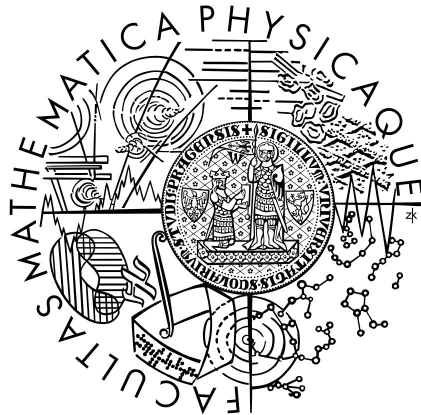
Charles University in Prague
Faculty of Mathematics and Physics



Mathematical Properties of Dependency Trees and their Application to Natural Language Syntax

Jiří Havelka

Ph.D. Thesis
Prague, June 2007

Doctoral thesis

Author: Jiří Havelka
Institute of Formal and Applied Linguistics
Charles University in Prague
Faculty of Mathematics and Physics
Malostranské náměstí 25, 118 00 Praha 1, Czech Republic
email: jiri.havelka@mff.cuni.cz

Supervisor: Prof. PhDr. Eva Hajičová, DrSc.
Institute of Formal and Applied Linguistics
Charles University in Prague
Faculty of Mathematics and Physics
Malostranské náměstí 25, 118 00 Praha 1, Czech Republic

Department: Institute of Formal and Applied Linguistics
Charles University in Prague
Faculty of Mathematics and Physics
Malostranské náměstí 25, 118 00 Praha 1, Czech Republic

Contents

List of Algorithms 4

List of Figures 5

List of Tables 6

Preface 8

Introduction 9

I Theoretical results 12

1 Preliminaries 13

1.1 Rooted trees and dependency trees 13

1.2 Data representations of rooted trees and dependency trees . . 18

1.2.1 Data representation of a node 18

1.2.2 Data representation of a whole dependency tree 19

1.3 Remark on processing rooted trees 20

2 Projectivity and basic properties of non-projective edges 22

2.1 Condition of projectivity in dependency trees 22

2.2 Non-projective edges and their gaps 24

2.3 Simple algorithm for finding non-projective edges and determining their gaps 26

3 Projectivity and projective orderings of dependency trees 27

3.1 An alternative condition of projectivity 27

3.2 Projective orderings of a rooted tree 30

3.3 Algorithm for projectivizing 34
3.4 Using the algorithm for checking projectivity 37

4 Level types of non-projective edges **38**
4.1 Basic properties of level types and their relationship to projectivity 38
4.2 Algorithm for finding non-projective edges of non-negative level type 42
4.3 Using the algorithm for checking projectivity and for finding all non-projective edges 47
4.4 Combining algorithms for finding non-projective edges of non-negative level types and for projectivizing 49

5 Planarity and non-projective edges **51**
5.1 Condition of planarity 51
5.2 Planarity and non-projective edges 54
5.3 Characterization of planarity using single non-projective edges 56
5.4 Checking planarity 58
5.5 Remark on NP-completeness of multiplanarity 60

6 Well-nestedness and non-projective edges **62**
6.1 Original formulation of well-nestedness 62
6.2 Reformulation of well-nestedness in terms of edges 64
6.3 Characterization of well-nestedness using pairs of non-projective edges 65
6.4 Sufficient condition for ill-nestedness 66
6.5 Characterization of well-nestedness using single edges 66
6.6 Checking well-nestedness 68

7 Partitioning of gaps of non-projective edges **71**
7.1 Partitioning of gaps into intervals 71
7.2 Partitioning of gaps into components 72
7.3 Combining levels of nodes and partitioning of gaps into intervals 74

8 Formulas for counting some classes of trees **76**
8.1 Unrestricted dependency trees 76
8.2 Projective and planar trees 77
8.3 Well-nested trees 77
8.4 Note on asymptotic growths 78

II Empirical results 80

9 Empirical evaluation of algorithms for finding non-projective edges 81

10 Evaluation of tree and edge properties on natural language data 83
10.1 Experimental setup . . . 83
10.1.1 Natural language treebanks . . . 84
10.1.2 Reported tree and edge properties . . . 85
10.1.3 Note on computing the tree and edge properties . . . 86
10.1.4 Program tools . . . 87
10.2 Empirical results . . . 87
10.2.1 Arabic . . . 88
10.2.2 Basque . . . 89
10.2.3 Bulgarian . . . 91
10.2.4 Catalan . . . 92
10.2.5 Czech . . . 93
10.2.6 Danish . . . 95
10.2.7 Dutch . . . 96
10.2.8 English . . . 98
10.2.9 German . . . 100
10.2.10 Greek . . . 102
10.2.11 Hungarian . . . 103
10.2.12 Italian . . . 105
10.2.13 Japanese . . . 106
10.2.14 Latin . . . 108
10.2.15 Portuguese . . . 113
10.2.16 Slovene . . . 115
10.2.17 Spanish . . . 116
10.2.18 Swedish . . . 117
10.2.19 Turkish . . . 119
10.3 Discussion . . . 121
10.3.1 Tree properties . . . 121
10.3.2 Edge properties . . . 121
10.4 Conclusion . . . 123

Index 124

Bibliography 127

List of Algorithms

1 Sketch of algorithm for determining gaps of non-projective edges 26
2 Sketch of algorithm for general projectivization 34
3 Algorithm for general projectivization 35
4 Sketch of algorithm for finding non-projective edges of non-negative level type 43
5 Algorithm for finding non-projective edges of non-negative level type 44
6 Sketch of algorithm for finding non-projective edges of non-negative level type and general projectivization 49
7 Sketch of algorithm for checking planarity 58
8 Sketch of algorithm for determining upper non-planar sets . . . 59
9 Sketch of algorithm for determining ill-nested sets 69
10 Sketch of algorithm for determining upper ill-nested sets . . . 69

List of Figures

1.1 Sample non-projective dependency tree and its data representation . . . 20

2.1 Sample projective and non-projective dependency trees 23

3.1 Sample non-projective dependency tree showing that condition (†) is not equivalent to projectivity . . . 29
3.2 Example of canonical projectivization . . . 31

4.1 Sample configurations with non-projective edges of negative, zero, and positive level types . . . 40
4.2 Sample minimal non-projective dependency tree with a non-projective edge of level type -1 . . . 41

5.1 Sample non-planar dependency trees . . . 52
5.2 Relationship between planar totally ordered unrooted trees and projective dependency trees . . . 53

6.1 Schematic visualization of well-nested and ill-nested dependency trees . . . 63
6.2 Sample ill-nested dependency trees . . . 64

7.1 Sample dependency trees showing mutual independence of interval degree and component degree of a non-projective edge . 73

10.1 Sample non-projective dependency tree considered planar in empirical evaluation . . . 87

List of Tables

8.1 Counts of trees of some classes of rooted trees on small numbers of nodes 79

9.1 Summary of non-projective dependency trees and edges on analytical layer of PDT 2.0 82
9.2 Running times of Algorithm 5 and simple Algorithm 1 for finding non-projective edges on PDT 2.0 82

10.1 Arabic: Counts of dependency trees violating global constraints 88
10.2 Arabic: Counts of properties of non-projective edges 88
10.3 Basque: Counts of dependency trees violating global constraints 89
10.4 Basque: Counts of properties of non-projective edges 89
10.5 Bulgarian: Counts of dependency trees violating global constraints 91
10.6 Bulgarian: Counts of properties of non-projective edges . . . 91
10.7 Catalan: Counts of dependency trees violating global constraints 92
10.8 Catalan: Counts of properties of non-projective edges 92
10.9 Czech: Counts of dependency trees violating global constraints 93
10.10 Czech: Counts of properties of non-projective edges 93
10.11 Danish: Counts of dependency trees violating global constraints 95
10.12 Danish: Counts of properties of non-projective edges 95
10.13 Dutch: Counts of dependency trees violating global constraints 96
10.14 Dutch: Counts of properties of non-projective edges 96
10.15 English: Counts of dependency trees violating global constraints 98
10.16 English: Counts of properties of non-projective edges 98

10.17 German: Counts of dependency trees violating global constraints 100
10.18 German: Counts of properties of non-projective edges 100
10.19 Greek: Counts of dependency trees violating global constraints 102
10.20 Greek: Counts of properties of non-projective edges 102
10.21 Hungarian: Counts of dependency trees violating global constraints 103
10.22 Hungarian: Counts of properties of non-projective edges . . . 103
10.23 Italian: Counts of dependency trees violating global constraints 105
10.24 Italian: Counts of properties of non-projective edges 105
10.25 Japanese: Counts of dependency trees violating global constraints 106
10.26 Japanese: Counts of properties of non-projective edges 106
10.27 Latin – Cicero: Counts of dependency trees violating global constraints 108
10.28 Latin – Cicero: Counts of properties of non-projective edges . 108
10.29 Latin – Caesar: Counts of dependency trees violating global constraints 109
10.30 Latin – Caesar: Counts of properties of non-projective edges 109
10.31 Latin – Vergil: Counts of dependency trees violating global constraints 110
10.32 Latin – Vergil: Counts of properties of non-projective edges . 110
10.33 Latin – Jerome: Counts of dependency trees violating global constraints 112
10.34 Latin – Jerome: Counts of properties of non-projective edges 112
10.35 Portuguese: Counts of dependency trees violating global constraints 113
10.36 Portuguese: Counts of properties of non-projective edges . . . 113
10.37 Slovene: Counts of dependency trees violating global constraints 115
10.38 Slovene: Counts of properties of non-projective edges 115
10.39 Spanish: Counts of dependency trees violating global constraints 116
10.40 Spanish: Counts of properties of non-projective edges 116
10.41 Swedish: Counts of dependency trees violating global constraints 117
10.42 Swedish: Counts of properties of non-projective edges 117
10.43 Turkish: Counts of dependency trees violating global constraints 119
10.44 Turkish: Counts of properties of non-projective edges 119

Preface

Both theoretical and empirical results presented in this thesis have been, at least partially, published.

Preliminary versions of our approach to projectivity were published in [Veselá and Havelka, 2003], [Veselá et al., 2004], and [Hajičová et al., 2004]; these papers contain an error rectified in [Havelka, 2005a], see also remark on page 29 in Section 3.1.

Preliminary results on our alternative condition of projectivity and level types of non-projective edges (Chapters 3 and 4) were first published in [Havelka, 2005a] and [Havelka, 2005b].

The characterization of the condition of well-nestedness through non-projective edges (Chapter 6) was presented in a shorter form in [Havelka, 2007b].

An empirical evaluation of formal means for describing non-projective dependency structures in natural language was presented in [Havelka, 2007a]. These means have been utilized in a continuing collaborative work on dependency parsing, published as [Hall et al., 2007].

Introduction

This thesis studies properties of mathematical structures used as models of syntactic structures in dependency analysis of natural languages. These mathematical structures are rooted trees, supplemented with a total order on the set of their nodes; as our primary field of application is computational linguistics, we call these mathematical structures simply dependency trees. We also empirically evaluate how mathematical properties of dependency trees can be used to describe dependency structures occurring in natural languages on treebanks from several languages.

In the theoretical part, we develop new formal tools capable of describing the interaction of the tree structure of a rooted tree with total orders on the set of its nodes. We also present several algorithms for related problems.

First, we develop tools for describing projective dependency trees and all projective orderings on arbitrary rooted trees. A novel alternative condition of projectivity for dependency trees is the key result, which allows us to derive easily the results for projective orderings.

Second and most importantly, we show that properties of non-projective edges, combined with levels of nodes, provide powerful mathematical tools for describing arbitrary non-projective dependency trees. We use them to characterize three global constraints on dependency trees, namely projectivity, planarity, and well-nestedness. (Although all three constraints can be applied to more general structures, we concentrate on their application to dependency trees.)

The approach to properties of rooted trees and dependency trees presented in this thesis can be characterized as *analytical* or *graph-theoretic* (an approach exemplified by Marcus [1967]), as opposed to *generative* or *formal-language-theoretic* (i.e., an approach using theory of formal languages, for dependency syntax starting with the work of Gaifman [1965]). The recently renewed interest in dependency analysis of natural languages has brought

also new results taking to the other approach, applying the theory of formal languages to dependency syntax [Kuhlmann and Möhl, 2007].

Even though our theoretical results for dependency trees and rooted trees are general and independent of any particular way of drawing the trees, to a large extent they have been inspired by geometric intuition. We use a visualization of dependency trees that utilizes the two dimensions of a plane to capture both the tree structure and the total order on all nodes explicitly. The reader is encouraged to draw pictures of dependency trees to get a better grasp of the presented results.

In the empirical part, we evaluate on natural language treebanks some of the presented theoretical results. Most importantly, we present an extensive evaluation on nineteen languages of some of the mathematical properties of dependency trees that can be used to describe non-projective structures occurring in natural languages. We show that our principal analytical tool, properties of non-projective edges combined with levels of nodes, are capable of delimiting non-projective structures in natural languages most accurately.

Outline of the thesis

We begin the theoretical part by Chapter 1 presenting formal preliminaries, i.e., basic notions and notation.

Chapter 2 presents formally the condition of projectivity; we also introduce the notion of a non-projective edge and its gap, which is the main analytical tool in this thesis.

In Chapter 3, we derive an alternative condition of projectivity, which allows us to develop formal tools for the characterization of all possible projective total orders on a rooted tree. We introduce the notion of a projectivization of a rooted tree, which can be informally described as an underspecified description of a projective total order of nodes of a general dependency tree, specified through total orders on parts of the rooted tree. We also present a linear algorithm for computing projective total orders, based on our characterization of projectivity.

Chapter 4 discusses the interaction of levels of nodes and gaps of non-projective edges in a dependency tree. We define level types of non-projective edges and show how they give yet another characterization of projectivity. We derive a linear algorithm for finding all non-projective edges of the level types that characterize non-projectivity.

Using non-projective edges and levels of nodes as tools, we give new characterizations of planarity in Chapter 5 and of well-nestedness in Chapter 6. We also derive algorithms for checking planarity and well-nestedness that utilize the algorithm for finding non-projective edges in a dependency tree.

In Chapter 7, we look at ways of partitioning the gaps of non-projective edges. These properties of individual non-projective edges can be used to describe non-projective dependency structures occurring in natural languages; they are, among others, used in the empirical evaluation.

To close the theoretical part, in Chapter 8 we briefly review formulas for counting trees in the classes of trees we are concerned with in this thesis.

In the empirical part, we first evaluate in Chapter 9 the actual running time of our algorithm for finding non-projective edges characterizing non-projectivity. We compare it with a naive algorithm for finding all non-projective edges to empirically verify its theoretical complexity bound.

In Chapter 10, we present an extensive evaluation of some of the tree and edge properties of non-projective dependency trees we study in the theoretical part. Our empirical results corroborate theoretical results and show that an edge-based approach using levels of nodes provides accurate and at the same time expressive tools for capturing non-projective structures in natural language.

Part I

Theoretical results

Chapter 1

Preliminaries

This chapter introduces basic notions and notation used in this thesis. The reader may skip it and consult it only when need arises, all terms are referenced in the index.

We presuppose knowledge of basics of graph theory and algorithm theory. In our analysis of algorithms, we assume a generic one-processor, *random-access machine* model of computation. The reader can consult any standard book on graph theory and any standard book on algorithms (we recommend the book [Cormen et al., 2001]) for further details not covered here.

1.1 Rooted trees and dependency trees

We give a definition of rooted trees and dependency trees, without proofs briefly review some basic properties of rooted trees, and introduce some further notions and notation.

1.1.1 Definition. A *rooted tree* T is a pair $(V, \rightarrow)$, where V is a finite set of *nodes* and $\rightarrow$ is a binary relation on V satisfying

a) relation $\rightarrow$ is *acyclic*;

b) for all $v \in V$, there is at most one node $p \in V$ such that $p \rightarrow v$; we call every such pair of nodes an *edge*;

c) there is a *root* node $r \in V$ such that for all nodes $v \in V$ there is a path $r \rightarrow^* v$ from the root node r to node v. (For any binary relation R, we denote its *reflexive and transitive closure* as R^*.)

If we drop the condition c) from Definition 1.1.1, we get a definition of a *forest*, which allows for multiple root nodes. In this thesis, we are by and large concerned with rooted trees; we will need the notion of a forest only at one place in the thesis.

For every node i of a rooted tree $T = (V, \to)$ we call the tree $T_i = (V_i, \to_i)$, where $V_i = \{v \in V \mid i \to^* v\}$ and $\to_i \, = \, \to \restriction V_i$ (i.e., relation $\to_i$ is the restriction of relation $\to$ to V_i), the *subtree* of rooted tree T rooted in node i. (The *restriction* of a binary relation R to a set S is the relation $R \cap (S \times S)$.)

Relation $\to$ models linguistic dependency, therefore we call it the *dependency* relation of a rooted tree T. Relation $\to^*$ is often called *subordination*.

Remark. Rooted trees can be conceived of both as undirected and directed. In the undirected case, the direction of edges can be induced on them using paths connecting all nodes with the root node. The definition of a directed, rooted tree we give here is one of many equivalent ways of defining rooted trees.

A dependency tree is a rooted tree supplemented with a total order on the set of its nodes.

1.1.2 Definition. A *dependency tree* is a triple $(V, \to, \preceq)$, where $(V, \to)$ is a rooted tree and $\preceq$ is a total order on V.

Dependency trees could be called *totally ordered rooted trees*, but throughout this thesis we prefer to use the shorter term dependency trees.

The set of nodes V is often taken to be $[n] = \{1, \ldots, n\}$, totally ordered by the natural total order on integers. Our formulation is equivalent and we use the symbol $\preceq$ to clearly mark the total order on the set of nodes V.

For every node i of a rooted tree $T = (V, \to, \preceq)$ we call the tree $T_i = (V_i, \to_i, \preceq_i)$, where $V_i = \{v \in V \mid i \to^* v\}$, $\to_i \, = \, \to \restriction V_i$, and $\preceq_i \, = \, \preceq \restriction V_i$, the *subtree* of dependency tree T rooted in node i (i.e., relations $\to_i$ and $\preceq_i$ are the restrictions of relations $\to$ and $\preceq$ to V_i, respectively).

For every node j in a rooted tree T such that $j \neq r$, we call the unique node p such that $p \to j$ the *parent* of node j. Analogously, for every node i we call any node c such that $i \to c$ a *child* of node i (we also say that node c *depends on* node i). Nodes with the same parent are called *siblings*. A node with no child nodes is called a *leaf*, a node which is not a leaf is an *internal* node.

We extend the terms parent and child also to edges: For an edge $i \to j$ in a rooted tree T, we call node i its *parent* node and node j its *child* node.

For any two nodes a, d in a dependency tree T such that $a \to^* d$, we

say that a is an *ancestor* of d, or that d is a *descendant* of a, or that d is *subordinated* to a. (Note that the relation of dependency $\rightarrow$ is irreflexive, whereas the relation of subordination $\rightarrow^*$ is defined as reflexive.)

In a rooted tree, there is a one-to-one correspondence between its edges and nodes different from the root node (edges correspond uniquely to their child nodes).

In a rooted tree T, there is a unique path from the root node r to every node i, say $v_0 = r, v_1, \ldots, v_k = i$, $k \geq 0$, where $v_l \rightarrow v_{l+1}$ for $0 \leq l < k$. Therefore every node i has a uniquely defined *level* equal to the length of the path connecting it with the root, i.e. k, which we denote level_i. The *height* of a rooted tree is defined as the maximum level occurring in it.

Although we define rooted trees as directed, it is often advantageous to treat them as undirected. We have already noted that both approaches to rooted trees are equivalent.

The symmetrization $\leftrightarrow \overset{\text{df}}{=} \rightarrow \cup \rightarrow^{-1}$ of relation $\rightarrow$ represents *undirected dependency relation*. It allows us to talk about edges without explicitly specifying their direction, i.e. their parent and child nodes. In other words, relation $\rightarrow$ represents directed edges and relation $\leftrightarrow$ represents undirected edges. We will use the symbol $\leftrightarrow$ exclusively for the symmetrization of dependency relation $\rightarrow$.

