typically deterministic and simple, which allows efficient implementations of the application of the rules, sometimes even linear and independent of the number of rules (Roche and Schabes, 1995); highly efficient algorithms are known even for Transformation-Based Parsing operating with subtree rewrite rules (Satta and Brill, 1996);
- interpretability: unlike many statistical learning methods, TBL produces lists of simple transformation rules, typically not very long, that can be easily examined manually, and indeed, often are linguistically meaningful.

Note that we follow the model of TBL fairly closely: we also extract transformations that consist of the rewrite part and a trigger (we call it constraint). Several important differences, however, are worth noting:

- whereas in most applications of TBL transformation templates are fairly simple and predefined (cf. the example transformation above), our rewrite rules in principle can be arbitrarily complex; remember that when considering candidate rules, we extract maximal connected left-subgraphs; these subgraphs might be of very different sizes and topologies, as we will see in the applications of our method in later chapters of the thesis;
- unlike TBL's triggers, our constraints are complex functions implemented by a machine learning classifier; this can be seen both as advantage (we use more flexible, and hence, potentially more accurate rules) and a drawback (it increases the application time of the rewrite rules, as each time a classifier needs to be invoked to check the constraint);
- while classic TBL methods extract one best rule per iteration, we extract rules in batches (20 in all applications in this thesis), mainly for the sake of performance. We are not aware of any work in the TBL community that studied possible effects of using TBL in such a batch mode.

Is it possible to run the classic TBL method for our graph transformation problems, addressing the issues above and empirically comparing the effect of our deviation from TBL?

While we cannot answer this question now, we believe that such a comparison would be very helpful in understanding the inner operation of our method and locating potential applications for TBL and its variants. We leave this question as an important aspect of future work.

Another area of research directly related to our work is graph-based data mining (Cook and Holder, 2000). Approaches developed within this fast-growing field of data mining address at least two different problems: identifying frequent subgraphs in a big graph (Kuramochi and Karypis, 2001), and identifying subgraphs maximizing specific information-theoretic measures (Yan and Han, 2002). Since in our TBL-like setting we are interested both in finding frequent graph patterns and in patterns that allow us to discriminate between different right-hand sides of rewrite rules as accurately as possible, both of these tasks are relevant for providing potential improvements to our simple frequency-based pattern selection, and moreover, they are important for a study of the relation between our method and TBL in future work.

5.10 Conclusions

In this chapter we have described a general approach to the task of learning graph transformations given a training corpora of input and output graphs. Our approach combines ideas from Transformation Based Learning with graph pattern matching and conventional supervised machine learning. The method iteratively compares input and output graphs, identifying useful rewrite rules and applying the rules to the input graphs. The result of the learning method is a sequence of graph rewrite rules that can be applied to previously unseen graphs.

In the following chapters of the thesis we will examine several instantiations of the method to different NLP tasks. In particular, in later chapters we present an application of our graph transformation method to the problem of recovering predicate argument structure that was addressed by an ad-hoc graph-based method in Chapter 4. We will show that the generality of the method does not reflect negatively on the performance for this task and, on the contrary, allows us to improve the results presented in Chapter 4. Moreover, we will illustrate the flexibility of our method by considering a variant of this task that works with syntactic phrase trees rather than with dependency structures, which will allow for a more direct comparison of our results with the state-of-the-art.

Chapter 6

Graph Transformations for Predicate Argument Structure Identifi cation

In Chapter 4 we defined the task of predicate argument structure (PAS) identification as recovery of empty nodes, non-local dependencies and grammatical and semantic tags in syntactic structures produced by a syntactic parser. The definition of the task and the data used for solving it were based on the PAS information annotated in the Penn Treebank II (PennTB). We presented our first approach to the task: a method to automatically learn graph transformations by comparing syntactic dependency graphs produced by a parser to dependency graphs derived from the Penn Treebank. Our method allowed us to learn a sequence of simple graph rewrite rules: correcting labels of dependencies (graph edges), adding new graph nodes and, finally, adding new graph edges.

In Chapter 5, the core chapter of the thesis, we described a more general, task-independent method for learning graph transformations. In contrast to the approach of Chapter 4, this method is not restricted to simple "one object at a time" transformations and does not require pre-defining the order of the transformation sequence. Instead, given a corpus of graph pairs, the method learns a sequence of graph rewrite rules that may involve arbitrary connected subgraphs. In Chapter 5 we gave general motivations for such a method. The purpose of the present chapter is to demonstrate the method at work and thus to provide empirical support for these motivations. In particular, we will apply the method to the task of PAS identification, compare its performance to the approach of Chapter 4, and show that the generality of the method is actually beneficial for the task. We will also apply our method to a variant of the task: PAS identification in constituency structures rather than in dependency graphs, demonstrating that the method is general enough to operate with different syntactic representations. All in all, the experiments described in this chapter will support our claim about the generality, flexibility and effectiveness of the graph transformation method proposed in Chapter 5.

The chapter is organized as follows. We start in Section 6.1 by reviewing some of the motivation stances from Chapter 5. Then, in Section 6.2 we recapitulate the key issues of the task and the transformation method of Chapter 5. In Section 6.3 we apply the method to the task of predicate argument identification in dependency structures, and in Section 6.4 to the same task for phrase trees. We conclude in Section 6.5.

6.1 Motivation

In Chapter 4 we gave a detailed description of the task of identifying predicate argument structure of English sentences in the output of a syntactic parser, and we presented our first graph-based method for addressing the task. The method used the Penn Treebank, a corpus with PAS annotation, to learn a sequence of atomic graph rewritings that correspond to recovery of the PAS. In spite of the state-of-the-art performance, this method for predicate-argument structure identification has a number of drawbacks.

First of all, the method is task- and corpus-dependent. We defined the types of possible graph transformations (changing edge labels, adding nodes, adding edges) based on our knowledge of the task and experience with the data (Penn Treebank) and syntactic parsers. The extraction of some of the features (such as verb subjects and objects) used during the machine learning stages is also corpus-specific. Applying a similar method to the output of a different parser, to a different syntactic formalism (e.g., constituency trees in place of dependency structures), or to a different corpus would require additional feature engineering.

In Chapter 4 we considered three graph transformation templates in sequence: first, changing edge labels, then adding nodes, and finally, adding edges. Of course, there is no guarantee that this particular sequence of transformations is indeed the best possible one, and that other types of transformations (removing edges, manipulating subgraphs with several nodes, etc.) do not suit the task better. Allowing different or more complex transformations may well prove beneficial, and so may changing the sequence of transformations, or using transformations of the same type several times in the sequence.

Let us give an example of such transformations. Consider the two dependency graphs in Figure 6.1, one derived from the output of Charniak's parser and the other from the Penn Treebank. Transforming the parser's graph to the Penn Tree-

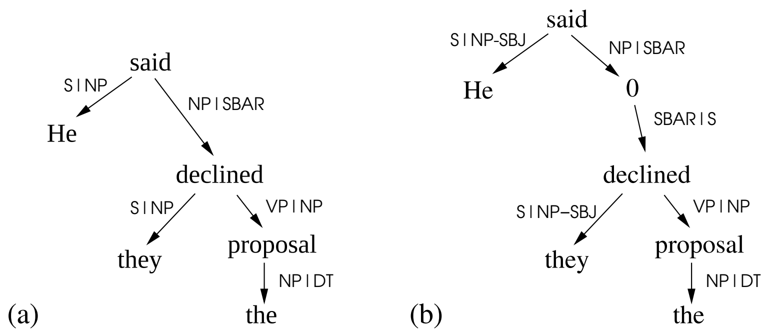


Figure 6.1: Example requiring a complex transformation: dependency structures derived from (a) the output of Charniak's parser and (b) the Penn Treebank, for the sentence "*He said they declined the proposal*."

bank graph involves inserting an empty complementizer (annotated as the empty element "0" in the Penn Treebank) in the subordinate clause. This transformation is most naturally described as a rewrite rule that removes an edge of a graph and replaces it with another subgraph consisting of one node and two edges. This particular transformation can be quite useful as empty complementizers abound in the Penn Treebank (constituting 10% of all empty elements). However, this natural transformation cannot be implemented with the sequence of the tree simple transformation templates used in Chapter 4. The method of Chapter 4 was capable of inserting the empty complementizer by splitting this transformation into two simpler ones (first adding 0 with the edge *said*→0 at the second iteration, and then adding the edge 0→*declined* at the third iteration), but it was unable to remove the remaining incorrect edge *said*→*declined*.

Yet another problem with the approach of Chapter 4 is that it strongly depends on our choice of syntactic formalism: dependency structures. Indeed, while our simple graph rewrite templates largely suffice for the PAS identification on dependency graphs, transformations involving more complex subgraphs are required for the task of PAS recovery in constituency trees (Johnson, 2002), which indicates that the approach of Chapter 4 is not directly applicable for this variant of the task.

Our general approach to graph transformations, described in detail in Chapter 5, allows us to address all these issues, as we will show in the course of the present chapter. But first, in the next section, we will recap our notation for graphs and patterns and the overall structure of our graph transformation method; this section may be skipped by readers who have just read Chapter 5.

6.2 Graphs and transformations: a recap

As described in detail in Chapter 3, we represent syntactic dependency structures and phrase trees of English sentences as node- and edge-labeled, node-ordered directed graphs, with nodes corresponding to words or constituents, and edges to syntactic dependencies or parent-child edges in phrase trees; the ordering of nodes reflects the left-to-right ordering of words and constituents (children of a single parent).

We represent graphs as sets (conjunctions) of first-order atomic formulas (atoms):

- node(x): declares a graph node identified by x,
- edge(x, y, z): declares a graph edge with id x, directed from the node with id y to the node with id z,
- attr(x, n, v): for the object (node or edge) with id x, declares an attribute with name n and string value v. We will also use the notation $n = v$, when x is clear from the context.

Nodes of a dependency graph correspond to words of the sentence and are labeled using the attributes word, lemma and pos, which specify the word, its lemma and its part of speech tag, respectively. Empty nodes derived from the Penn Treebank are identified using the attribute empty $= 1$. Edges of a dependency graph correspond to word-word dependencies. They are marked with the attribute type $=$ dep and labeled using the attribute label. Traces (non-local dependencies) derived from the Penn TB are similar to dependency edges, but are additionally marked with the attribute trace $=$ 1. Nodes are ordered using edges with the attribute type $=$ ord.

Nodes of constituency trees (phrase structures) correspond to terminals and non-terminals of the trees. Terminals (words) are marked with the attribute type $=$ word and labeled using attributes word, lemma and pos, the word, its lemma and its part of speech tag. Non-terminals (phrases) are identified by the attribute type $=$ ph and are labeled with label. Parent-child relations in the constituency tree are represented by edges with type $=$ child from parents to children, and traces derived from the Penn Treebank by edges with type $=$ trace. Empty nodes (terminals, as well as non-terminals without any non-empty terminal descendants) are marked with empty $=$ 1. Terminal nodes are ordered according to the surface word order using edges with type $=$ ord and label $=$ surf. Children of the same parent node are ordered using type $=$ ord and label $=$ sibling. We refer to Section 3.4 on page 37 for more details and examples of dependency and phrase structure graphs.

When comparing two graphs, two possibly different analyses of the same sentence (e.g., one produced by a parser and another derived from a treebank), we merge some of the nodes, edges and attributes of the two graphs as identical. The resulting structure (a graph merge) is a graph such that each element (a node, edge or attribute) stores information whether it belongs to only one of the graphs or to both of them. See Section 3.5.2 on page 42 for more details.

As described in Chapter 5, we use a corpus of graph pair merges to automatically learn which rewrite rules should be applied to input graphs to obtain output graphs. Each graph rewrite rule involves a connected subgraph (i.e., a set of graph elements, the left-hand side of the rule, LHS for short) that should be substituted in a source graph with another subgraph (another set of atoms, the right-hand side of the rule, RHS for short). The decision whether a specific occurrence of a left-hand side of a rewrite rule should be replaced with the corresponding right-hand side, is made by a machine learner. The machine learner uses a context of such an occurrence (i.e., information about the entire local neighborhood of the occurrence, encoded as a feature vector) to make its decision. An LHS, an RHS and a trained machine learner implementing the constraint together define a graph rewrite rule.

Our method for learning graph transformations (Chapter 5) works iteratively. We start by comparing input and output graph pairs of the training corpora and extracting possible left- and right-hand sides of rewrite rules. Next, we determine the most frequent LHS's and RHS's and train a machine learner for each pair of LHS and RHS. The resulting rewrite rules are stored and applied to the training corpus, resulting in a new corpus of graph pair merges, which is, in turn, used for the extraction of a next series of rewrite rules at the next iteration. The method iterates until some task-specific convergence criterion is met. The result of the learning method is, thus, a sequence of graph rewrite rules that can be applied to new input graphs.

When extracting possible LHS's and RHS's from a corpus of graph merges, as described in Chapter 5, we consider the following transformation types:

- Any possible modification of attributes of single objects (nodes and edges). In this case, possible LHS's and RHS's are objects and the attributes that need changing.
- Modifying (adding, removing, replacing) connected subgraphs. In this case, LHS's are connected subgraphs and RHS's arbitrary subgraphs, specifying nodes, edges, their types and labels (values of the type and label attributes), along with all other attributes that are to be modified by a transformation rule.

In this chapter we describe the application of our graph transformation method to the two different flavors for the PAS identification task: one operating on dependency structures and another that identifies PAS using constituency trees. We start with the PAS identification using dependency graphs in the next section.

6.3 Identifying PAS using dependency graphs

In Chapter 2 we introduced a procedure for converting constituency trees as they are annotated in the Penn Treebank and produced by syntactic parsers of Charniak (2000) and Collins (1999) to dependency graphs, with dependency labels derived from the treebank's or parser's constituent labels. With this choice of the syntactic formalism, we re-formulated the task of identifying PAS in a parser's output as a graph transformation problem: adding the missing information that defines the predicate argument structure of a sentence. We refer to Chapter 2, and specifically to Section 2.3 and Figure 2.5 on page 28 for more details and an example.

Now we describe an application of our method for learning graph transformations to the same task and the same data.

As in Chapter 4, we used the Penn Treebank annotations and the output of Charniak's parser on the strings of the corpus, both converted to dependency graphs, to create a corpus of graph pair merges. To allow a meaningful comparison of the results of our method to the approach of Chapter 4, we used the same standard split of the corpus: Penn Treebank sections 02–21 were used for training, sections 00–01 for development and performance estimation during training, and section 23 was used once to obtain the final evaluation results.

6.3.1 Learning transformations

Starting with the training corpus of graph merges from sections 02–21, and the held-out development corpus from sections 00–01, we iterated the graph transformation method of Chapter 5 as follows:

- At each iteration at most 20 most frequent LHS's among all rewrite rules extracted from the current corpus of merges were selected as LHS's of possible transformations.

- For each selected LHS, at most 20 most frequent RHS's were selected.

- For each of the resulting (LHS, RHS) pairs, a separate SVM classifier (we used the SVM-Light package by Joachims (1999)) was trained using all occurrences of the LHS in the training corpus of the current iteration: each classifier predicts whether a particular occurrence of the LHS should be replaced with the RHS.

	Development corpus		Test corpus	
Stage	unlab P / R	lab P / R	unlab P / R	lab P / R
Initial	88.3 / 83.5	69.5 / 65.7	87.6 / 83.2	68.0 / 63.5
1	89.4 / 87.7	82.4 / 80.8	88.8 / 87.3	82.1 / 80.7
2	89.9 / 88.7	84.5 / 83.3	89.3 / 88.3	84.3 / 83.4
3	90.2 / 88.9	84.8 / 83.6	89.6 / 88.6	84.7 / 83.7
4	90.2 / 88.9	84.8 / 83.6	89.6 / 88.6	84.7 / 83.7

Table 6.1: Evaluation results for PAS identification in dependency graphs derived from Charniak's parser: labeled and unlabeled precision and recall of dependency relations (including non-local dependencies).

- At each iteration, all rewrite rules were applied in parallel to the current training and development corpora to prepare for the next iteration. The resulting new development corpus of graph merges was then evaluated on labeled dependencies (Section 2.1.1 on page 21). We calculated the F_1 score of precision and recall for all graph edges of $\mathsf{type} = \mathsf{dep}$ on the development corpus. In case the improvement of the F_1 score over the score at the previous iteration is smaller than 0.1%, the learning process was terminated. Otherwise a new iteration was started.

Because of the convergence criterion (no increase in F_1 score of at least 0.1% on the development corpus), the application of the method stopped at the fourth iteration. Table 6.1 shows labeled and unlabeled precision and recall, evaluated both on the development and test corpora.

How does the performance of our general method at the PAS identification task compare to the task-specific method of Chapter 4? Table 6.2 shows the labeled precision and recall scores for the two methods as they changed through the iterations. Note that the method of Chapter 4 involved three iterations whereas the new method stopped after four iterations, each with patterns with 20 different left-hand sides.

The two systems demonstrate a very similar performance at their final iterations, with the system of Chapter 4 showing a slightly better recall (83.86 vs. 83.74), but lower precision (84.16 vs. 84.66). This results in a better F_1 score for our graph transformation-based system, though the differences between the two are not statistically significant using the t-test with $p = 0.05$.

In Chapter 4, apart from providing the overall dependency evaluation results, we gave a comparison of the performance of the system on several subtasks to the related work: the recovery of the Penn Treebank function and grammatical

	Chapter 4			Here		
Stage	P	R	F_1	P	R	F_1
Initial	68.0	63.5	65.7	68.0	63.5	65.7
1	83.8	78.2	80.9	82.1	80.7	81.4
2	84.1	79.8	81.9	84.3	83.4	83.9
3	84.2	**83.9**	84.0	84.7	83.7	84.2
4	–	–	–	**84.7**	83.7	**84.2**

Table 6.2: Comparison of the results for PAS identification in dependency graphs for the method of Chapter 4 and of this chapter.

tags (Blaheta and Charniak, 2000), empty nodes and non-local dependencies (Dienes, 2004; Dienes and Dubey, 2003a). We will not go into such a detailed analysis in this section, since later in this chapter, in Section 6.4, we will describe our approach to the PAS identification on constituency trees, which allows for a much more direct comparison to the results in literature, which are also presented for constituency structures.

But before we turn to phrase structures, we will first have a closer look at the rewrite rules that were automatically extracted by our system for the PAS identification in dependency graphs. In the four iterations used by the method, in total 80 LHS's of the rewrite rules were identified. Below is a summary of the sizes of the LHS's: the number of patterns containing 1 to 4 nodes (no patterns were extracted with 5 or more nodes in the LHS).

Size of pattern (nodes)	Number of patterns
1	25
2	49
3	4
4	2

As can be expected, most of the patterns are quite simple and correspond to changing edge (dependency) labels: 33 of the 49 two-node patterns were of this type. The second most frequent transformation pattern involved single nodes: in most cases this corresponds to the rewrite rules that only change node attributes.

In the following section we give examples and a detailed analysis of some of the transformation rules extracted during the iterations of our method.

Left-hand side	Count	Right-hand sides	Count	Example
edge(e_0, n_0, n_1), attr$(e_0$, label, S|NP$)$	67850	edge(e_0, n_0, n_1), attr$(e_0$, label, S|NP-SBJ$)$	66649	Figure 6.2
		edge(e_0, n_0, n_1), attr$(e_0$, label, S|NP-TMP$)$	785	Figure 6.3
node(n_0), attr$(n_0$, label, AUX$)$	34178	node(n_0), attr$(n_0$, label, VBZ$)$	11952	Figure 6.4
		node(n_0), attr$(n_0$, label, VBD$)$	8120	
		node(n_0), attr$(n_0$, label, VBP$)$	7956	
edge(e_0, n_0, n_1), attr$(e_0$, label, VP|PP$)$	37788	edge(e_0, n_0, n_1), attr$(e_0$, label, VP|PP-CLR$)$	11171	Figure 6.2
		—	9345	
		edge(e_0, n_0, n_1), attr$(e_0$, label, VP|PP-TMP$)$	4307	
		edge(e_0, n_0, n_1), attr$(e_0$, label, VP|PP-DIR$)$	4194	Figure 6.3
node(n_0), node(n_1), edge(e_0, n_0, n_1), attr$(e_0$, label, SBAR|S$)$	20300	—	12015	(see text)
		node(n_0), node(n_1), edge(e_0, n_0, n_1), attr$(e_0$, label, SBAR|S$)$, edge(E_1, n_1, n_0), attr$(E_1$, label, S|NP-SBJ$)$	5709	Figure 6.5
		node(n_0), node(n_1), edge(e_0, n_0, n_1), attr$(e_0$, label, SBAR|S$)$, edge(E_1, n_1, n_0), attr$(E_1$, label, VP|ADVP-TMP$)$	997	Figure 6.6
		node(n_0), node(n_1), edge(e_0, n_0, n_1), attr$(e_0$, label, SBAR|S$)$, edge(E_1, n_1, n_0), attr$(E_1$, label, VP|NP$)$	537	

Table 6.3: Most frequent left-hand sides of extraction transformation rules, with counts in the training corpus and several of the possible right-hand sides.

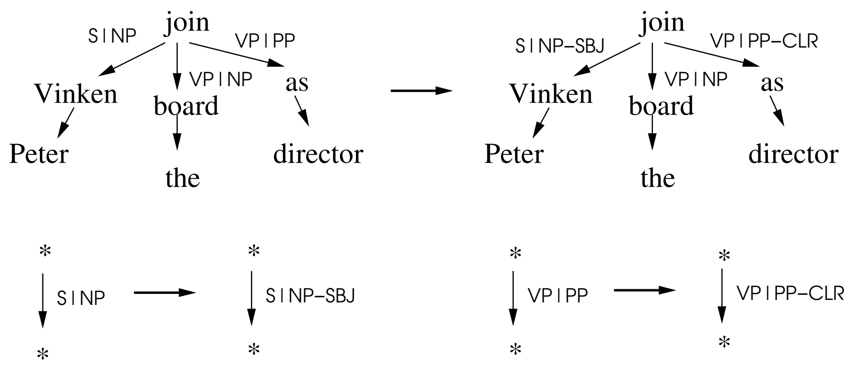


Figure 6.2: Sentence "*Pierre Vinken will join the board as director*": dependency graphs (top) and the extracted transformations (bottom). Some dependency labels omitted.