To retain the ability to talk about the direction of edges when using the relation $\leftrightarrow$, for any edge $i \leftrightarrow j$ in a rooted tree $T = (V, \rightarrow)$ we define

$$\mathsf{Parent}_{i \leftrightarrow j} = \begin{cases} i & \text{if } i \rightarrow j \\ j & \text{if } j \rightarrow i \end{cases}$$

and

$$\mathsf{Child}_{i \leftrightarrow j} = \begin{cases} j & \text{if } i \rightarrow j \\ i & \text{if } j \rightarrow i \end{cases} .$$

We write $i \rightarrow j \in S$ as a shortcut for $i \rightarrow j \;\&\; i, j \in S$; analogous convention applies also to undirected edges represented by relation $\leftrightarrow$.

To make the exposition clearer through using geometric intuition (and to enhance intelligibility by avoiding overuse of the symbol $\rightarrow$), we introduce notation for sets of *descendants* and *ancestors* of both nodes and edges in a

rooted tree $T = (V, \to)$

$$\mathsf{Subtree}_i = \{v \in V \mid i \to^* v\} \ ,$$

$$\mathsf{Subtree}_{i \leftrightarrow j} = \{v \in V \mid \mathsf{Parent}_{i \leftrightarrow j} \to^* v\} \ ,$$

$$\mathsf{Anc}_i = \{v \in V \mid v \to^* i\} \ ,$$

$$\mathsf{Anc}_{i \leftrightarrow j} = \{v \in V \mid v \to^* \mathsf{Parent}_{i \leftrightarrow j}\} \ .$$

Note that both notions of ancestors and descendants of an edge are defined relative to the parent node of the edge.

When talking about the tree structure of a rooted tree $T = (V, \to)$ (and thus also of a dependency tree), we use vertical-axis terms such as "above", "below", "upper", "lower" etc.; we draw dependency trees as is usual with the root node at the top, cf. Convention on page 17.

To be able to talk concisely about the total order $\preceq$ on the set of nodes V in a dependency tree, we define *open* and *closed intervals* in a total order $\preceq$ on V whose endpoints need not be in a prescribed order

$$(i, j) = \{v \in V \mid \min_{\preceq}\{i, j\} \prec v \prec \max_{\preceq}\{i, j\}\} \ ,$$

$$[i, j] = \{v \in V \mid \min_{\preceq}\{i, j\} \preceq v \preceq \max_{\preceq}\{i, j\}\} \ .$$

We do not mark explicitly with respect to which total order we take the intervals, it will be always clear from context.

For an edge $i \to j$ in a dependency tree T, we also refer informally to an interval (be it open or closed) delimited by the endpoints of the edge also as the *span* of edge $i \to j$.

When talking about the total order $\preceq$ on nodes of a dependency tree $T = (V, \to, \preceq)$, we use horizontal-axis terms such as "left", "right", "in between" etc. (with the obvious meaning: we say that i is to the left from j when $i \prec j$, etc.).

For total orders, we freely switch between a total order and its reflexive reduction, denoted by the presence and absence of an "or equals" part of the total order symbol, respectively. (A *reflexive reduction* of a binary relation R is the minimal binary relation R' such that the *reflexive closure* of R' is R; i.e., R' is obtained from R by removing all self-loops.)

We also use the notion of *transitive reduction* of a binary relation R, which is any minimal binary relation R' such that the *transitive closure* of R' is R (for total orders, the transitive reduction is unique). We use the notation R^{tr} for the transitive reduction of a binary relation R.

For the sake of brevity of formulas concerning projectivity of dependency trees, we introduce an auxiliary ternary predicate representing the "being

siblings" relation

$$\text{Sibl}(j_1, j_2, i) \overset{\text{df}}{\iff} (i \to j_1 \ \&\ i \to j_2 \ \&\ j_1 \neq j_2) \ .$$

An expanded reading of this predicate is "nodes j_1 and j_2 are different child nodes of their common parent node i". Note that predicate Sibl is symmetric in its first and second arguments.

Convention. In sample figures, vertically nodes are drawn top-down according to their increasing level, with nodes on the same level being the same vertical distance from the root; horizontally nodes are drawn from left to right according to the total order on nodes. We draw edges as solid lines, non-trivial paths as dotted curves. Subtrees may be schematically drawn as triangles with their upper tips representing their roots.

Remark. Figures may represent only *subgraphs* of dependency trees, not their full subtrees (i.e., subtrees in the above defined sense).

We conclude this section with a simple property of rooted trees we will use in proofs presented further in the thesis.

1.1.3 Proposition. *Let i be a node and $u \leftrightarrow v$ any edge disjoint from i in a rooted tree $T = (V, \to)$. Then*

$$u \in \mathsf{Subtree}_i \iff v \in \mathsf{Subtree}_i \ .$$

Proof. From the assumption $u \neq i \neq v$ it immediately follows that $u, v \in \mathsf{Subtree}_i \iff \mathsf{Parent}_{u \leftrightarrow v} \in \mathsf{Subtree}_i$, which is equivalent to the statement of the proposition. $\square$

Remark. Note that the notion of a dependency tree (which can be thought of as a *totally ordered rooted tree*, cf. Definition 1.1.2) differs from the notion of an *ordered rooted tree*, where for every internal node only a total order on the set of its child nodes is given (i.e. there is not a single total order on all nodes of the tree, there are only total orders on sets of sibling nodes). In this thesis, we are concerned with rooted trees with a total order on the set of all their nodes.

1.2 Data representations of rooted trees and dependency trees

Since we are concerned mainly with the interaction of the tree structure with a total order on nodes, we will discuss in detail only the data representation of dependency trees.

For rooted trees, we do not presuppose a specific data representation. The reader may assume the standard *left-child, right-sibling representation* of rooted trees, or a slight modification thereof described below as a subset of the data representation of dependency trees.

The data representation of dependency trees described in this section is a minimal representation of both the tree structure of a dependency tree and the total order on its nodes. Hence the representation subsumes a data representation of rooted trees. The presented data representation is the data representation we presuppose in the detailed versions of presented algorithms.

First we give the data structure for the representation of single nodes and then we describe the requirements we pose on the data representation of whole dependency trees.

If a different data representation is used, it either has to be transformed into the described one and the complexity of the transformation has to be added to the complexity of the presented algorithms, or some of the operations used in the algorithms that are assumed to be atomic have to be implemented in a different way and again their computational cost has to be included in the resulting complexity of the modified algorithms.

In this thesis, we present some algorithms only as high-level sketches. For those algorithms, no specific data representation is presupposed, but we assume that the data representation can be converted to the one described in this section in linear time; cf. Section 1.2.2 for more details.

1.2.1 Data representation of a node

A node of a rooted tree or a dependency tree is represented as an object with fields containing pointers. Unless explicitly specified otherwise, we assume that the object contains the following pointer fields:

`left_child` pointer to the linked list of the node's child nodes to the left from the node (for dependency trees, the linked list is ordered inversely to the total order on nodes; cf. below)

`right_child` pointer to the linked list of the node's child nodes to the right from the node (for dependency trees, the linked list is ordered according to the total order on nodes; cf. below)

`sibling` pointer to a sibling (left sibling if the node is a left child of its parent, right sibling if it is a right child)

`prev` pointer to previous node in the total order (used for dependency trees)

`next` pointer to next node in the total order (used for dependency trees)

We list here only those pointer fields which are presupposed by the algorithms concerning projectivity, rooted trees and dependency trees that we present in full detail. In other algorithms, we need to be able to determine the relative order of an arbitrary pair of nodes (this can be achieved e.g. by assigning each node an `order` field containing an integer representing its order); we also need access to parent nodes (e.g. by using a `parent` field containing for each node except the root node a pointer to its parent node). In practical applications, there would also be fields containing other information associated with nodes or edges (represented by their child nodes).

Convention. A pointer with no value assigned (i.e. an undefined pointer) is considered as a null reference; we do not use a special value for the null reference. The notation `field`$[i]$ is used for the contents of the `field` field of the object represented by pointer i.

1.2.2 Data representation of a whole dependency tree

We assume that for each node i in a dependency tree $T = (V, \rightarrow, \preceq)$, its child nodes can be traversed according to the total order $\preceq$ in linear time. This assumption allows us to disregard any possible cost of ordering sibling nodes according to $\preceq$.

One possible way of achieving this is to require that in the data representation of a dependency tree the linked list of left (right) child nodes of a node be ordered inversely (according) to the total order on nodes (respectively); cf. the data representation of a node described in the previous section. In fact, such a representation is only a minor variation of the standard left-child, right-sibling representation of rooted trees and we use it for ease of exposition. We could equivalently use the standard representation with the sibling list ordered according to the total order on nodes.

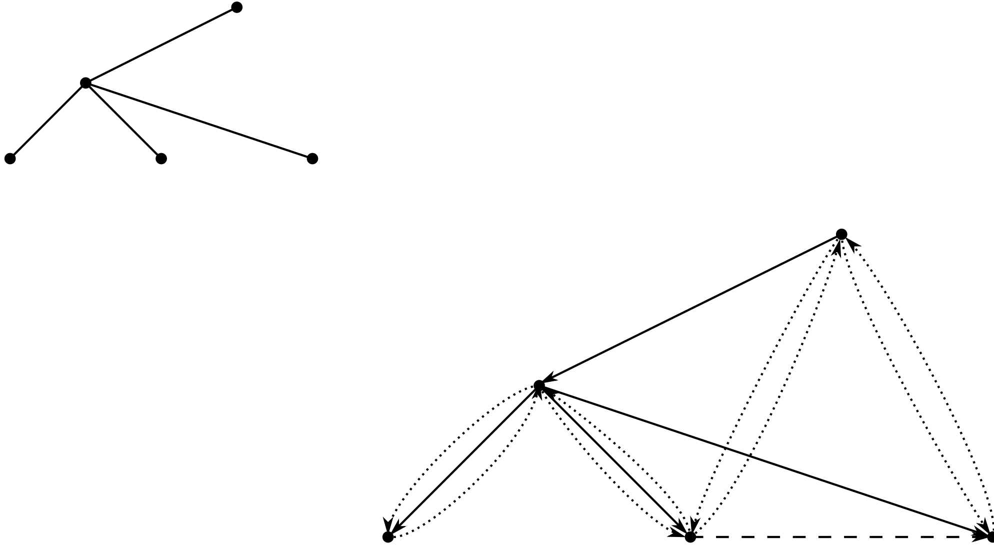


Figure 1.1: Sample non-projective dependency tree and its data representation

This requirement on ordering of nodes is natural, but may not be present in a different data representation of dependency trees. Transforming different data representations so as to meet this requirement is usually quite straightforward.

Figure 1.1 presents a sample a dependency tree and its data representation. Pointers to child nodes are depicted by solid arrows, pointers to siblings by dashed arrows, and pointers representing the total order by dotted arrows.

1.3 Remark on processing rooted trees

In the algorithms presented in this thesis, we use some simple facts concerning the processing of rooted trees.

We take advantage of the fact that a rooted tree can be processed in linear time using a tree traversal. Two natural ways of traversing a general rooted tree are *pre-order* and *post-order* tree traversals. Since the out-degree of nodes in a rooted tree is not bounded, sibling nodes can be processed in any order.

We use the term *general post-order traversal* of a rooted tree to stress that in post-order traversal sibling nodes may be processed in any relative order before processing their parent node.

As a special case of a general post-order traversal, we will be using *traversal by levels* from the deepest level to the root node, which can be achieved e.g. using a queue as an auxiliary data structure.

Here is a sketch how to build such a queue: put the root node in the queue; until you have reached the end of the queue, move to the next node in the queue, append its child nodes (e.g., in the order corresponding to their relative order in the total order on nodes; the actual order is irrelevant, as in the case of general post-order traversal) at the end of the queue. If you want to process nodes by levels bottom up, read the queue backwards.

Since we are also concerned with levels of nodes, we make a note that using a pre-order traversal of a rooted tree, it is easy to compute levels of all nodes in linear time. (The computation of levels for all nodes can by also straightforwardly incorporated into the construction of a queue for bottom-up traversal described above.)

In some of the algorithms discussed below, we need to be able to determine for a pair of nodes of a rooted tree whether one is subordinated to the other one. Since it does not affect the worst-case time bounds we derive for the relevant algorithms, we assume a simple quadratic algorithm for pre-computing the subordination relation $\to^*$ in a rooted tree $T = (V, \to)$.

We would like to remark that a more thorough analysis of the relevant algorithms presented below might take advantage of more efficient algorithms for determining subordination, or of more general algorithms for determining least common ancestors (in this thesis called lowest common ancestors).

Chapter 2

Projectivity and basic properties of non-projective edges

This section reviews the condition of projectivity in dependency trees. We present classical definitions of projectivity and introduce the notion of a non-projective edge. Properties of non-projective edges will be studied in further detail in subsequent chapters.

2.1 Condition of projectivity in dependency trees

We begin by giving a definition of projectivity using three conditions proposed in the early 1960's and proved to be equivalent by Marcus [1965]; we denote the conditions by the names of those to whom Marcus attributes their authorship.

2.1.1 Definition (Marcus [1965]). A dependency tree $T = (V, \to, \preceq)$ is *projective* if the following equivalent conditions hold

(Harper & Hays)

$$(\forall i, j, v \in V)\big(i \to j \ \& \ v \in (i, j) \implies v \in \mathsf{Subtree}_i\big) \ ,$$

(Lecerf & Ihm)

$$(\forall i, j, v \in V)\big(j \in \mathsf{Subtree}_i \ \& \ v \in (i, j) \implies v \in \mathsf{Subtree}_i\big) \ ,$$

(Fitialov)

$$(\forall i, j_1, j_2, v \in V)\big(j_1, j_2 \in \mathsf{Subtree}_i \ \& \ v \in (j_1, j_2) \implies v \in \mathsf{Subtree}_i\big) \ .$$

A dependency tree not satisfying the conditions is called *non-projective*.

Figure 2.1: Sample projective and non-projective dependency trees

See Figure 2.1 for examples of projective and non-projective dependency trees. From Definition 2.1.1 it immediately follows that a non-projective dependency tree has to have at least 3 nodes; the non-projective dependency tree in Figure 2.1(b) is an example of a minimal non-projective dependency tree.

Remark. Marcus [1965] derived the equivalence of conditions in Definition 2.1.1 for more general structures than dependency trees: totally ordered finite sets with any binary relation $\to$.

We do not give a proof of the equivalence of the three conditions in Definition 2.1.1, it is quite straightforward and relies on the fact that for every two nodes in the relation of subordination there exists a unique finite path between them formed by edges of the dependency relation $\to$.

We see that the antecedents of the projectivity conditions in Definition 2.1.1 move from edge-focused to subtree-focused (i.e. from talking about dependency to talking about subordination).

It is the condition of Fitialov that has been mostly explored when studying so-called relaxations of projectivity. However, we find the condition of Harper & Hays to be the most appealing from the linguistic point of view because it gives prominence to the primary notion of dependency edges over the derived notion of subordination. The condition of Harper & Hays is also the basis for the formal tools for describing non-projective dependency trees that we develop in this thesis.

Remark. The condition of Fitialov is perhaps the most transparent as far as regards the structure of the whole tree. It can be easily reformulated to

make the point even more clear*

(Fitialov') $(\forall i, j_1, j_2 \in V)$
$$\big(j_1, j_2 \in \mathsf{Subtree}_i \implies \neg(\exists v \in V)(v \in (j_1, j_2)\ \&\ v \notin \mathsf{Subtree}_i)\big)\ .$$

This reformulation gives immediately the following commonly used condition of projectivity: A dependency tree is projective if the nodes of all its subtrees constitute contiguous intervals in the total order on nodes.

2.2 Non-projective edges and their gaps

We introduce the notion of a non-projective edge and show its simple properties we use in subsequent chapters.

2.2.1 Definition. For any edge $i \leftrightarrow j$ in a dependency tree $T = (V, \rightarrow, \preceq)$ we define its *gap* as follows

$$\mathsf{Gap}_{i\leftrightarrow j} = \{v \in V \mid v \in (i, j)\ \&\ v \notin \mathsf{Subtree}_{i\leftrightarrow j}\}\ .$$

An edge with an empty gap is *projective*, an edge whose gap is non-empty is *non-projective*.

We see that non-projective are those edges $i \leftrightarrow j$ for which there is a node v such that together they violate the condition of Harper & Hays; we group all such nodes v into $\mathsf{Gap}_{i\leftrightarrow j}$, the gap of the non-projective edge $i \leftrightarrow j$.

2.2.2 Observation. *Let $i \rightarrow j$ be a non-projective edge in a dependency tree $T = (V, \rightarrow, \preceq)$ with root node r. Then $i \neq r$.*

Convention. In figures with sample configurations we adopt this convention: for a non-projective edge, we draw all maximal nodes in its gap explicitly and assume that for any upward path from the maximal nodes, no node

*The equivalence of conditions (Fitialov) and (Fitialov') is straightforward, see the following first-order-logic reasoning:

$$\begin{aligned}
&(\forall i, j_1, j_2, v \in V)(j_1, j_2 \in \mathsf{Subtree}_i\ \&\ v \in (j_1, j_2) \Rightarrow v \in \mathsf{Subtree}_i)\\
&\Leftrightarrow (\forall i, j_1, j_2, v \in V)(j_1, j_2 \in \mathsf{Subtree}_i \Rightarrow (v \in (j_1, j_2) \Rightarrow v \in \mathsf{Subtree}_i))\\
&\Leftrightarrow (\forall i, j_1, j_2, v \in V)(j_1, j_2 \in \mathsf{Subtree}_i \Rightarrow (v \notin (j_1, j_2) \vee v \in \mathsf{Subtree}_i))\\
&\Leftrightarrow (\forall i, j_1, j_2, v \in V)(j_1, j_2 \in \mathsf{Subtree}_i \Rightarrow \neg(v \in (j_1, j_2)\ \&\ v \notin \mathsf{Subtree}_i))\\
&\Leftrightarrow (\forall i, j_1, j_2 \in V)(j_1, j_2 \in \mathsf{Subtree}_i \Rightarrow (\forall v \in V)(\neg(v \in (j_1, j_2)\ \&\ v \notin \mathsf{Subtree}_i)))\\
&\Leftrightarrow (\forall i, j_1, j_2 \in V)(j_1, j_2 \in \mathsf{Subtree}_i \Rightarrow \neg(\exists v \in V)(v \in (j_1, j_2)\ \&\ v \notin \mathsf{Subtree}_i))\ .
\end{aligned}$$

on the path lies in the span of the non-projective edge. (By a *maximal* node in a gap we mean any node in the gap such that its parent node does not belong to the gap.)

Remark. The notion of gap is defined differently for subtrees of a dependency tree [Holan et al., 1998, Bodirsky et al., 2005]. There it is defined through the nodes of the whole dependency tree not in the considered subtree that intervene between its nodes in the total order on nodes $\preceq$.

Formally, based on the condition of Fitialov', the gap of a subtree T_i rooted in node i of a dependency tree $T = (V, \rightarrow, \preceq)$ is defined as follows

$$\mathsf{Gap}_i = \{v \in V \mid v \in (\min_{\preceq} \mathsf{Subtree}_i, \max_{\preceq} \mathsf{Subtree}_i) \;\&\; v \notin \mathsf{Subtree}_i\} \ .$$

This notion of gap of a subtree in a dependency tree was first used by Holan et al. [1998], who introduce measures of non-projectivity based on it and present a class of dependency-based formal grammars allowing for a varying degree of word-order freedom. Holan et al. [2000] present linguistic considerations concerning Czech and English with respect to this notion.

Next we present a couple of simple properties of non-projective edges. We will use them in proofs of theorems we present in subsequent chapters.

2.2.3 Proposition. *Let $i \leftrightarrow j$, $u \leftrightarrow v$ be disjoint edges in a dependency tree $T = (V, \rightarrow, \preceq)$. If $u, v \in (i, j)$, then*

$$u \in \mathsf{Gap}_{i\leftrightarrow j} \iff v \in \mathsf{Gap}_{i\leftrightarrow j} \ .$$

Proof. The statement follows immediately from the definition of $\mathsf{Gap}_{i\leftrightarrow j}$ and Proposition 1.1.3. □

2.2.4 Proposition. *Let $i \leftrightarrow j$, $u \leftrightarrow v$ be disjoint edges in a dependency tree $T = (V, \rightarrow, \preceq)$. If $u \in \mathsf{Gap}_{i\leftrightarrow j}$ and $v \notin \mathsf{Gap}_{i\leftrightarrow j}$, then $v \notin [i, j]$.*

Proof. For the sake of argument assume $v \in (i, j)$ and arrive at a contradiction with Proposition 2.2.3. □

2.3 Simple algorithm for finding non-projective edges and determining their gaps

Algorithm 1 presents a high-level sketch of a straightforward way of determining gaps of non-projective edges in a dependency tree. It is based directly on Definition 2.2.1.