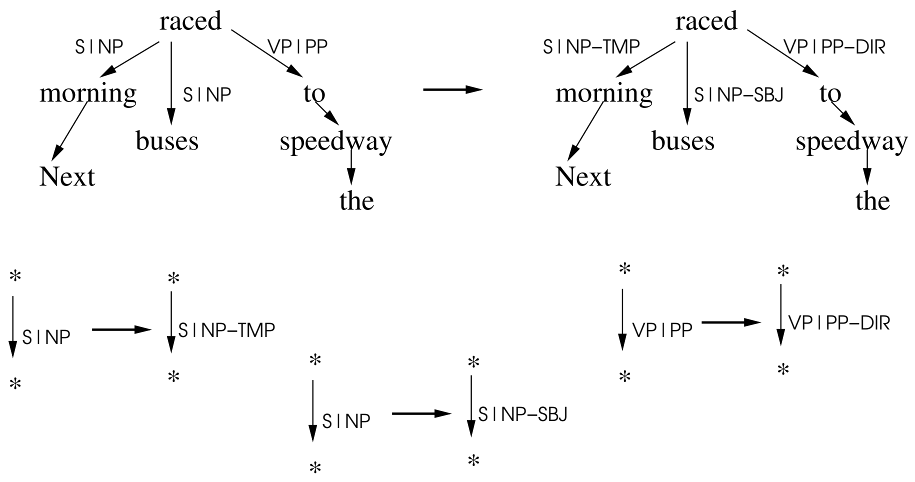


Figure 6.3: Sentence "*Next morning buses raced to the speedway*": dependency graphs (top) and the extracted transformations (bottom). Some dependency labels omitted.

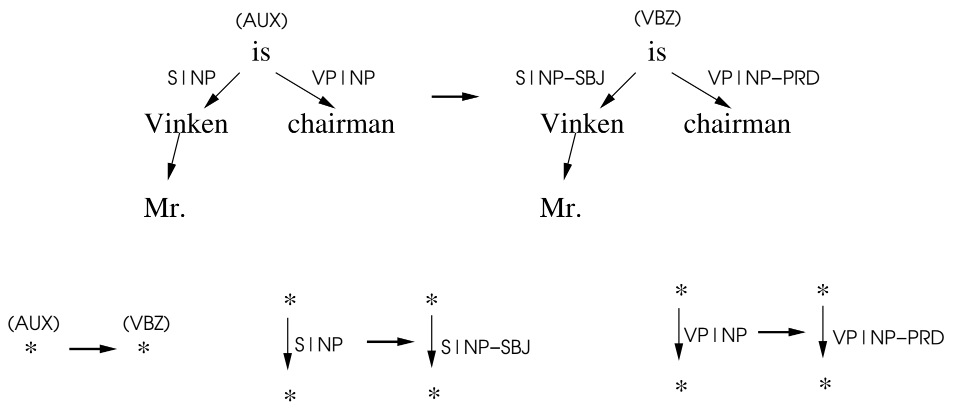


Figure 6.4: Sentence "*Mr. Vinken is chairman*": dependency graphs (top) and the extracted transformations (bottom). Some dependency labels omitted. Parentheses indicate node labels, representing part of speech tags.

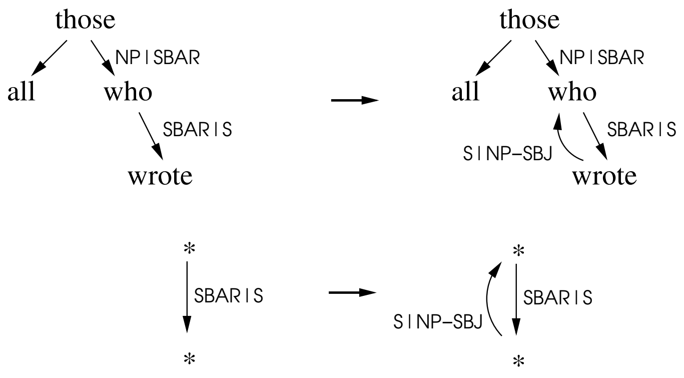


Figure 6.5: Noun phrase "*all those who wrote*": dependency graphs (top) and the extracted transformations (bottom). Some dependency labels omitted.

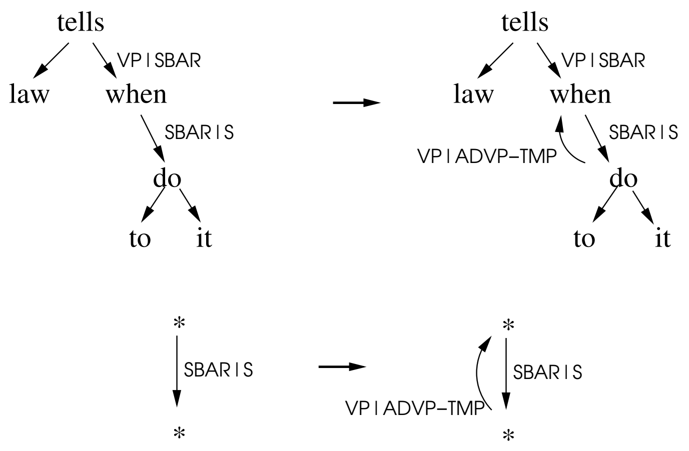


Figure 6.6: Sentence "*law tells when to do it*": dependency graphs (top) and the extracted transformations (bottom). Some dependency labels omitted.

6.3.2 Extracted transformations: a closer look

Table 6.3 on page 97 shows some of the frequent rewrite rules extracted from the training corpus. The most frequent left-hand side is the pattern $\{$ edge(e_0, n_0, n_1), attr$(e_0,$ label, S|NP$)\}$, the first group of rows in Table 6.3 on page 97. With its right-hand sides, it defines changes of edge labels from S|NP to S|NP-SBJ, S|NP-TMP (see the example in Figure 6.3 on page 98), S|NP-ADV, S|NP-VOC, all corresponding to adding functional tags (subject, temporal, adverbial, vocative, resp.). Other right-hand sides for this LHS introduce transformations caused by more or less systematic parsing errors. The most frequent such transformation is changing edge label S|NP to S|", PRN|NP-SBJ, S|S-ADV, SINV|NP-SBJ, S|S-NOM-SBJ etc. These transformations correspond to parsing errors such as incorrect constituent labels (e.g., S in place of PRN, or NP in place of nominalized S).

The pattern $\{$ node(n_0), attr$(n_0,$ label, AUX$)\}$ (the second left-hand side in Table 6.3) defines changes of node labels, which correspond to part of speech tags of words. Charniak's parser assigns the part of speech tag AUX to auxiliary verbs *be, did, has, been* etc., which are annotated as common verbs in the Penn Treebank (tags VB, VBD, VBZ, VBN, respectively). The extracted transformation rules try to fix these mismatches.

The third left-hand side in Table 6.3 is the pattern $\{$ edge(e_0, n_0, n_1), attr$(e_0,$ label, VP|PP$)\}$ that allows the system to add Penn functional tags to prepositional phrases (PP) attached to verb phrases (VP). The functional tags added by these rules are: -CLR, -TMP, -DIR, -LOC, etc. (closely related, temporal, direction, locative, resp.). Some of the PP's in the Penn Treebank are not marked with functional tags—these cases correspond to the empty right-hand side ("—"). There are 14 different possible functional tags detected by our system for VP|PP edges, each defining a rule with the corresponding RHS.

Finally, the fourth group of rows in Table 6.3 is described by the LHS pattern $\{\text{node}(n_0), \text{node}(n_1), \text{edge}(e_0, n_0, n_1), \text{attr}(e_0, \text{label}, \text{SBAR|S}\}$ and gives an example of a more complicated transformation: adding a dependency relation between a relative pronoun and the head of a relative or small clause (see examples in Figures 6.5 and 6.6). Apart from those mentioned in Table 6.3, there are several other possible labels of the dependency relations added in the right-hand sides of the transformations, such as VP|NP (object, as in "*questions that we considered*") or VP|ADVP-MNR (manner adjunct, as in "*explain how to deal with it*").

Table 6.4 shows some other examples of left- and right-hand sides extracted in the first iteration of the learning cycle. The pattern in the first group of rows corresponds to subject control constructions (annotated in the Penn Treebank using empty nodes and co-indexing, and represented with dependency relations in dependency graphs). Note that since our atomic transformations always involve connected graphs with edge labels, the transformations introducing controlled subjects also correct existing labels (e.g., changing S|NP to S|NP-SBJ, Figure 6.7), thus possibly duplicating the effect of other, simpler transformations (e.g., see Figure 6.2). As a result, for a given edge, modifications such as changing edge labels from S|NP to S|NP-SBJ can be identified more than once, by different rules with different left-hand sides. This does not lead to a clash when applying transformations to actual data, as long as the modifications are not conflicting. Clashes (e.g., when one pattern requires a change of a label to S|NP-SBJ and another to S|NP-TMP) are resolved by ordering transformations according to frequencies of left-hand sides in the training corpus. In other words, although several transformations are applied in parallel at each iteration of the learning loop, the most frequent one wins in case of conflict.

The second group of rows in Table 6.8 introduces transformations related to passive constructions (see example in Figure 6.8). Similar to the control construction, a new edge (with label VP|NP, corresponding to verb object) is added to the dependency graph.

In the current section we have described the application of the general graph transformation method to the task of PAS identification in dependency graphs. Now we turn to a related, but different application: PAS identification in phrase structures.

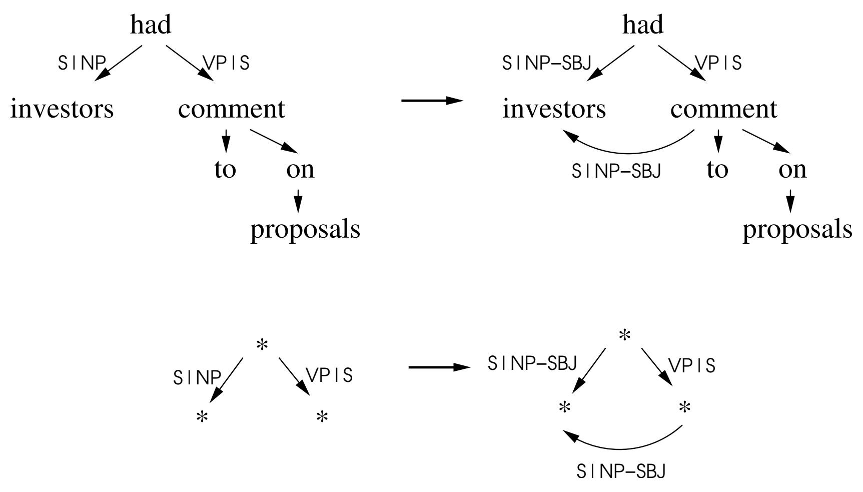


Figure 6.7: Sentence "*investors had to comment on proposals*": dependency graphs (top) and the extracted transformations (bottom). Some dependency labels omitted.

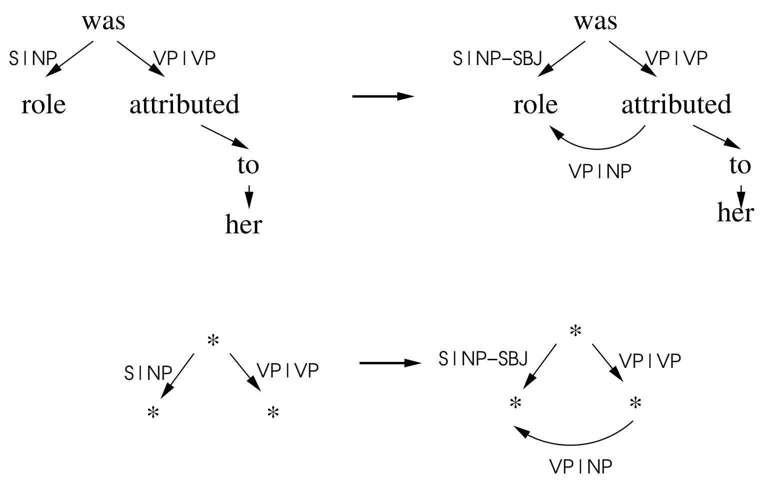


Figure 6.8: Sentence "*role was attributed to her*": dependency graphs (top) and the extracted transformations (bottom). Some dependency labels omitted.

Left-hand side	Count	Right-hand sides	Count	Ex.
node(n_0), node(n_1), node(n_2), edge(e_0, n_0, n_1), attr(e_0, label, S|NP), edge(e_1, n_0, n_2), attr(e_1, label, VP|S)	9564	node(n_0), node(n_1), node(n_2), edge(e_0, n_0, n_1), attr(e_0, label, S|NP-SBJ), edge(e_1, n_0, n_2), attr(e_1, label, VP|S) edge(e_2, n_2, n_1), attr(e_2, label, S|NP-SBJ) attr(e_2, trace, 1)	4067	Fig. 6.7
		—	3654	
		node(n_0), node(n_1), node(n_2), edge(e_0, n_0, n_1), attr(e_0, label, S|NP-SBJ), edge(e_1, n_0, n_2), attr(e_1, label, VP|S-ADV) edge(e_2, n_2, n_1), attr(e_2, label, S|NP-SBJ) attr(e_2, trace, 1)	761	
node(n_0), node(n_1), node(n_2), edge(e_0, n_0, n_1), attr(e_0, label, S|NP), edge(e_1, n_0, n_2), attr(e_1, label, VP|VP)	7433	node(n_0), node(n_1), node(n_2), edge(e_0, n_0, n_1), attr(e_0, label, S|NP-SBJ), edge(e_1, n_0, n_2), attr(e_1, label, VP|VP) edge(e_2, n_0, n_2), attr(e_2, label, VP|NP) attr(e_2, trace, 1)	4005	Fig. 6.8
		—	2028	

Table 6.4: Some of the extracted patterns.

6.4 Identifying PAS using phrase structures

In Chapter 4 and in the previous section we presented two solutions for the problem of identifying predicate argument structure (including function labels, empty nodes and non-local dependencies) in dependency structures, using the Penn Treebank as the source of the annotated PAS.

Both of our methods involved a conversion step: annotations of the Penn Treebank, as well as the outputs of two phrase structure parsers, were first converted to dependency graphs, using phrase labels and function tags to derive dependency labels, and preserving information about empty nodes and non-local dependencies. Although this conversion step allowed us to reduce the PAS identification task to a graph transformation problem with relatively simple local rewrite rules (such as adding nodes or edges, or changing dependency labels), this approach based on dependency graphs suffers from several methodological shortcomings.

One problematic issue with the conversion to dependencies is that our gold standard corpus, the Penn Treebank, is modified both for training and for the evaluation of the systems using a set of imperfect heuristic rules that are bound to make mistakes. As a result, it is not completely clear whether the comparison of the output of our systems to the test section of this semi-automatically created gold standard corpus actually reflects the performance: the gold standard data has not been directly created by human annotators. In fact, our dependency-based evaluation scores indicate how close our system gets to the data in a rather artificially generated corpus.

Another methodological problem is comparing our results to the performance of other systems that have addressed various sub-problems of the PAS identification task: recovery of empty nodes and non-local dependencies (Dienes, 2004; Dienes and Dubey, 2003a; Johnson, 2002) and function tags (Blaheta, 2004; Blaheta and Charniak, 2000). These systems also used the Penn Treebank as the training corpus for their respective subtasks, but all of them were applied *directly* to the corpus, annotated with constituency trees, and to the output of phrase structure parsers. In Chapter 4 we gave a detailed analysis of the performance of our method on specific subtasks, but the comparison to previous work was rather indirect due to this mismatch of the syntactic representation formalisms (Sections 4.5.1, 4.6.1 and 4.7.1).

Can we use our general graph transformation method to address the PAS identification problem formulated in terms of phrase structures rather than dependency graphs, in order to address these problems? In this section we will give a positive answer to this question.

6.4.1 Definition of the task

We define the tasks of PAS identification in the output of a syntactic parser, as the recovery of *all* of the information annotated in the Penn Treebank, including function and semantic tags of constituents, empty nodes and non-local dependencies. We consider the annotation of the Penn Treebank and the output of a parser for each sentence to be directed labeled phrase structure graphs (see Section 3.4 on page 37), and use our automatic graph transformation method to map phrase trees produced by a parser to the Penn Treebank annotation.

As described in Section 3.4, our representation of phrase structure graphs uses nodes with attributes to represent terminals, non-terminals and empty elements, and edges to represent parent-child relations, antecedents of traces, surface order of the terminals and the order of children of each non-terminal. Figure 6.9 on the following page shows the original Penn Treebank annotation of the sentence "*Directors this month planned to seek more seats*" and the corresponding directed labeled phrase structure graph. Figure 6.10 shows the output of Charniak's parser and the graph for the same sentence.

While converting the bracketed annotations of the Penn Treebank and the output of a parser to our graph representation, the following modifications were made:

- We replaced Penn Treebank co-indexing information (e.g., "-1" in labels NP-SBJ-1 and *-1 in Figure 6.9) with edges with type $=$ antecedent, in order to make the semantics of this Penn Treebank feature explicit in the graphs;

- parent-child edges connecting constituents with their head children were marked with the attribute head $= 1$, and lexical heads of constituents were indicated using edges with the attribute type $=$ lexhead; we used the same heuristic method for head identification and head lexicalization as before, see Section 2.1.2 for details.

When Penn Treebank sentences and the output of the parser are encoded as directed labeled graphs as described above, the task of PAS identification can be formulated as transforming phrase structure graphs produced by a parser into the Penn Treebank graphs.

6.4.2 Learning transformations

As before, we parse the strings of the Penn Treebank with Charniak's parser and then use the data from sections 02–21 of the Penn Treebank to training the graph transformation system, sections 00–01 for development and section 23 for testing.

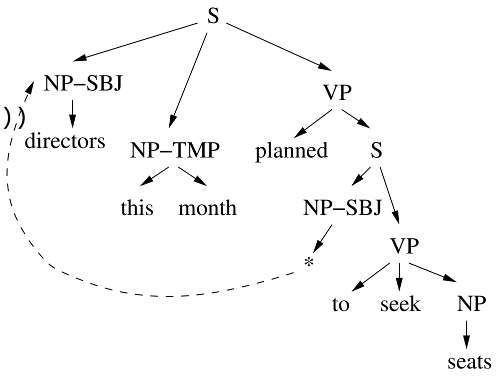
We iterated the transformation method as follows:

- At each iteration we only considered rewrite rules that involved non-local dependencies or function tags; this was implemented by a simple filter that only allowed rules whose RHS contains objects with the attribute empty $= 1$ (i.e., empty nodes) or objects with the label attribute containing a function tag (as indicated by a dash, as in NP-TMP).

- Similar to Section 6.3, at each iteration we extracted at most 20 most frequent LHS's and for each of them 20 most frequent RHS's of possible rewrite rules.

- We used the same classifier as before, SVM-Light (Joachims, 1999) to learn constraints of each of the extracted transformation rules.

```
(S
  (NP-SBJ-1 (NNS Directors))
  (NP-TMP ((DT this) (NN month))
  (VP (VBD planned)
    (S
      (NP-SBJ (-NONE- *-1))
      (VP (TO to) (VB seek)
        (NP (JJR more)
            (NNS seats))))))
```



$$
\begin{aligned}
F = \{ & \text{node}(n_0), \text{node}(n_1), \ldots, \text{node}(n_{15}), \\
& \text{attr}(n_0, \text{type}, \text{phrase}), \text{attr}(n_0, \text{label}, \text{S}), \text{attr}(n_1, \text{type}, \text{phrase}), \text{attr}(n_1, \text{label}, \text{NP-SBJ}), \\
& \text{attr}(n_3, \text{type}, \text{phrase}), \text{attr}(n_3, \text{label}, \text{NP-TMP}), \text{attr}(n_6, \text{type}, \text{phrase}), \text{attr}(n_6, \text{label}, \text{VP}), \\
& \text{attr}(n_2, \text{type}, \text{word}), \text{attr}(n_2, \text{word}, \text{directors}), \text{attr}(n_2, \text{pos}, \text{NNS}), \\
& \text{attr}(n_4, \text{type}, \text{word}), \text{attr}(n_4, \text{word}, \text{this}), \text{attr}(n_4, \text{pos}, \text{DT}), \\
& \text{attr}(n_5, \text{type}, \text{word}), \text{attr}(n_5, \text{word}, \text{month}), \text{attr}(n_5, \text{pos}, \text{NN}), \\
& \text{attr}(n_7, \text{type}, \text{word}), \text{attr}(n_7, \text{word}, \text{planned}), \text{attr}(n_7, \text{pos}, \text{VBD}), \\
& \text{attr}(n_{10}, \text{type}, \text{word}), \text{attr}(n_{10}, \text{word}, *), \text{attr}(n_{10}, \text{pos}, \text{-NONE-}), \text{attr}(n_{10}, \text{empty}, 1), \\
& \ldots \\
& \text{edge}(e_0, n_0, n_1), \text{attr}(e_0, \text{type}, \text{child}), \text{edge}(e_{16}, n_0, n_7), \text{attr}(e_{16}, \text{type}, \text{lexhead}), \\
& \text{edge}(e_{24}, n_1, n_3), \text{attr}(e_{24}, \text{type}, \text{ord}), \text{attr}(e_{24}, \text{label}, \text{sibling}), \\
& \text{edge}(e_{25}, n_3, n_6), \text{attr}(e_{25}, \text{type}, \text{ord}), \text{attr}(e_{25}, \text{label}, \text{sibling}), \\
& \text{edge}(e_1, n_1, n_2), \text{attr}(e_1, \text{type}, \text{child}), \text{attr}(e_1, \text{head}, 1), \\
& \text{edge}(e_{17}, n_1, n_2), \text{attr}(e_{17}, \text{type}, \text{lexhead}), \\
& \text{edge}(e_3, n_3, n_4), \text{attr}(e_3, \text{type}, \text{child}), \\
& \text{edge}(e_4, n_3, n_5), \text{attr}(e_4, \text{type}, \text{child}), \text{edge}(e_{18}, n_3, n_5), \text{attr}(e_{18}, \text{type}, \text{lexhead}), \\
& \ldots \\
& \text{edge}(e_{15}, n_{10}, n_1), \text{attr}(e_{15}, \text{type}, \text{antecedent}), \\
& \ldots \}
\end{aligned}
$$

Figure 6.9: Example of the Penn Treebank annotation, corresponding graph and its logical representation for the sentence "*Directors this month planned to seek more seats*" (ord-edges, lexhead-edges and some attributes are not shown in the graph diagram).

```
(S
  (NP (NNS Directors))
  (NP ((DT this) (NN month))
  (VP (VBD planned)
    (S
      (VP (TO to) (VB seek)
        (NP (JJR more)
            (NNS seats))))))
```

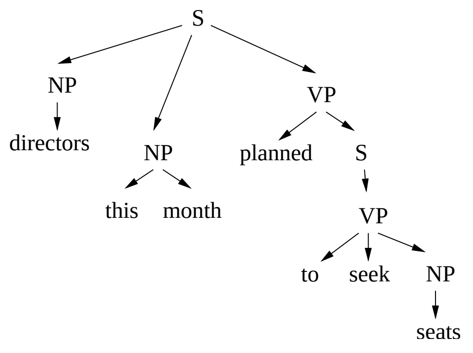


Figure 6.10: Example of the output of Charniak's parser and the corresponding phrase structure graph for the sentence "*Directors this month planned to seek more seats*." (ord-edges, lexhead-edges and attributes are not shown).