Algorithm 1 Determine gaps – high-level sketch

Input: dependency tree T
Output: gaps of non-projective edges in T

1: **for** each edge $i \leftrightarrow j$ **do**
2:     **for** each node $v \in (i, j)$ **do**
3:         check $v \in \mathsf{Gap}_{i\leftrightarrow j}$
4:     **end for**
5: **end for**

The time complexity of Algorithm 1 is quadratic: there are linearly many edges and at most linearly many nodes in the span of any edge, which gives the quadratic bound. (We assume that checking subordination on line 3 takes constant time, e.g. using pre-computation; cf. Section 1.3.) Let us express this result as a theorem.

2.3.1 Theorem. *Algorithm 1 returns for a dependency tree T the gaps of all its non-projective edges; its time complexity is $O(n^2)$.*

There are also other possibilities of looking for non-projective edges, but we do not discuss them here. We will return to the problem of finding non-projective edges in detail in Chapter 4.

Chapter 3

Projectivity and projective orderings of dependency trees

We introduce a new reformulation of the condition of projectivity and show its equivalence with the classical ones. We define the notion of a projectivization of a rooted tree and show its uniqueness, and using our alternative condition of projectivity we derive a characterization of all projective orderings of a rooted tree.

3.1 An alternative condition of projectivity

All three conditions in Definition 2.1.1 have in common the following: in a configuration where two (or three) nodes have some structural relationship (i.e. a relationship via the dependency relation) and there is a node v between them in the total order, the conditions predicate that the node v be in an analogous structural relationship to one of the nodes in the antecedent.

Let us now present in the form of a theorem another condition which is equivalent to the conditions in Definition 2.1.1.

3.1.1 Theorem. *A dependency tree $T = (V, \rightarrow, \preceq)$ is projective if and only if the following condition holds*

$$\text{(ACP–3.1)}\quad (\forall i, j_1, j_2, u_1, u_2 \in V) \Big(\big[i \rightarrow j_1 \;\&\; u_1 \in \mathsf{Subtree}_{j_1} \;\&\; \big([j_2 = i \;\&\; u_2 = i] \vee [\mathrm{Sibl}(j_1, j_2, i) \;\&\; u_2 \in \mathsf{Subtree}_{j_2}] \big) \big] \Longrightarrow [j_1 \prec j_2 \Leftrightarrow u_1 \prec u_2] \Big) \;.$$

Before giving the proof of this theorem, let us give in words the meaning of condition (ACP–3.1): it says that for any subtree rooted in a node i the relative order of all nodes in distinct subtrees of the child nodes of i with respect to each other and to i has to be the same as the relative order of i and its child nodes. To put it succinctly, condition (ACP–3.1) requires that the total order of nodes in every subtree respect the order of the root node and the nodes on the first level of the subtree.

The substantial difference between condition (ACP–3.1) and the conditions in Definition 2.1.1 is that while the conditions in Definition 2.1.1 predicate that nodes in a structural and ordering relationship have a structural relationship, condition (ACP–3.1) predicates that nodes in a structural relationship have an ordering relationship, i.e. the antecedent of condition (ACP–3.1) is concerned only with the tree structure, while the consequent only with the total order. We will take advantage of this fact when proving theorems and when discussing the algorithms concerning projectivity of dependency trees further below.

Proof of Theorem 3.1.1. (Harper & Hays) $\Rightarrow$ (ACP–3.1): We proceed by contradiction. Let us suppose that the antecedent in the implication in condition (ACP–3.1) holds, but the consequent does not, and arrive at a contradiction with condition (Harper & Hays).

First let us discuss the case where $j_2 = u_2 = i$. Let us suppose w.l.o.g. that $j_1 \prec j_2$ (we can proceed using duality in the opposite case). Then by assumption $u_1 \succ u_2 = j_2$, and hence $u_1 \neq j_1$. As $u_1 \in \mathsf{Subtree}_{j_1}$, there is a non-trivial path $y_0 = j_1, y_1, \ldots, y_k = u_1$, $k > 0$, from j_1 to u_1. Let l be the smallest integer such that $y_l \prec j_2 \prec y_{l+1}$. Edge $y_l \rightarrow y_{l+1}$ and node j_2 are in contradiction with condition (Harper & Hays).

Now let us discuss the remaining case, i.e. suppose that $\mathrm{Sibl}(j_1, j_2, i)$ & $u_2 \in \mathsf{Subtree}_{j_2}$. Using the symmetry of predicate Sibl in its first two arguments and duality for $\preceq$, let us assume w.l.o.g. that $j_1 \prec j_2$ and $u_1 \neq j_1$ (from the assumption that the consequent of (ACP–3.1) does not hold, for at least one node u_k it has to hold that $u_k \neq j_k$, $k = 1, 2$). There are two cases:

- $u_1 \succ j_2$: Then there is a non-trivial path $y_0 = j_1, y_1, \ldots, y_k = u_1$, $k > 0$, from j_1 to u_1. Let l be the smallest integer such that $y_l \prec j_2 \prec y_{l+1}$. Edge $y_l \rightarrow y_{i+1}$ and node j_2 are in contradiction with condition (Harper & Hays).

- $u_1 \prec j_2$: By assumption we have $u_2 \prec u_1$, thus $u_2 \neq j_2$ and there is a non-trivial path $z_0 = j_2, z_1, \ldots, z_m = u_2$, $m > 0$, from j_2 to u_2. Let n be the smallest integer such that $z_{n+1} \prec u_1 \prec z_n$. Edge $z_n \rightarrow z_{n+1}$ and node u_1 are in contradiction with condition (Harper & Hays).

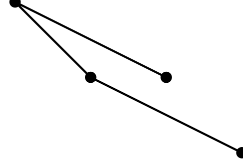


Figure 3.1: Sample non-projective dependency tree satisfying condition (†)

(ACP–3.1) ⇒ (Harper & Hays): Let us again proceed by contradiction. Using duality for $\preceq$, let us w.l.o.g. suppose that for nodes of edge $i \to j$ it holds that $i \prec j$, v is such a node that $i \prec v \prec j$, and $v \notin \mathsf{Subtree}_i$. There are two cases to be discussed:

- $i \in \mathsf{Subtree}_v$: Then there is a non-trivial path $x_0 = v, x_1, \ldots, x_k = i$, $k > 0$, from v to i, with two possibilities: if $x_1 \prec v$, then the assumption $j \succ v$ is in contradiction with (ACP–3.1); if $v \prec x_1$, then the assumption $v \succ i$ is in contradiction with (ACP–3.1).

- $i \notin \mathsf{Subtree}_v$: Let a be the lowest common ancestor of i and v. Then there are non-trivial paths $y_0 = a, y_1, \ldots, y_l = i$, $l > 0$, from a to i and $z_0 = a, z_1, \ldots, z_m = v$, $m > 0$, from a to v. If $y_1 \prec z_1$, then the assumption $j \succ v$ is in contradiction with (ACP–3.1); if $y_1 \succ z_1$, then the assumption $i \prec v$ is in contradiction with (ACP–3.1).

This finishes the proof. □

Remark. After giving the proof of the equivalence of condition (ACP–3.1) and the conditions in Definition 2.1.1, let us remark that the preliminary results concerning this approach to projectivity presented in [Veselá et al., 2004], [Hajičová et al., 2004], and [Veselá and Havelka, 2003] contain an error: the condition presented therein and claimed to be equivalent to the conditions discussed above is in fact weaker; the sketches of the algorithm for projectivization, which will be discussed in detail below, are correct.

The condition presented in the articles has the following form

$$(\dagger) \qquad (\forall i, j, v \in V)\Big(\big(i \to j \;\&\; j \prec i \;\&\; v \in \mathsf{Subtree}_j \Longrightarrow v \prec i\big) \\ \&\; \big(i \to j \;\&\; j \succ i \;\&\; v \in \mathsf{Subtree}_j \Longrightarrow v \succ i\big)\Big)\,.$$

We do not prove formally that condition (†) is weaker than the above presented conditions of projectivity; it follows easily from comparing condi-

tion (†) with condition (ACP–3.1). For a minimal counterexample, see Figure 3.1: it is easy to verify that the sample dependency tree satisfies condition (†), but is non-projective according to the conditions in Definition 2.1.1 and condition (ACP–3.1).

3.2 Projective orderings of a rooted tree

In this section, we address the issue of what possible projective orderings of a given rooted tree there are. Using condition (ACP–3.1), we derive a characterization of all projective total orders on a given rooted tree.

First, we introduce the notion of canonical projectivization of a dependency tree, which is defined as a projective dependency tree naturally related to a given (possibly non-projective) dependency tree. Then, we generalize it to the notion of a general projectivization of a rooted tree. We prove the existence and uniqueness of a general projectivization of a rooted tree, which gives as a corollary the existence and uniqueness of the canonical projectivization of a dependency tree. Using these results, we then characterize all projective total orders on a rooted tree.

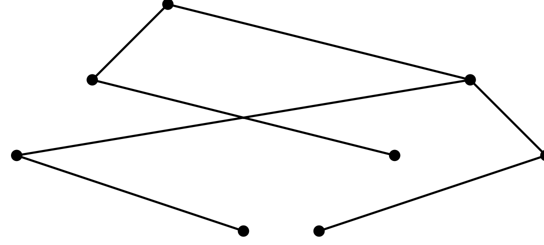
3.2.1 Definition. For a dependency tree $T = (V, \to, \preceq)$, we call a tree $T^{\mathrm{can}} = (V, \to, \leq)$ the *canonical projectivization* of T if it is projective and the following condition holds

$$(\mathsf{CanP}\text{–}3.2) \quad (\forall i, j_1, j_2 \in V)$$
$$\Big(\big[i \to j_1 \;\&\; (j_2 = i \vee \mathrm{Sibl}(j_1, j_2, i)) \big] \implies [j_1 \prec j_2 \Leftrightarrow j_1 < j_2] \Big) .$$

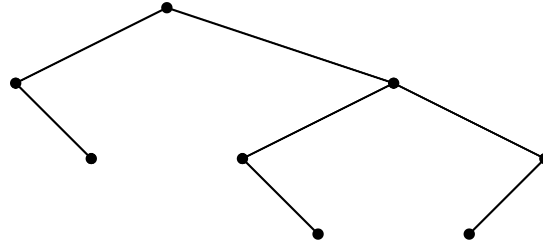
In the canonical projectivization of a dependency tree, its original total order gets possibly modified in such a way that a projective total order is obtained and at the same time for all nodes the original relative ordering of the node and its child nodes is preserved.

For an example of a canonical projectivization, see Figure 3.2. The projective dependency tree in figure (b) is the canonical projectivization of the non-projective dependency tree in figure (a).

Condition (CanP–3.2) can be generalized so as to take any total order for each node and its child nodes (the orders induced on these sets by the total order on all nodes of a dependency tree are a special case). The following definitions formalize this idea for rooted trees supplemented with *local* total orders.



(a) non-projective dependency tree



(b) its canonical projectivization

Figure 3.2: Example of canonical projectivization

3.2.2 Definition. Let $T = (V, \rightarrow)$ be a rooted tree. For every node $i \in V$, let $\preceq_i$ be a total order on the set $L_i = \{i\} \cup \{v \in V \mid i \rightarrow v\}$, the *local tree* of node i. We call any such total order $\preceq_i$, $i \in V$, a *local order* of the local tree of node i, and the set $\{\preceq_i \mid i \in V\}$ of local orders for all nodes a *local ordering* of the rooted tree T. We denote the local ordering induced by $\mathcal{L}$ on a subtree T_i as $\mathcal{L}_i = \{\prec_v \mid v \in \mathsf{Subtree}_i\}$.

3.2.3 Definition. Let $T = (V, \rightarrow)$ be a rooted tree and $\mathcal{L} = \{\preceq_i \mid i \in V\}$ a local ordering of T. We call a dependency tree $T^{\text{gen}_\mathcal{L}} = (V, \rightarrow, \leq)$ the *general projectivization* of T with respect to $\mathcal{L}$ if it is projective and the following condition holds

(GenP–3.3) $(\forall i, j_1, j_2 \in V)$

$$\Big(\big[i \rightarrow j_1 \;\&\; (j_2 = i \vee \mathrm{Sibl}(j_1, j_2, i))\big] \implies [j_1 \prec_i j_2 \Leftrightarrow j_1 < j_2]\Big) \,.$$

Note that the rooted tree T and the dependency tree $T^{\text{gen}_\mathcal{L}}$ share the set of nodes V and dependency relation $\rightarrow$.

Using these notions we can present a theorem stating the existence and uniqueness of a general projectivization of a rooted tree. The existence and

uniqueness of canonical projectivization of a dependency tree is just its corollary.

3.2.4 Theorem. *Let $T = (V, \to)$ be a rooted tree and $\mathcal{L} = \{\preceq_i \mid i \in V\}$ a local ordering of T. Then the general projectivization $T^{\mathrm{gen}_\mathcal{L}} = (V, \to, \preccurlyeq)$ exists and is unique. Furthermore, for every node i in T, the subtree $T_i^{\mathrm{gen}_\mathcal{L}}$ of $T^{\mathrm{gen}_\mathcal{L}}$ rooted in node i is the general projectivization with respect to $\mathcal{L}_i$ of the subtree T_i of T rooted in node i (i.e., the total orders of $(T^{\mathrm{gen}_\mathcal{L}})_i$ and $(T_i)^{\mathrm{gen}_{\mathcal{L}_i}}$ are isomorphic).*

Proof. We proceed by induction on the height n of rooted trees. Throughout the proof, we say that a node $i \in V$ satisfies the *projectivization condition for subtrees* if for all nodes $u \in \mathsf{Subtree}_i$ it holds that the total orders of $(T^{\mathrm{gen}_{\mathcal{L}_i}})_u$ and $(T_u)^{\mathrm{gen}_{\mathcal{L}_u}}$ are isomorphic.

First let us prove the basis step: A rooted tree of height $n = 0$ consists of a single node; there is only one total order on one node, which corresponds at the same time to the general projectivization of the rooted tree. Since in a one-node tree there are no proper subtrees, the projectivization condition for subtrees is vacuously true.

Now let us prove the induction step: Suppose that for every rooted tree of height at most n there is a unique general projectivization given any local ordering and the projectivization condition for subtrees holds. Let $T = (V, \to, \preceq)$ be a rooted tree of height $n+1$ and let $\mathcal{L} = \{\preceq_i \mid i \in V\}$ be a local ordering of T.

Let $i_1, \ldots, i_m$, $m > 0$, be the child nodes of the root node r of T ordered as follows

$$i_1 \prec_r \cdots \prec_r i_j \prec_r r \prec_r i_{j+1} \prec_r \cdots \prec_r i_m, \ 1 \leq j \leq m \ .$$

From the induction hypothesis there are unique general projectivizations $T_{i_k}^{\mathrm{gen}_{\mathcal{L}_{i_k}}}$ of subtrees rooted in i_k, $1 \leq k \leq m$, which satisfy the projectivization condition for subtrees. Let us consider a dependency tree $T' = (V, \to, \leq)$, where $\leq$ is obtained by concatenating the general projectivizations of the subtrees (i.e. the total orders on their nodes) and the root r in the following order

$$T_{i_1}^{\mathrm{gen}_{\mathcal{L}_{i_1}}}, \ldots, T_{i_j}^{\mathrm{gen}_{\mathcal{L}_{i_j}}}, r, T_{i_{j+1}}^{\mathrm{gen}_{\mathcal{L}_{i_{j+1}}}}, \ldots, T_{i_m}^{\mathrm{gen}_{\mathcal{L}_{i_m}}} \ .$$

From the definition of $\leq$ tree T' satisfies the condition (ACP–3.1) for its root r, and from the induction hypothesis the condition also holds for all other nodes in T', hence T' is projective. Since $\leq$ is defined in the only way in which condition (GenP–3.3) can be satisfied for $i = r$, tree T' is uniquely

determined. Obviously, T' also satisfies the projectivization condition for subtrees. This finishes the proof that $T' = T^{\text{gen}_\mathcal{L}}$. □

We obtain easily as a corollary the following result for the canonical projectivization of a dependency tree.

3.2.5 Corollary. *For every dependency tree $T = (V, \rightarrow, \preceq)$, the canonical projectivization $T^{\text{can}} = (V, \rightarrow, \leq)$ of T exists and is unique. Furthermore, for every node i in T, the subtree T_i^{can} of T^{can} rooted in i is the canonical projectivization of the subtree T_i of T rooted in node i (i.e., the total orders of $(T^{\text{can}})_i$ and $(T_i)^{\text{can}}$ are isomorphic).*

Proof. Observe that condition (CanP–3.2) is equivalent to condition (GenP–3.3) with local ordering $\{\preceq \upharpoonright L_i \mid i \in V\}$, where $L_i = \{i\} \cup \{v \in V \mid i \rightarrow v\}$ is the local tree of node i. The statement thus follows from Theorem 3.2.4. □

We are ready to derive the characterization of all projective total orders on a rooted tree. By a *projective* total order on a rooted tree $T = (V, \rightarrow)$ we mean any total order $\preceq$ on V for which the dependency tree $T' = (V, \rightarrow, \preceq)$, obtained by supplementing the rooted tree T with the total order $\preceq$, is projective.

3.2.6 Theorem. *Every projective total order on a dependency tree $T = (V, \rightarrow)$ corresponds uniquely to a local ordering of T.*

Proof. We will show that there is a one-to-one correspondence between projective total orders on T and local orderings of T.

Theorem 3.2.4 states that for every local ordering $\mathcal{L} = \{\preceq_i \mid i \in V\}$ of T there is a unique general projectivization $T^{\text{gen}_\mathcal{L}}$ of T, i.e. a projective total order on T satisfying condition (GenP–3.3) with respect to $\mathcal{L}$. For different local orderings the corresponding projective total orders obviously also differ.

Each projective order $\preceq$ on T specifies a local ordering $\mathcal{L} = \{\preceq \upharpoonright L_i \mid i \in V\}$; we know from Theorem 3.2.4 that $\preceq$ is isomorphic to the projective total order of the corresponding general projectivization. We will show that for two differing projective total orders $\preceq_1, \preceq_2$ on T, the local orderings $\mathcal{L}_1, \mathcal{L}_2$ corresponding to $\preceq_1, \preceq_2$, respectively, also differ.

Let $u_1, u_2 \in V$ be nodes such that $u_1 \prec_1 u_2$ and $u_1 \succ_2 u_2$. Let us consider their lowest common ancestor i. Then there are two possible cases:

- Node i is disjoint from nodes u_1, u_2, i.e., $u_1 \neq i \neq u_2$. Let j_1, j_2 be ancestors of u_1, u_2, respectively, such that $i \rightarrow j_1$ and $i \rightarrow j_2$. Since both total orders $\preceq_i, \preceq_2$ are projective, according to condition (ACP–3.1)

it holds that $j_1 \prec_1 j_2$ and $j_1 \succ_2 j_2$, and hence the local orderings $\mathcal{L}_1 = \{\preceq_1 \upharpoonright L_i \mid i \in V\}$, $\mathcal{L}_2 = \{\preceq_2 \upharpoonright L_i \mid i \in V\}$ differ in the local orders for the local tree L_i.

- Node i is equal to one of nodes u_1, u_2, w.l.o.g. assume that $i = u_2$. Let j_1 be an ancestor of u_1 such that $i \to j_1$. Since both total orders $\preceq_i, \preceq_2$ are projective, according to condition (ACP–3.1) it holds that $j_1 \prec_1 i$ and $j_1 \succ_2 i$, and hence the local orderings $\mathcal{L}_1, \mathcal{L}_2$ differ in their local orders for the local tree L_i.

This finishes the proof. □

3.3 Algorithm for projectivizing

In this section, we present an algorithm for computing general projectivizations. A high-level sketch of the algorithm is given as Algorithm 2.

Algorithm 2 Projectivize – high-level sketch

Input: rooted tree T, local ordering $\mathcal{L}$ of T
Output: general projectivization of T with respect to local ordering $\mathcal{L}$
1: **for** each node i in a general post-order traversal **do**
2: concatenate i and subtrees rooted in its child nodes according to the local order for i in $\mathcal{L}$
3: **end for**

The algorithm returns for an input rooted tree T and a local ordering $\mathcal{L}$ of T its general projectivization with respect to $\mathcal{L}$. The projective total order on nodes is constructed recursively using local orders.

We assume that local trees can be processed in linear time according to the corresponding local orders in a given local ordering; we do not assume any particular data representation of local orders. In our data representation, this is guaranteed for the special case of canonical projectivization, where we assume that local orders are represented by pointer fields `left_child`, `right_child`, and `sibling`, thanks to our requirement on the data representation of dependency trees.

Algorithm 3 presents a detailed version of the algorithm. For the resulting total order it uses pointer fields `prev` and `next`. It consists of two subsequent loops over all nodes, the first one on lines 1–18 is the main one, performing the projectivization itself, the second one on lines 19–22 is just an auxiliary loop for deleting auxiliary pointers `left_span` and `right_span` for the leftmost

Algorithm 3 Projectivize

Input: rooted tree T, local ordering $\mathcal{L}$ of T

Output: general projectivization of T with respect to local ordering $\mathcal{L}$

1: **for** each node i in a general post-order traversal **do**
    ▷ *first process left child nodes*
2:     `left_span`$[i] \leftarrow i$ ▷ *initialize auxiliary pointer*
3:     $d \leftarrow i$ ▷ *auxiliary node variable for creating projective total order*
4:     **for** each left child c_{left} of i inversely to local order for i in $\mathcal{L}$ **do**
5:         `prev`[`left_span`$[d]$] $\leftarrow$ `right_span`$[c_{\text{left}}]$
6:         `next`[`right_span`$[c_{\text{left}}]$] $\leftarrow$ `left_span`$[d]$
7:         $d \leftarrow c_{\text{left}}$
8:     **end for**
9:     `left_span`$[i] \leftarrow$ `left_span`$[d]$
    ▷ *analogously process right child nodes*
10:     `right_span`$[i] \leftarrow i$ ▷ *initialize auxiliary pointer*
11:     $d \leftarrow i$ ▷ *auxiliary node variable*
12:     **for** each right child c_{right} of i according to local order for i in $\mathcal{L}$ **do**
13:         `next`[`right_span`$[d]$] $\leftarrow$ `left_span`$[c_{\text{right}}]$
14:         `prev`[`left_span`$[c_{\text{right}}]$] $\leftarrow$ `right_span`$[d]$
15:         $d \leftarrow c_{\text{right}}$
16:     **end for**
17:     `right_span`$[i] \leftarrow$ `right_span`$[d]$
18: **end for**
19: **for** each node i **do** ▷ *loop for deleting auxiliary pointers*
20:     delete `left_span`$[i]$
21:     delete `right_span`$[i]$
22: **end for**

and rightmost nodes in the subtree of a node, i.e. for the span of the subtree. In the main loop, the algorithm processes nodes in any general post-order traversal; cf. Section 1.3. For the data representation used in the detailed versions of algorithms and the requirement on it, cf. Section 1.2.

3.3.1 Theorem. *Algorithm 3 returns for a rooted tree T and a local ordering $\mathcal{L}$ of T the general projectivization $T^{\text{gen}_{\mathcal{L}}}$ of T with respect to $\mathcal{L}$. Its time complexity is $O(n)$. (We assume that local trees can be processed in linear time according to the corresponding local orders.)*

Proof. First let us prove the correctness of the algorithm, i.e. that for any

input rooted tree T and a local ordering $\mathcal{L}$ of T it returns general projectivization $T^{\mathrm{gen}_{\mathcal{L}}}$ of T with respect to $\mathcal{L}$.

We will prove by induction the following invariant for the main loop on lines 1–18: after processing each node, the pointers `prev` and `next` of all nodes in the subtree rooted in this node represent the general projectivization of the subtree with respect to $\mathcal{L}$, the doubly linked list representing the total order on the subtree is contiguous, and the pointers `left_span` and `right_span` contain the leftmost and rightmost nodes of the subtree, respectively.

Let us prove the invariant by discussing the main loop in detail. As it consists of two analogous sub-blocks, processing left and right child nodes of node i, we will go line by line only through the sub-block processing the left child nodes (lines 2–9); the processing of the right child nodes (lines 10–17) is analogous, using duality for total order on nodes.

First, the auxiliary pointer `left_span` of the processed node i is initialized to the node itself (line 2) and the auxiliary pointer d is initialized to i (line 3). The pointer d is used in the loop over left child nodes of node i (lines 4–8) for setting the pointers `prev` and `next` of the doubly linked list representing the total order on nodes.

The doubly linked list representing the total order on the nodes of the subtree rooted in i is created using the information already stored at the child nodes of i (when processing i, all its child nodes have already been processed thanks to the general post-order traversal, and so by induction satisfy the invariant). In the loop over left child nodes of i, the child nodes are processed according to the data representation (and therefore reversely with respect to the total order on nodes). On lines 5 and 6, the doubly linked lists of the subtrees rooted in c_{left} and d are concatenated (in the first pass, d is equal to i, thus i gets correctly concatenated with the subtree of its first left child thanks to the initialization of `left_span` on line 2). On line 7, the auxiliary pointer d is reset. After processing all left child nodes of i, the auxiliary pointer d contains the leftmost child of i, and therefore line 9 sets the `left_span` pointer correctly for i (if i does not have any left child nodes, `left_span` is set correctly thanks to the initialization on line 2).

After the iteration of the main loop for node i, the subtree is projective according to condition (ACP–3.1), and since it respects the local order for i in $\mathcal{L}$, it is the general projectivization of the subtree rooted in i with respect to $\mathcal{L}$. Furthermore, the doubly linked list representing the total order is constructed in such a way (subtrees are concatenated) that the total order on the subtree rooted in i is contiguous. Thus we have proved the invariant for the algorithm, and so its correctness.

In the auxiliary loop on lines 19–22, the auxiliary pointers `left_span` and `right_span` are deleted for all nodes.

Now let us show that the time complexity of the algorithm is $O(n)$.

There are two consecutive loops in the algorithm: the main loop on lines 1–18 and the auxiliary loop on lines 19–22. The time complexity of the auxiliary loop is obviously $O(n)$. Therefore it remains to show the time complexity of the main loop.

The main loop is processed exactly as many times as there are nodes in the tree. To see that the sub-loops for processing left and right child nodes (lines 4–8 and 12–16, respectively) have overall linear time complexity, let us observe that the two loops together are processed exactly as many times as there are nodes (except the root). (Every node except the root node is a child node of some other node, and therefore it is processed exactly once by one of these loops). Hence the time complexity of the main loop is $O(n)$.

The overall time complexity of the algorithm is therefore $O(n)$ and the proof is finished. □

The auxiliary pointers `left_span` and `right_span` can be deleted already in the main loop, but this would only unnecessarily complicate the presented algorithm due to technicalities.

3.4 Using the algorithm for checking projectivity

Algorithm 3 can also serve for linear-time checking of projectivity of a dependency tree. We just have to compare the total orders of the original tree and the projectivized one. If they differ, the original tree is non-projective.

The main disadvantage of this way of checking projectivity is that for a non-projective tree, we do not learn much about the causes of its non-projectivity. If we are interested in which edges are non-projective, we have to use some other algorithm, e.g. the simple Algorithm 1 for finding non-projective edges.

In the next section, we introduce level types of non-projective edges and we also present an efficient algorithm for finding non-projective edges of non-negative level types. We also give a hint on using its output if we want to find all non-projective edges, i.e. also non-projective edges of negative level types.

Chapter 4

Level types of non-projective edges

We introduce level types of non-projective edges, which combine properties of their gaps with levels of nodes. A fundamental property of level types of non-projective edges allows us to derive another characterization of the condition of projectivity. We present a linear algorithm for finding non-projective edges of non-negative level types (the ones that characterize non-projectivity), which will be the basis of efficient algorithms for checking planarity and well-nestedness presented in subsequent chapters.

4.1 Basic properties of level types and their relationship to projectivity

We discuss the relationship of properties of non-projective edges, levels of nodes, and projectivity. Level types of non-projective edges give yet another characterization of projectivity.

4.1.1 Definition. The *level type* of a non-projective edge $i \leftrightarrow j$ in a dependency tree $T = (V, \rightarrow, \preceq)$ is defined as follows

$$\mathsf{Type}_{i\leftrightarrow j} = \mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} - \min_{u \in \mathsf{Gap}_{i\leftrightarrow j}} \mathsf{level}_u \ .$$

Level type of an edge is the distance of its child node and a node in its gap closest to the root (distance here means relative difference in levels). Note that there may be more than one node witnessing an edge's level type. Level type of an edge is not bounded—it can take any integer value. In any given dependency tree, however, it is bounded by the height of the dependency tree.

We use the word *witness* in two slightly different meanings, which will be clear from context: (a) when talking about nodes testifying the exact value of level type of an edge, and (b) when talking about nodes testifying a lower bound on the level type of an edge.

4.1.2 Observation. *For any non-projective edge $i \leftrightarrow j$ in a dependency tree $T = (V, \to, \preceq)$ it holds*

$$\mathsf{Type}_{i \leftrightarrow j} = \max_{u \in \mathsf{Gap}_{i \leftrightarrow j}} (\mathsf{level}_{\mathsf{Child}_{i \leftrightarrow j}} - \mathsf{level}_u) \ .$$

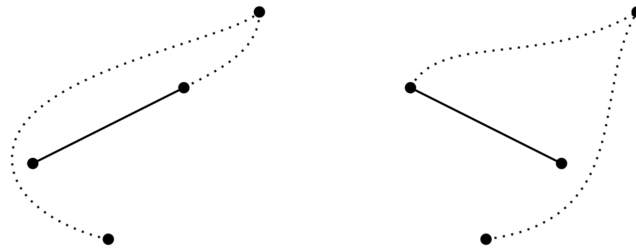
Figure 4.1 schematically shows sample configurations in dependency trees of different level types of non-projective edges. All edges are examples of edges of the corresponding level type, but let us stress that they do not represent all possible configurations with non-projective edges.

Let us look in more detail at the first configuration with a non-projective edge of level type 0 in figure (b): the first edge on the upward path from the maximal node in the gap of the non-projective edge (this edge is not shown explicitly) is also non-projective, of level type 1. The two edges can be said to “cause” each other's non-projectivity. As this simple example shows, relationships between non-projective edges in a dependency tree can be rather complex.

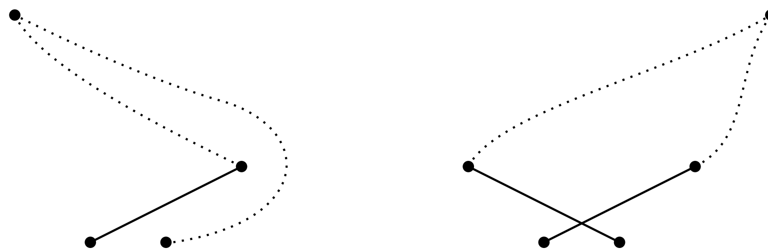
It is easy to see that a non-projective edge of negative level type can occur only in dependency trees with height at least 3 and at least 6 nodes—see Figure 4.2 with a sample minimal dependency tree with a non-projective edge of level type -1. Note that the sample dependency tree contains also a non-projective edge of level type 1. This is in fact an example of a general property of non-projective edges of non-positive level type.

Remark. Algorithm 1 for determining gaps of non-projective edges can be easily modified to compute also the level types of non-projective edges. The level types of non-projective edges can be, e.g., determined on the fly in the inner loop on lines 2–4; or they can be computed only after all gaps have been fully determined. The time complexity bound for the modified algorithm remains quadratic.

We now show a fundamental property of level types of non-projective edges: In a dependency tree, the presence of a non-projective edge of non-positive level type implies the presence of a non-projective edge of non-negative level type. The sample configurations with non-projective edges of negative and zero level types in Figure 4.1 illustrate well the proof.



(a) negative level type



(b) level type 0

(c) positive level type

Figure 4.1: Sample configurations with non-projective edges of negative, zero, and positive level types

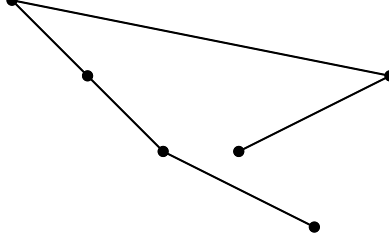


Figure 4.2: Sample minimal non-projective dependency tree with a non-projective edge of level type -1

4.1.3 Theorem. *Let $i \leftrightarrow j$ be a non-projective edge in a dependency tree $T = (V, \rightarrow, \preceq)$ of non-positive level type, i.e., $\mathsf{Type}_{i\leftrightarrow j} \leq 0$. Then for each edge $v \rightarrow u$ in T such that $u \in \arg\min_{n \in \mathsf{Gap}_{i\leftrightarrow j}} \mathsf{level}_n$ it holds that one of the endpoints of edge $i \leftrightarrow j$ (i.e., either i, or j) is in $\mathsf{Gap}_{u\leftrightarrow v}$ and it witnesses that*

$$\mathsf{Type}_{u\leftrightarrow v} \geq -\mathsf{Type}_{i\leftrightarrow j} \ .$$

Proof. Let u be any node in $\arg\min_{n \in \mathsf{Gap}_{i\leftrightarrow j}} \mathsf{level}_n$. From the assumption that $\mathsf{Type}_{i\leftrightarrow j} \leq 0$, node u has a parent node v, which satisfies $v \notin \mathsf{Gap}_{i\leftrightarrow j}$. Obviously, edges $i \leftrightarrow j$, $v \rightarrow u$ are disjoint, thus from Proposition 2.2.4 we have that $v \notin [i, j]$, and so either $i \in (u, v)$, or $j \in (u, v)$. Since $\mathsf{level}_v \geq \mathsf{level}_{\mathsf{Parent}_{i\leftrightarrow j}}$, we have that $\mathsf{Parent}_{i\leftrightarrow j} \notin \mathsf{Subtree}_v$, and so either $i \in \mathsf{Gap}_{u\leftrightarrow v}$, or $j \in \mathsf{Gap}_{u\leftrightarrow v}$. Immediately from definition we obtain that $\mathsf{Type}_{u\leftrightarrow v} \geq \mathsf{level}_u - \mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} = -\mathsf{Type}_{i\leftrightarrow j}$. The simple facts that $\mathsf{level}_i \leq \mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}}$, $\mathsf{level}_j \leq \mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}}$ imply that the endpoint of edge $i \leftrightarrow j$ in the span of edge $v \rightarrow u$ indeed witnesses the inequality for the level types, which finishes the proof. $\square$

From Theorem 4.1.3 we get as a corollary the following important theorem, giving a characterization of projectivity of a dependency tree in terms of presence of non-projective edges of non-negative level types. It also allows us to check projectivity of a tree by looking only for non-projective edges of non-negative level types, which will be important in algorithms we present further below.

4.1.4 Theorem. *A dependency tree is projective if and only if it contains no non-projective edges of non-negative level type.*

Proof. Let T be an arbitrary non-projective tree. According to Theorem 4.1.3,

if T contains a non-projective edge of negative level type, then it also contains a non-projective edge of positive level type. Hence every non-projective tree contains at least one non-projective edge of non-negative level type, which is equivalent to the statement of the theorem. □

The previous theorem reveals the importance of non-projective edges of non-negative level types. It will show also later in the algorithms utilizing properties of non-projective edges. The following notion of upper gap straightforwardly incorporates non-negative level types into the notion of gap.

4.1.5 Definition. For any edge $i \leftrightarrow j$ in a dependency tree $T = (V, \rightarrow, \preceq)$ we define its *upper gap* as follows

$$\mathsf{Gap}^{\uparrow}_{i\leftrightarrow j} = \{v \in V \mid v \in (i,j) \ \&\ v \notin \mathsf{Subtree}_{i\leftrightarrow j} \ \&\ \mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} \geq \mathsf{level}_v\} \ .$$

4.1.6 Observation. *Let $i \leftrightarrow j$ be an edge in a dependency tree $T = (V, \rightarrow, \preceq)$. Then $\mathsf{Gap}^{\uparrow}_{i\leftrightarrow j} \neq \emptyset$ if and only if edge $i \leftrightarrow j$ is non-projective of non-negative level type.*

The notion of upper gap of an edge leads to the following reformulation of Theorem 4.1.4.

4.1.7 Theorem. *A dependency tree $T = (V, \rightarrow, \preceq)$ is non-projective if and only if $\mathsf{Gap}^{\uparrow}_{i\leftrightarrow j} \neq \emptyset$ for some edge $i \leftrightarrow j$ in T.*

4.2 Algorithm for finding non-projective edges of non-negative level type

We present a linear algorithm for finding non-projective edges of non-negative level types. From Theorem 4.1.4 we obtain that finding non-projective edges of non-negative level types is sufficient for checking projectivity. At the end of this section we suggest how the output of the algorithm can be used to find also non-projective edges of negative level type.

Algorithm 4 is a high-level sketch of the algorithm. It processes the input dependency tree by levels bottom up. For all nodes on the processed level, it goes through all edges going down from the nodes (i.e. through their child nodes) and checks against the total order on nodes whether there is a node causing some non-projectivity of non-negative level type. After processing all edges with parent nodes on one level, the nodes below this level are deleted from the total order on nodes. This is in fact the crucial point that

Algorithm 4 Find non-projective edges of non-negative level types – high-level sketch

Input: dependency tree T
Output: non-projective edges of non-negative level type in T

1: **for** each level of nodes in T bottom up **do**
2: **for** each edge $i \rightarrow c$ with parent node i on processed level **do**
3: check $\mathsf{Gap}^{\uparrow}_{i \rightarrow c} \neq \emptyset$ using nodes on the same level as or above c
4: **end for**
5: delete all nodes below processed level from the total order on T
6: **end for**

allows the check for projectivity to be performed without explicitly checking subordination. Note that the algorithm does not determine the exact level types of returned non-projective edges.

Algorithm 5 presents a detailed version of the algorithm. For the sake of simplicity of exposition, the algorithm as presented here is destructive in the sense that it destroys the original total order on nodes of the tree. This can be easily remedied either by working with a copy of the whole input tree, or by working with copies of the pointers `prev` and `next` representing the total order for all nodes of the tree.

4.2.1 Theorem. *Algorithm 5 returns for any dependency tree T the set of all non-projective edges of non-negative level types occurring in T, and its time complexity is $O(n)$.*

Proof. First let us prove the correctness of the algorithm, i.e. that for any input dependency tree it returns exactly all its non-projective edges of non-negative level type.

We will prove the following invariant holding after processing each level of the input tree: the algorithm finds all non-projective edges of non-negative level type whose parent nodes are on this level, all nodes on lower levels are deleted from the total order, and all non-projective edges of non-negative level type whose parent nodes are on higher levels are preserved in the total order (restricted to the nodes on the processed and higher levels).

The algorithm processes nodes by levels bottom-up. Lines 1, 2 and 3 represent this traversal decomposed into two embedded loops, the outer one over levels, the inner one over nodes on individual levels.

In order to prove the invariant, we have to discuss the main loop on lines 2–30. For nodes on the lowest level the invariant is vacuously true

Algorithm 5 Find non-projective edges of non-negative level types

Input: dependency tree T

Output: non-projective edges of non-negative level types in T

```
 1: l_max ← maximal level in T
 2: for l = l_max, ..., 0 do                                   ▷ main loop
 3:     for each node i on level l do                          ▷ loop over nodes on one level
                                                               ▷ first process left child nodes
 4:         d ← i            ▷ auxiliary variable for keeping the previously processed node
 5:         p ← true                       ▷ suppose that edges to the left are projective
 6:         for each left child node c_left of i inversely to total order do
 7:             if (c_left = prev[d] & p) then                 ▷ edge is projective
 8:                 d ← c_left                                 ▷ keep the processed node
 9:             else             ▷ we found a non-projectivity and mark remaining edges
10:                 p ← false                         ▷ important only in the first pass
11:                 mark c_left as non-projective
12:             end if
13:         end for
                                                  ▷ analogously process right child nodes
14:         d ← i            ▷ auxiliary variable for keeping the previously processed node
15:         p ← true                      ▷ suppose that edges to the right are projective
16:         for each right child node c_right of i according to total order do
17:             if (c_right = next[d] & p) then                ▷ edge is projective
18:                 d ← c_right
19:             else             ▷ we found a non-projectivity and mark remaining edges
20:                 p ← false
21:                 mark c_right as non-projective
22:             end if
23:         end for
24:     end for
                                         ▷ delete nodes on level l + 1 from the total order
25:     for each node i on level l do
26:         for each child node c of i do
27:             delete c from the total order on T
28:         end for
29:     end for
30: end for
```

(there are no edges with parent nodes on the lowest level of a tree), which proves the basis step.