- At each iteration, the system was evaluated on the development corpus with respect to both non-local dependencies and function tags of constituents (Section 6.4.3 describes the evaluation measures in detail). If the improvements of the F_1 score for either evaluation measure was smaller than 0.1, the learning cycle was terminated, otherwise a new iteration was started.

6.4.3 Evaluation measures

The commonly accepted evaluation measure for phrase structure parsers, PARSEVAL (see Section 2.1.1), is based on comparing lexical spans of constituents detected by a parser to the gold standard and does not address possible non-local dependencies. Especially for the evaluation of the recovery of non-local dependencies in phrase trees, Johnson (2002) proposed a simple measure, that calculates how well a system identifies empty nodes and their antecedents. We take Johnson's evaluation approach, used also by Dienes and Dubey (2003a) and Dienes (2004), in order both to guide our transformation method through iterations, and to present the final evaluation results and compare them to the earlier work.

More specifically, the evaluation measure for non-local dependencies is the F_1 score of precision and recall of co-indexed empty nodes, where an empty node (a terminal node in a phrase structure graph with pos $=$ -NONE-) is considered correctly identified if:

- the type of the empty node (the value of the word attribute: *T*, *, *?*,0, *U*, etc.) matches the gold standard;

- it is attached as child node to a constituent with the correct head word;

- the head of the antecedent (or NULL, in case the empty node has no antecedents) is the correct word.

The use of this measure enables a straightforward comparison of our results to earlier work.

Unlike the approach of Johnson (2002), that focused only on detection and insertion of empty nodes and their antecedents, our method may also affect the constituents of phrase trees, since the transformations found by the method in principle are not restricted only to empty nodes. It is possible that our system, while adding empty nodes and detecting their antecedents, accidentally modifies the trees, and thus, in principle, a good score for the detection of non-local dependencies might come at the price of decreased PARSEVAL score. Indeed, this effect has been noticed by Dienes (2004), who reports a decrease in PARSEVAL F_1 score from 88.0 to 87.1 when making the parser aware of empty nodes, with the parsing model that shows best results on non-local dependencies. Because of this potential trade-off between accuracy of a system on local structures and on non-local dependencies, we give both measures when reporting results.

Apart from detecting empty nodes and non-local dependencies, our method also corrects labels of constituents, which corresponds to adding grammatical and function tags (-SBJ, -TMP, etc.). For a separate evaluation of the tag detection we use another measure: precision and recall for attaching tags to *correctly identified constituents* (Blaheta, 2004; Blaheta and Charniak, 2000). We consider only those constituents in the parser's output, that have the same lexical span and label as in the gold standard corpus, and for each of these constituents, we count as true positives the tags that are attached to the constituent both in the gold standard and in the results of transforming the parser's output using our system.

6.4.4 Results and comparison to previous work

We applied our graph transformation method using the trees produced by Charniak's parser as input corpus and the Penn Treebank as the output corpus, both converted to phrase structure graphs as described above. We iterated the method until the convergence criteria were met: both the increase in the evaluation score for function tags and the increase of the evaluation score for non-local dependencies are not higher than 0.1. Remember that we used the development corpus (sections 00–01) for these performance estimations at every iteration of the method.

The learning cycle terminated after 12 iterations. Table 6.5 shows the evaluation results on all iterations: identification of function tags, detection of non-local dependencies, and the standard PARSEVAL measure for constituent bracketing.

Unlike the method of Chapter 4, our present system works directly with phrase

	Function tags			Non-local deps			PARSEVAL
Stage	P	R	F_1	P	R	F_1	F_1
Initial	0.0	0.0	0.0	0.0	0.0	0.0	88.7
1	93.3	89.0	91.2	88.2	38.6	53.7	88.4
2	93.4	89.0	91.1	87.2	48.6	62.5	88.4
3	93.4	88.9	91.1	87.5	51.9	65.2	88.4
4	93.4	89.0	91.1	86.7	52.1	65.1	88.4
5	93.4	89.0	91.2	86.1	56.3	68.1	88.3
6	93.4	89.2	91.3	86.0	57.2	68.7	88.4
7	93.4	89.3	91.3	86.3	61.3	71.7	88.4
8	93.4	89.3	91.3	86.6	63.4	73.2	88.4
9	93.4	89.3	91.3	86.7	64.6	74.0	88.4
10	93.4	89.4	91.3	86.7	64.9	74.2	88.4
11	93.4	89.4	91.3	86.6	65.1	74.3	88.4
12	93.3	89.6	91.4	86.7	65.2	74.4	88.4

Table 6.5: Evaluation results for PAS identification in phrase structure graphs produced by Charniak's parser: precision and recall function tags and non-local dependencies, and the standard PARSEVAL score for the parses.

trees produced by Charniak's parser, without converting them to dependency structures. How do the results compare to results of other systems? Below we discuss our results for identification on non-local dependencies and for function tagging.

Non-local dependencies

Table 6.6 on the next page gives a detailed description of our results specifically for the identification of empty nodes and their antecedents. We compare the performance of three systems:

- the system of Dienes and Dubey (2003a), combining a MaxEnt-based tagger that inserts empty elements in sentences, and a lexicalized trace-aware parser extending Model 3 of Collins (1997);
- the system of Dienes (2004), extending the previous one by integrating the tagger and the parser into a probabilistic framework; to our knowledge, this system demonstrates the best published F_1-scores for the identification of non-local dependencies;
- finally, our present system based on the graph transformation method applied to the phrase trees by Charniak's parser and the original Penn Treebank annotation.

		Dienes and Dubey			Dienes (2004)			Here		
Type	Count	P	R	F_1	P	R	F_1	P	R	F_1
PARSEVAL	44278	-	-	86.3	-	-	87.3	88.5	88.2	**88.4**
All empty	3864	81.8	67.9	74.3	82.5	**70.1**	**75.8**	**86.7**	65.2	74.4
NP-NP	1148	74.9	68.3	71.5	78.9	**71.6**	**75.5**	**87.4**	59.5	70.8
COMP-SBAR	545	**93.9**	76.9	84.6	78.6	**83.7**	81.7	89.9	81.3	**85.4**
WH-NP	508	89.6	73.4	80.7	91.6	**77.8**	**84.1**	**94.8**	74.8	83.6
PRO-NP	477	72.4	**69.2**	70.7	74.7	68.1	**71.3**	**75.4**	47.2	58.0
UNIT	388	**99.7**	**93.6**	**96.5**	94.4	91.2	92.8	93.0	92.3	92.6
TOP-S	277	83.9	79.1	81.4	**87.8**	**85.9**	**86.9**	87.0	79.8	83.2
WH-ADVP	171	73.2	41.5	53.0	85.9	46.2	60.1	**92.0**	**70.1**	**79.6**
COMP-WHNP	107	67.8	37.4	48.2	70.4	35.5	47.2	**85.3**	**48.6**	**61.9**

Table 6.6: A detailed comparison of the results of our system (using Charniak's parser) and the systems of Dienes and Dubey (2003a) and Dienes (2004): PARSEVAL bracketing scores and identification of empty nodes and their antecedents in phrase trees.

The column *Count* gives the number of evaluated elements in the test corpus (section 23 of the Penn Treebank): for the PARSEVAL measure, this is the total number of constituents, and for all other rows, the number of corresponding empty elements in the corpus.

While our system outperforms (Dienes and Dubey, 2003a), and therefore our system of Chapter 4, the overall performance on all empty elements is 1.4% worse than that of the best combined system of Dienes (2004). Interestingly, however, the systems show different behavior with respect to precision and recall of non-locals: our transformation-based method allows us to achieve 4% better precision, but the recall is about 5% lower. The performance on specific types of non-local dependencies indicates a similar trend. Moreover, while for all types of empty elements the precision of our system is close to, or substantially better than, the precision of Dienes (e.g., 87.4 vs. 78.9 for NP-NP), the situation is opposite with the recall.

This analysis indicates that our system is cautious in making changes to the graphs. The system seems to apply rewrite rules only when there is enough evidence and the decision is well supported by the training corpus. This allows us to achieve good precision for the recovery of empty nodes (86.7%), but hurts recall. Another explanation to this imbalance is the iterative nature of our method: we learn the most frequent graph transformations first, terminating the learning cycle when the improvement of the F_1 score is lower than 0.1, thus, ignoring less frequent linguistic constructions. Note also that for relatively infrequent types of empty elements (the bottom rows of Table 6.6) that nevertheless "made it" into our transformation system, the results are significantly better than for the system of Dienes (2004), both for precision and recall. It seems that while infrequent language phenomena are difficult for probabilistic approaches (due to the data sparseness problems or because more frequent constructions blur the model), our pattern-based approach allows us to isolate specific phenomena and treat each of them separately, with separate rewrite rules that use pattern-specific classifiers. Our method shares this feature of avoiding over-generalizations with other non-probabilistic approaches to learning for complex tasks: Transformation-Based Learning (Brill, 1995) and Memory-Based Learning (Daelemans and van den Bosch, 2005).

Function tagging

How accurately does our system assign Penn Treebank function tags? In Table 6.7 on the following page we summarize the results of our method and the best MaxEnt-based model of Blaheta (2004). As in Section 4.5.1 on page 56, we use the evaluation measure of Blaheta and Charniak (2000), considering function tags only for constituents correctly identified by the parser with respect to the word span and the phrase label (88.5% of the constuents returned by Charniak's parser on section 23 of the Penn Treebank). Since the system of Blaheta (2004) does not assign tags to empty constituents (i.e., containing only empty elements), as they are not produced by the parser, we exclude those from the evaluation.

Neither of the two systems outperforms the other: the ranking is different for syntactic (SBJ, PRD, LGS, etc.) and semantic (ADV, LOC, TMP) tags. Apart from presenting the results of the MaxEnt model, Blaheta (2004) describes extensive experiments with voted perceptrons (Freund and Schapire, 1998) with different combinations of features. Perceptrons demonstrate a different behavior, with some of the models achieving F_1 score of 98.8% for the identification of syntactic tags, but none of the trained perceptrons performing better than 78.6% for semantic tags. Blaheta reports the perceptron evaluation results only for the development corpus (section 24 of the Penn Treebank).

6.4.5 Extracted transformations: a closer look

We now describe the graph rewrite rules extracted by the system in more detail.

During the 12 iterations of the learning cycle, 240 left-hand sides of rewrite rules were extracted, with a total of 1840 right-hand sides, 7.7 RHS's for one LHS on average. The distribution of the sizes (the number of nodes) of the left-hand sides was as follows:

		(Blaheta, 2004)			Here		
Type	Count	P	R	F_1	P	R	F_1
All tags	8480	-	-	-	93.3	89.6	91.4
Syntactic	4917	**96.5**	95.3	**95.9**	95.4	**95.5**	95.5
Semantic	3225	86.7	80.3	83.4	**89.7**	**82.5**	**86.0**

Table 6.7: A detailed comparison of our results and the results of the MaxEnt model of Blaheta (2004) for assigning function tags to constituents in phrase trees (both using Charniak's parser on section 23 of the Penn Treebank).

Number of nodes:	1	2	3	4	5	6	7	8	9	10	≥11
Number of patterns:	61	39	29	23	24	34	18	7	3	2	0

Rules with one-node left hand sides most often modify labels of constituents (e.g., adding function tags) or part of speech tags. Below are the five most frequent left-hand sides extracted from the training corpus, each with their two most frequent right-hand sides.

$$\begin{aligned}
LHS &= \text{node}(n_0), \text{attr}(n_0, \text{label}, \text{NP}), \text{attr}(n_0, \text{type}, \text{ph})\\
RHS_1 &= \text{node}(n_0), \text{attr}(n_0, \text{label}, \text{NP}), \text{attr}(n_0, \text{type}, \text{ph})\\
RHS_2 &= \text{node}(n_0), \text{attr}(n_0, \text{label}, \text{NP-SBJ}), \text{attr}(n_0, \text{type}, \text{ph})\\
RHS_3 &= \text{node}(n_0), \text{attr}(n_0, \text{label}, \text{NP}), \text{attr}(n_0, \text{type}, \text{ph}),\\
&\quad \text{node}(N_1), \text{attr}(N_1, \text{type}, \text{word}), \text{attr}(N_1, \text{empty}, 1), \text{attr}(N_1, \text{pos}, \text{-NONE-}),\\
&\quad \text{attr}(N_1, \text{word}, \text{*U*}), \text{edge}(E_0, N_1, n_0), \text{attr}(E_0, type, child)
\end{aligned}$$

The LHS above matches phrase nodes labeled NP. The right-hand side RHS_1 is identical to LHS, i.e., it defines an empty rewrite rule. In RHS_2, the label of the node is changed to NP-SBJ. In RHS_3, a new node (N_1) is inserted as a new child of the phrase node: this is an empty unit (*U* with part of speech tag -NONE-).

Below are more examples of rules with a one-node left-hand side:

$$\begin{aligned}
LHS &= \text{node}(n_0), \text{attr}(n_0, \text{pos}, \text{AUX}), \text{attr}(n_0, \text{type}, \text{word})\\
RHS_1 &= \text{node}(n_0), \text{attr}(n_0, \text{pos}, \text{VBZ}), \text{attr}(n_0, \text{type}, \text{word})\\
RHS_2 &= \text{node}(n_0), \text{attr}(n_0, \text{pos}, \text{VBD}), \text{attr}(n_0, \text{type}, \text{word})\\
RHS_3 &= \text{node}(n_0), \text{attr}(n_0, \text{pos}, \text{VBP}), \text{attr}(n_0, \text{type}, \text{word})
\end{aligned}$$

These rules change the part of speech tag AUX, used by Charniak's parser for auxiliary verbs (e.g., *does, had, is*), to one of the tags of the Penn Treebank tag set (VBZ, VBD, VBP, respectively).

Another example of a simple left-hand side:

$$\begin{aligned}
LHS &= \text{node}(n_0), \text{attr}(n_0, \text{label}, \text{S}), \text{attr}(n_0, \text{type}, \text{ph})\\
RHS_1 &= \text{node}(n_0), \text{attr}(n_0, \text{label}, \text{S-TPC}), \text{attr}(n_0, \text{type}, \text{ph})\\
RHS_2 &= \text{node}(n_0), \text{attr}(n_0, \text{label}, \text{S}), \text{attr}(n_0, \text{type}, \text{ph}),\\
&\quad \text{node}(N_1), \text{attr}(N_1, \text{type}, \text{ph}), \text{attr}(N_1, \text{empty}, 1), \text{attr}(N_1, \text{label}, \text{NP-SBJ}),\\
&\quad \text{node}(N_2), \text{attr}(N_2, \text{empty}, 1), \text{attr}(N_2, \text{pos}, \text{-NONE-}),\\
&\quad \text{attr}(N_2, \text{type}, \text{word}), \text{attr}(N_2, \text{word}, *),\\
&\quad \text{edge}(E_0, N_1, n_0), \text{attr}(E_0, \text{type}, \text{child}),\\
&\quad \text{edge}(E_1, N_2, N_1), \text{attr}(E_1, \text{head}, 1), \text{attr}(E_1, \text{type}, \text{child}),\\
&\quad \text{edge}(E_2, N_2, N_1), \text{attr}(E_2, \text{type}, \text{lexhead})
\end{aligned}$$

In the example above, the rule $LHS \to \mathrm{RHS}_1$ changes constituent label from S to S-TPC, adding the function tag that marks topicalized clauses. The rule $LHS \to \mathrm{RHS}_2$ adds a new child of the S node: an empty phrase node NP-SBJ containing a single empty word, which is also marked as the head of the new NP-SBJ.

As these examples demonstrate, the transformation rules our system learns are often interpretable and appear natural for the task we consider. A list of most frequent extracted transformation patterns is available online at `http://staff.science.uva.nl/~jijkoun/thesis/patterns`.

6.5 Conclusions

In this chapter we presented two applications of the general method for learning graph transformations of Chapter 5: identification of Penn Treebank-style predicate argument structure in dependency graphs and in phrase trees produced by a parser. Our method learned a sequence of graph rewrite rules, with constraints of the rules implemented using an SVM classifier. The system does improve the results for the task of PAS identification, as compared to the ad-hoc transformation system described in Chapter 4. Moreover, evaluated separately on the recovery of non-local dependencies and on assigning Penn function tags, our system demonstrates results close to, or better than, the results in the literature.

Although the two applications we considered are similar in the problem addressed, they were defined using different syntactic formalisms: dependency graphs (Section 6.3) and phrase structures (Section 6.4). We showed that in both cases our graph transformation method can handle the task successfully with essentially the same machinery. Therefore, the experiments described in this chapter allow us to give a positive answer to the Research Question 1 on page 14: the learning method of Chapter 5 can indeed be applied to different linguistic structures, without modification of the method itself.

Comparing the results of our method to other approaches in the literature, we showed that its generality does not clash with effectiveness, thereby addressing our Research Question 4 about the limitations of the method.

Our method is based on extraction of graph patterns—left-hand sides of rewrite rules—and training classifiers that select possible rewrites. In this chapter we have seen that this approach has two attractive consequences. First, the rewrite rules produced by our learning method can be interpreted linguistically and they often correspond directly to linguistic phenomena. Unlike some purely statistical approaches to machine learning for NLP problems, our method produces clearer and more interpretable list of rewrite rules, where only a part of a rule is "hidden" in a trained model of a classifier implementing the rule's constraint. We believe that our method inherits this interpretability of learning results from the Transformation-Based Learning paradigm (Brill, 1995).

A second consequence of the pattern-based nature of our method is that it allows us to separate different linguistic contexts for learning. We use statistical classifiers (SVM's in this chapter) as one of the steps inside the method, but we train a separate classifier for each rule, thereby dynamically dividing a complex learning problem into smaller, and possibly simpler, sub-problems. We believe that this accounts for the high precision figures for some of the tasks we considered above. Moreover, while dividing the task into smaller, more digestible chunks, we do not assume independence between the sub-problems: the iterative structure of the method allows for complex dependencies, as rules of later iterations operate on the result of the application of earlier rules. This feature is directly inherited from the Transformation-Based Learning paradigm.

Analyzing the NLP applications of the method in the present chapter, we also identified some of the potential problems. In particular, we noticed that although the precision for a learning task in question may be good, the recall values are often substantially lower than those of state-of-the-art methods. It is not clear whether this phenomenon is intrinsic to the method itself, or can be attributed to our imperfect frequency-based rule selection mechanism. We leave this important question for future research, but will come back it more than once in later chapters of the thesis, where we examine applications of our graph transformation method to other NLP problems. In particular, in the next chapter we will present and analyze the application of our method to a more general task: automatic conversion between different syntactic formalisms.

Chapter 7

Transforming Syntactic Structures

We have already described our general method for learning graph transformations (Chapter 5) and its application to the problem of predicate argument structure (PAS) identification in the output of a syntactic parser (Chapter 6). In fact, we described two applications of our method to two variants of the problem based on different syntactic formalisms: the PAS identification using dependency structures (Section 6.3) and using phrase trees (Section 6.4). Our results showed that the problem can be described as a graph transformation problem and effectively addressed by our transformation learning method, with either syntactic formalism. At an abstract level, this result indicates that at least some NLP tasks can be formulated and solved using methods that do not strongly depend on the underlying syntactic formalism (dependency or phrase structures).

In this chapter we continue our exploration of the applicability of our graph transformation method, and turn to yet another application: automatic conversion between different syntactic representations. As a case study, we take the problem of conversion between two particular types of syntactic dependency formalisms: dependency graphs produced by Minipar (Lin, 1994) and dependency graphs derived from the Penn Treebank (Marcus *et al.*, 1994). The goal of this chapter is to demonstrate the flexibility of our method for learning graph transformations by applying it to another NLP problem.

The chapter is organized as follows. We first describe our motivation for the task in Section 7.1. In Section 7.2 we discuss related work. In Section 7.3 we describe the problem in detail and give examples. We describe the actual application of the graph transformation method to the problem in Section 7.4, and present the results in Section 7.5 and give an analysis of the learned transformation rules in

Section 7.6. We conclude in Section 7.7.

7.1 Motivation

In this chapter we consider the generic problem of transforming syntactic information from one form or formalism into another. The need for effective methods for performing such transformations comes up in various areas, both in applied language technology and in computational linguistics.

Consider a complex NLP system, such as corpus-based Question Answering (QA) system, that uses a syntactic parser as one of its components. Syntactic analysis is essentially one of the key components of such a system and the basis for many other modules and subsystems: offline information extraction (Jijkoun *et al.*, 2004), document retrieval (Katz and Lin, 2003), passage and sentence re-ranking (Bouma *et al.*, 2005), reasoning using lexical chains (Moldovan *et al.*, 2003b). Some of these modules require offline syntactic parsing of the entire text collection available to the QA system, others use a syntactic parser online, while answering a user's question, and only require parsing the most relevant documents or passages. In either case, all modules depend on the exact details of the syntactic analyzer: the representation and encoding of the structures (e.g., trees) and the set of syntactic labels assigned by the parser. For example, Jijkoun *et al.* (2004) use manually created patterns, simple labeled dependency trees, to extract information from dependency graphs generated by Minipar (Lin, 1994). Bouma *et al.* (2005) use manually created tree rewrite rules to generate syntactic paraphrases given constituency trees produced by a syntactic parser, Alpino (Bouma *et al.*, 2001). Clearly, for such systems, changing to a different syntactic parser, e.g., with a different set of syntactic labels, may require re-parsing of the entire text collection or manual re-designing of these processing modules. In order to avoid this, for example, when assessing a new parser and estimating its effect on the performance of the entire system, one may want to use a wrapper around the new parser, that will automatically convert its output into the format of the old parser. With such a wrapper, it is possible to substitute the old parser with the new one without the elaborate process of adjusting various modules of the system to the new syntactic representation. In some cases, when the syntactic representations used by the old and by the new parser are well-defined and clearly documented, and moreover, the differences between the two parsers are systematic, a rule-based conversion mechanism may suffice. In other cases, we may prefer to induce such a wrapper automatically.

Another motivation for the task comes from within computational linguistics itself. There is no *lingua franca* of syntactic formalisms that has been adopted universally. Even if we restrict ourselves to dependency syntax, we come across many different approaches to the formalization of dependency grammars, different analyses of language constructions and different labels of dependency relations (Carroll *et al.*, 2003; Hudson, 1990; Lin, 1998; Mel'cuk, 1988; Nivre and Scholz, 2004; Schneider, 2003; Sleator and Temperley, 1991; Yamada and Matsumoto, 2003). How different are these formalisms? Are the differences systematic? Is it possible to devise a mapping of one type of structure to another? Can this mapping be induced automatically?