Now let us prove the induction step: Let us suppose that we process nodes on level l. From the induction hypothesis we get that in the tree there are only nodes on levels $0, \ldots, l+1$ left (in the previous iteration of the main loop nodes on level $l+1$ were processed, and so all nodes on lower levels got deleted) and all non-projective edges of non-negative level type whose parent nodes are on levels $0, \ldots, l$ are preserved.

The main loop contains two sub-loops: the loop on lines 3–24 finds the non-projective edges and the loop on lines 25–29 deletes the nodes on the level below the processed one from the total order on nodes.

Now let us discuss in detail the loop for finding non-projective edges of non-negative level types. It goes through all nodes on the current level (recall that the order of processing is irrelevant) and for each node processes all edges going down from it.

Let us go line by line through the sub-block on lines 4–13 processing the left child nodes of i (i.e. edges going down and left from node i); the processing of the right child nodes of i (lines 14–23) is analogous, using duality for the total order on nodes.

The variable d initialized on line 4 is an auxiliary variable used when looking for a non-projectivity of non-negative level type. The variable p initialized on line 5 is a boolean auxiliary variable used for storing the information whether we already have found a non-projective edge (at first we suppose that the edges are projective).

The loop on lines 6–13 processes all left child nodes of i according to the data representation, i.e. inversely to the total order on nodes. The condition on line 7 is crucial: it checks whether the edge $i \to c_{\text{left}}$ has a non-empty upper gap. If $c_{\text{left}} = \texttt{prev}[d]$ holds (and we have not found a non-projective edge yet, i.e. p is true), we know that all left child nodes up the current child c_{left} form with i a contiguous interval in the total order on nodes. If $c_{\text{left}} \neq \texttt{prev}[d]$, there is some node v between d (either i itself, or the previous left child) and the currently processed child c_{left}: from the induction hypothesis node v is on level $l+1$ or higher, and obviously it is not subordinated to i, which means that the edge $i \to c_{\text{left}}$ is a non-projective edge of non-negative level type (its gap containing at least v). When we find a non-projective edge, then all remaining edges going down and left from i are also non-projective of non-negative level type (because their gaps contain as a subset the gap of the first found non-projective edge) – by setting the auxiliary variable p to false, the algorithm marks all remaining edges as non-projective in the subsequent iterations of the loop over left child nodes.

Obviously, in this way we find all non-projective edges of non-negative

level types whose parent nodes are on level l (using the induction hypothesis that all such non-projectivities are preserved for edges with parent nodes on levels $0, \ldots, l$).

The loop on lines 25–29 deletes the nodes on level $l+1$ from the doubly linked list representing the total order on nodes. Obviously, this does not destroy any non-projective edges of non-negative level type with parent nodes on levels $0, \ldots, l-1$, because any such non-projectivity involves nodes on the same or higher levels than the level of the child node of the non-projective edge. Thus all non-projectivities of non-negative level type with parent nodes on levels $0, \ldots, l-1$ are preserved, and since all nodes below level l are deleted from the total order on nodes, this finishes the proof of the invariant, and so proves the correctness of the algorithm.

Now let us show that the time complexity of the algorithm is $O(n)$:

The main loop on lines 2–30 contains two sub-loops on lines 3–24 and 25–29. These two sub-loops are processed exactly as many times as there are nodes in the tree. The innermost sub-loops contained in these two loops do not add to the overall complexity, because for both these loops they are processed exactly as many times as there are nodes (except the root node).

The overall time complexity of the algorithm is therefore $O(n)$ and the proof is finished. □

Let us remark that for Algorithm 5 to work it is essential to process the input dependency tree by levels bottom up. To be able to find all non-projective edges of non-negative level types, we can make do for any such edge with nodes on the same or higher levels as the child node of the edge; thus removing nodes on the already processed levels from the total order on nodes allows us not to perform explicit subordination checks.

No edge from the root can be non-projective (of any type), therefore the main loop of Algorithm 5 over levels on lines 2–30 need not process level 0, i.e. the root. This modification does not however affect the overall time complexity of the algorithm in the general case (although for trees of height 1 it indeed does).

Algorithm 5 does not return the gaps of the found non-projective edges of non-negative level type. If we want to fully determine the gaps, we can e.g. perform subordination checks for all nodes in the spans of the returned non-projective edges. In the next section, we show how the output of Algorithm 5 can be used to find all non-projective edges.

4.3 Using the algorithm for checking projectivity and for finding all non-projective edges

From Theorem 4.1.4 we know that for checking projectivity it suffices to look for non-projective edges of non-negative level type. Algorithm 5 finds non-projective edges of non-negative level type; however, it does not find non-projective edges of negative level type, nor it determines the exact level types of the found non-projective edges.

Algorithm 5 can therefore also serve for linear-time checking of projectivity of a dependency tree. In comparison with using an algorithm for projectivizing for this purpose (cf. Algorithm 3 in Section 3.3), it provides much more detailed information about the causes of non-projectivity: all non-projective edges of non-negative level type.

If we are interested in finding all non-projective edges, one way of looking for them is the simple quadratic algorithm, derived directly from the definition of gap of a non-projective edge (cf. Algorithm 1 in Section 2.3).

But we can do better: Theorem 4.1.3 gives us a useful hint where to look for non-projective edges of negative level type. Based on this hint, we can take some advantage of the output of Algorithm 5.

Let us suppose that for a dependency tree $T = (V, \rightarrow, \preceq)$ we know the set of all its non-projective edges of non-negative level type; let us denote it as

$$\mathsf{Nonneg} = \{u \leftrightarrow v \in T \mid \mathsf{Gap}^{\uparrow}_{u\leftrightarrow v} \neq \emptyset\} \ .$$

We will show that the following set contains all non-projective edges of negative level type:

$$\begin{aligned}\mathsf{CandNeg} = \{i \leftrightarrow j \in T \mid i \leftrightarrow j \notin \mathsf{Nonneg} \ \& \ (\exists\, v \rightarrow u \in T) & \\ (v \neq r \ \& \ u \in (i,j) \ \& \ i \in (u,v) & \\ \& \ \mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} < \mathsf{level}_u \leq \mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} + \mathsf{Type}_{u\leftrightarrow v})\} \ .\end{aligned}$$

To see that set CandNeg indeed contains all non-projective edges of negative level type, i.e.

$$\{i \leftrightarrow j \in T \mid \mathsf{Gap}_{i\leftrightarrow j} \neq \emptyset \ \& \ \mathsf{Type}_{i\leftrightarrow j} < 0\} \subseteq \mathsf{CandNeg} \ ,$$

observe that Theorem 4.1.3 tells us that for any non-projective edge of negative level type $i \leftrightarrow j$ there is a non-projective edge of positive level type $v \rightarrow u$ such that $u \in \arg\min_{n\in\mathsf{Gap}_{i\leftrightarrow j}} \mathsf{level}_n$ and $\mathsf{Type}_{u\leftrightarrow v} \geq -\mathsf{Type}_{i\leftrightarrow j}$, witnessed by an endpoint of edge $i \leftrightarrow j$ (recall also that any non-projective

edge is disjoint from the root node). An easy manipulation with the inequality for level types, using the fact that $\mathsf{Type}_{i\leftrightarrow j} = \mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} - \mathsf{level}_u$, gives $\mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} + \mathsf{Type}_{u\leftrightarrow v} \geq \mathsf{level}_u$. Since node u witnesses the negative level type of the non-projective edge $i \leftrightarrow j$, it obviously holds that $\mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} < \mathsf{level}_u$. This shows that any non-projective edge of negative level type in the dependency tree T satisfies the defining conditions of set $\mathsf{CandNeg}$, and thus belongs to it.

We now have a set with "candidate" non-projective edges of negative level type. For each edge in $\mathsf{CandNeg}$ we still have to verify whether it really is non-projective of negative level type. The reasoning in the previous paragraph shows that if an edge $i \leftrightarrow j \in \mathsf{CandNeg}$ is non-projective of negative level type, then the set

$$\begin{aligned}\mathsf{CandGap}_{i\leftrightarrow j} = \{u \in T \mid (\exists v \in T)(&v \rightarrow u \;\&\; v \neq r \;\&\; u \in (i,j)\\ &\&\; [i \in (u,v) \vee j \in (u,v)]\\ &\&\; \mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} < \mathsf{level}_u \leq \mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} + \mathsf{Type}_{u\leftrightarrow v})\}\end{aligned}$$

contains all nodes witnessing its exact level type (in other words, all nodes lying on the highest level in its gap). Therefore, if we only want to find out whether an edge $i \leftrightarrow j \in \mathsf{CandNeg}$ is non-projective of negative level type, it suffices to check that at least one node in the set $\mathsf{CandGap}_{i\leftrightarrow j}$ is not subordinated to edge $i \leftrightarrow j$; if we want to determine the exact level type of edge $i \leftrightarrow j$, we must check all nodes in $\mathsf{CandGap}_{i\leftrightarrow j}$.

Although Algorithm 5 returns all non-projective edges of non-negative level type, it does not specify level types of the non-projective edges. We can still use its output by considering the following modifications to the sets defined above:

Since we do not know the exact level types of edges of non-negative level types, we can consider the set

$$\begin{aligned}\mathsf{CandNeg}' = \{i \leftrightarrow j \in T \mid i \leftrightarrow j \notin \mathsf{Nonneg} \;\&\; (&\exists\, v \rightarrow u \in T)\\ &(v \neq r \;\&\; u \in (i,j) \;\&\; i \in (u,v)\\ &\&\; \mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} < \mathsf{level}_u)\}\ .\end{aligned}$$

which removes from the defining conditions the upper bound on levels of child nodes of considered edges $v \rightarrow u$. Obviously $\mathsf{CandNeg} \subseteq \mathsf{CandNeg}'$, so set $\mathsf{CandNeg}'$ contains all non-projective edges of negative level type.

Analogously, to determine whether a "candidate" edge $i \leftrightarrow j$ really is

non-projective of negative level type, we can consider the set of nodes

$$\begin{aligned}\mathsf{CandGap}'_{i\leftrightarrow j} = \{u \in T \mid (\exists v \in T)(v \to u \;\&\; v \neq r \;\&\; u \in (i,j) \\ \&\; [i \in (u,v) \vee j \in (u,v)] \\ \&\; \mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} < \mathsf{level}_u)\}\ .\end{aligned}$$

Again, since $\mathsf{CandGap}_{i\leftrightarrow j} \subseteq \mathsf{CandGap}'_{i\leftrightarrow j}$ for any edge $i \leftrightarrow j \in \mathsf{CandNeg}$, we can use the set $\mathsf{CandGap}'_{i\leftrightarrow j}$ to find out whether edge $i \leftrightarrow j$ is non-projective of negative level type and to determine its exact level type in the same way as described above.

We have only sketched a way of taking advantage of the output of Algorithm 5 for finding all non-projective edges. We do not give a detailed analysis of the time complexity of the resulting algorithm, but let us note that it is obviously $O(n^2)$.

4.4 Combining algorithms for finding non-projective edges of non-negative level types and for projectivizing

Algorithms 3 and 5 presented above can be straightforwardly combined, preserving the linear time complexity. A high-level sketch of the algorithm is presented as Algorithm 6.

Algorithm 6 Find non-projective edges of non-negative level type and projectivize – high-level sketch

Input: dependency tree T, local ordering $\mathcal{L}$ of T

Output: non-projective edges of non-negative level type in T, general projectivization of T with respect to local ordering $\mathcal{L}$

1: **for** each level of nodes in T bottom up **do**
2: **for** each edge $i \to c$ with parent node i on processed level **do**
3: check $\mathsf{Gap}^{\uparrow}_{i\to c} \neq \emptyset$ using nodes on the same level as or above c
4: **end for**
5: delete all nodes below processed level from original total order on T
6: concatenate i and subtrees rooted in its child nodes according to the local order for i in $\mathcal{L}$ to create new projective total order
7: **end for**

Algorithm 6 performs both detection of non-projective edges of non-negative level types and computes the general projectivization of the input

dependency tree. As the original total order gets destroyed during the detection of non-projective edges, the projective total order of the projectivization has to be returned in another doubly linked list (using another couple of pointers, e.g. `prev_proj` and `next_proj`).

The algorithm consists of a main loop processing levels bottom-up. The sub-loop on lines 2–4 and line 5 perform the detection of non-projective edges (the lines correspond to the main loop of Algorithm 5); line 6 performs the projectivization (it corresponds to the main loop of Algorithm 3), only the projectivized total order has to use another set of pointers.

Since the combination of the algorithms is perspicuous, we will not delve into unnecessary detail. The correctness of the algorithm follows easily from Theorems 3.3.1 and 4.2.1.

The time complexity $O(n)$ of Algorithm 6 follows easily from the fact that all sub-loops of the main loop on lines 1–7 have overall linear time complexities.

4.4.1 Theorem. *Algorithm 6 returns for a dependency tree T and a local ordering $\mathcal{L}$ of T the canonical projectivization $T^{\mathrm{gen}_{\mathcal{L}}}$ of T with respect to $\mathcal{L}$ and the set of all non-projective edges of non-negative level types occurring in T. Its time complexity is $O(n)$.*

Chapter 5

Planarity and non-projective edges

Planarity is a relaxation of projectivity that can be characterized as the "no crossing edges" constraint. Although it might get confused with projectivity, it is in fact a strictly weaker constraint.

In this chapter, we show how planarity relates to non-projective edges and their level types. First we study the properties of pairs of edges forming a non-planar pair of edges, then we show how planarity can be characterized using properties of single non-projective edges.

Our results allow us to derive an original algorithm that can be used for checking planarity; its running time is linear for all projective dependency trees and scales gracefully with increasing number of non-projective edges with non-empty upper gaps; its worst-case complexity bound is quadratic.

We conclude the chapter with a remark on intractability of multiplanarity, which has been proposed as a measure of the complexity of natural language syntax.

5.1 Condition of planarity

Planarity forbids crossing of edges when we draw nodes on a horizontal line and edges as arcs in one half-plane (e.g., above the nodes). Beware, this notion differs from the graph-theoretic notion of planarity of a graph, where one is allowed to use the whole plane.

Planarity is a recent name for a constraint studied under different names already in the 1960's. We are aware of independent work in the USSR (*weakly non-projective dependency trees*) and in Czechoslovakia (*smooth dependency trees*). The paper by Dikovsky and Modina [2000] surveys mathematical results on dependency syntax obtained in the former Soviet Union; it also

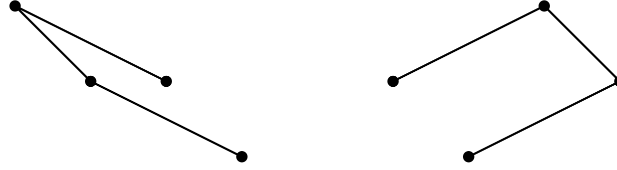


Figure 5.1: Sample non-planar dependency trees

gives references to many works hardly accessible in the former West. Nebeský [1979] presents a survey of his results on projective and smooth trees.

5.1.1 Definition. A dependency tree $T = (V, \rightarrow, \preceq)$ is *non-planar* if there are two edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ in T such that

(npp–5.1) $$i_1 \in (i_2, j_2) \;\;\&\;\; i_2 \in (i_1, j_1)\ .$$

Otherwise a dependency tree T is *planar*. We say that any two edges satisfying (npp–5.1) form a *non-planar pair of edges*.

From the definition of planarity it follows that a non-planar dependency tree has to have at least 4 nodes. Figure 5.1 shows sample minimal non-planar dependency trees.

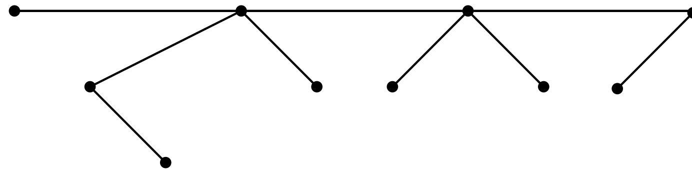
Planarity reduces to projectivity for dependency trees with their root node at either the left or right fringe of the tree. We prove this observation as Proposition 5.2.4 in the following section.

Remark. We would like to mention informally a close relationship between projective and planar dependency trees, which was derived by Ladislav Nebeský.

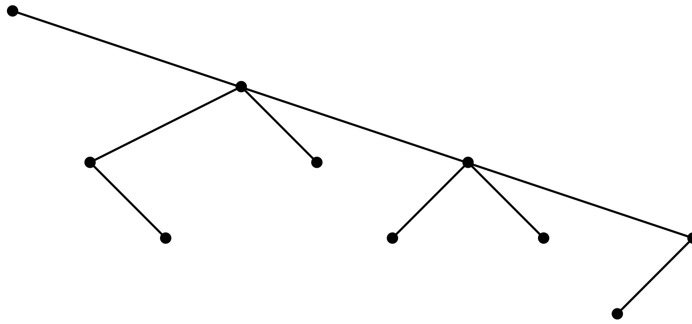
To make the idea most perspicuous, we will also use totally ordered unrooted trees (i.e., also undirected). Obviously, condition (npp–5.1) can be applied to such structures.

The relationship between planar totally ordered unrooted trees and planar dependency trees is straightforward: A planar totally ordered unrooted tree can be rooted in any of its nodes to form a planar dependency tree (i.e., a totally ordered rooted tree). But a dependency tree obtained in this way need not be projective.

The key notion in the relationship between planar totally ordered unrooted trees and projective dependency trees is the *chief path* of a planar totally ordered unrooted tree. It is defined as the path in a totally ordered



(a) planar totally ordered unrooted tree



(b) two of corresponding projective dependency trees

Figure 5.2: Relationship between planar totally ordered unrooted trees and projective dependency trees: (a) sample planar totally ordered unrooted tree with its chief path and (b) two of the corresponding planar dependency trees

unrooted tree that connects its leftmost and rightmost nodes (i.e., minimal and maximal nodes in the total order on the tree).

The nodes on the chief path of a planar totally ordered unrooted tree correspond uniquely to roots of projective dependency trees with the same set of nodes, edges, and total order on nodes. Rooting a planar totally ordered unrooted tree in a node that is not on its chief path results in a non-projective dependency tree.

Figure 5.2 gives a sample totally ordered unrooted tree and two of the

projective dependency trees corresponding to it (there are four in total, as its chief path has four nodes). The visualization of the sample totally ordered unrooted tree does not follow fully our convention for drawing dependency trees; for convenience, we draw the chief path on a horizontal line, all other nodes below it with vertical distances corresponding to the length of the path connecting a node with the closest node on the chief path.

5.2 Planarity and non-projective edges

In this section, we study the properties of non-planar pairs of edges. We show how planarity relates to properties of non-projective edges and their level types.

5.2.1 Proposition. *Any non-planar pair of edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ in a dependency tree $T = (V, \rightarrow, \preceq)$ satisfy*

$$i_1 \in \mathsf{Gap}_{i_2 \leftrightarrow j_2} \ \vee \ i_2 \in \mathsf{Gap}_{i_1 \leftrightarrow j_1} \ . \tag{5.2}$$

Proof. Suppose for the sake of contradiction that $i_1 \notin \mathsf{Gap}_{i_2 \leftrightarrow j_2}$ and $i_2 \notin \mathsf{Gap}_{i_1 \leftrightarrow j_1}$. This implies that $i_1 \in \mathsf{Subtree}_{i_2 \leftrightarrow j_2}$ and $i_2 \in \mathsf{Subtree}_{i_1 \leftrightarrow j_1}$. Since condition (npp–5.1) implies that edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ are disjoint, using Proposition 1.1.3 we get a non-trivial cycle $\mathsf{Parent}_{i_1 \leftrightarrow j_1} \rightarrow^* \mathsf{Parent}_{i_2 \leftrightarrow j_2} \rightarrow^* \mathsf{Parent}_{i_1 \leftrightarrow j_1}$, which is a contradiction with T being a tree. □

5.2.2 Corollary. *If a dependency tree $T = (V, \rightarrow, \preceq)$ is projective, then it is also planar.*

Proof. Follows immediately from Proposition 5.2.1, which states that in any non-planar pair of edges at least one of them is non-projective. □

5.2.3 Corollary. *Any two edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ in a dependency tree $T = (V, \rightarrow, \preceq)$ form a non-planar pair if and only if the following condition (possibly after renaming the edges) holds*

$$i_1 \in (i_2, j_2) \ \ \& \ \ i_2 \in \mathsf{Gap}_{i_1 \leftrightarrow j_1} \ . \tag{npp'–5.3}$$

Proof. Direction (npp'–5.3) ⇒ (npp–5.1) is obvious. Direction (npp–5.1) ⇒ (npp'–5.3) follows from Proposition 5.2.1 (possibly after renaming the edges), which states that in any non-planar pair of edges at least one of them is non-projective with an endpoint of the other edge as a witness. □

5.2.4 Proposition. *Let $T = (V, \to, \preceq)$ be a dependency tree such that its root node r is either at its left or right fringe (i.e., $r = \min_{\preceq} V$, or $r = \max_{\preceq} V$). Then T is planar if and only if it is projective.*

Proof. From Corollary 5.2.2 we know that every projective dependency tree is also planar.