Computational linguistics provides yet another type of motivation for our interest in transforming syntactic structures. Today, many syntactic parsers are available for use. How do we choose between them? Comparing two parsers is often a difficult task because of possibly different formalisms. Typically, performance of both parsers would be evaluated against a manually annotated corpus, such as the Penn Treebank (Marcus *et al.*, 1994), or English Parser Evaluation Corpus (Carroll *et al.*, 2003). This, however, requires converting the output of the parsers to the syntactic representation used in the evaluation corpus. Such a conversion is in many cases non-trivial and is usually implemented using theory-specific heuristics and rule-based methods. In particular, automatic head identification using head assignment tables has been used to convert phrase trees to dependency structures (Collins, 1997; Magerman, 1995), with dependency labels derived from phrase labels (we used the same method to obtain a dependency corpus from the Penn Treebank in Section 2.1.2). The dependency-based parser evaluation method of Lin (1998) is based on converting the output of a parser and the gold standard corpus to a single dependency formalism and evaluating the parser using precision and recall of identified dependency relations. Carroll *et al.* (2003) proposed a parser evaluation scheme based on grammatical relations, organized hierarchically and presented a manually annotated corpus of grammatical relations that can be used as a gold standard. In order to use these evaluation schemes for a given parser, one need to devise complex mappings between the labels produced or derived from a parser to the relation or dependency labels used in the gold standard corpus (Bohnet, 2003; Kübler and Telljohann, 2002).

Several methods have been described for automatically transforming syntactic structures in the opposite direction: from dependency structures to phrase trees (Collins *et al.*, 1999; Covington, 1994; Xia and Palmer, 2001). Because of the complex nature of the task, these methods are theory-dependent and also use heuristics such as projection tables (specifying labels of possible projection phrases for each word category) and, in some cases, explicit argument-modifier distinction.

In this section of the thesis we address a different task than the methods mentioned above: converting between two different syntactic dependency formalisms.

Syntactic dependency formalisms may differ at various levels of the grammar. The easiest mismatches are systematic differences in dependency labels: e.g., NP-SBJ is used by Nivre and Scholz (2004) to denote subject dependencies, while S|NP-SBJ is used in our dependency version of the Penn Treebank (Section 2.1.2), ncsubj in the relational evaluation scheme of Briscoe *et al.* (2002) and subj by the parser of Lin (1994).

A more complicated though systematic mismatch is due to the differing granularity of the dependency labels. E.g., while in our dependency version of the Penn Treebank we distinguish types of adjuncts (temporal -TMP, locative -LOC, manner -MNR, purpose -PRP, etc.), other dependency formalisms do not make this distinction. Moreover, the standard conversion of the phrase trees produced, e.g., by the parser of Charniak (2000) to dependency structures, as we describe in Section 2.1.2, results in dependency labels that do not allow us to distinguish arguments and modifiers, making a meaningful comparison even harder.

Yet more complex mismatches between syntactic formalisms occur when they produce different analyses of constructions like control, clausal complements and relative clauses. Some of the formalisms (e.g., Lin's parser and Carroll et al.'s evaluation scheme) explicitly include non-local dependencies, while others ignore them.

7.2 Related work

The issue of automatically converting between different syntactic formalisms has not received much attention yet in the NLP community. Often, when the need for such conversion arises, some task-specific rule-based heuristics are used. E.g., Collins (1999) describes heuristics for converting phrase structure parses into labeled dependency graphs (we describe an extension of these heuristics in Section 2.1.2). Jijkoun and de Rijke (2004) report on a conversion between two different dependency formalisms, one based on Penn Treebank labels and another based on grammatical relations of Carroll *et al.* (2003), using a system with 40 rules. Such task-specific rule-based approaches are often difficult to implement due to the (possibly) differing degrees of granularity of different syntactic analyses. E.g., the syntactic analysis provided the Penn Treebank distinguishes various types of modifiers (temporal, manner, etc.), while the dependency parser Minipar (Lin, 1994) does not make this distinction. It is not obvious how to map the structures provided by the latter into Penn Treebank-like annotations using a simple set of rules.

Automatic transformations of syntactic trees and graphs have previously been used for the recovery of non-local dependencies, the task we also addressed in Chapters 4 and 6. Johnson (2002) was the first to present a method for adding empty nodes and finding possible antecedents in parse trees. Johnson's method

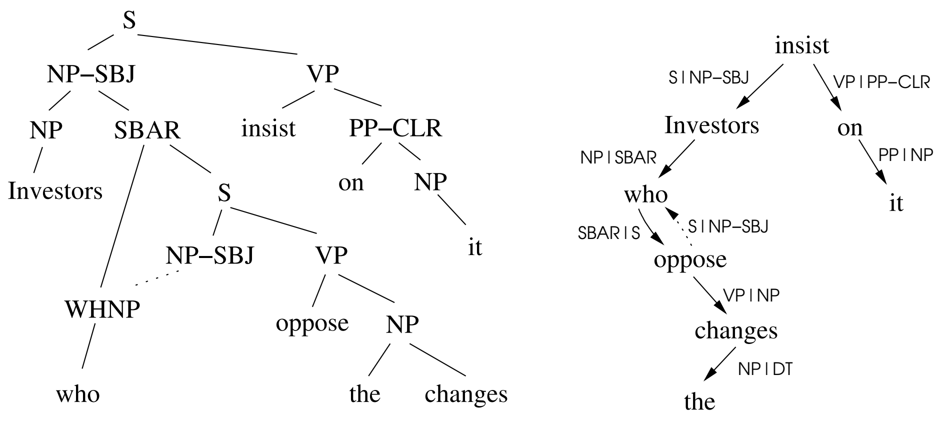


Figure 7.1: (Left): Penn Treebank annotation of the sentence "*Investors who oppose the changes insist on it.*" (Right): the result of its conversion to a dependency graph. Dotted edge corresponds to the co-indexed trace in the Penn Treebank annotation.

works by identifying in the training corpus which local contexts require the insertion of empty nodes and which ones license non-local dependencies as they occur in control or wh-extraction. Later, Jijkoun and de Rijke (2004) described a machine learning method for a similar task of identifying non-local relations in dependency graphs, where it is the task of a machine learner to identify licensing contexts.

7.3 Problem description

In this chapter we address a problem of automatically converting between different syntactic dependency formalisms. Phrased at an abstract level, given a parser P that produces analyses in the source formalism, and a corpus C that provides sample analyses of a number of sentences in the target formalism, our task is to build a transformation function that maps structures provided by P into those annotated in C.

In order to make the task concrete, we consider a specific instance of the general problem outlined above: for the parser P, we take the wide-coverage dependency parser Minipar (Lin, 1994) and our gold standard corpus consist of dependency structures derived from the Penn Treebank as described in Section 2.1.2.

Figure 7.1 shows an example of the original Penn Treebank phrase tree (with function tags, empty nodes and non-local dependencies) and the derived dependency graph for the sentence "*Investors who oppose the changes insist on it.*" La-

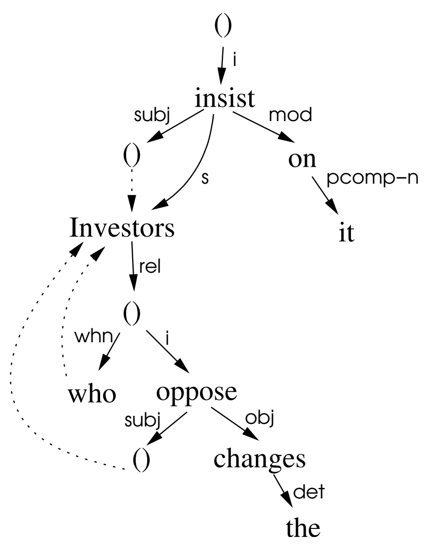


Figure 7.2: The output of Minipar for the sentence "*Investors who oppose the changes insist on it.*" Dotted edges correspond to antecedents as given by Minipar.

bels of the dependency relations are obtained by concatenating phrase labels of corresponding constituents. Figure 7.2 (top) shows the output of Minipar for the same sentence.

Let's take a close look at the differences between the dependency graphs in Figures 7.1 and 7.2. First of all, some of the differences are simply due to different dependency labels, as summarized in Table 7.1. To further illustrate the differences in dependency labels, Table 7.2 describes the most frequent dependency labels in our dependency version of the Penn Treebank, and Table 7.3 does the same for Minipar's output. The frequencies are calculated using sections 00–01 of the Penn Treebank.

Different dependency labels and their different granularity, however, are not the only possible mismatches between the two formalisms. Often Minipar identifies empty nodes (labeled with "()" in Figure 7.2) in positions where no empty nodes exist in the dependency version of the Penn Treebank. In particular, Minipar inserts an empty node with the category C (clause) as a root of the sentence, attaching the main verb "*insist*" as an i-dependent of this empty node. Another empty node of the category C is inserted as the root of the relative clause "*who oppose the changes*." Two other empty nodes of the category N are inserted as subj-dependents of the verb of the two clauses of the sentence.

Probably, the linguistically most interesting difference for this example is the analysis of the WH-extraction in the two formalisms. In the dependency version of

Penn TB label	Minipar's label	Description
S|NP-SBJ	s	surface subject
VP|PP-CLR	mod	prepositional modifier
PP|NP	pcomp-n	nominal complement of a preposition
VP|NP	obj	direct object of a verb
NP|DT	det	determiner modifying a noun

Table 7.1: Some of the correspondences between Minipar's dependency labels and labels derived from the conversion of the Penn Treebank to a dependency corpus.

the Penn Treebank, following the original phrase structure annotation, the extracted subject is co-indexed with the relative pronoun, e.g., *who* is the non-local subject of *oppose* in Figure 7.1. On the other hand, in Minipar's analysis the subject is co-indexed with the noun modified by the relative clause, e.g., *Investors* is the subject of *oppose* in Figure 7.2. Moreover, while in the Penn Treebank version we choose the relative pronoun to be the head of SBAR and therefore the immediate NP|SBAR-dependent of the modified noun, Minipar adds a separate empty node as a rel-dependent.

In principle it is possible to manually create a list of graph rewrite rules that try to account for such systematic differences in the analysis of various constrictions, though this list is likely to be very long and transformations are likely to be far from trivial. Moreover, the creation of such a list has to be example-driven, since neither dependency formalism provides a full description of its analysis of all constructions. The task we address in this chapter of the thesis can be formulated as creating such a list of rewrite rules automatically, applying our general framework for learning graph transformations described in Chapter 5. In the next section we define this task in detail.

7.4 Experimental setting

We use the graph transformation method described in Chapter 5 to learn graph rewrite rules that would allow us to convert graphs produced by Minipar into the dependency formalism derived from the Penn Treebank annotation.

As in the applications of the method to the predicate argument identification problem (Chapter 6), we take the dependency version of the Penn Treebank as the corpus of output graphs, and Minipar's parses of the strings of the Penn Treebank as the corpus of input graphs.

We split each of the input and output corpora in a standard way into develop-

Penn TB label	Frequency	Description
All	98742	total dependency relations in sections 00–01
S|NP-SBJ	9460	subject
PP|NP	8487	nominal complement of a preposition
NP|DT	7877	determiner modifying a noun
NP|NNP	5045	proper noun modifying a noun (e.g., apposition)
VP|NP	5037	object (direct or indirect)
NP|JJ	4652	adjective modifying a noun
S|.	3942	punctuation mark: period
NP|PP	3629	prepositional phrase modifying a noun
SBAR|S	3236	relative clause
NP|NN	3039	noun-noun modifier
NP|,	2273	comma at an NP level
S|,	1718	comma at sentence level
NP|NP	1682	noun-noun modifier
VP|TO	1322	*to* of an infinitive
NP|CD	1272	number modifying a noun
VP|PP-CLR	1223	prepositional argument of a verb
VP|SBAR	1192	verb complement: a clause
VP|S	1172	verb complement: control or raising
NP|CC	1118	conjunction between nouns
aux	1078	auxiliary verb attached to a main verb
NP|SBAR	1009	relative clause or clausal compl. of an NP
VP|MD	947	modal verb modifying a main verb
VP|VP	841	past participle in passive
VP|CC	808	conjunction between verbs
COORD|NP	776	NP conjunct
S|CC	773	conjunction between clauses
NP|PRP$	758	personal possessive pronoun modifying a noun
VP|PP	708	prepositional phrase modifying a verb
VP|,	701	comma at a VP level
QP|CD	670	number in a quantifier phrase
NP|-NONE-	650	empty unit in an NP
NP|PP-LOC	608	locative PP modifying a noun
VP|ADJP-PRD	522	adjectival predicate of copular verbs
VP|PP-TMP	515	temporal PP modifying a verb
NP|ADJP	510	complex adjectival phrase modifying a noun
VP|NP-PRD	498	noun predicate of copular verbs
NP|VP	460	reduced relative clauses
QP|$	449	currency sign $ in quantifier phrase
VP|RB	413	adverb modifying a verb
VP|PP-LOC	409	locative PP modifying a verb

Table 7.2: Forty most frequent dependency labels, out of total 618, in our dependency version of sections 00–01 of the Penn Treebank.

Minipar label	Frequency	Description
All	112053	total dependency relations in sections 00–01
mod	14611	adjunct modifier
lex-mod	12322	part of a proper name or a compound
antecedent	11662	antecedent of an empty node
i	8657	relation between a clause and its inflectional phrase
subj	8635	subject of a verb
s	8346	surface subject
pcomp-n	8327	nominal complement of a preposition
det	7562	determined modifying a noun
obj	5515	direct object
Empty	3438	used to connect fragments of a parse
nn	3120	mainly noun-noun compound
aux	2131	auxiliary verb
conj	1953	conjunct
fc	1432	clausal complement
gen	1330	modification by a possessive (*his*, *Jane's*)
pred	1219	predicate of a copular verb
be	1014	*be* in progressive tense
rel	937	relative clause
comp1	886	complement
amod	806	adverb modifying a verb
guest	737	adjunct modifier (similar to mod)
poss	634	relation between a noun and an *'s* in possessive
c	632	complementizer (*that*)
appo	579	apposition
whn	540	WH-pronoun of a clause (*what*, *who*)
have	531	auxiliary *have*
title	494	title (*Mr.*, *Mrs.*)
post	347	special modifiers (*fi rst*, *most*)
pcomp-c	320	clausal complement of a preposition
sc	276	small clause (e.g., in control)
vrel	234	passive verb modifier of nouns
obj2	222	second object of ditransitive verbs
pre	197	special modifiers (*such*, *only*)
num	186	number modifying a noun (*2 copies*)
lex-dep	150	mainly conjuncts
desc	150	some predicates (*grew popular*, *get burned*)
wha	148	WH-pronoun of a clause (*when*, *how*)
by-subj	146	by-PP, a logical subject in passive
pnmod	128	post-nominal modifier

Table 7.3: Forty most frequent dependency labels, out of total 94, in the dependency graphs produced by Minipar on the strings of sections 00–01 of the Penn Treebank.

ment (sections 00–01 of the treebank), training (sections 02–21), and test (section 23) corpora. We will use the training corpus to extract transformation patterns and to learn transformation rules, the development corpus to guide the learning method, and the test corpus to evaluate the resulting automatic graph conversion system.

For the evaluation of our graph transformation method on this task we take the standard dependency-based approach (Lin, 1998), calculating the precision and recall of labeled word-word relations. More precisely, in order to compare a graph produced by Minipar and transformed by the system to the gold standard, we merge (Section 3.5.2) the two graphs, aligning and identifying corresponding words, and consider as correct only labeled edges present in both system's output and the gold standard.

We ran iterations of our method, as in the earlier applications, at each iteration considering at most 20 most frequent left-hand sides of possible rewrite rules, and for each left-hand side at most 20 most frequent possible right-hand sides. After each iteration we evaluated the system using the development corpus (sections 00–01 of the Penn Treebank), calculating the F_1 score of precision and recall of labeled dependencies. As before, we iterated the method until the improvement of the F_1 score was not larger than 0.1, and arbitrary chosen threshold. With this termination criterion, the method stopped after the 18-th iteration and we then used the test corpus to evaluate the performance of the final system.

7.5 Results

Table 7.4 shows the final evaluation results of our transformation system at different iterations of the learning method, on the test corpus (section 23 of the Penn Treebank). Remember that during the learning of graph transformations from the training corpus, we use the development corpus to estimate the accuracy of the learned transformations and determine whether the termination criterion is met.

Given that Minipar and the Penn Treebank-derived dependency corpus provide very different analyses of sentences, the final evaluation results for precision/recall (72/61 on labeled dependencies, 78/66 on unlabeled dependencies) seem interesting, although it is difficult to interpret the scores in the absence of a baseline for this task.

Nevertheless, we can directly compare the evaluation results to the performance of the system described in Section 6.3, where we described a system that learns to transform the output of the parser of Charniak (2000) to the same dependency version of the Penn Treebank. Table 7.5 summarizes the performance with the two parsers.

Iterations	Labeled		Unlabeled	
	P	R	P	R
Initial	0.1	0.1	43.4	49.3
1	43.2	38.7	55.4	49.6
2	52.2	46.2	62.5	55.4
3	55.7	50.9	63.9	58.4
4	61.1	53.9	68.1	60.0
5	64.3	55.8	71.1	61.7
6	66.1	57.0	72.2	62.3
7	67.1	57.7	73.1	62.9
8	67.9	58.2	73.8	63.3
9	68.4	58.5	74.2	63.5
10	68.7	58.8	74.5	63.7
11	69.2	59.1	75.0	64.0
12	69.5	59.3	75.2	64.2
13	69.8	59.6	75.6	64.6
14	70.0	59.8	75.8	64.7
15	70.3	60.0	76.0	64.9
16	70.4	60.0	76.2	65.0
17	71.2	60.7	77.1	65.6
18	71.8	60.6	77.6	65.6

Table 7.4: Evaluation results: transforming Minipar's output to the Penn Treebank-derived dependency formalism.

The final results of the graph transformation method on the output of Charniak's parser are substantially better than with Minipar. First, Charniak's parser produces syntactic analysis much closer to the Penn Treebank that the analysis of Minipar, which makes our transformation task harder. Second, Minipar is not a statistical parser, and presumably produces less accurate syntactic structure. However, it is difficult to determine which factor might account for most errors.

7.6 Extracted transformations: a close look

During 18 iterations, our transformation learning method identified a total of 4984 graph rewrite rules with 360 different left-hand sides. Below is an overview of the extracted rules:

- 34% of all rewrite rules are edge attribute modifications; these are modifica-

	Labeled		Unlabeled	
Parser	P	R	P	R
Charniak's parser, see Section 6.3	84.7	83.7	89.6	88.6
Minipar, this chapter	71.8	60.6	77.6	65.6

Table 7.5: Comparing the two systems for transforming the parsers' output into the dependency version of the Penn Treebank: precision and recall of the dependency relations.

tions of dependency labels;

- 16% of all rules are node attribute modifications; all these modification are corrections of part-of-speech tags (pos attribute of word nodes);
- 26% of all rules contain exactly three nodes on the left-hand side;
- 5% of all rules contain four or more nodes on the left-hand side;
- 8% of all rules involve Minipar's antecedent edges or Penn Treebank traces and empty nodes.

Let us also have a look at the performance of our method for the example in Figure 7.2: the sentence *Investors who oppose the changes insist on it*. Figure 7.3 shows three dependency graphs:

(a) the output of Minipar,

(b) the result of its transformation by our system, and

(c) the gold standard Penn Treebank dependency graph.

Our graph transformation method is capable of producing a dependency graph that is very close to the gold standard annotation. The only differences between the graph (b) and the gold standard graph (c) in Figure 7.3 are the following:

- the dependency label between *insist* and the PP *on it* is VP|PP rather than VP|PP-CLR, i.e., the labels lacks the Penn Treebank tag -CLR ("closely related", used to mark constituents that *occupy some middle ground between argument and adjunct* (Bies *et al.*, 1995));
- an empty node with label "*" and the dependency label VP|NP was erroneously inserted as a dependent of *insist*. This error was caused by our system misinterpreting *insist* as a head of a reduced relative clause (cf. *the company based * in Los Angeles*).

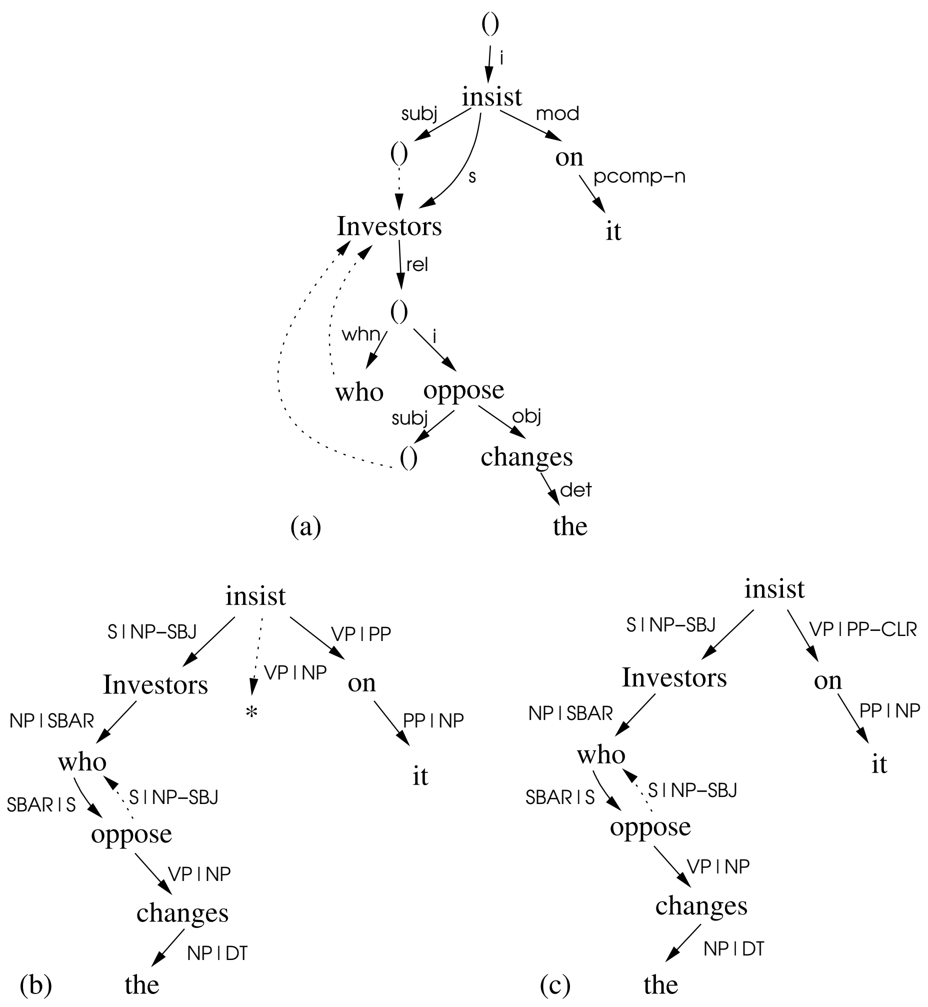


Figure 7.3: Three dependency analyses for the sentence *Investors who oppose the changes insist on it*: (a) Minipar's output, (b) Minipar's output transformed by our system, and (c) dependency graph derived from the Penn Treebank.