Let us suppose that T is planar and at the same time non-projective. Then it contains a non-projective edge $i \to j$; from Observation 2.2.2 we know that $i \neq r$. Let u be any maximal node in $\mathsf{Gap}_{i \leftrightarrow j}$. From the assumption that r is at a fringe of T it follows that $u \neq r$, so there is a parent node v of u and it satisfies $v \notin [i, j]$. Edges $i \to j$, $v \to u$ form a non-planar pair and hence T is non-planar. □

5.2.5 Lemma. *Let $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ be a non-planar pair of edges in a dependency tree $T = (V, \to, \preceq)$ satisfying*

$$i_1 \notin \mathsf{Gap}_{i_2 \leftrightarrow j_2} \ \ \& \ \ i_2 \in \mathsf{Gap}_{i_1 \leftrightarrow j_1} \ .$$

Then

$$\mathsf{level}_{\mathsf{Child}_{i_1 \leftrightarrow j_1}} - \mathsf{level}_{i_2} > 0 \ ,$$

i.e., $\mathsf{Type}_{i_1 \leftrightarrow j_1} > 0$ *with node i_2 as a witness.*

Proof. Using Proposition 1.1.3 and disjointness of edges $i_1 \leftrightarrow i_2$, $j_1 \leftrightarrow j_2$, from the assumption $i_1 \notin \mathsf{Gap}_{i_2 \leftrightarrow j_2}$ we get that $i_1 \leftrightarrow j_1 \in \mathsf{Subtree}_{i_2 \leftrightarrow j_2}$. The disjointness of edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ further implies $\mathsf{level}_{\mathsf{Parent}_{i_1 \leftrightarrow j_1}} > \mathsf{level}_{\mathsf{Parent}_{i_2 \leftrightarrow j_2}}$, which is equivalent to $\mathsf{level}_{\mathsf{Child}_{i_1 \leftrightarrow j_1}} > \mathsf{level}_{\mathsf{Child}_{i_2 \leftrightarrow j_2}}$. Since $\mathsf{level}_{\mathsf{Child}_{u \leftrightarrow v}} \geq \mathsf{level}_u$ holds for any edge, we finally get $\mathsf{level}_{\mathsf{Child}_{i_1 \leftrightarrow j_1}} > \mathsf{level}_{i_2}$. □

5.2.6 Lemma. *Let $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ be a non-planar pair of edges in a dependency tree $T = (V, \to, \preceq)$ satisfying*

$$i_1 \in \mathsf{Gap}_{i_2 \leftrightarrow j_2} \ \ \& \ \ i_2 \in \mathsf{Gap}_{i_1 \leftrightarrow j_1} \ .$$

Then for at least one of the edges, say edge $i_1 \leftrightarrow j_1$, it holds that

$$\mathsf{level}_{\mathsf{Child}_{i_1 \leftrightarrow j_1}} - \mathsf{level}_{i_2} \geq 0 \ ,$$

i.e., $\mathsf{Type}_{i_1 \leftrightarrow j_1} \geq 0$ *with node i_2 as a witness.*

Proof. Let edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ satisfy the assumption. Let us first suppose that $\mathsf{level}_{\mathsf{Child}_{i_1 \leftrightarrow j_1}} \geq \mathsf{level}_{\mathsf{Child}_{i_2 \leftrightarrow j_2}}$. Since $\mathsf{level}_{\mathsf{Child}_{u \leftrightarrow v}} \geq \mathsf{level}_u$ for any edge

$u \leftrightarrow v$, we have that $\mathsf{level}_{\mathsf{Child}_{i_1 \leftrightarrow j_1}} \geq \mathsf{level}_{i_2}$, which we wanted to prove. If it is the case that $\mathsf{level}_{\mathsf{Child}_{i_1 \leftrightarrow j_1}} \leq \mathsf{level}_{\mathsf{Child}_{i_2 \leftrightarrow j_2}}$, it is analogously proved that edge $i_2 \leftrightarrow j_2$ is of non-negative level type. □

5.2.7 Lemma. *Let $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ be a non-planar pair of edges in a dependency tree $T = (V, \rightarrow, \preceq)$. Then for at least one of the edges, say edge $i_1 \leftrightarrow j_1$, it holds that*

$$i_2 \in \mathsf{Gap}^{\uparrow}_{i_1 \leftrightarrow j_1} ,$$

i.e., $\mathsf{Type}_{i_1 \leftrightarrow j_1} \geq 0$ *with node i_2 as a witness.*

Proof. From Lemma 5.2.1 we know that every non-planar pair of edges satisfy condition (5.2). In this condition, either only one, or both of the disjuncts are true. These two possible cases are dealt with by Lemmas 5.2.5 and 5.2.6; in either case, possibly after renaming the edges, it holds that $i_2 \in \mathsf{Gap}_{i_1 \leftrightarrow j_1}$ and $\mathsf{level}_{\mathsf{Child}_{i_1 \leftrightarrow j_1}} - \mathsf{level}_{i_2} \geq 0$, i.e., $i_2 \in \mathsf{Gap}^{\uparrow}_{i_1 \leftrightarrow j_1}$. □

5.2.8 Theorem. *Any two edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ in a dependency tree $T = (V, \rightarrow, \preceq)$ form a non-planar pair if and only if the following condition (possibly after renaming the edges) holds*

(npp"–5.4) $$i_1 \in (i_2, j_2) \ \ \& \ \ i_2 \in \mathsf{Gap}^{\uparrow}_{i_1 \leftrightarrow j_1} .$$

(Specifically, $\mathsf{Type}_{i_1 \leftrightarrow j_1} \geq 0$ *with node i_2 as a witness.)*

Proof. Direction (npp"–5.4) $\Rightarrow$ (npp–5.1) is obvious. Direction (npp–5.1) $\Rightarrow$ (npp"–5.4) follows from Lemma 5.2.7. □

5.2.9 Corollary. *A dependency tree $T = (V, \rightarrow, \preceq)$ is non-planar if and only if there are two edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ in T that satisfy any of the equivalent conditions* (npp–5.1)*,* (npp'–5.3)*, and* (npp"–5.4)*.*

5.3 Characterization of planarity using single non-projective edges

We show that non-planarity can be characterized in terms of properties of single non-projective edges only. The non-planar set of an edge presents an immediate characterization. Using our results on planarity and level types of non-projective edges, we show that even upper non-planar sets characterize planarity; they will also allow us to check planarity effectively.

5.3.1 Definition. The *non-planar set* of any edge $i \leftrightarrow j$ in a dependency tree $T = (V, \to, \preceq)$ is defined as follows

$$\mathsf{Np}_{i\leftrightarrow j} = \{u \leftrightarrow v \in T \mid u \in \mathsf{Gap}_{i\leftrightarrow j} \ \& \ v \notin [i,j]\} \ .$$

The non-planar set of a non-projective edge $i \leftrightarrow j$ can be thought of as the set of all edges with which the edge interacts through nodes in its gap.

5.3.2 Observation. *Let $i \leftrightarrow j$ be an edge in a dependency tree $T = (V, \to, \preceq)$. Then $\mathsf{Np}_{i\leftrightarrow j} \neq \emptyset$ only if edge $i \leftrightarrow j$ is non-projective.*

The next proposition exposes the relationship of edges in $\mathsf{Np}_{i\leftrightarrow j}$ to the gap of $i \leftrightarrow j$.

5.3.3 Proposition. *For any edge $i \leftrightarrow j$ in a dependency tree $T = (V, \to, \preceq)$ it holds that*

$$\mathsf{Np}_{i\leftrightarrow j} = \{u \leftrightarrow v \in T \mid u \in \mathsf{Gap}_{i\leftrightarrow j} \ \& \ v \notin \mathsf{Gap}_{i\leftrightarrow j} \ \& \ v \neq \mathsf{Parent}_{i\leftrightarrow j}\} \ .$$

Proof. The inclusion $\subseteq$ follows immediately from the observation that $v \notin [i,j]$ implies both $v \notin \mathsf{Gap}_{i\leftrightarrow j}$ and $v \neq \mathsf{Parent}_{i\leftrightarrow j}$.

Inclusion $\supseteq$: The assumption $u \in \mathsf{Gap}_{i\leftrightarrow j}$ implies that $u \notin \mathsf{Subtree}_{i\leftrightarrow j}$, and specifically $u \neq \mathsf{Parent}_{i\leftrightarrow j}$. Together with the assumption $v \neq \mathsf{Parent}_{i\leftrightarrow j}$ this gives us that edge $u \leftrightarrow v$ is disjoint from $\mathsf{Parent}_{i\leftrightarrow j}$, so using Proposition 1.1.3 we get that $v \notin \mathsf{Subtree}_{i\leftrightarrow j}$. Hence edges $i \leftrightarrow j$, $u \leftrightarrow v$ are disjoint, and from Proposition 2.2.4 it follows that $v \notin [i,j]$. □

5.3.4 Theorem. *A dependency tree $T = (V, \to, \preceq)$ is non-planar if and only if $\mathsf{Np}_{i\leftrightarrow j} \neq \emptyset$ for some non-projective edge $i \leftrightarrow j$ in T.*

Proof. Direction $\Leftarrow$ follows from the simple fact that edge $i \leftrightarrow j$ and any edge $u \leftrightarrow v$ from $\mathsf{Np}_{i\leftrightarrow j}$ form a non-planar pair. Direction $\Rightarrow$ follows from Proposition 5.2.1, which states that in any non-planar pair of edges at least one of them is non-projective with an endpoint of the other edge as a witness. □

Theorem 5.2.8 justifies the following notion of upper non-planar set and will allow us to speed up planarity checking.

5.3.5 Definition. The *upper non-planar set* of any edge $i \leftrightarrow j$ in a dependency tree $T = (V, \to, \preceq)$ is defined as follows

$$\mathsf{Np}^{\uparrow}_{i\leftrightarrow j} = \{u \leftrightarrow v \in T \mid u \in \mathsf{Gap}^{\uparrow}_{i\leftrightarrow j} \ \& \ v \notin [i,j]\} \ .$$

5.3.6 Theorem. *A dependency tree* $T = (V, \to, \preceq)$ *is non-planar if and only if* $\mathsf{Np}^{\uparrow}_{i\leftrightarrow j} \neq \emptyset$ *for some non-projective edge* $i \leftrightarrow j$ *in* T.

Proof. Direction $\Leftarrow$ follows from the simple fact that edge $i \leftrightarrow j$ and any edge $u \leftrightarrow v$ from $\mathsf{Np}^{\uparrow}_{i\leftrightarrow j}$ form a non-planar pair. Direction $\Rightarrow$ follows from Lemma 5.2.7, which states that in any non-planar pair of edges at least one of them is non-projective of non-negative level type with an endpoint of the other edge as a witness. □

Note that an analogy of Proposition 5.3.3 holds obviously also for upper non-planar sets.

5.4 Checking planarity

In this section, we address the problem of checking planarity of a dependency tree. First we present a straightforward way of checking planarity, then we show how our results on upper non-planar sets lead to a simple yet effective algorithm for determining them, which can also be used to check planarity.

Algorithm 7 Check planarity – high-level sketch

Input: dependency tree T
Output: planarity of T

```
1: for each edge i ↔ j do
2:     for each edge u ↔ v s.t. u ∈ (i, j) do
3:         check whether i ↔ j, u ↔ v cross
4:     end for
5: end for
```

Algorithm 7 presents a straightforward way of checking planarity, based directly on Definition 5.1.1. Its time complexity is quadratic: each of the embedded loops processes at most linearly many edges, and the check on line 3 obviously takes constant time. So we have the following theorem.

5.4.1 Theorem. *Algorithm 7 returns for a dependency tree* T *all its non-planar pairs of edges; its time complexity is* $O(n^2)$.

Algorithm 7 can be easily modified to determine non-planar sets of non-projective edges (change line 3 to: check $u \leftrightarrow v \in \mathsf{Np}_{i\leftrightarrow j}$). Although this modification does not increase the worst-case bound, it complicates matters by the need to check subordination; we can compute subordination relation on demand for the subtree of the processed edge $i \leftrightarrow j$ (once for all edges

$u \leftrightarrow v$ processed in loop on lines 2–4), preserving worst-case quadratic complexity.

Using our results concerning upper non-planar sets and non-projective edges of non-negative level types, we can devise an algorithm that is linear for projective trees and faster for random input and that remains worst-case quadratic. We can achieve this by incorporating the determination of upper non-planar sets into the algorithm for finding non-projective edges of non-negative level types from Section 4.2.

Algorithm 8 Determine upper non-planar sets – high-level sketch

Input: dependency tree T
Output: upper non-planar sets of edges in T

1: **for** each level of nodes in T bottom up **do**
2:     **for** each edge $i \rightarrow c$ with parent node i on processed level **do**
3:         compute $\mathsf{Gap}^{\uparrow}_{i \rightarrow c}$ using nodes on the same level as or above c
4:     **end for**
5:     **for** each edge $i \rightarrow c$ with parent node i on processed level **do**
6:         **for** each edge $u \leftrightarrow v$ s.t. $u \in \mathsf{Gap}^{\uparrow}_{i \rightarrow c}$ **do**
7:             check $u \leftrightarrow v \in \mathsf{Np}^{\uparrow}_{i \rightarrow c}$
8:         **end for**
9:     **end for**
10:    delete all nodes below processed level from original total order on T
11: **end for**

Algorithm 8 modifies and expands the loop over levels on lines 2–4 of Algorithm 4.

First, in the loop on lines 2–4, upper gaps of edges with parent nodes on the processed level are computed. We can proceed similarly to the detailed Algorithm 5 when computing upper gaps by processing child nodes of a node from the closest to the farthest one, as their upper gaps are included in one another.

The loop on lines 5–9 computes upper non-planar sets of edges on the processed level. Note that for the purpose of determining membership of some edge $u \leftrightarrow v$ in the upper non-planar set of edge $i \rightarrow c$ on line 7, total order also for nodes one level below c is needed; we can, for example, keep a copy of the whole total order for this purpose.

Worst-case time complexity of Algorithm 8 remains quadratic, but the actual running time is linear for projective dependency trees and scales with the number of edges whose endpoints are in the upper gap of some non-projective edge of non-negative level type. Hence we obtain the following theorem.

5.4.2 Theorem. *Algorithm 8 returns for a dependency tree T all its upper non-planar sets; its time complexity is $O(n^2)$.*

Both presented algorithms can also serve to simply check planarity of a dependency tree (for Algorithm 8 this follows from Theorem 5.3.6). If used for this purpose, they can be modified to finish as soon as the first non-planar pair of edges or non-empty upper non-planar set is found, respectively.

5.5 Remark on NP-completeness of multiplanarity

Yli-Jyrä [2003] proposes a generalization of planarity that he calls *multiplanarity*. We show that determining the multiplanarity of a graph is a special case of a general problem for chord graphs, and therefore it is NP-complete.

In this section, we use general totally ordered undirected graphs; the statements for them specialize to dependency trees. We say that $G = (V, \leftrightarrow)$ is a simple, *undirected graph* with set of nodes V and relation $\leftrightarrow$ representing undirected edges. (Note that in this section we use the symbol $\leftrightarrow$ in a different sense than elsewhere in this thesis; the presented notation is non-standard.) We call an undirected graph supplemented with a total order on the set of its nodes a *totally ordered undirected graph*.

We say that a totally ordered undirected graph $G = (V, \leftrightarrow, \preceq)$ is *non-planar* if it contains two edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ such that $i_1 \in (i_2, j_2)$ and $i_2 \in (i_1, j_1)$; otherwise G is *planar* (cf. Definition 5.1.1).

5.5.1 Definition (Yli-Jyrä [2003]). A totally ordered undirected graph $G = (V, \leftrightarrow, \preceq)$ is *k-planar* if G can be split into k planar totally ordered undirected graphs $G_1 = (V, \leftrightarrow_1, \preceq)$, ..., $G_k = (V, \leftrightarrow_k, \preceq)$ such that $\leftrightarrow\, = \,\leftrightarrow_1 \uplus \cdots \uplus \leftrightarrow_k$ ($\uplus$ is disjoint union).

Next, using our terminology, we present an NP-completeness result for vertex-colouring of circle graphs, or equivalently edge-colouring of chord graphs [Garey et al., 1980]. In current graph-theoretic literature, the terms *book embedding* and *stack layout* are used for colourings of edges such that no two edges with the same colour cross (form a non-planar pair in our terminology). For a review paper on edge-colouring problems in graphs with totally ordered sets of nodes and current terminology in the field, see Dujmović and Wood [2004].

5.5.2 Theorem (Garey et al. [1980]). *Determining k-planarity of totally ordered undirected graphs is NP-complete.*

Dependency trees are a special case of totally ordered undirected graphs. Hence, it is NP-complete to find minimal k such that a dependency tree is k-planar. For this reason, we do not include multiplanarity in our empirical evaluation presented in Part II.

Chapter 6

Well-nestedness and non-projective edges

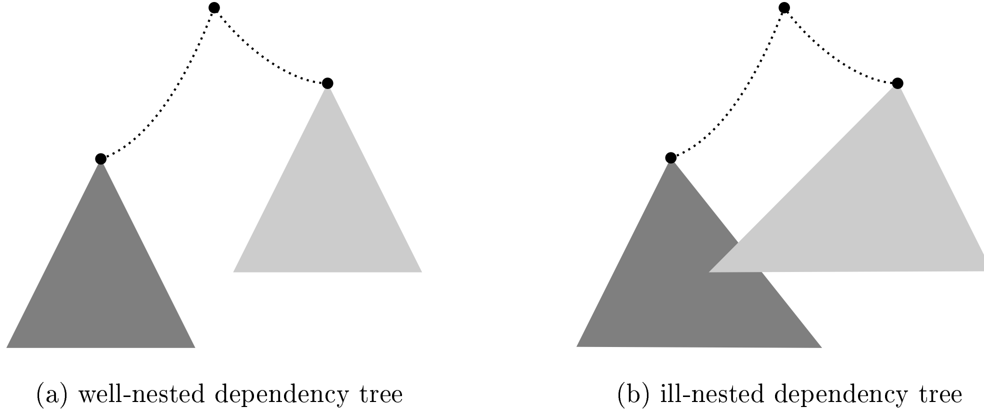
The condition of well-nestedness was proposed by Bodirsky et al. [2005]. They used it to characterize trees derivable by Tree Adjoining Grammars (together with one more constraint, namely gap degree for subtrees at most 1). The condition of well-nestedness can be also applied to dependency trees. Informally, it can be characterized as the "no interleaving of disjoint subtrees" constraint.

In this chapter, we show how well-nestedness relates to properties of non-projective edges and their level types. We derive characterizations of well-nestedness in terms of pairs of non-projective edges and in terms of properties of single non-projective edges. We also show that the presence of a non-projective edge of non-positive level type implies ill-nestedness of the whole dependency tree.

Our results allow us to derive original algorithms that can be used for checking well-nestedness. Both have worst-case quadratic time complexities; analogously to Algorithm 8, the running time of one of them is linear for all projective dependency trees and scales gracefully with increasing number of non-projective edges with non-empty upper gaps.

6.1 Original formulation of well-nestedness

In this section, we present the original formulation of the condition of well-nestedness; we also give its reformulation we will use further below.



(a) well-nested dependency tree (b) ill-nested dependency tree

Figure 6.1: Schematic visualization of well-nested and ill-nested dependency trees

6.1.1 Definition (Bodirsky et al. [2005]). A dependency tree $T = (V, \rightarrow, \preceq)$ is *ill-nested* if there are disjoint subtrees T_1, T_2 of T and nodes $x_1, y_1 \in T_1$ and $x_2, y_2 \in T_2$ such that

$$x_1 \prec x_2 \prec y_1 \prec y_2 \ .$$

Otherwise the dependency tree T is *well-nested*.

The constraint of well-nestedness can be schematically visualized as shown in Figure 6.1

We will be using the following straightforward reformulation of well-nestedness. It allows us to disregard alternative orderings of endpoints of edges.

6.1.2 Observation. *A dependency tree $T = (V, \rightarrow, \preceq)$ is ill-nested if and only if there are disjoint subtrees T_1, T_2 of T and nodes $x_1, y_1 \in T_1$ and $x_2, y_2 \in T_2$ such that*

$$x_1 \in (x_2, y_2) \ \ \& \ \ x_2 \in (x_1, y_1) \ .$$

From the definition of well-nestedness it follows that an ill-nested dependency tree has to have at least two disjoint subtrees with at least two nodes each, i.e., at least 5 nodes in total. Figure 6.2 shows sample minimal ill-nested dependency trees.

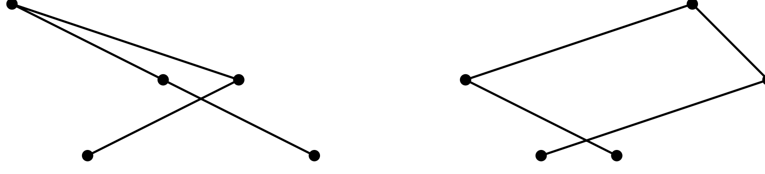


Figure 6.2: Sample ill-nested dependency trees

6.2 Reformulation of well-nestedness in terms of edges

Well-nestedness can be expressed in terms of edges—it will prove crucial in subsequent sections.

6.2.1 Lemma. *A dependency tree $T = (V, \to, \preceq)$ is ill-nested if and only if there are two edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ in disjoint subtrees T_1, T_2 of T, respectively, such that*

$$i_1 \in (i_2, j_2) \;\;\&\;\; i_2 \in (i_1, j_1) \ .$$

Proof. Direction $\Leftarrow$ is obvious.

Direction $\Rightarrow$: We show the existence of a pair of edges satisfying the condition by construction. Let r_k be the root node of subtree T_k, $k = 1, 2$.

We will first find an edge $i_1 \leftrightarrow j_1$ in subtree T_1. There are two possible cases. Let us first suppose that $r_1 \in (x_2, y_2)$. Consider the first edge $v_k \to v_{k+1}$ on the downward path $v_0 = r_1, v_1, \ldots, v_m = y_1$, $m > 0$, such that $v_k \in (x_2, y_2)$ and $v_{k+1} \notin [x_2, y_2]$. If it is the case that $r_1 \notin [x_2, y_2]$, consider the first edge $v_{k+1} \to v_k$ on the upward path $v_0 = x_1, v_1, \ldots,$ $v_n = r_1$, $n > 0$, such that $v_k \in (x_2, y_2)$ and $v_{k+1} \notin [x_2, y_2]$. Let us denote $i_1 = v_k$ and $j_1 = v_{k+1}$, and possibly rename x_2, y_2 so that $i_1 \in (x_2, y_2)$ and $x_2 \in (i_1, j_1)$.

In order to find an edge $i_2 \leftrightarrow j_2$ in subtree T_2 such that $i_1 \in (i_2, j_2)$ and $i_2 \in (i_1, j_1)$, let us proceed similarly as above. As the construction is analogous to the one presented above, we do not give it in full detail. Again there are two possible cases: either $r_2 \in (i_1, j_1)$, or $r_2 \notin [i_1, j_1]$. In the former case, let us consider the downward path from r_2 to y_2, in the latter case, the upward path from x_2 to r_2, and find an edge $i_2 \leftrightarrow j_2$ with the desired properties. Obviously, edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ are in disjoint subtrees. $\square$

6.3 Characterization of well-nestedness using pairs of non-projective edges

In this section, we show that the condition of well-nestedness can be expressed using pairs of non-projective edges.

First we give a characterization of pairs of edges in Lemma 6.2.1 in terms of their gaps.

6.3.1 Theorem. *Let $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ be two edges in a dependency tree $T = (V, \rightarrow, \preceq)$. They are in disjoint subtrees T_1, T_2, respectively, and satisfy $i_1 \in (i_2, j_2)$, $i_2 \in (i_1, j_1)$ if and only if the following condition holds*

(inp–6.1) $$i_1 \in \mathsf{Gap}_{i_2 \leftrightarrow j_2} \ \ \& \ \ i_2 \in \mathsf{Gap}_{i_1 \leftrightarrow j_1} \ .$$

Proof. Direction $\Leftarrow$: Let us consider subtrees T_k rooted in $\mathsf{Parent}_{i_k \leftrightarrow j_k}$, $k = 1, 2$. Condition (inp–6.1) obviously implies $i_1 \in (i_2, j_2)$, $i_2 \in (i_1, j_1)$, which in turn implies that edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ are disjoint. From Property 1.1.3 we get that both $\mathsf{Parent}_{i_2 \leftrightarrow j_2} \notin \mathsf{Subtree}_{i_1 \leftrightarrow j_1}$ and $\mathsf{Parent}_{i_1 \leftrightarrow j_1} \notin \mathsf{Subtree}_{i_2 \leftrightarrow j_2}$, hence subtrees T_1, T_2 are disjoint.

Direction $\Rightarrow$: Let us consider edge $i_2 \leftrightarrow j_2$ and node i_1. Since T_1 is disjoint from T_2, we have that $i_1 \notin \mathsf{Subtree}_{i_2 \leftrightarrow j_2}$, and hence $i_1 \in \mathsf{Gap}_{i_2 \leftrightarrow j_2}$. The proof that $i_2 \in \mathsf{Gap}_{i_1 \leftrightarrow j_1}$ is analogous. □

Condition (inp–6.1) allows us to talk about pairs of edges causing ill-nestedness and so characterize well-nestedness using properties of pairs of non-projective edges.

6.3.2 Definition. We say that any two non-projective edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ in a dependency tree $T = (V, \rightarrow, \preceq)$ satisfying condition (inp–6.1) form an *ill-nested pair of edges*.

6.3.3 Corollary. *A dependency tree $T = (V, \rightarrow, \preceq)$ is ill-nested if and only if it contains an ill-nested pair of edges.*

Proof. Follows from Lemma 6.2.1 and Theorem 6.3.1. □

Note that condition (inp–6.1) is identical to the case of planarity dealt with by Lemma 5.2.6. Therefore, in any ill-nested pair of edges, at least one of the edges is of non-negative level type, witnessed by an endpoint of the other edge.

6.3.4 Theorem. *Any two edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ in a dependency tree $T = (V, \rightarrow, \preceq)$ form an ill-nested pair if and only if the following condition (possibly after renaming the edges) holds*

(inp'–6.2) $$i_1 \in \mathsf{Gap}_{i_2 \leftrightarrow j_2} \ \ \& \ \ i_2 \in \mathsf{Gap}^{\uparrow}_{i_1 \leftrightarrow j_1} \ .$$

(Specifically, $\mathsf{Type}_{i_1 \leftrightarrow j_1} \geq 0$ *with node* i_2 *as a witness.)*

Proof. Direction (inp'–6.2) ⇒ (inp–6.1) is obvious. Direction (inp–6.1) ⇒ (inp'–6.2) follows from Lemma 5.2.6, which states that in any ill-nested pair of edges at least one of them is non-projective of non-negative level type with an endpoint of the other edge as a witness. □

6.3.5 Corollary. *A dependency tree $T = (V, \rightarrow, \preceq)$ is ill-nested if and only if there are two edges $i_1 \leftrightarrow j_1$, $i_2 \leftrightarrow j_2$ in T satisfying any of the equivalent conditions* (inp–6.1) *and* (inp'–6.2).

6.4 Sufficient condition for ill-nestedness

The results of Section 4.1 give the following relationship between level types of non-projective edges and well-nestedness.

6.4.1 Theorem. *If a dependency tree contains a non-projective edge of non-positive level type, then it is ill-nested.*

Proof. From Theorem 4.1.3 it follows that when a dependency tree contains a non-projective edge of non-positive level type, then it contains another non-projective edge such that the two edges form an ill-nested pair of edges. This in turn by Corollary 6.3.3 implies that the dependency tree is ill-nested. □

We see that types of non-projective edges and well-nestedness share a common ground; however, the statement of Theorem 6.4.1 cannot be strengthened to equivalence. It is easy to see that also two edges of arbitrary positive level type can satisfy condition (inp–6.1)—cf. the first dependency tree in Figure 6.2.

6.5 Characterization of well-nestedness using single edges

We show that well-nestedness can be characterized in terms of properties of single non-projective edges only. We define the ill-nested set of an edge

and show that it gives the desired characterization. Analogously to upper non-planar sets, we define upper ill-nested sets and show that they also characterize well-nestedness.

6.5.1 Definition. The *ill-nested set* of any edge $i \leftrightarrow j$ in a dependency tree $T = (V, \to, \preceq)$ is defined as follows

$$\mathsf{In}_{i\leftrightarrow j} = \{u \leftrightarrow v \in T \mid u \in \mathsf{Gap}_{i\leftrightarrow j} \ \& \ v \notin [i,j] \ \& \ u, v \notin \mathsf{Anc}_{i\leftrightarrow j}\} \ .$$

6.5.2 Observation. *Let $i \leftrightarrow j$ be an edge in a dependency tree $T = (V, \to, \preceq)$. Then $\mathsf{In}_{i\leftrightarrow j} \neq \emptyset$ only if edge $i \leftrightarrow j$ is non-projective.*

The next proposition exposes the relationship of edges in $\mathsf{In}_{i\leftrightarrow j}$ to the gap of $i \leftrightarrow j$.

6.5.3 Proposition. *For any edge $i \leftrightarrow j$ in a dependency tree $T = (V, \to, \preceq)$ it holds that*

$$\mathsf{In}_{i\leftrightarrow j} = \{u \leftrightarrow v \in T \mid u \in \mathsf{Gap}_{i\leftrightarrow j} \ \& \ v \notin \mathsf{Gap}_{i\leftrightarrow j} \ \& \ u, v \notin \mathsf{Anc}_{i\leftrightarrow j}\} \ .$$

Proof. The inclusion $\subseteq$ follows immediately from the observation that $v \notin [i,j]$ implies $v \notin \mathsf{Gap}_{i\leftrightarrow j}$.

The prove the inclusion $\supseteq$, observe that $u \in \mathsf{Gap}_{i\leftrightarrow j}$ implies u is disjoint from edge $i \leftrightarrow j$, and that $v \notin \mathsf{Anc}_{i\leftrightarrow j}$ and $v \notin \mathsf{Gap}_{i\leftrightarrow j}$ imply v is disjoint from edge $i \leftrightarrow j$. So edges $i \leftrightarrow j$, $u \leftrightarrow v$ are disjoint, and the desired inclusion follows from Proposition 2.2.4. □

We are ready to formulate the main result of this section, which gives as a corollary a characterization of well-nestedness using properties of single edges.

6.5.4 Theorem. *Let $i \leftrightarrow j$ be an edge in a dependency tree $T = (V, \to, \preceq)$. The edges that form an ill-nested pair with the edge $i \leftrightarrow j$ are exactly the edges in $\mathsf{In}_{i\leftrightarrow j}$.*

Proof. Direction $\Rightarrow$: Let $u \leftrightarrow v$ be an edge forming an ill-nested pair with the edge $i \leftrightarrow j$, i.e. $i \in \mathsf{Gap}_{u\leftrightarrow v}$ and $u \in \mathsf{Gap}_{i\leftrightarrow j}$. This implies $i \in (u,v)$ and $u \in (i,j)$, which immediately gives $v \notin [i,j]$. Supposing $u \in \mathsf{Anc}_{i\leftrightarrow j}$ or $v \in \mathsf{Anc}_{i\leftrightarrow j}$ we get $i \in \mathsf{Subtree}_{u\leftrightarrow v}$, which is in contradiction with $i \in \mathsf{Gap}_{u\leftrightarrow v}$, and hence $u, v \notin \mathsf{Anc}_{i\leftrightarrow j}$. Therefore $u \leftrightarrow v \in \mathsf{In}_{i\leftrightarrow j}$.

Direction $\Leftarrow$: Let $u \leftrightarrow v \in \mathsf{In}_{i\leftrightarrow j}$ (i.e. $u \in \mathsf{Gap}_{i\leftrightarrow j}$, $v \notin [i,j]$, and $u, v \notin \mathsf{Anc}_{i\leftrightarrow j}$; without loss of generality assume $i \in (u,v)$). From the assumptions $u \in \mathsf{Gap}_{i\leftrightarrow j}$ and $v \notin [i,j]$ we get that edges $i \leftrightarrow j$, $u \leftrightarrow v$ are disjoint. Using

Property 1.1.3, from the assumption $u, v \notin \mathsf{Anc}_{i \leftrightarrow j}$ we get $i \notin \mathsf{Subtree}_{u \leftrightarrow v}$, thus $i \in \mathsf{Gap}_{u \leftrightarrow v}$. Therefore $i \leftrightarrow j$, $u \leftrightarrow v$ satisfy (inp–6.1). □

6.5.5 Corollary. *A dependency tree $T = (V, \to, \preceq)$ is ill-nested if and only if $\mathsf{In}_{i \leftrightarrow j} \neq \emptyset$ for some non-projective edge $i \leftrightarrow j$ in T.*

Proof. Follows directly from Theorem 6.5.4. □

Next we define upper ill-nested sets and show that they also capture well-nestedness. We will use the notion of upper ill-nested set to derive an effective algorithm for checking well-nestedness.

6.5.6 Definition. The *upper ill-nested set* of any edge $i \leftrightarrow j$ in a dependency tree $T = (V, \to, \preceq)$ is defined as follows

$$\mathsf{In}^{\uparrow}_{i \leftrightarrow j} = \{u \leftrightarrow v \in T \mid u \in \mathsf{Gap}^{\uparrow}_{i \leftrightarrow j} \;\&\; v \notin [i, j] \;\&\; u, v \notin \mathsf{Anc}_{i \leftrightarrow j}\} \ .$$

6.5.7 Theorem. *A dependency tree $T = (V, \to, \preceq)$ is ill-nested if and only if $\mathsf{In}^{\uparrow}_{i \leftrightarrow j} \neq \emptyset$ for some non-projective edge $i \leftrightarrow j$ in T.*

Proof. Direction $\Leftarrow$ follows from the simple fact that edge $i \leftrightarrow j$ and any edge $u \leftrightarrow v$ from $\mathsf{In}^{\uparrow}_{i \leftrightarrow j}$ form an ill-nested pair. Direction $\Rightarrow$ follows from Theorem 6.5.5 (recall that this implication amounts to Lemma 5.2.6). □

Note that an analogy of Proposition 6.5.3 holds obviously also for upper ill-nested sets.

6.6 Checking well-nestedness

In this section, we address the problem of checking well-nestedness of a dependency tree. Our characterizations of well-nestedness give novel ways of checking it; the results on ill-nested sets and upper ill-nested sets lead to simple yet effective algorithms. The algorithms are analogous to the ones for planarity presented in Section 5.4.

Algorithm 9 presents a straightforward way of determining ill-nested sets of non-projective edges in a dependency tree. Its time complexity is $O(n^2)$ because of the two embedded loops processing at most linearly many edges each (the check on line 3 can be implemented so as to take constant time, e.g. by pre-computing subordination $\to^*$, which can be done in $O(n^2)$ time; cf. Section 1.3). Hence we obtain the following theorem.

Algorithm 9 Determine ill-nested sets – high-level sketch

Input: dependency tree T
Output: ill-nested sets of non-projective edges in T

1: **for** each edge $i \leftrightarrow j$ **do**
2:     **for** each edge $u \leftrightarrow v$ s.t. $u \in (i, j)$ **do**
3:         check $u \leftrightarrow v \in \mathsf{In}_{i \leftrightarrow j}$
4:     **end for**
5: **end for**

6.6.1 Theorem. *Algorithm 9 returns for a dependency tree T all its ill-nested sets; its time complexity is $O(n^2)$.*

The bound is the same as for the reported algorithms for checking well-nestedness, but without relying on any assumptions about the operations used, namely that bit-vector operations are $O(1)$ [Möhl, 2006].

Using Theorem 6.5.7 for upper ill-nested sets, we can present an algorithm that is linear for projective trees and faster for random input and that remains worst-case quadratic. Analogously to Algorithm 8, this is achieved by incorporating the determination of upper ill-nested sets into the algorithm for finding non-projective edges of non-negative level type from Section 4.2.

Algorithm 10 Determine upper ill-nested sets – high-level sketch

Input: dependency tree T
Output: upper ill-nested sets of edges in T

1: **for** each level of nodes in T bottom up **do**
2:     **for** each edge $i \to c$ with parent node i on processed level **do**
3:         compute $\mathsf{Gap}^{\uparrow}_{i \to c}$ using nodes on the same level as or above c
4:     **end for**
5:     **for** each edge $i \to c$ with parent node i on processed level **do**
6:         **for** each edge $u \leftrightarrow v$ s.t. $u \in \mathsf{Gap}^{\uparrow}_{i \to c}$ **do**
7:             check $u \leftrightarrow v \in \mathsf{In}^{\uparrow}_{i \to c}$
8:         **end for**
9:     **end for**
10:     delete all nodes below processed level from original total order on T
11: **end for**

First, in the loop on lines 2–4, upper gaps of edges with parent nodes on the processed level are computed. Second, the loop on lines 5–9 computes upper ill-nested sets of edges on the processed level. Again, we point out that for the purpose of determining membership of some edge $u \leftrightarrow v$ in the

upper ill-nested set of edge $i \to c$ on line 7, total order also for nodes one level below c is needed.

In the check on line 7, ancestry check is the only trickier part. We can, for example, compute ancestry relation on demand for ancestors of the processed edge $i \leftrightarrow j$ (once for all edges $u \leftrightarrow v$ processed in the loop on lines 6–8). We can also optimize the order in which we process edges incident on node u: first process the edge for which u is its child node, then the edges for which u is their parent node, and always check for ancestry the parent node of the considered edge (this way, we can utilize the simple fact that if an edge's child node is an ancestor of a node, so is its parent node).

Worst-case time complexity of Algorithm 10 remains quadratic, but the actual running time is linear for projective dependency trees and scales with the number of edges whose endpoints are in the upper gap of some non-projective edge of non-negative level type. Hence we get the following theorem.

6.6.2 Theorem. *Algorithm 10 returns for a dependency tree T all its upper ill-nested sets; its time complexity is $O(n^2)$.*

Both presented algorithms can also serve to simply check ill-nestedness of a dependency tree (this follows from Corollary 6.5.5 and Theorem 6.5.7, respectively). If used for this purpose, they can be modified to finish as soon as a non-empty ill-nested set or a non-empty upper ill-nested set is found, respectively.

Chapter 7

Partitioning of gaps of non-projective edges

In this chapter, we study properties of individual non-projective edges that can serve as edge-based measures of non-projectivity of dependency trees; we will use them in the empirical evaluation presented in Chapter 10. Some of the edge-based measures have corresponding tree-based counterparts, which we briefly mention.

We present two ways of partitioning gaps of non-projective edges: into intervals and into components. We show how levels of nodes can be combined with the partitioning of gaps into components; we propose a new edge-based measure of non-projectivity of dependency trees, level signatures.

7.1 Partitioning of gaps into intervals

7.1.1 Definition. For any edge $i \leftrightarrow j$ in a dependency tree $T = (V, \rightarrow, \preceq)$ we define its *interval degree* as follows

$$\mathsf{ideg}_{i\leftrightarrow j} = \text{number of intervals in } \mathsf{Gap}_{i\leftrightarrow j} \text{ with respect to } \preceq \ .$$

By an interval we mean a maximal contiguous interval in $\preceq$, i.e. a maximal set of nodes comprising all nodes between its endpoints in the total order on nodes $\preceq$. (By definition, the interval degree of a projective edge is 0.)

Formally, the interval degree of edge $i \leftrightarrow j$ in a dependency tree $T = (V, \rightarrow, \preceq)$ is the number of equivalence classes of the equivalence relation

$$((\prec^{\mathrm{tr}} \cup \succ^{\mathrm{tr}}) \restriction \mathsf{Gap}_{i\leftrightarrow j})^* \ .$$

This measure corresponds to the tree-based *gap degree* measure, which was first introduced by Holan et al. [1998]. The gap degree of a subtree is the number of maximal contiguous intervals in the gap of the subtree. Obviously, the interval degree of an edge is bounded from above by the gap degree of the subtree rooted in its parent node.