In total, 16 graph rewrite rules learned by the system were applied to the graph in Figure 7.3(a). We describe these rules below in the application order:

- First, 6 rewrite rules of the following form are applied to change part-of-speech tags of Minipar to the Penn Treebank tag set:

$$LHS = \{\, \mathsf{node}(n), \mathsf{attr}(n, \mathsf{type}, \mathsf{word}), \mathsf{attr}(n, \mathsf{pos}, T_1) \,\}$$
$$RHS = \{\, \mathsf{node}(n), \mathsf{attr}(n, \mathsf{type}, \mathsf{word}), \mathsf{attr}(n, \mathsf{pos}, T_2) \,\}$$

  Specifically,

  - for the word *Investors*, the tag was changed from N to NNP (i.e., the above rule with $T_1 = \mathsf{N}$ and $T_2 = \mathsf{NNP}$ was applied with n mapped to the word node *Investors*);
  - for the word *opposed*, V was changed to VBG;
  - for the word *the*, Det was changed to DT;
  - for the word *changes*, N was changed to NN;
  - for the word *insist*, V was changed to VBN;
  - for the word *on*, Prep was changed to IN;
  - for the word *it*, N was changed to NN;

  Note that the tags NNP, VBG, NN and VBN were assigned incorrectly.

- Next, five more rules of the following form change edge labels:

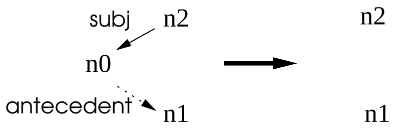
$$\begin{aligned} LHS = \{\, & \mathsf{node}(n_0), \mathsf{node}(n_1), \mathsf{attr}(n_0, \mathsf{type}, \mathsf{word}), \mathsf{attr}(n_1, \mathsf{type}, \mathsf{word}), \\ & \mathsf{edge}(e_0, n_0, n_1), \mathsf{attr}(e_0, \mathsf{type}, \mathsf{dep}), \mathsf{attr}(e_0, \mathsf{label}, T_1) \,\} \\ RHS = \{\, & \mathsf{node}(n_0), \mathsf{node}(n_1), \mathsf{attr}(n_0, \mathsf{type}, \mathsf{word}), \mathsf{attr}(n_1, \mathsf{type}, \mathsf{word}), \\ & \mathsf{edge}(e_0, n_0, n_1), \mathsf{attr}(e_0, \mathsf{type}, \mathsf{dep}), \mathsf{attr}(e_0, \mathsf{label}, T_2) \,\} \end{aligned}$$

  Specifically,

  - the label of the edge from *insist* to *on* is changed from mod to VP|PP (i.e., the above rule with $T_1 = \mathsf{mod}$ and $T_2 = \mathsf{VP|PP}$ is applied);
  - for the edge from *on* to *it*, changed from pcomp-n to PP|NP;
  - for the edge from *changes* to *the*, changed from det to NP|DT;
  - for the edge from *insist* to *Investors*, changed from s to S|NP-SBJ;
  - for the edge from *oppose* to *changes*, changed from obj to VP|NP;

- The next rule, applied twice, removed two empty nodes, the subject of *oppose* and the subject of *insist*, together with their incident edges:

$$\begin{aligned} LHS = \{ & \text{node}(n_0), \text{node}(n_1), \text{node}(n_2), \\ & \text{attr}(n_0, \text{type}, \text{word}), \text{attr}(n_1, \text{type}, \text{word}), \text{attr}(n_2, \text{type}, \text{word}), \\ & \text{edge}(e_0, n_0, n_1), \text{attr}(e_0, \text{type}, \text{dep}), \text{attr}(e_0, \text{label}, \text{antecedent}), \\ & \text{edge}(e_1, n_2, n_0), \text{attr}(e_0, \text{type}, \text{dep}), \text{attr}(e_0, \text{label}, \text{subj}) \} \\ RHS = \{ & \text{node}(n_1), \text{node}(n_2), \\ & \text{attr}(n_1, \text{type}, \text{word}), \text{attr}(n_2, \text{type}, \text{word}) \} \end{aligned}$$



The rule was applied to the occurrences of its left-hand side, where

- n_1 is mapped to *insist*, n_2 to *Investors*,
- n_1 is mapped to *oppose*, n_2 to *Investors*,

and n_0 to the corresponding () nodes.

- The next rule removes the empty node, the root of the Minipar's dependency tree, together with the incident edge labeled i, connecting () and *insist*:

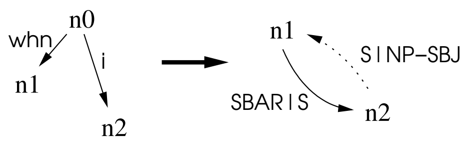
$$\begin{aligned} LHS = \{ & \text{node}(n_0), \text{node}(n_1), \text{attr}(n_0, \text{type}, \text{word}), \text{attr}(n_1, \text{type}, \text{word}), \\ & \text{edge}(e_0, n_0, n_1), \text{attr}(e_0, \text{type}, \text{dep}), \text{attr}(e_0, \text{label}, \text{i}) \} \\ RHS = \{ & \text{node}(n_1), \text{attr}(n_1, \text{type}, \text{word}) \} \end{aligned}$$

- The next rule adds a Penn Treebank empty node * as a dependent of *insist*:

$$\begin{aligned} LHS = \{ & \text{node}(n_0), \text{attr}(n_0, \text{type}, \text{word}) \} \\ RHS = \{ & \text{node}(n_0), \text{attr}(n_0, \text{type}, \text{word}), \\ & \text{node}(n_1), \text{attr}(n_1, \text{type}, \text{word}), \\ & \text{attr}(n_1, \text{empty}, 1), \text{attr}(n_1, \text{pos}, \text{-NONE-}), \\ & \text{edge}(e_0, n_0, n_1), \text{attr}(e_0, \text{type}, \text{dep}), \text{attr}(e_0, \text{label}, \text{VP|NP}) \} \end{aligned}$$

- The next rule changes the annotation of the relative clause, removing Minipar's empty node with its incident edges, and making the WH-pronoun *who* a non-local subject of the verb *oppose* and the head of the relative clause:

$$\begin{aligned} LHS = \{ & \textsf{node}(n_0), \textsf{node}(n_1), \textsf{node}(n_2), \\ & \textsf{attr}(n_0, \textsf{type}, \textsf{word}), \textsf{attr}(n_1, \textsf{type}, \textsf{word}), \textsf{attr}(n_2, \textsf{type}, \textsf{word}), \\ & \textsf{edge}(e_0, n_0, n_1), \textsf{attr}(e_0, \textsf{type}, \textsf{dep}), \textsf{attr}(e_0, \textsf{label}, \textsf{whn}), \\ & \textsf{edge}(e_1, n_0, n_2), \textsf{attr}(e_0, \textsf{type}, \textsf{dep}), \textsf{attr}(e_0, \textsf{label}, \textsf{i}) \} \\ RHS = \{ & \textsf{node}(n_1), \textsf{node}(n_2), \\ & \textsf{attr}(n_1, \textsf{type}, \textsf{word}), \textsf{attr}(n_2, \textsf{type}, \textsf{word}), \\ & \textsf{edge}(e_2, n_1, n_2), \textsf{attr}(e_2, \textsf{type}, \textsf{dep}), \textsf{attr}(e_2, \textsf{label}, \textsf{SBAR|S}), \\ & \textsf{edge}(e_3, n_2, n_1), \textsf{attr}(e_3, \textsf{type}, \textsf{dep}), \textsf{attr}(e_3, \textsf{label}, \textsf{S|NP-SBJ}) \} \end{aligned}$$



- Finally, the next rule attaches the pronoun *who* that heads the relative clause, to the modified noun *Investors*, making use of the remaining Minipar's antecedent edge from *who* to *investors*, in fact, simply changing the label of the edge connecting *who* and *Investors* from antecedent to NP|SBAR:

$$\begin{aligned} LHS = \{ & \textsf{node}(n_0), \textsf{node}(n_1), \textsf{attr}(n_0, \textsf{type}, \textsf{word}), \textsf{attr}(n_1, \textsf{type}, \textsf{word}), \\ & \textsf{edge}(e_0, n_0, n_1), \textsf{attr}(e_0, \textsf{type}, \textsf{dep}), \textsf{attr}(e_0, \textsf{label}, \textsf{antecedent}) \} \\ RHS = \{ & \textsf{node}(n_0), \textsf{node}(n_1), \textsf{attr}(n_0, \textsf{type}, \textsf{word}), \textsf{attr}(n_1, \textsf{type}, \textsf{word}), \\ & \textsf{edge}(e_0, n_0, n_1), \textsf{attr}(e_0, \textsf{type}, \textsf{dep}), \textsf{attr}(e_0, \textsf{label}, \textsf{NP|SBAR}) \} \end{aligned}$$

To give an overview of the transformations for this example, Figure 7.4 shows the graph at different stages of the transformation (changes only in node attributes omitted).

7.7 Conclusions and future work

In this chapter we described yet another application of our method for learning graph transformations, to the task of automatic conversion between different syntactic dependency formalisms. We used our method to learn a sequence of transformations that allows us to convert the output of Minipar, a wide-coverage dependency parser for English, to the kind of structures used in a dependency version of the Penn Treebank.

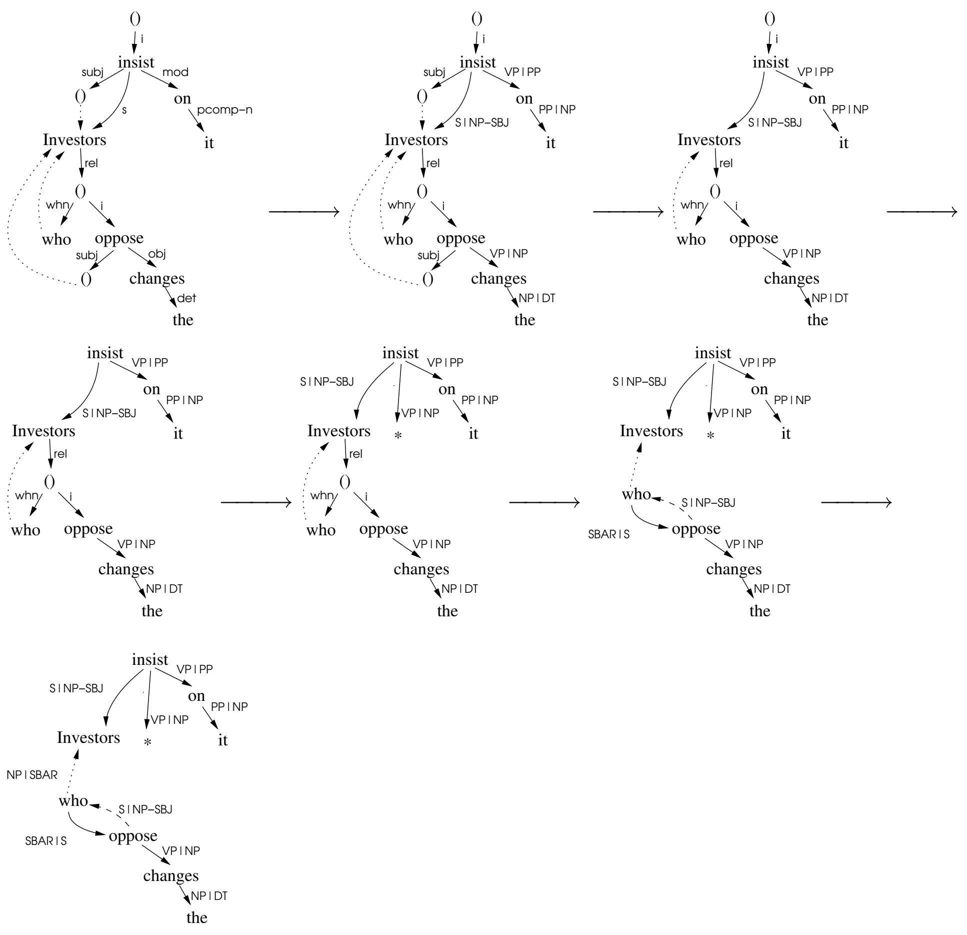


Figure 7.4: Transforming Minipar's output for the sentence *Investors who oppose the changes insist on it* into Penn Treebank dependency graph, step by step.

Although the two dependency formalisms are quite different, our system, used as a post-processor for Minipar, achieved labeled precision 70.5 and labeled recall 60.0 for dependency relations, evaluated using a dependency version of the Penn Treebank. These numbers are modest compared to the scores obtained by a similar system applied to the output of the Charniak's parser (precision and recall 84.1, see Chapter 6).

An interesting aspect of our transformation method is that, similar to Transformation Based Learning, the rewrite rules extracted by the method, or more precisely, their left- and right-hand sides can be interpreted linguistically, as we have seen for the example in the previous section.[1]

At this point in the thesis, we have already presented our general graph transformation method in Chapter 5 and described two applications of the method to concrete NLP problems: the identification of the predicate argument structure in the output of a syntactic parser (in Chapter 6) and conversion between two different syntactic dependency formalisms (in the present chapter).

In the next chapter we will demonstrate that our general method can also be used to identify semantic roles in English sentences.

[1]It is difficult to support a claim of this kind objectively, but we suggest interested readers to examine the extracted patterns on the Web: `http://staff.science.uva.nl/~jijkoun/thesis/patterns`.

Chapter 8

Identifying Semantic Roles

In this thesis we have repeatedly argued that many Natural Language Processing tasks can be viewed as instances of a general graph transformation problem: transforming graphs of one type into graphs of another type. In Chapter 2 we gave an overview of different representations of linguistic structures as directed labeled graphs and described how some NLP tasks can be formulated as graph transformation problems. Then, in Chapter 5 we presented a general method for learning graph transformations, given a corpus of transformation instances (pairs of input and output graphs). In Chapters 6 and 7 we described applications of this general method to concrete NLP problems:

- predicate argument structure (PAS) identification in dependency graphs,
- PAS identification in phrase trees, and
- transformation of one syntactic dependency formalism into another.

In the present chapter we describe yet another application of our graph transformation method: to the tasks of identification of PropBank (Kingsbury *et al.*, 2002) and FrameNet (Johnson *et al.*, 2003) semantic roles in English sentences. We will demonstrate possible ways of formulating these NLP tasks as graph transformation problems, and show how our general learning method performs on these tasks.

In particular, we will argue that a graph-based view on the annotations of PropBank and FrameNet allows for a very precise and lossless encoding of the information in the corpora. We will show that the graph transformation method is applicable to the task if identification of PropBank semantic roles. When applying the method to FrameNet, however, we will find that it is not well suited for FrameNet-based shallow parsing, whereas a more ad-hoc task-specific graph transformation method (similar to the task-specific method of Chapter 4 for the predicate argument structure identification) gives a state-of-the-art performance. This will raise important issue concerning the default parameters of the graph transformation method, such as criteria for the selection of rewrite rules.

8.1 Background and motivation

Semantic analysis of text is one of the long-term goals of Computation Linguistics. Most NLP researchers agree that extracting deep semantic structures from free text requires (at least, as a first step) identification of semantic predicates and their arguments. With the success of statistical corpus-based methods for syntactic analysis (Bod *et al.*, 2003a; Charniak, 2000; Collins, 1999; Magerman, 1995; Manning, 2003; Sampson, 1986), and with the emergence of semantically annotated corpora such as PropBank (Kingsbury *et al.*, 2002) and FrameNet (Baker *et al.*, 1998), it has become possible to address the task of semantic parsing (or, rather, shallow semantic parsing, as it only focuses on extraction of predicates and their arguments) in a similar, corpus-driven way. The growing number of text processing systems making use of ProbPank and FrameNet data and their high performance are clear indications that the problem is salient and within reach for the current state-of-the-art in NLP (Ahn *et al.*, 2004; Bejan *et al.*, 2004; Gildea and Hockenmaier, 2003; Gildea and Jurafsky, 2002; Gildea and Palmer, 2002; Kwon *et al.*, 2004a; Litkowski, 2004; Palmer *et al.*, 2005; Pradhan *et al.*, 2004; Toutanova *et al.*, 2005).

Put very broadly, the task of shallow semantic parsing, or semantic role identification, consists in detecting and labeling simple predicates: *Who did what to whom, where, when, how, why*, etc. There is no single definition of a universal set of semantic roles and moreover, different NLP applications may require different granularity of role labels. In the present chapter we will examine two semantic parsing tasks with different views on semantic predicates and their arguments: PropBank-based and FrameNet-based semantic parsing.

The Proposition Bank project (Kingsbury *et al.*, 2002; Palmer *et al.*, 2005) aims at creating a corpus with explicit annotations of semantic predicates, without emphasizing any specific semantic theory, but rather providing analyses that are in principle consistent with all theories of argument structure.

The following example from (Palmer *et al.*, 2005) illustrates the PropBank annotation of a sentence with respect to the target verb *bought*:

$[_{\mathrm{Arg0}}$ Chuck$]$ have *bought* $[_{\mathrm{Arg1}}$ a car$]$ $[_{\mathrm{Arg2}}$ from Jerry$]$ $[_{\mathrm{Arg3}}$ for \$1000.$]$

Essentially, PropBank annotates semantic arguments of verbs, without giving them any specific semantic interpretation such as thematic roles *Agent, Patient, Theme,* *Experiencer,* etc., as, for example, in VerbNet (Kipper *et al.*, 2000), but using numbered arguments and preserving the numbering in alternations of syntactic realization with the same verb (Levin, 1993). Nevertheless, generally the argument $Arg0$ corresponds to a prototypical Agent, the argument $Arg1$ to a Patient and a Theme, and moreover, for each specific sense of each verb (a frameset), PropBank defines a mapping from the numbered arguments to thematic roles. For example, the verb *buy* is specified to take up to five thematic arguments:

Arg0: Agent	Arg2: Source	Arg4: Beneficiary
Arg1: Theme	Arg3: Asset	

At the present (summer 2006), PropBank provides semantic annotations for verbs only, with the annotation of nominalizations and other noun predicates in progress.

The FrameNet project (Baker *et al.*, 1998) takes a different approach to annotating semantic arguments. Following the Frame Semantics of Fillmore (1982), it defines a set of semantic frames, a conceptual structures corresponding to particular types of situations, objects or events, such as *Motion* or *Commerce-Good-Transfer*. Each frame specifies a set of possible roles, frame elements, and a set of lexical units, i.e., target words that evoke the frame. FrameNet annotates a corpus of sentences with respect to particular target words, specifying the evoked frame and all frame elements. Below is an example of the FrameNet annotation for the target word *bought* from the *Commerce-buy* semantic frame.

$$[_{\text{Buyer}}\ \text{Chuck}]\ \text{have}\ \textit{bought}\ [_{\text{Goods}}\ \text{a car}]\ [_{\text{Seller}}\ \text{from Jerry}]\ [_{\text{Payment}}\ \text{for \$1000.}]$$

In general, for the frame *Commerce-buy*, FrameNet specifies 2 core frame elements (i.e., conceptually necessary for the frame) and 13 non-core elements. Some of them are listed below:

Buyer (core)	Money	Recipient	...
Goods (core)	Seller	Rate	

FrameNet annotates lexical units realized as verbs, nouns and adjectives.

In this chapter we will consider two variants of the shallow semantic parsing task: PropBank-based and FrameNet-based. We will demonstrate a way of representing semantic role information using graphs, re-formulate the semantic role identification problem as a graph transformation problem and apply our method for learning graph transformations.

8.2 PropBank: an annotated corpus of semantic roles

The goal of the Proposition Bank project is defined in (Kingsbury *et al.*, 2002) as "*adding a layer of predicate-argument information, or semantic role labels, to* *the syntactic structures of the Penn Treebank*." The choice of the set of role labels was guided by earlier research into the linking between semantic arguments of verbs and their syntactic realization, in particular the study of verb classes and alternations by Levin (1993).

PropBank does not aim at cross-verb semantically consistent labeling of arguments, but rather at annotating the different ways arguments of a verb can be realized syntactically in the corpus. Therefore, the architects of the project chose a theory-neutral numbered labels (e.g., *Arg0, Arg1,* etc.), rather than mnemonic names (*Agent, Patient, Theme*); these labels can be mapped easily into any specific theory of argument structure, such as Lexical-Conceptual Structure (Rambow *et al.*, 2003) or Tectogrammatics (Hajičová and Kučerová, 2002). In addition to arguments, PropBank annotates various adjunct-like modifiers of the basic predicate argument structure: temporal, location, manner (e.g., "John ate his peas *yesterday*"), as well as verb-level negation (e.g., "John did*n't* eat his peas") and modal verbs (e.g., "John *would* eat his peas").

PropBank takes a corpus-based approach to the annotation of semantic roles: for all verbs (except copular) of the syntactically annotated sentences of the Wall Street Journal section of the Penn Treebank (Marcus *et al.*, 1994), semantic arguments are marked using references to the syntactic constituents of the Penn Treebank. For the 49,208 syntactically annotated sentences of the Penn Treebank, the PropBank annotated 112,917 verb predicates (2.3 predicates per sentence on average), with the total of 292,815 semantic arguments (2.6 arguments per predicate on average).

While most PropBank semantic argument correspond to separate syntactic phrases as they are annotated in the Penn Treebank, for 49,353 (16.9%) of the annotated arguments, the set of the Penn Treebank phrase tree nodes that comprise the argument consists of more than one. In most cases this violation of the one-to-one mapping between syntactic constituents and semantic arguments is due to the PropBank policy towards Penn Treebank traces: whenever a trace (an empty node in the Penn Treebank annotation) is assigned a semantic roles, the co-indexed constituent (the antecedent of a trace) is also annotated as a part of the semantic argument. Moreover, multi-constituent PropBank arguments occur in the case of split constituents. In total 6.2% of all annotated arguments consist of more that one Penn Treebank non-empty constituent.

The stand-off annotation format of the PropBank directly corresponds to the predicates and their arguments. More specifically, for each predicate PropBank specifies:

- the reference to the verb, the head of the predicate; in most cases, the head consists of a single word, though 2.8% of the heads are phrasal verb constructions (e.g., *set up*, *kick in*) consisting of two words, in which case PropBank lists both;

- the frameset of the predicate: the lemma of the head verb and the identifier of a specific role set for the lemma, the result of disambiguation between different senses of the verb;
- the information about the inflection of the head verb: form (infinitive, gerund, participle, finite), tense (future, past ,present), aspect (perfect, progressive, both perfect and progressive), person (3rd person or other), voice (active or passive);
- for each argument, the name of the argument and references to the Penn Treebank constituents comprising the argument.

We refer to Section 8.4, in particular to Figures 8.1 and 8.1 on page 140 for examples of Proposition Bank annotations.

8.3 Related work

Most systems for PropBank-based semantic analysis described in the literature are based on reducing the problem of semantic role identification to a multi-class classification problem: given a constituent in a syntactic tree (or a chunk identified by a shallow parser), assign it to a specific role with respect to a given verb in the sentence.

The first such system was described in (Gildea, 2001; Gildea and Jurafsky, 2002) for FrameNet-based shallow parsing and it was also applied to the PropBank data (Gildea and Palmer, 2002). The system estimates probabilities of constituents taking semantic roles, based on a number of features such as the constituent label and label of its parent, path in the syntactic tree between the constituent and the verb defining the predicate, lexical heads, etc., combining these features in a back-off tree. Moreover, in order to take into account possible dependencies between different arguments of the same verb, the system also incorporates the prior probability of *sets* of possible arguments, estimated from the training corpus.

Gildea and Hockenmaier (2003) describe a system that makes use of a different, richer syntactic representation: Combinatory Category Grammar (CCG), which is a form of a dependency grammar capable of handling long-distance dependencies. The system uses a similar probabilistic approach, but the features describing constituents are derived from CCG representations provided by a CCG parser (Hockenmaier and Steedman, 2002a).

Chen and Rambow (2003) describe a system that uses yet another syntactic representation extracted from a Tree Adjoining Grammar (TAG). They also use a role assignment method similar to (Gildea and Palmer, 2002), but extract their features from TAG analyses.

The system of Surdeanu *et al.* (2003) uses a feature set similar to (Gildea and Jurafsky, 2002), but applies a decision tree classifier to the role assignment problem, instead of estimating parameters of a probabilistic model.

Pradhan *et al.* (2004) present a system that also uses parse trees, but trains a Support Vector Machine (SVM) classifier that predicts labels of constituents (as SVMs are binary classifiers, each classification task is reduced to a sequence of "one vs. all" classifications). They also use heuristics (e.g., disallowing overlapping constituents as arguments) and add new features like partial tree path between the constituent and the verb, named entities, temporal triggers, verb clusters, labels of constituent's siblings, etc. The system has been extended in Pradhan *et al.* (2005c) to include possible dependencies between arguments of the same word, by training a trigram language model that is used to estimates probabilities of possible sequences of labels of a predicate's arguments. In Pradhan *et al.* (2005a) a similar system is described, one that combines features derived from several syntactic representations: parse trees from the parser of Charniak (2000), analyses from the CCG parser of Hockenmaier and Steedman (2002b), dependency parses from Minipar (Lin, 1994), and the results of the semantic chunker of Hacioglu (2004). The experiments show that the combined system outperforms systems based only on a single syntactic representation.

The system of Toutanova *et al.* (2005) addresses the semantic role labeling task using a discriminative log-linear joint model that incorporates possible complex dependencies between role labels of different constituents. The authors show that such a joint model outperforms local models that assume independence of the role assignment.

8.4 Encoding PropBank using graphs

We translated the PropBank stand-off annotation scheme to a graph representation in a straightforward way, extending the graph representation of the Penn Treebank phrase structures (see Section 3.4 on page 37).

As we mention above, for each predicate PropBank annotates its head and the list of labeled arguments. Both the head and each argument contain a list of references to the corresponding nodes of the Penn Treebank trees. We mirror this structure for each predicate by adding to a corresponding Penn Treebank phrase structure graph the following nodes and edges:

- A *predicate node* with type = propbank and label = pred, labeled also with the attributes inflection and frameset, as defined in the PropBank;
- A *head node* with type = propbank and label = head and an edge with type = propbank from the predicate node to the head node;
- For each PropBank argument, a separate *argument node* with type = propbank and the label attribute specifying the PropBank argument label: ARG0, ARG1, . . . , ARG5 or ARGM. Additionally, the feature attribute specifies the feature of the argument if present in the PropBank annotation: TMP, LOC, DIR, etc., for arguments with the label ARGM, and the preposition (on, to, etc.) for numbered arguments realized as prepositional phrases.
- For each of the head and argument nodes, edges with type = propbank, from the PropBank node to the corresponding phrase and word nodes of the Penn Treebank annotation.

Figure 8.1 shows an example of our encoding of the PropBank annotation on top of the phrase structure graphs of the Penn Treebank: here, the predicate introduced by the verb *join* has five arguments: an agent ARG0, a patient ARG1 and three adjunct-like arguments ARGM of different subtypes. In a more complex example in Figure 8.2, the annotated predicate introduced by the verb *criticized* and having two arguments: a patient ARG1 and temporal adjunct ARGM. Notice that the PropBank specifies four nodes in the phrase structure graph that together constitute the ARG1 argument: two of them are empty nodes (object extraction in passive and WH-extraction in a relative clause), and the other two are phrases *the buyers* and *who*.

By adding the PropBank annotations to the original Penn Treebank phrase structures encoded as graphs, we combine the two corpora, i.e., we create a single corpus of graphs that annotates the sentences with both Penn Treebank and PropBank information. In Section 8.5 we will use this corpus to train a learner that will be able to *add* the PropBank semantic argument information to the Penn Treebank structures automatically.

As we have seen, adding the PropBank annotation to the original Penn Treebank phrase trees using our graph-based encoding is straightforward, since the Proposition Bank explicitly refers to the constituents of the Penn Treebank that serve as semantic role fillers. How do we perform a similar merge of two corpora annotated with syntactic and semantic information if the phrase trees come from a parser and, thus, are noisy?

The main problem with imperfect parse trees is that in the case of misattachments or other parsing errors, semantic arguments no longer correspond to single syntactic constituents. Note, however, that this correspondence is not one-to-one

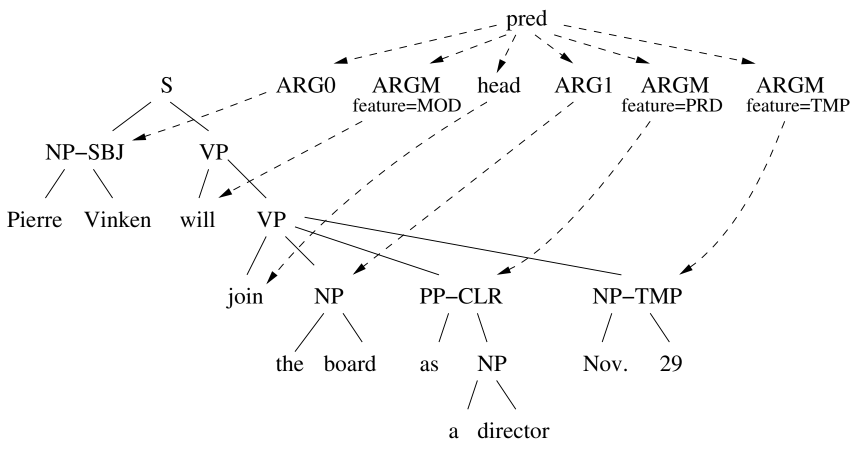


Figure 8.1: Example of an encoding of the PropBank annotation of the Penn Treebank sentence *Pierre Vinken will join the board as a director Nov. 29.* Dashed arrows represent PropBank edges (type=propbank).

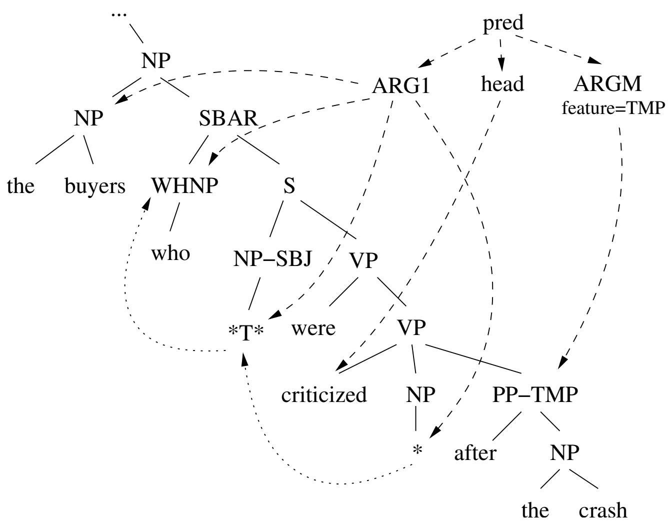


Figure 8.2: Example of an encoding of the PropBank annotation of the Penn Treebank sentence fragment ... *the buyers who were criticized after the crash.* Dotted edges represent Penn Treebank traces (type=antecedent), dashed edges are PropBank edges (type=propbank).

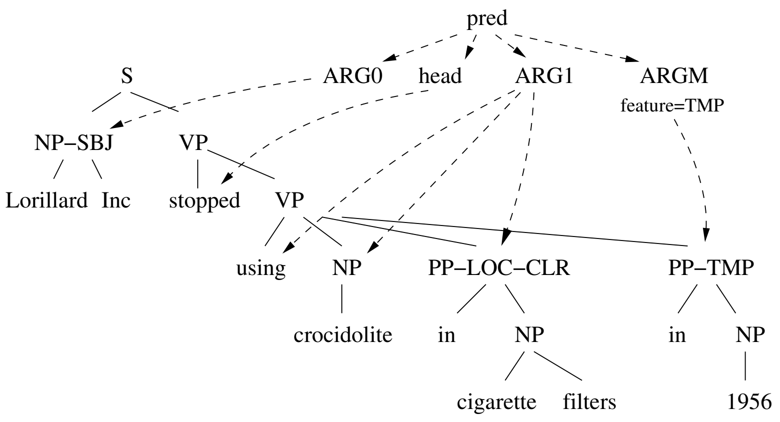


Figure 8.3: The merged PropBank and Penn Treebank annotation of the sentence *Lorillard Inc. stopped using crocidolite in cigarette filters in 1956* for the verb *stopped*. PropBank resolves an incorrect Penn Treebank attachment of the temporal PP *in 1956*.

even in the case of the manual annotation of the Penn Treebank, both due to the specifics of the annotation guidelines (e.g., multiple constituents in relative clause constructions, see example in Figure 8.2) and possible annotation errors in the Penn Treebank itself. Consider the example in Figure 8.3, where the PropBank semantic argument ARG0 is annotated to include three different constituents of the Penn Treebank tree, due to a misattachment of the temporal PP *in 1956* in the syntactic annotation. Interestingly, the Charniak parse of the same sentence (Figure 8.4) attaches the temporal PP correctly, although making a different attachment error.

Note that neither the original PropBank annotation, nor our graph-based encoding of PropBank semantic arguments disallows using multiple constituents per argument. We use this flexibility of our representation, and encode the semantic information precisely as it is annotated in the PropBank with respect to the lexical spans of the annotations. Specifically, when adding a PropBank argument to an imperfect parse tree (the output of the parser), we link the corresponding node to *all maximal constituents* that cover the same word span as the argument. For the example in Figure 8.4, e.g., this gives us two syntactic constituents comprising the argument ARG1 of the verb *stopped*, because according to the PropBank annotation, ARG1 spans the phrase *using crocidolite in cigarette filters*.

Whereas, as we mentioned above, for the manually annotated trees, only 6.2% of the PropBank arguments correspond to more than one syntactic constituent, for the output of Charniak's parser for the Penn Treebank sentences, 11.9% of all argument spans require more than one constituent to cover the span precisely as

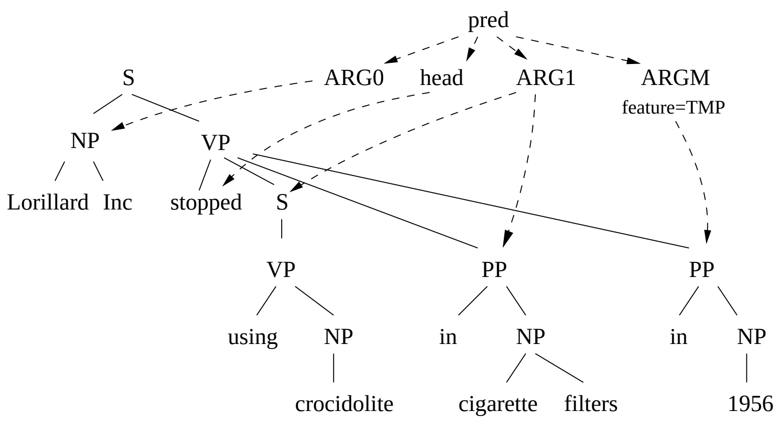


Figure 8.4: The parse tree produced by Charniak's parser, merged with the PropBank annotation of the sentence *Lorillard Inc. stopped using crocidolite in cigarette filters in 1956* for the verb *stopped*. The merged annotation resolves the misattachment of *in cigarette filters*.

specified in the PropBank. Note that this might pose an upper bound on the performance (specifically, on the recall) of some of the semantic parsing methods based on constituent classification (Section 8.3).

In the following section we describe the application of the general graph transformation method to the task of PropBank-based semantic parsing, that uses the corpora with merged syntactic and semantic information for training.

8.5 PropBank-based parsing via graph transformations

For a given sentence, the task of PropBank-based shallow semantic parsing consists in finding semantic arguments of all its verbs, as they are annotated in the PropBank. For example, for the sentence *Lorillard Inc. stopped using crocidolite in cigarette filters in 1956* and the verb *stopped*, the ideal output of a semantic parsing system is

- ARG0: Lorillard Inc.
- ARG1: using crocidolite in cigarette filters
- ARGM-TMP: in 1956

indicating the agent, the theme and the time of the stopping event.

We translate this task into our graph-based framework as follows. Given a phrase structure graph, which can be either a gold standard parse from the Penn Treebank, or an output of an automatic parser, our task is to transform the graph so that it includes semantic annotation (i.e., nodes and edges with type=propbank, as described in Section 8.4 above). Then, for the evaluation and comparison of our system, we read off the resulting semantic annotations from the transformed graphs, taking all descendant words of each resulting semantic argument node as the word span of this argument.

Formulated in this way, the task becomes an instance of a general graph transformation problem. We apply our method for learning graph transformations to this problem, as described below.

Our graph transformation method (Chapter 5) requires sets of input and output graphs for training. We take a corpus of syntactic analyses of sentences as the input corpus, and the result of its merging with PropBank annotations as the output corpus. In Section 8.4 we defined the encoding of semantic arguments in such a way that we can merge it with either gold standard phrase trees from the Penn Treebank, or with the output of an imperfect phrase structure analyzer, the only requirement being that the syntactic analysis contains phrase nodes with identifiable word spans.

Below we report on three experiments with different syntactic corpora as input:

- Penn Treebank: phrase structure graphs with complete Penn Treebank annotation (constituent labels with function tags, empty nodes, traces);
- Charniak: the output of the parser of Charniak (2000) on the strings of the Penn Treebank;
- Charniak+PAS: the output of Charniak's parser after applying the graph transformations for predicate argument structure identification learned as described in Chapter 6 (adding function tags, empty nodes and traces).

The three experiments will allow us to evaluate the effect of imperfect parses and the richer input on the shallow semantic parsing task.

We used a standard split of the Penn Treebank and the Proposition Bank data: sections 02–21 for training, sections 00–01 for development and performance estimation during training, section 23 for testing. We ran the graph transformation method of Chapter 5 as before, separately for each of the three variants of the task. Specifically,

- at each iteration of the method, we extracted 20 most frequent left-hand sides of possible graph rewrite rules, and for each LHS at most 20 most frequent right-hand sizes;

- we used SVM-Light (Joachims, 1999) to learn constraints for each extracted rewrite rule;

- at each iteration, we evaluated the system on the development corpus. We calculated the precision and recall for the identification of PropBank arguments, and terminated the learning process if the improvement of the F_1 score for the current iteration was smaller than 0.1.

8.5.1 Results and analysis

Table 8.1 presents the evaluation results on the test corpus for the three versions of the task: with the Penn Treebank trees, with Charniak's parser and with Charniak's parser post-processed by our system from Chapter 6.

At the time of writing the best published results on the PropBank semantic role labeling using the output of Charniak's parser are the results of Pradhan *et al.* (2005b): precision 80.9, recall 76.8, F_1-score 78.8. Our best results for Charniak's parser are: precision 81.0, recall 70.4 and F_1-score 75.3.

Interestingly, comparison to the state-of-the-art indicates the same situation as was discussed in Section 6.4.4 on page 109: while our system shows good precision, the recall falls behind. Taking into account the iterative nature of our system an imperfect rule selection criteria (we simply take most frequent left-hand sides), we believe that it is the rule selection and learning termination condition that account for relatively low recall values. Indeed, for all of the three tasks in Table 8.1 the learning stops while the recall is still on the rise, albeit very slowly. It seems that a more careful rule selection mechanism is needed to address this issue.

8.5.2 Extracted transformations: a closer look

For Charniak's parser with traces (Charniak+PAS in Table 8.1), during the 11 iterations 220 left-hand sides were extracted and 1444 right-hand sides of rewrite rules. In total, 76% of the LHS's were patterns with 3 to 7 nodes. Interestingly, 55% of the left-hand sides of the extracted patterns included PropBank ARGx nodes, which means that half of the rewrite rules potentially were correcting previous decisions of the system. This is a characteristic property of Transformation-Based Learning and its relatives.

8.6 Experiments with FrameNet

In this section we describe our experiments with FrameNet-based shallow semantic parsing. We will first present a task-specific method, similar to the method for

	Penn Treebank		Charniak		Charniak+PAS	
Iterations	P	R	P	R	P	R
1	90.0	70.7	79.5	58.6	79.9	59.1
2	90.7	76.5	81.2	63.9	81.0	64.2
3	90.7	78.1	81.3	65.6	81.1	65.8
4	90.6	78.9	81.4	66.5	81.2	66.7
5	90.5	80.4	81.4	67.0	81.2	68.3
6	90.4	81.2	81.4	68.3	81.1	68.8
7	90.3	81.9	81.3	68.9	81.0	69.3
8	90.3	82.2	81.3	69.3	81.0	69.8
9	90.3	82.5	81.3	69.6	81.0	70.1
10	90.3	82.8	81.4	69.8	81.0	70.3
11	90.3	83.0	81.3	69.9	81.0	70.4
12	90.3	83.2				

Table 8.1: Identification of semantic arguments using Penn Treebank trees, Charniak's parser and the parser extended with the PAS identification system of Chapter 6.

predicate argument structure identification of Chapter 4 Specifically,

- we will define the task of FrameNet-based parsing using graphs representing dependency structures of sentences;
- we will use a task-specific type of dependency graph transformation, namely, adding edges representing FrameNet semantic roles;
- we will train memory-based classifiers to predict which paths in syntactic dependency graphs correspond to which semantic roles.

Then we will report of experiments with applying the general graph transformation method of Chapter 5 and discuss why the method with the default transformation rule selection criteria is not applicable to FrameNet.

We start with the description of representing the FrameNet data in our graph-based framework.

8.6.1 Encoding FrameNet using dependency graphs

The FrameNet 1.1 corpus (Johnson *et al.*, 2003) includes information about 487 semantic frames, 696 frame elements with distinct names and provides semantic role annotations of 132,968 sentences. Each annotated sentence specifies a single target word, the name of the semantic frame it evokes, and character spans of semantic arguments of the frame. Unlike PropBank, the FrameNet annotation does not use or refer to the syntactic structure of the sentences, but is specified using simple character spans.

We parsed the sentences of the FrameNet corpus using the parser of Charniak (2000), converted the resulting trees to dependency graphs as described in Section 2.1.2 and applied the system of Chapter 4, that identifies Penn Treebank-style predicate argument structure, adding Penn grammatical and semantic function tags, empty nodes and non-local dependencies.

Let G be a dependency graph (see Section 3.4 on page 37). For a node n in G, we define the *span* of n to be the set of all nodes k in G such that there is a directed path from n to k, consisting only of dependency edges, excluding non-local dependencies, i.e., consisting of edges e_i with $\mathsf{attr}(e_i, \mathsf{type}, \mathsf{dep}) \in G$ and $\mathsf{attr}(e_i, \mathsf{trace}, \mathtt{1}) \notin G$. In other words, a span of a node is the set of all its descendants with respect to the local dependency edges. Spans in dependency graphs correspond to constituents in phrase trees, though not for every constituent there is a identical span in the dependency tree of a sentence.

FrameNet annotations of the sentences were merged with the resulting dependency graphs in a way similar to merging PropBank data to phrase trees in Section 8.4. More specifically, for every annotated FrameNet sentence:

- for the target T of the sentence, we identified a node t in the dependency parse of the sentence, such that its span is the smallest span of the graph, containing all words of T;
- similarly, for every annotated semantic role R_i that spans a string P_i in a sentence, we identified a node r_i that spans the minimal set of words containing all words of P_i;
- for every such r_i, we added a new edge e from t to r_i, with $\mathsf{type} = \mathsf{framenet}$ and $\mathsf{label} = R_i$.

Figures 8.5 and 8.6 show examples of the encoding of FrameNet information using extra edges in dependency graphs. Note that for the example in Figures 8.5, the target word *grandmother*, evoking the frame *Kinship*, is also the argument *Alter* of the frame, which results in a loop edge in the graph. For the example in Figure 8.6, the graph contains two parsing errors: first, the phrasal verb '*bring about*', the target of the frame *Causation*, is erroneously analyzed as the verb *bring* with a prepositional phrase '*about an end...*'; second, the phrase *to history* is erroneously attached to *end* instead of *bring*. Nevertheless, our heuristics using spans in dependency graphs allowed us to produce an appropriate encoding of FrameNet roles for this example.

As our learning method, described below, operates with the graph-based encoding of FrameNet roles, we will also have to perform an opposite operation:

Frame:	*Kinship*
Target:	*grandmother*
Ego:	*Your*
Alter:	*grandmother*

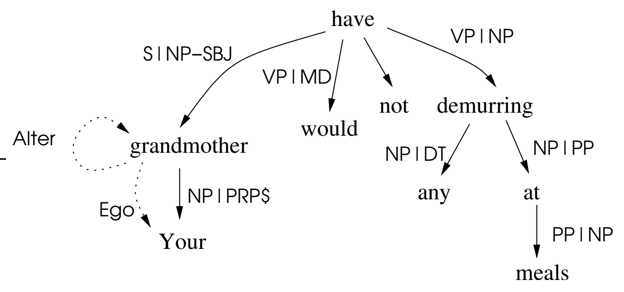


Figure 8.5: Dependency parse of the sentence "*Your grandmother would not have any demurring at meals*" with the FrameNet annotation added for the noun *grandmother* evoking frame *Kinship*. The dotted edges are FrameNet edges (type = framenet).

Frame:	*Causation*
Target:	*bring about*
Cause:	*History*
Effect:	*an end*
Affected:	*to history*

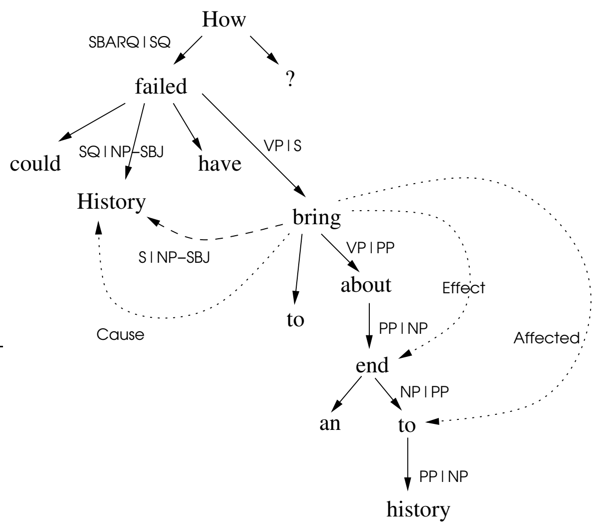


Figure 8.6: Dependency parse of the sentence "*How could History have failed to bring about an end to history?*" with the FrameNet annotation added for the phrasal verb *bring about* evoking frame *Causation*. The dotted edges are FrameNet edges (type = framenet); the dashed edge is a non-local dependency (trace = 1).

given a graph encoding of semantic arguments of a sentence (e.g., in Figure 8.6, right), produce a flat list of character spans (Figure 8.6, left). We implement this in a straightforward way, computing spans of the nodes at the end points of FrameNet edges. If one of the resulting argument spans contains another span or the target, we remove the subsumed span from the subsumer, making all semantic arguments disjoint from each other and from the target. If the subsumer becomes discontinuous after the removal, we leave only the right-most continuous part. The only exception when we allow overlapping spans is when two or more spans are exactly the same: this corresponds to the situation of one span playing more than one role in a semantic frame.

Consider an example graph in Figure 8.6 (right). In order to generate character spans of the arguments, we first compute spans of the corresponding nodes:

$$\begin{aligned} \text{Cause} &= \textit{History}, \\ \text{Effect} &= \textit{an end to history}, \\ \text{Affected} &= \textit{to history}. \end{aligned}$$

Since the Effect argument subsumes the Affected, we remove the subsumed, obtaining the final argument spans:

$$\begin{aligned} \text{Cause} &= \textit{History}, \\ \text{Effect} &= \textit{an end}, \\ \text{Affected} &= \textit{to history}. \end{aligned}$$

Having described our graph-based encoding of FrameNet information, we now reformulate the task of semantic role identification as a graph transformation problem: adding FrameNet edges to dependency graphs.

8.6.2 Learning to add FrameNet edges

We experimented with the data of the Senseval-3 Automatic Labeling of Semantic Roles task (Litkowski, 2004): 24,558 annotated sentences for training and 8,002 test sentences, both corpora from 40 of the 100 most frequent frames in FrameNet. Our system considered all FrameNet edges in the dependency graphs of the training corpus, and for each such edge, it extracted the shortest path between its endpoints: the target word and the word representing a semantic argument. For example, for the graphs in Figures 8.5 and 8.6, the following paths were extracted:

$$\begin{aligned}
\text{Ego} &= \big\{\,\mathsf{node}(t), \mathsf{node}(r), \mathsf{edge}(e_0, t, r), \mathsf{attr}(e_0, \mathsf{label}, \mathsf{NP|PRP\$})\big\}\\
\text{Alter} &= \big\{\,\mathsf{node}(t)\big\}\\
\text{Cause} &= \big\{\,\mathsf{node}(t), \mathsf{node}(r), \mathsf{edge}(e_0, t, r), \mathsf{attr}(e_0, \mathsf{label}, \mathsf{S|NP\text{-}SBJ})\big\}\\
\text{Effect} &= \big\{\,\mathsf{node}(t), \mathsf{node}(n_0), \mathsf{edge}(e_0, t, n_0), \mathsf{attr}(e_0, \mathsf{label}, \mathsf{VP|PP}),\\
&\qquad \mathsf{node}(r), \mathsf{edge}(e_0, n_0, r), \mathsf{attr}(e_0, \mathsf{label}, \mathsf{PP|NP})\big\}\\
\text{Affected} &= \big\{\,\mathsf{node}(t), \mathsf{node}(n_0), \mathsf{edge}(e_0, t, n_0), \mathsf{attr}(e_0, \mathsf{label}, \mathsf{VP|PP}),\\
&\qquad \mathsf{node}(n_1), \mathsf{edge}(e_0, n_0, n_1), \mathsf{attr}(e_0, \mathsf{label}, \mathsf{PP|NP}),\\
&\qquad \mathsf{node}(r), \mathsf{edge}(e_0, n_1, r), \mathsf{attr}(e_0, \mathsf{label}, \mathsf{NP|PP})\big\}
\end{aligned}$$

We then proceeded similarly to step 3 of the graph transformation method of Chapter 4 (see Section 4.7 on page 59). We considered only paths containing 3 or less edges and occurring 10 or more times in the training corpus. For each path, we extracted all its occurrences as a pattern in the training corpus, with the first node of each occurrence (i.e., the node that matched the target node when the pattern was extracted; t in the paths above) only mapped to targets of sentences. For each occurrence of a path, we checked whether the occurrence corresponds to a FrameNet edge, and if so, what the label of the edge is. Moreover, for each occurrence of each path we also extracted the following features:

- the pattern defining the path;
- the name of the semantic frame of the sentence;
- the words of the nodes of the occurrence and their part of speech tags: as we only restrict our method to paths with 3 or less edges, each occurrence contains at most 4 nodes;
- presence of subject and object dependents, for each node of the occurrence;
- for each noun in the occurrence, its semantic class: one of 19 manually defined hypernyms in WordNet (Fellbaum, 1998): *animal, person, property,* etc.

The entire resulting set of path occurrences was used to train a memory-based classifier, whose task was to predict a semantic role label, or lack thereof. As in Chapter 4, we used TiMBL (Daelemans *et al.*, 2003) for classification.

From the testing corpus without FrameNet edges, we extracted all occurrences of the paths starting at the target node of a sentence, along with their features. We applied the trained classifier to the extracted occurrences, and for those positively

	boundary overlap				exact match		
System	P	R	Overlap	Attempted	P	R	F
Bejan *et al.* (2004)	**89.9**	**77.2**	**88.2**	85.9	82.4	71.1	76.3
Here	86.9	75.2	84.7	86.4	73.5	63.6	68.2
Moldovan *et al.* (2004)	80.7	78.0	77.7	**96.7**			
Kwon *et al.* (2004b)	80.2	65.4	78.4	81.5			

Table 8.2: Performance of the four systems with the highest results on the Senseval-3 Semantic Role Labeling task.

classified, added a FrameNet edge from the first to the last node, with the label predicted by the learner.

At the application stage our system takes as input dependency graphs with marked target word and known semantic frame. It extracts occurrences of the paths and uses a classifier to predict which of them correspond to FrameNet semantic roles for a given target word. The result of the system are dependency graphs with added FrameNet edges.

For the evaluation of the system, we compute the final argument spans as described in Section 8.6.1 above.

8.6.3 Evaluation results

We report here the results of our official run at the Senseval-3 Semantic Role Labeling task (Ahn *et al.*, 2004; Litkowski, 2004), created as described above. We only took part in the *Restricted* task, where the systems needed to identify boundaries of semantic arguments, as well as to assign role labels.

The evaluation measures follow the method of Gildea and Jurafsky (2002). A frame element is considered correctly identified if it overlaps with the gold standard span of the argument with the same label. Precision and recall of the correctly identified frame elements are then calculated in a standard way. Another measure is *Overlap*, the average character overlap of the correct frame elements (FEs), calculated as the number of characters overlapping in an FE returned by the system and the gold standard FE divided by the length of the gold standard FE, averaged for all FEs. Finally, the measure *Attempted* is the number of FEs generated by the system divided by the number of FEs in the gold standard.

We also evaluated our run with a more strict measure, where the boundaries of frame elements are required to match the gold standard exactly. The results for our system and the system of Bejan *et al.* (2004) are presented in Table 8.2, columns *exact match*.

8.6.4 Analysis and further experiments

Although our method for FrameNet-based semantic parsing using dependency graphs allows us to achieve high accuracy if partial overlap of FEs with the gold standard is allowed, the system shows a drop of about 12% in both precision and recall with the exact boundary requirement. It is tempting to attribute this drop to parsing errors (e.g., misattachments, which might render an FE incorrect even if we have identified the headword of the FE correctly). Indeed, our evaluation showed that only 85% of FEs in the gold standard correspond to spans that our system can generate, even after applying some simple heuristics to compensate frequent mismatches. However, a much smaller drop in precision and recall for the system of Bejan *et al.* (2004) indicates that this is not necessarily the case. We conjecture that the deterioration of the performance of our system with the "exact match" evaluation can be attributed to our choice of dependency rather than phrase structure syntactic representation. Some of the constituents (namely, those which are not maximal projections) are irrecoverably lost during the conversion to dependency structures. To check whether this is indeed the case, however, requires the development of a similar system based on phrase trees rather than dependency graphs. We leave this for future work.

As we have seen, it is possible to formulate the task of FrameNet-based shallow parsing as a graph transformation problem. However, the method we applied here to this problem is again ad-hoc and task dependent, much in the same way as the method for PAS identification in Chapter 4. Is it possible to apply our general method for learning graph transformation to the task of semantic role labeling?

Unfortunately, our preliminary experiments with the graph transformation method of Chapter 5 gave disappointing results: we ran the method with the phrase trees produced by Charniak's parser and the same split of training and test FrameNet data as for our Senseval experiments described above (we did not make a separate development set for estimate performance through iterations). We stopped the learning cycle after the second iteration since the performance on the test set was very low (precision 76% and recall 33%, for the exact match of frame elements' borders) and was increasing very slowly: 2% improvement of the F_1-score between the first and the second iterations and less than 1% improvement between the second and the third iterations. It seems that our graph transformation method cannot be applied 'as is' to the FrameNet data. But why?

Our hard-wired heuristics for selecting the most frequent rewrite rules at each iteration of the learning cycle and terminating the learning process unless the improvement with the learned rewrite rules is substantial, are appropriate for the Penn Treebank and PropBank: relatively few rewrite rules with many training examples for each. The FrameNet learning task appears to be of an essentially different nature: it seems to require many graph rewrites each of which is 'small' (i.e., comes with few examples). For our generic graph transformation method as defined in Chapter 5, to be more competitive at the FrameNet learning task we would need to explore a two dimensional parameter space: how many LHS candidates of potential rewrites to consider, and for each, how many possible RHS. We leave this exploration as future work, but anecdotal evidence suggests that substantial gains can be had by dropping the heuristics defined in Chapter 5.

8.7 Conclusions

In this chapter we examined two PropBank- and FrameNet-based shallow semantic parsing tasks. We presented different ways of embedding the tasks into our graph framework. For PropBank, we carefully transfered the original annotation into graphs, preserving all of the information in this conversion. Unlike most other approaches to shallow semantic parsing, we did not make any assumptions about one-to-one match between syntactic phrases and semantic elements, but rather preserved exact word spans, allowing more than one constituent (or even separate words) to make up a semantic argument. According to our evaluation, 12% of the PropBank arguments did not match the constituents produced by Charniak's parser. About half of these problematic elements seem to be truly discontinuous (as 6% of the elements do not match gold standard Penn Treebank constituents either), and the other half can be attributed to parsing errors. We believe that such mismatches between annotations should not be ignored or discarded in favor of simpler representations. This is important because the mismatches may be systematic and pertinent to specific language constructions (cf. example in Figure 8.2).

The results of the application of our general graph transformation method to the two tasks of shallow semantic parsing are inconclusive. For the PropBank roles the method shows reasonable precision, but the recall is as low as 70.4. We notice, however, that this was also the case for the application in Chapter 6. We believe this to be an indication that our mechanism for rule selection (hard-wired to select 20 most frequent potential left-hand sides in the training corpus) needs to be revised.

The results are different for FrameNet. While our graph-based but ad-hoc method for semantic role identification demonstrates competitive results, a direct application of the graph transformation method seems to fail, producing very low recall scores. We attribute this to the structure of FrameNet annotations: the corpus consists of a large number of independent frames, thus reasonable rewrite rules might exist but they are not necessarily among the most frequent throughout the iterations.

Chapter 9

Conclusions

In this thesis we presented an approach to natural language processing tasks based on graphs and graph transformations. We considered different types of linguistic structures and demonstrated that they can be straightforwardly and without information loss represented using directed labeled graphs. In particular, we used a graph formalism to represent dependency graphs, phrase trees with empty nodes and non-local dependencies, predicate argument structure annotations of PropBank and semantic frames of FrameNet.

In this final chapter we summarize our main findings, discuss the strengths and weaknesses of our graph transformation-based take on natural language processing tasks, and discuss future work.

9.1 Main findings

We presented a view on several NLP tasks as graph transformation problems and described a method to learn graph transformations from a corpus of graphs. The method builds on ideas of Transformation-Based Learning, extending them to the domain of graphs and complementing simple rewrite rules with machine learners.

We demonstrated and analyzed applications of our graph transformation method to several tasks:

- recovery of Penn Treebank function tags, empty nodes and non-local dependencies in the output of a parser; we applied the method both to dependency structures and phrase trees;

- converting between different syntactic formalisms; we trained a system that converts the output of a dependency parser (Minipar) to dependency formalisms derived from the Penn Treebank II;

- shallow semantic parsing with PropBank and FrameNet.

Turning to the research questions listed in Chapter 1, we answered the first one ("Can a general framework and processing method for NLP problems be developed?") in the affirmative: we proposed a single, unified approach to language processing, that is generic but does allow for task-specific knowledge to be brought in.

In answer to our second research question ("How well is our proposed graph-based framework suited for representing various types of linguistic structures?"), we have shown that our proposed modeling framework in terms of labeled directed graphs is sufficiently flexible and rich to be able to cater for both a broad range of syntactic and semantic features.

As to our third research question ("How well and how natural can different language processing tasks be formulated as graph transformation problems?"), we considered several NLP tasks related to the syntactic and semantic analysis of text at different levels. We demonstrated how a graph-based encoding of linguistic structures makes it possible to re-cast these tasks as instances of a general graph transformation problem. For each of the three tasks listed above, we put our graph transformation-based method to work, achieving state-of-the-art performance in several cases, thus partly addressing our fourth research question ("How well does our general graph transformation-based method for solving language processing tasks perform compared to other, task-specific methods?"). Further answers to this question were offered by comparing both a task-specific and a generic graph transformation-based solution for two of the three tasks listed above: in one case, generality helped to improve the results, on the other it hurt, mainly because of specific heuristics hardwired into our method.

As to the limitations of our graph transformation-based approach to NLP tasks (our fourth research question), one of our main findings was that, with the heuristics currently hardwired into it, the approach is effective for the NLP tasks where there are relatively few potential rewrite rules, each with many examples (such as the recovery of Penn Treebank-style predicate argument structures, converting between syntactic dependency formalisms, and semantic parsing with PropBank), whereas its underlying heuristics need revising to be able to deal effectively with tasks such as FrameNet parsing, where we face many possible rewrite rules, each with a small number of examples.

9.2 Strengths and weaknesses of our approach

Below we summarize strengths and weaknesses of our method identified while experimenting with the NLP tasks mentioned above:

+ A graph-based view on different types of linguistic structure allows us to naturally model various aspects of text and combine different structures of text in a systematic way.

+ Our graph transformation method, being an extension of Transformation-Based Learning, exhibits similar properties, in particular, using a combination of rule-based and statistical machine learning instruments, the method identifies meaningful and linguistically interpretable and justifiable rewrite rules.

+ Our method for learning graph transformations is applicable to different NLP tasks and demonstrates competitive results for some of them even without any task- and data-specific fine-tuning.

– However, our graph transformation method relies on a small but specific set of heuristics, which worked well for a number of tasks, but needs revisiting for other tasks. Specifically, the method in its current version is not capable of effectively addressing the task of FrameNet-based semantic parsing.

– Due to the iterative nature of the method and frequency-based criteria for the selection of rewrite rules, for most of the tasks considered in the thesis, the method demonstrates low recall values.

9.3 Conclusions on the tasks

For the three specific language processing tasks we considered in the thesis (in Chapters 6, 7 and 8), the behavior of our graph transformation method is indicative of the nature of the NLP tasks and the distribution of frequencies of the linguistic phenomena involved in the tasks. Specifically, the first few iterations of our transformation algorithm allow us to account for a substantial number of the most systematic mismatches between input and output graphs. However, during later iterations the rewrite rules detected by the algorithm become less and less frequent and, therefore, the subsequent iterations are less effective. This leads to early termination of the algorithm and low recall. We believe that this behavior is yet another manifestation of the Pareto principle, or the 80–20 rule, stating that, as a rule of thumb, 80% of consequences stem from 20% of causes. A deeper analysis of the performance of the algorithm is needed to confirm this claim and to see whether the frequency distribution of the learned patterns follows Zipf's law (Manning and Schütze, 1999, pages 23–29).

9.4 Further work

In the thesis we presented a novel method and showed it at work. We have answered some of the questions about the method and, quite naturally, many new ones were generated. The three most important, in our opinion, are listed below.

- As our method takes its general architecture from Transformation-Based Learning, it is interesting and informative to study the relation between the two. One specific question is, whether it is possible to apply the classic TBL paradigm to graph transformation tasks, for example, using Graph-Based Relational Learning (Cook and Holder, 2000) for extraction of simpler rewrite rules.

- The weaknesses of the method that we identified above suggest another direction for further work. Is it possible to factor out or to parametrize in a natural way the current heuristics for the rule selection? Our hope is that such an analysis of heuristics and assumptions of our method will help to address the low recall problem, allowing us to handle corpora like FrameNet that have so far prove difficult.

- The relatively low recall values our method demonstrates on various tasks indicate that either the current rule selection strategy does not cope well with less frequent rewrite rules, or the types of the rewrite rules we consider are not sufficient. We need to experiment both with rule types and selection strategies to determine the cause of low recall scores and to improve the performance of our method.

These additional experiments with our general graph transformation method will help us to identify differences and similarities between the language processing tasks to which we apply our methods. They will allow us to make explicit what parameter settings and biases are necessary for our method to perform well. Moreover, we believe that this is essential for a deeper understanding of the NLP tasks themselves by uncovering regularities and patterns in the underlying language usage.

Appendix A

Penn Treebank II Annotation: Function Tags and Empty Nodes

In this appendix we briefly describe two aspects of the Penn Treebank II annotation: function tags and empty nodes. We refer to the Penn Treebank II annotation guidelines Bies *et al.* (1995) for a detailed description.

A.1 Function tags

Function tags are attached to phrase labels in the Penn Treebank annotation to indicate a specific grammatical role or semantic aspects of the corresponding constituents. A constituent may have more than one function tag. Technically, attaching tags to a constituent can be viewed as changing the constituent's label, e.g., from NP to NP-SBJ.

The following function tags are defined in the Penn Treebank II annotation guidelines:

-ADV	(adverbial) — marks constituents used adverbially
-NOM	(nominal) — marks relatives and gerunds that act nominally
-DTV	(dative) — marks the dative object in the double object construction
-LGS	(logical subject) — marks the logical subject in passives
-PRD	(predicate) — marks any predicate that is not VP
-PUT	— marks the locative complement of *put*
-SBJ	(subject) — marks the surface subject of clauses
-TPC	(topicalized) — elements that appear before subjects in declarative sentences
-VOC	(vocative) — marks nouns of address

-BNF	(benefactive) — marks the benefactive of an action
-DIR	(direction) — adverbials answering “from where” or “to where” questions
-EXT	(extent) — phrases describing the spacial extent of an activity
-LOC	(locative) — adverbials indicating place/setting of an event
-MNR	(manner) — adverbial that indicate manner
-PRP	(purpose) — purpose or reason clauses or PPs
-TMP	(temporal) — temporal or aspectual adverbs
-CLR	(closely related) — constituents occupying some middle ground between argument and adjuncts of a verb phrase
-CLF	(cleft) — marks it-clefs
-HLN	(headline) — marks headlines and datelines
-TTL	(title) – titles appearing inside running text

A.2 Empty nodes

Empty nodes in the Penn Treebank II are realized using a special part-of-speech tag -NONE-. The “lexical” content of the node indicates its function:

T	Traces of A' movement. Example: *Which story about tribbles did you read* *T**?*
*	Traces of NP movement and PRO. Example: *John was hit* * *by a ball.*
0	Null complementizer. Example: *. . . the bird* 0 *I saw*.
U	Interpreted position of a unit symbol. Example: *. . . more that $ 5* *U*.
?	Placeholder for ellipsed material. Example: *John is taller than I am* *?*.
NOT	Placeholder element in parallel constructions.
EXP	“Expletive”: marks *it*-extraposition.
ICH	“Interpret Constituent Here”: indicates discontinuous dependency.
PPA	“Permanent Predictable Ambiguity”: indicates ambiguity of attachment of a trace.
RNR	“Right Node Raising”: indicates simultaneous interpretation at multiple e attachment sites.

Bibliography

Ahn, D., Fissaha, S., Jijkoun, V., and de Rijke, M. (2004). The University of Amsterdam at Senseval-3: semantic roles and logic forms. In *Proceedings of the Third International Workshop on the Evaluation of Systems for the Semantic Analysis of Text*, pages 49–53.

Ahn, D., Alink, W., Jijkoun, V., de Rijke, M., Boncz, P., and de Vries, A. (2006a). Representing and querying multi-dimensional markup for question answering. In *NLPXML-2006: EACL 2006 Workshop on Multi-dimensional Markup in Natural Language Processing*.

Ahn, D., Jijkoun, V., Müller, K., de Rijke, M., and Sang, E. T. K. (2006b). Towards an Offline XML-based Strategy for Answering Questions. In C. Peters, F. Gey, J. Gonzalo, H. Müller, G. Jones, M. Kluck, B. Magnini, and M. D. Rijke, editors, *Accessing Multilingual Information Repositories*, volume 4022 of *Lecture Notes in Computer Science*, pages 449–456. Springer.

Baker, C. F., Fillmore, C. J., and Lowe, J. B. (1998). The Berkeley FrameNet project. In *Proceedings of the Thirty-Sixth Annual Meeting of the Association for Computational Linguistics and Seventeenth International Conference on Computational Linguistics*, pages 86–90.

Bejan, C. A., Moschitti, A., Morărescu, P., Nicolae, G., and Harabagiu, S. (2004). Semantic parsing based on FrameNet. In *Senseval-3: Third International Workshop on the Evaluation of Systems for the Semantic Analysis of Text*, pages 73–76.

Bies, A., Ferguson, M., Katz, K., and MacIntyre, R. (1995). Bracketing guidelines for Treebank II style Penn Treebank project. Technical report, University of Pennsylvania.

Blaheta, D. (2004). *Function Tagging*. Ph.D. thesis, Brown University.

Blaheta, D. and Charniak, E. (2000). Assigning function tags to parsed text. In *Proceedings of the 1st Annual Meeting of the North American Chapter of the ACL (NAACL)*, pages 234–240.

Bod, R., Scha, R., and Sima'an, K., editors (2003a). *Data-Oriented Parsing*. CSLI Publications. University of Chicago Press.

Bod, R., Hay, J., and Jannedy, S., editors (2003b). *Probabilistic Linguistics*. The MIT Press.

Bohnet, B. (2003). Mapping phrase structures to dependency structures in the case of free word order language. In *Proceedings of The First International Conference on Meaning-Text Theory*, Paris.

Bouma, G., Noord, G. V., and Malouf, R. (2001). Alpino: Wide-coverage computational analysis of Dutch. In *Computational Linguistics in The Netherlands 2000*, pages 45–59.

Bouma, G., Mur, J., van Noord, G., van der Plas, L., and Tiedemann, J. (2005). Question Answering for Dutch using dependency relations. In *Proceedings of CLEF: Cross-Language Evaluation Forum*.

Brill, E. (1995). Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, **21**(4), 543–565.

Brill, E. (1996). *Learning to Parse With Transformations*. Kluwer Academic Publishers.

Brill, E. and Resnik, P. (1994). A rule-based approach to prepositional phrase attachment disambiguation. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING'94)*, pages 1198–1204.

Brill, Eric, J. C. H. and Ngai, G. (2000). Automatic grammar induction: Combining, reducing and doing nothing. In *Proceedings of the Sixth International Workshop on Parsing Technologies (IWPT'2000)*.

Briscoe, T., Carroll, J., Graham, J., and Copestake, A. (2002). Relational evaluation schemes. In *Proceedings of the LREC 2002 Workshop Beyond PARSEVAL – Towards Improved Evaluation Measures for Parsing Systems*, pages 4–8.

Buchholz, S. (2002). *Memory-based Grammatical Relation Finding*. Ph.D. thesis, Tilburg University.

Carroll, J., Minnen, G., and Briscoe, T. (2003). Parser evaluation using a grammatical relation annotation scheme. In A. Abeillé, editor, *Building and Using Parsed Corpora*, pages 299–316. Kluwer.

Charniak, E. (2000). A maximum-entropy-inspired parser. In *Proceedings of the 1st Annual Meeting of the North American Chapter of the Association for Computational Lingustics (NAACL)*, pages 132–139.

Chen, J. and Rambow, O. (2003). Use of deep linguistic features for the recognition and labeling of semantic arguments. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 41–48.

Collins, M. (1997). Three generative, lexicalized models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 16–23.

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

Collins, M., Ramshaw, L., Hajič, J., and Christoph, T. (1999). A statistical parser for Czech. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 505–512.

Cook, D. J. and Holder, L. B. (2000). Graph-based data mining. *IEEE Intelligent Systems*, **15**(2), 32–41.

Covington, M. A. (1994). An empirically motivated reinterpretation of dependency grammar. Technical Report AI-1994-01, Athens, GA.

Curran, J. R. and Wong, R. K. (1999). Transformation-based learning in document format processing. In *Working notes of the AAAI 1999 Fall Syposium on Using Layout for the Generation, Understanding or Retrieval of Documents*, pages 75–79.

Curran, J. R. and Wong, R. K. (2000). Formalisation of transformation-based learning. In *Proceedings of the 2000 Australian Computer Science Conference (ACSC)*, pages 51–57.

Daelemans, W. and van den Bosch, A. (2005). *Memory-Based Language Processing*. Cambridge University Press.

Daelemans, W., Zavrel, J., van der Sloot, K., and van den Bosch, A. (2003). *TiMBL: Tilburg Memory Based Learner, version 5.0, Reference Guide*. ILK Technical Report 03-10. Available from `http://ilk.kub.nl/downloads/pub/papers/ilk0310.ps.gz`.

Dienes, P. (2004). *Statistical Parsing with Non-local Dependencies*. Ph.D. thesis, Universität des Saarlandes, Saarbrücken, Germany.

Dienes, P. and Dubey, A. (2003a). Antecedent recovery: Experiments with a trace tagger. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing*, pages 33–40.

Dienes, P. and Dubey, A. (2003b). Deep syntactic processing by combining shallow methods. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*.

Drewes, F., Kreowski, H.-J., and Habel, A. (1997). *Hyperedge Replacement Graph Grammars*, chapter 2, pages 95–162. Volume 1 of Rozenberg (1997).

Engelfriet, J. and Rozenberg, G. (1997). *Node Replacement Graph Grammars*, chapter 1, pages 1–94. Volume 1 of Rozenberg (1997).

Fellbaum, C., editor (1998). *WordNet: An Electronic Lexical Database*. The MIT Press.

Ferro, L., Vilain, M., and Yeh, A. (1999). Learning transformation rules to find grammatical relations. In *Proceedings of the 3rd International Workshop on Computational Natural Language Learning (CoNLL)*, pages 43–52.

Fillmore, C. J. (1982). Frame semantics. *Linguistics in the Morning Calm*, pages 111–137.

Freund, Y. and Schapire, R. E. (1998). Large margin classification using the perceptron algorithm. In *Proceedings of the 11th Annual Conference on Computational Learning Theory (COLT)*, pages 277–296.

Gildea, D. (2001). *Statistical Language Understanding Using Frame Semantics*. Ph.D. thesis, University of California, Berkeley.

Gildea, D. and Hockenmaier, J. (2003). Identifying semantic roles using combinatory categorial grammar. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Gildea, D. and Jurafsky, D. (2002). Automatic labeling of semantic roles. *Computational Linguistics*, **28**(3), 245–288.

Gildea, D. and Palmer, M. (2002). The necessity of syntactic parsing for predicate argument recognition. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Hacioglu, K. (2004). A lightweight semantic chunker based on tagging. In *Proceedings of the Human Language Technology Conference/North American chapter of the Association of Computational Linguistics (HLT/NAACL)*.

Hajičová, E. and Kučerová, I. (2002). Argument/valency structure in propbank, lcs database and prague dependency treebank: A comparative pilot study. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC)*, pages 846–851.

Hockenmaier, J. (2003). Parsing with generative models of predicate-argument structure. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 359–366.

Hockenmaier, J. and Steedman, M. (2002a). Acquiring compact lexicalized grammars from a cleaner treebank. In *Proceedings of Third International Conference on Language Resources and Evaluation (LREC)*, pages 1974–1981.

Hockenmaier, J. and Steedman, M. (2002b). Generative models for statistical parsing with combinatory categorial grammar. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL)*, pages 335–342.

Hudson, R. A. (1990). *English Word Grammar*. Blackwell, Oxford.

Jijkoun, V. (2003). Finding non-local dependencies: Beyond pattern matching. In *Proceedings of the ACL-2003 Student Research Workshop*, pages 37–43.

Jijkoun, V. and de Rijke, M. (2004). Enriching the output of a parser using memory-based learning. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL)*, pages 311–318.

Jijkoun, V. and de Rijke, M. (2005a). Recognizing textual entailment using lexical similarity. In *Proceedings of the PASCAL Recognising Textual Entailment Challenge*, pages 73–76.

Jijkoun, V. and de Rijke, M. (2005b). Retrieving answers from frequently asked questions pages on the web. In *Proceedings of the Fourteenth ACM Conference on Information and Knowledge Management (CIKM 2005)*. ACM Press.

Jijkoun, V. and de Rijke, M. (2006). Recognizing textual entailment: Is lexical similarity enough? In I. Dagan, F. Dalche, J. Quinonero Candela, and B. Magnini, editors, *Evaluating Predictive Uncertainty, Textual Entailment and Object Recognition Systems*, volume 3944 of *LNAI*, pages 449–460. Springer.

Jijkoun, V., de Rijke, M., and Mur, J. (2004). Information Extraction for Question Answering: Improving recall through syntactic patterns. In *Proceedings of the 20th International on Computational Linguistics (COLING 2004)*, pages 1284–1290.

Joachims, T. (1999). Making large-scale svm learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, pages 169–184. MIT-Press.

Johnson, C. R., Petruck, M. R. L., Baker, C. F., Ellsworth, M., Ruppenhofer, J., and Fillmore, C. J. (2003). FrameNet: Theory and Practice. `http://www.icsi.berkeley.edu/~framenet`.

Johnson, M. (2002). A simple pattern-matching algorithm for recovering empty nodes and their antecedents. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 136–143.

Katz, B. and Lin, J. (2003). Selectively using relations to improve precision in question answering. In *Proceedings of the EACL-2003 Workshop on Natural Language Processing for Question Answering*.

Kingsbury, P., Palmer, M., and Marcus, M. (2002). Adding semantic annotation to the Penn Treebank. In *Proceedings of the Human Language Technology Conference (HLT'02)*.

Kipper, K., Trang Dang, H., and Palmer, M. (2000). Class-based construction of a verb lexicon. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI)*.

Klein, D. and Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st Meeting of the Association for Computational Linguistics (ACL)*, pages 423–430.

Kübler, S. and Telljohann, H. (2002). Towards a dependency-based evaluation for partial parsing. In *Proceedings of the LREC 2002 Workshop Beyond PARSEVAL – Towards Improved Evaluation Measures for Parsing Systems*.

Kuramochi, M. and Karypis, G. (2001). Frequent subgraph discovery. In *Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM)*, pages 313–320.

Kwon, N., Fleischman, M., and Hovy, E. (2004a). FrameNet-based semantic parsing using Maximum Entropy models. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING)*, pages 1233–1239.

Kwon, N., Fleischman, M., and Hovy, E. (2004b). Senseval automatic labeling of semantic roles using Maximum Entropy models. In *Senseval-3: Third International Workshop on the Evaluation of Systems for the Semantic Analysis of Text*.

Levenshtein, V. (1966). *Binary codes capable of correcting deletions, insertions and reversals*, volume 10, pages 707–710.

Levin, B. (1993). *English Verb Classes and Alternations: A Preliminary Investigation*. University of Chicago Press, Chicago.

Levy, R. and Manning, C. (2004). Deep dependencies from context-free statistical parsers: Correcting the surface dependency approximation. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 327–334.

Lin, D. (1994). PRINCIPAR – an efficient, broad-coverage, principle-based parser. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING-94)*.

Lin, D. (1998). A dependency-based method for evaluating broad-coverage parsers. *Natural Language Engineering*, **4**(2), 97–114.

Litkowski, K. (2004). Senseval-3 task: Automatic labeling of semantic roles. In *Proceedings of the Third International Workshop on the Evaluation of Systems for the Semantic Analysis of Text*.

Magerman, D. (1995). Statistical decision-tree models for parsing. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 276–283.

Manning, C. and Schütze, H. (1999). *Foundations of statistical natural language processing*. The MIT press.

Manning, C. D. (2003). *Probabilistic Syntax*. In Bod *et al.* (2003b).

Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Fergusson, M., Katz, K., and Schasberger, B. (1994). The Penn Treebank: Annotating predicate argument structure. In *Proceedings of the 1994 Human Language Technology Workshop*, pages 110–115.

Mel'cuk, I. A. (1988). *Dependency syntax: theory and practice*. SUNY Series in Linguistics. State University of New York Press, Albany.

Moldovan, D., Harabagiu, S., Girju, R., Morarescu, P., Lacatusu, F., Novischi, A., Badulescu, A., and Bolohan, O. (2003a). LCC Tools for Question Answering. In *Proceedings of the Eleventh Text REtrieval Conference (TREC 2002)*.

Moldovan, D., Paşca, M., Harabagiu, S., and Surdeanu, M. (2003b). Performance issues and error analysis in an open-domain question answering system. *ACM Transactions on Information Systems*, **21**, 133–154.

Moldovan, D., Gîrju, R., Olteanu, M., and Fortu, O. (2004). SVM classification of FrameNet semantic roles. In *Senseval-3: Third International Workshop on the Evaluation of Systems for the Semantic Analysis of Text*, pages 167–170.

Nivre, J. and Scholz, M. (2004). Deterministic dependency parsing of English text. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING)*, pages 64–70.

Palmer, M., Gildea, D., and Kingsbury, P. (2005). The Proposition Bank: An annotated corpus of semantic roles. *Computational Linguistics*, **31**(1), 71–106.

Pradhan, S., Ward, W., Hacioglu, K., Martin, J., and Jurafsky, D. (2004). Shallow semantic parsing using support vector machines. In *Proceedings of the Human Language Technology Conference/North American chapter of the Association of Computational Linguistics (HLT/NAACL)*.

Pradhan, S., Ward, W., Hacioglu, K., Martin, J. H., and Jurafsky, D. (2005a). Semantic role labeling using different syntactic views. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*.

Pradhan, S., Ward, W., Hacioglu, K., Martin, J., and Jurafsky, D. (2005b). Semantic role labeling using different syntactic views. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 581–588.

Pradhan, S., Hacioglu, K., Krugler, V., Ward, W., Martin, J. H., and Jurafsky, D. (2005c). Support vector learning for semantic argument classification. *Machine Learning*, **60**(1-3), 11–39.

Rambow, Owen, Dorr, B. J., Kipper, K., Kučerová, I., and Palmer, M. (2003). Automatically deriving tectogrammatical labels from other resources: A comparison of semantic labels across frameworks. In *The Prague Bulletin of Mathematical Linguistics*, volume 79–80, pages 23–35.

Ratnaparkhi, A. (1996). A maximum entropy part-of-speech tagger. In *Proceedings of the Empirical Methods in Natural Language Processing Conference*, pages 133–142.

Roche, E. and Schabes, Y. (1995). Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, **21**(2), 227–253.

Rozenberg, G., editor (1997). *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific.

Sampson, G. (1986). A stochastic approach to parsing. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING)*, pages 151–155.

Satta, G. and Brill, E. (1996). Efficient transformation-based parsing. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 255–262.

Schneider, G. (2003). A low-complexity, broad coverage probabilistic dependency parser for English. In *Proceedings of the Student Workshop of the Human Language Technology Conference/North American chapter of the Association of Computational Linguistics (HLT/NAACL)*, pages 31–36.

Schürr, A. (1997). *Programmed Graph Replacement Systems*, chapter 7, pages 479–546. Volume 1 of Rozenberg (1997).

Sleator, D. and Temperley, D. (1991). Parsing English with a link grammar. Technical Report CMU-CS-91-196, Department of Computer Science, Carnegie Mellon University.

Surdeanu, M., Harabagiu, S., Williams, J., and Aarseth, P. (2003). Using predicate-argument structures for information extraction. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics (ACL)*, pages 8–15.

Thomas H. Cormen, Charles E. Leiserson, R. L. R. and Stein, C. (2001). *Introduction to Algorithms, Second Edition*. The MIT Press.

Toutanova, K., Haghighi, A., and Manning, C. (2005). Joint learning improves semantic role labeling. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 589–596.

Voorhees, E. and Trang Dang, H. (2006). Overview of the TREC 2005 Question Answering track. In *Proceedings of the Fourteenth Text REtrieval Conference (TREC 2005)*.

Webber, B., Gardent, C., and Bos, J. (2002). Position statement: Inference in question answering. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC)*.

Williams, K., Dozier, C., and McCulloh, A. (2004). Learning transformation rules for semantic role labeling. In *Proceedings of the 7th International Workshop on Computational Natural Language Learning (CoNLL)*.

Xia, F. and Palmer, M. (2001). Converting dependency structures to phrase structures. In *Proceedings of the First International Conference on Human Language Technology Research*, pages 1–5.

Yamada, H. and Matsumoto, Y. (2003). Statistical dependency analysis with support vector machines. In *Proceedings of the 8th International Workshop on Parsing Technologies*.

Yan, X. and Han, J. (2002). gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM)*, page 721.

Samenvatting

Dit proefschrift beschrijft een methode voor het leren van graaftransformaties en onderzoekt de toepassingen van deze methode binnen de natuurlijke taalverwerking (Natural Language Processing, NLP). We beschouwen syntactische en semantische taalstructuren als gelabelde gerichte grafen, en herformuleren NLP-problemen als afbeeldingen tussen verschillende typen grafen. Bijvoorbeeld, een semantische analyse-component leest als invoer een graaf die de syntactische structuur van een zin weergeeft, en produceert als uitvoer een andere graaf met een semantische annotatie van de zin. Zo gezien is een NLP-algoritme simpelweg een systeem dat grafen transformeert. Deze uniforme presentatie van NLP-problemen maakt het mogelijk om een algemene methode te introduceren voor de constructie van graaftransformatie-tools.

We beschrijven een algemene methode voor het leren van graaftransformaties op basis van geannoteerde corpora, oftewel collecties van voorbeeldinvoer- en uitvoergrafen. Deze methode voor machinaal leren combineert het Transformation Based Learning paradigma met traditionele algoritmes voor "supervised" machinaal leren, om automatisch te herkennen welke reeks atomaire graaftransformaties de invoergraaf omzet in de juiste uitvoergraaf. Atomaire transformaties zijn eenvoudige regels die kleine wijzigingen aanbrengen in de verbindingen tussen subgrafen. Het toepassen van deze regels wordt gestuurd door middel van traditionele meervoudige classificatie.

Eerst beschrijven we de algemene methode voor het leren van graaftransformaties. Vervolgens beschouwen we drie specifieke NLP-toepassingen:

- Het herkennen van predicaat-argumentstructuren in het resultaat van semantische analyse, zowel voor syntactische dependentiestructuren als voor syntactische frasestructuren;
- het automatisch omzetten van syntactische structuren van het ene dependentieformalisme naar het andere; en
- het herkennen van semantische argumenten gegeven een syntactische parse,

waarbij we gebruik maken van PropBank en FrameNet als bronnen van semantische annotaties.

We tonen aan dat de prestatie van graaftransformatie op het eerstgenoemde probleem vergelijkbaar is met de best bekende technieken. Bovendien demonstreren we de flexibiliteit van deze methode door haar toe te passen op verschillende soorten syntactische representaties. Toepassing van de methode op de overige twee problemen wijst uit dat de parameters van onze leermethode zorgvuldige afgesteld moeten worden om voor die problemen goede resultaten te bereiken.

SIKS Dissertation Series

1998

1998-1 J̄ohan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM)
Memory versus Search in Games

1998-5 E.W.Oskamp (RUL)
Computerondersteuning bij Straftoemeting

1999

1999-1 Mark Sloof (VU)
Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR)
Classifi cation using decision trees and neural nets

1999-3 Don Beal (UM)
The Nature of Minimax Search

1999-4 Jacques Penders (UM)
The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB)
Empowering Communities: A Method for the Legitimate User-Driven Specifi cation of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-7 David Spelt (UT)
Verifi cation support for object database design

1999-8 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000

2000-1 Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE)
Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UVA)
Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval.

2000-6 Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-7 Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coup (EUR)
Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

2001

2001-1 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA)
Learning as problem solving

2001-4 Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU)
Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU)
Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 Pieter Jan 't Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA)
Knowledge Management: The Role of Mental Models in Business Systems Design

2002

2002-01 Nico Lassing (VU)
Architecture-Level Modifi ability Analysis

2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU)
The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL)
Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB)
Integrating Modern Business Applications with Objectifi ed Legacy Systems

2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (UvA)
Processing XML in Database Systems

2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT)
Semantics and Verifi cation of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA)
Understanding, Modeling, and Improving Main-Memory Database Performance

2003

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA)
Causation in Artifi cial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specifi cation of virtual environments

2003-07 Machiel Jansen (UvA)
Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM)
Repair Based Scheduling

2003-09 Rens Kortmann (UM)
The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT)
Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM)
Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN)
Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD)
Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI)
Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT)
Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM)
Learning Search Decisions

2004

2004-01 Virginia Dignum (UU)
A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT)
Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU)
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UVA)
Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR)
Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD)
The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM)
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM)
Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU)
For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UVA)
Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU)
Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT)
Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT)
Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU)
Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU)
Multi-Relational Data Mining

2004-16 Federico Divina (VU)
Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM)
Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA)
Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT)
Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode)
Learning from Design: facilitating multidisciplinary design teams

2005

2005-01 Floor Verdenius (UVA)
Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM)
AI techniques for the game of Go

2005-03 Franc Grootjen (RUN)
A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT)
Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UVA)
Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM)
Adaptive Game AI

2005-07 Flavius Frasincar (TUE)
Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE)
A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU)
Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UVA)
Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU)
Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR)
Distributed Simulation in Industry

2005-13 Fred Hamburg (UL)
Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU)
Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU)
Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU)
Usability of XML Query Languages

2005-17 Boris Shishkov (TUD)
Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU)
Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM)
Situated Representation

2005-20 Cristina Coteanu (UL)
Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT)
Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006

2006-01 Samuil Angelov (TUE)
Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU)
Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UVA)
The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU)
Building Web Service Ontologies

2006-05 Cees Pierik (UU)
Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU)
Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT)
XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT)
Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM)
Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU)
Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT)
Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU)
Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU)
Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU)
Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU)
CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU)
Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU)
User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UVA)
Graph Transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU)
Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT)
Monotone models for prediction in data mining

2006-21	Bas van Gils (RUN) *Aptness on the Web*
2006-22	Paul de Vrieze (RUN) *Fundaments of Adaptive Personalisation*
2006-23	Ion Juvina (UU) *Development of Cognitive Model for Navigating on the Web*
2006-24	Laura Hollink (VU) *Semantic Annotation for Retrieval of Visual Resources*
2006-25	Madalina Drugan (UU) *Conditional log-likelihood MDL and Evolutionary MCMC*
2006-26	Vojkan Mihajlovič (UT) *Score Region Algebra: A Flexible Framework for Structured Information Retrieval*
2006-27	Stefano Bocconi (CWI) *Vox Populi: generating video documentaries from semantically annotated media repositories*