Interval degree takes non-negative integer values; it is 0 exactly for projective edges. Interval degree is unbounded; in any given dependency tree, however, it is bounded by its size.

7.2 Partitioning of gaps into components

7.2.1 Definition. For any edge $i \leftrightarrow j$ in a dependency tree $T = (V, \rightarrow, \preceq)$ we define its *component degree* as follows

$$\mathsf{cdeg}_{i\leftrightarrow j} = \text{number of components in } \mathsf{Gap}_{i\leftrightarrow j} \text{ with respect to } \leftrightarrow .$$

By a component we mean a connected component in the relation $\leftrightarrow$, in other words a weak component in the relation $\rightarrow$ (we consider relations induced on the set $\mathsf{Gap}_{i\leftrightarrow j}$ by relations on T). (By definition, the component degree of a projective edge is 0.)

Formally, the component degree of edge $i \leftrightarrow j$ in a dependency tree $T = (V, \rightarrow, \preceq)$ is the number of equivalence classes of the equivalence relation

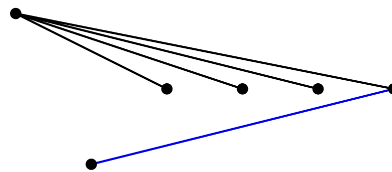
$$(\leftrightarrow \restriction \mathsf{Gap}_{i\leftrightarrow j})^* .$$

This measure was introduced by Nivre [2006]; he uses it to characterize a whole dependency tree by taking maximum over all its edges.

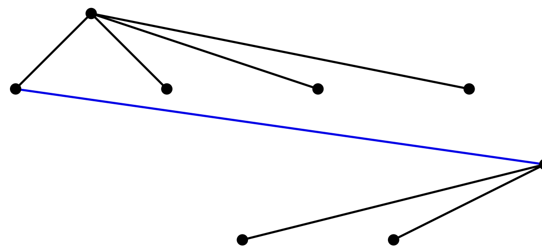
Similarly to interval degree, component degree takes non-negative integer values and is 0 exactly for projective edges. Component degree is unbounded; in any given dependency tree, however, it is bounded by its size.

Each component of a gap can be represented by a single node, its *root* in the dependency relation induced on the nodes of the gap (i.e. a node of the component closest to the root of the whole tree). Note that a component need not constitute a full subtree of the dependency tree (there may be nodes in the subtree of the component root that lie outside the span of the particular non-projective edge).

Remark. Partitioning of the gap of a non-projective edge into intervals and components is independent of each other. Figure 7.1 shows two sample dependency trees; tree (a) contains a non-projective edge with interval degree 1 and component degree 3, tree (b) contains a non-projective edge with



(a) dependency tree with a non-projective edge (in blue) with interval degree 1 and component degree 3



(b) dependency tree with a non-projective edge (in blue) with interval degree 3 and component degree 1

Figure 7.1: Sample dependency trees showing mutual independence of interval degree and component degree of a non-projective edge

interval degree 3 and component degree 1 (and also other non-projective edges).

7.3 Combining levels of nodes and partitioning of gaps into intervals

We propose a new edge-based measure of non-projectivity combining level types and component degrees. (We do not use interval degrees, i.e. the partitioning of gaps into intervals, because we cannot specify a unique representative of an interval with respect to the tree structure.)

7.3.1 Definition. The *level signature* of an edge $i \leftrightarrow j$ in a dependency tree $T = (V, \rightarrow, \preceq)$ is a mapping $\mathsf{Signature}_{i\leftrightarrow j} : \mathcal{P}(V) \rightarrow {}^{\mathbb{Z}}\mathbb{N}_0$ defined as follows

$$\mathsf{Signature}_{i\leftrightarrow j} = \{\mathsf{level}_{\mathsf{Child}_{i\leftrightarrow j}} - \mathsf{level}_r \mid r \text{ is component root in } \mathsf{Gap}_{i\leftrightarrow j}\} \ .$$

The right-hand side is considered as a multiset, i.e. elements may repeat. We call the elements of a level signature *component levels*.

The signature of an edge is a multiset consisting of the relative distances in levels of all component roots in its gap from its child node. Level components are not bounded; in any given dependency tree, however, level components of any non-projective edge are bounded in absolute value by the height of the dependency tree. Level signature is the empty multiset exactly for projective edges.

Further, we disregard any possible orderings on signatures and concentrate only on the relative distances in levels. In Chapter 10, we present signatures as non-decreasing sequences and write them in angle brackets $\langle\ \rangle$, component levels separated by commas (by doing so, we avoid combinatorial explosion).

Another natural ordering on level signatures could for example be to order component levels according to the order of the corresponding component roots in the total order on nodes $\preceq$.

Notice that level signatures subsume level types: the level type of a non-projective edge is the component level of any of possibly several component roots closest to the root of the whole tree. In other words, the level type of an edge is equal to the largest component level occurring in its level signature.

Level signatures share interesting formal properties with level types of non-projective edges. The following result is a direct generalization of Theorem 4.1.3.

7.3.2 Theorem. *Let $i \leftrightarrow j$ be a non-projective edge in a dependency tree $T = (V, \rightarrow, \preceq)$ with a non-positive component level in its level signature. Then for each edge $v \rightarrow r_c$ in T such that r_c is root of component c in $\mathsf{Gap}_{i \leftrightarrow j}$ with component level $l_c = \mathsf{level}_{\mathsf{Child}_{i \leftrightarrow j}} - \mathsf{level}_{r_c} \leq 0$ it holds that one of the endpoints of edge $i \leftrightarrow j$ (i.e., either i, or j) is in $\mathsf{Gap}_{v \leftrightarrow r_c}$ and it witnesses that*

$$\mathsf{Type}_{v \leftrightarrow r_c} \geq -l_c \ .$$

Proof. The proof proceeds along similar lines as the proof of Theorem 4.1.3.

From the assumptions that r_c is maximal in $\mathsf{Gap}_{i \leftrightarrow j}$ and $l_c = \mathsf{level}_{\mathsf{Child}_{i \leftrightarrow j}} - \mathsf{level}_{r_c} \leq 0$, node r_c has a parent node v, which satisfies $v \notin \mathsf{Gap}_{i \leftrightarrow j}$. Obviously, edges $i \leftrightarrow j$, $v \rightarrow r_c$ are disjoint, thus from Proposition 2.2.4 we have that $v \notin [i, j]$, and so either $i \in (v, r_c)$, or $j \in (v, r_c)$. Since $\mathsf{level}_v \geq \mathsf{level}_{\mathsf{Parent}_{i \leftrightarrow j}}$, we have that $\mathsf{Parent}_{i \leftrightarrow j} \notin \mathsf{Subtree}_v$, and so either $i \in \mathsf{Gap}_{v \leftrightarrow r_c}$, or $j \in \mathsf{Gap}_{v \leftrightarrow r_c}$. Immediately from definition we obtain that $\mathsf{Type}_{v \leftrightarrow r_c} \geq \mathsf{level}_{r_c} - \mathsf{level}_{\mathsf{Child}_{i \leftrightarrow j}} = -l_c$. The simple facts that $\mathsf{level}_i \leq \mathsf{level}_{\mathsf{Child}_{i \leftrightarrow j}}$, $\mathsf{level}_j \leq \mathsf{level}_{\mathsf{Child}_{i \leftrightarrow j}}$ imply that the endpoint of edge $i \leftrightarrow j$ in the span of edge $v \rightarrow r_c$ indeed witnesses the inequality for the level types, which finishes the proof. □

This result links level signatures to well-nestedness: it tells us that whenever an edge's level signature contains a non-positive component level, the whole dependency tree is ill-nested.

7.3.3 Corollary. *If a dependency tree contains a non-projective edge with a non-positive component level in its level signature, then it is ill-nested.*

Proof. From Theorem 7.3.2 it follows that when a dependency tree contains a non-projective edge with a non-positive component level, then it contains an ill-nested pair of edges, which in turn by Corollary 6.3.3 implies that the dependency tree is ill-nested. □

Chapter 8

Formulas for counting some classes of trees

In this chapter, we review known formulas for the numbers of trees of given sizes for classes of trees that we study in this thesis: projective, planar, well-nested, and unrestricted. We use the notation C_n for the number of trees with n nodes in the class of trees specified as superscript to the symbol.

The chapter is intended to provide the reader with some further intuitions regarding the relative "distances" between different classes of trees, based on the numbers of trees on a given number of nodes in the classes.

8.1 Unrestricted dependency trees

Unrestricted dependency trees are in fact rooted labelled trees—consider labelling nodes of a dependency tree of size n by natural numbers $1, \ldots,$ n; this labelling induces a total order on the set of nodes.

Labelled trees are counted by the well-know Cayley's formula. The formulas for labelled unrooted (LU) and rooted (LR) trees are closely related, since an unrooted tree can be rooted in any or its nodes. For reference, see sequences A000272 and A000169 in the On-Line Encyclopedia of Sequences [Sloane, 2007], respectively.

$$C_n^{\mathrm{LU}} = n^{n-2}$$

$$C_n^{\mathrm{LR}} = n^{n-1}$$

8.2 Projective and planar trees

To our knowledge, the general formula for the number of projective trees (Pr) was first derived by Jiřička [1975]. His derivation uses the relationship between projective dependency trees and planar totally ordered unrooted trees (PlU) we mentioned in Section 5.1 on page 52. Jiřička also derived the formula for planar unrooted trees we present in the next section; he attributes a previous derivation of the formula to Ladislav Nebeský. (Recall that a planar totally ordered unrooted tree can be rooted in any node to give a planar dependency tree; we denote the class of planar totally ordered rooted trees as PlR.)

It seems that these results had been forgotten for many years. They re-appeared in the field of enumerative combinatorics under the name of noncrossing trees with the work of Noy [1998]. For further references on recent results concerning these two mutually related classes of trees, see sequences A006013 (offset 1) and A001764 (offset 1) for projective dependency trees and planar totally ordered unrooted trees, respectively, in the On-Line Encyclopedia of Sequences [Sloane, 2007].

$$C_n^{\mathrm{Pr}} = \frac{1}{n}\binom{3n-2}{2n-1}$$

$$C_n^{\mathrm{PlU}} = \frac{1}{n-1}\binom{3n-3}{2n-1}$$

$$C_n^{\mathrm{PlR}} = nC_n^{\mathrm{PlU}} = \frac{n}{n-1}\binom{3n-3}{2n-1}$$

The relationship between planar totally ordered unrooted trees and projective dependency trees shows also in the simple relationship between formulas counting them

$$C_n^{\mathrm{Pr}} = \frac{3n-2}{n}\,C_n^{\mathrm{PlU}}$$

8.3 Well-nested trees

For well-nested dependency trees, only a recursive formula enumerating them is known.* For further details and references, see sequence A113882 in the On-Line Encyclopedia of Sequences [Sloane, 2007].

*For more details on deriving the recurrence, see Manuel Bodirsky's web page `http://www.informatik.hu-berlin.de/~bodirsky/publications/drawings.html`.

The recursive formulas are given below. To give the reader at least a feeling for them, here is what the sequences count: t_n is the number of well-nested dependency trees on n nodes; w_n is the number of well-nested totally ordered forests on n nodes; $w_{n,k}$ is the number of k-tuples of well-nested dependency forests with n as the total number of nodes.

$$\begin{aligned} t_n &= n w_{n-1} \\ w_n &= w_{n,1} \\ w_{n,1} &= \sum_{i=1}^{n} t_i w_{n-i,i} \\ w_{n,k} &= \sum_{i=0}^{n} w_i w_{n-i,k-1} \quad \text{for } k \geq 2 \end{aligned}$$

where

$$\begin{aligned} t_0 &= t_1 = 1 \\ w_0 &= w_1 = 1 \\ w_{0,k} &= 1 \quad \text{for } k \geq 0 \end{aligned}$$

No closed-form formula is known, nor is the exact asymptotic growth of t_n and w_n. The scatter plot from the On-Line Encyclopedia of Sequences suggests that the growth is super-exponential.

8.4 Note on asymptotic growths

Using Stirling formula, it is easy to show that numbers of projective and planar trees of any sort have exponential asymptotic growth. Labelled rooted trees grow super-exponentially, as do (most probably) well-nested trees; cf. remark in last section.

Table 8.1 shows counts of trees on small numbers of nodes for the classes in which we are interested in this thesis. It is worth noticing that even the number of projective dependency trees, the smallest class, grows enormously with increasing number of nodes.

The On-Line Encyclopedia of Sequences [Sloane, 2007] provides longer initial subsequences for all the sequences that we briefly presented in this chapter.

n	C_n^{Pr}	C_n^{PlR}	C_n^{WnR}	C_n^{LR}
1	1	1	1	1
2	2	2	2	2
3	7	9	9	9
4	30	48	64	64
5	143	275	605	625
6	728	1,638	6,996	7,776
7	3,876	9,996	94,556	117,649
8	21,318	62,016	1,452,928	2,097,152
9	120,175	389,367	24,921,765	43,046,721
10	690,690	2,466,750	471,091,360	1,000,000,000
11	4,032,015	15,737,865	9,720,039,120	25,937,424,601
12	23,841,480	100,975,680	217,285,778,700	743,008,370,688

Table 8.1: Counts of trees of some classes of rooted trees on small numbers of nodes

Part II

Empirical results

Chapter 9

Empirical evaluation of algorithms for finding non-projective edges

To check the theoretical results on the worst-case bound of Algorithm 5, we implemented both this algorithm and the simple Algorithm 1 and compared their running times on data from PDT 2.0, a dependency treebank of Czech [Hajič et al., 2006]. Let us remind the reader that Algorithm 5 finds only non-projective edges of non-negative level types.

Algorithm 1 looks for all non-projective edges by checking subordination for all nodes in their spans. Due to memory limitation, we implemented it with cubic worst-case time complexity and linear space complexity (we computed subordination on the fly).

The algorithms were run on a 2.2GHz 4-CPU Dual Core AMD Opteron Processor 275 computer with 10GB of memory. The test was performed on the whole analytical layer (i.e., surface-syntax layer) of PDT 2.0.

Relevant characteristics of the analytical layer of PDT 2.0 are shown in Table 9.1. From average tree height and average edge span we see that on this data our implementation of the simple algorithm does not reach its cubic worst-case time complexity.

Level types of non-projective edges in the whole PDT 2.0 are distributed as follows: there are $28,426$ edges of positive level types, 82 edges of level type 0, and 5 edges of negative level types (very rare!). We see that on this data set Algorithm 5 finds virtually all non-projective edges. A more detailed evaluation of formal properties of non-projective structures in language syntax using dependency treebanks in several languages can be found in Chapter 10.

Both algorithms were implemented in Perl within the tree editor TrEd [Pajas, 2007]. The data occupy about 5GB of memory (in the standard in-

	all trees	non-projective trees
# trees	87,980	20,380
# nodes	1,592,827	483,030
# edges	1,330,878	421,712
avg. tree height	5.97 (sd 2.70)	7.41 (sd 2.46)
avg. edge span	1.59 (sd 3.31)	1.79 (sd 3.55)

Table 9.1: Summary of non-projective dependency trees and edges on analytical layer of PDT 2.0 (edges from artificial root are omitted, edge span excludes the edge's endpoints, sd means standard deviation)

	all trees	non-projective trees
simple Algorithm 1	52.38s (sd 0.25)	20.78s (sd 0.16)
Algorithm 5	25.31s (sd 0.16)	11.00s (sd 0.15)

Table 9.2: Running times on analytical layer of PDT 2.0 (averages of 100 runs, sd means standard deviation)

memory representation of TrEd); for the implementation of Algorithm 5, we created additional data structures as described in Section 1.2.

Table 9.2 gives running times of both algorithms on the data; the running times exclude the time of loading the data into memory. Algorithm 5 outperforms the simple algorithm, the relative gain seems to be justified by the average edge span and tree height reported in Table 9.1. The relative deterioration of performance for non-projective trees only might be caused by the time cost of accessing the trees in memory (accessing all trees took average 3.93s, accessing non-projective trees took average 2.41s, although they amount to only about a quarter of all trees).

Our empirical results on real natural language data show that the constant in the $O(n)$ bound of Algorithm 5 is small. On the other hand, they suggest that for the purposes of analysis of natural language data, due to the characteristics of the data, even simple algorithms with worse than optimal complexity bounds work well enough.

Chapter 10

Evaluation of tree and edge properties on natural language data

This chapter presents an extensive empirical evaluation on data from nineteen natural languages of some of the tree and edge properties of non-projective dependency trees that can be used to describe and/or delimit non-projective structures occurring in natural language; it is an extension of results published in [Havelka, 2007a]. The experiments show that properties of non-projective edges provide accurate and expressive tools for capturing non-projective structures occurring in natural language.

Although we evaluate only properties derived directly from the tree structure and total order of a dependency tree, we would like to point out that properties of individual non-projective edges provide tools allowing for a more detailed and linguistically appropriate analysis. In particular, properties of a non-projective edge allow for "local" lexicalization, be it of its endpoints, of nodes in its gaps (e.g., the component roots), of edges in its non-planar set, or any other sets of nodes defined relative to the edge.

10.1 Experimental setup

First, we briefly list the languages that we use in our empirical evaluation. Second, we describe the properties we report. Third, we describe how we deal with the fact that the data formats of all treebanks we work with use an artificial root node for each sentence. Last, we mention the program tools for computing properties of non-projective edges that we used for the evaluation.

10.1.1 Natural language treebanks

We evaluate the tree and edge properties described further below on dependency treebanks in nineteen languages. All but one of the treebanks were made available in CoNLL 2006 and 2007 shared tasks on dependency parsing [Buchholz and Marsi, 2006, Nivre et al., 2007];* the only exception is the Latin Dependency Treebank.†

Here is the list of all the languages and corresponding references:

Arabic	Hajič et al. [2004], Smrž et al. [2002]
Basque	Aduriz et al. [2003]
Bulgarian	Simov et al. [2005], Simov and Osenova [2003]
Catalan	Martí et al. [2007]
Czech	Böhmová et al. [2003], Hajič et al. [2001]
Danish	Kromann [2003]
Dutch	van der Beek et al. [2002b,a]
English	Marcus et al. [1993], Johansson and Nugues [2007]
German	Brants et al. [2002]
Greek	Prokopidis et al. [2005]
Hungarian	Csendes et al. [2005]
Italian	Montemagni et al. [2003]
Japanese	Kawata and Bartels [2000]
Latin	Bamman and Crane [2006, 2007]
Portuguese	Afonso et al. [2002]
Slovene	Džeroski et al. [2006]
Spanish	Civit Torruella and Martí Antonín [2002], Navarro et al. [2003]
Swedish	Nilsson et al. [2005]
Turkish	Oflazer et al. [2003], Atalay et al. [2003]

We take the data "as is", although we are aware that structures occurring in different languages depend on the annotations and/or conversions used.

*All data sets are the train parts of the CoNLL 2006 and 2007 shared tasks on dependency parsing. The following treebanks were used in versions from year 2006: Bulgarian, Czech, Danish, Dutch, German, Japanese, Portuguese, Slovene, Spanish, and Swedish; the following treebanks in versions from year 2007: Arabic, Basque, Catalan, English, Greek, Hungarian, Italian, and Turkish.

†The Latin Treebank is comprised of four excerpts from work by four different authors. To be able to attest the large stylistic variation between the texts, we evaluate them separately.

Some languages in the CoNLL shared tasks were not originally annotated with dependency syntax, but only converted to a unified dependency format from other representations; and even for dependency treebanks the annotation schemata can vary considerably.

In all the treebanks, an artificial root node for each sentence placed before the sentence itself is used. To this root node, possibly several dependency analyses of parts of the sentence are attached. Equivalently, the representation of a sentence can be viewed as a forest consisting of dependency trees representing dependency analyses of parts of the sentence.

10.1.2 Reported tree and edge properties

We evaluate all global constraints dependency trees we studied in this thesis: we give counts of trees violating projectivity, planarity, and well-nestedness. The counts are also expressed in percentages relative to the total numbers of dependency trees.

Reported edge properties are: component and interval degrees; level types and counts of non-projective edges with negative and non-positive level types; level signatures and level signatures complemented with ancestry information (explained below); counts of non-projective edges with non-empty non-planar and ill-nested sets. Because of their theoretical importance, we also give counts of non-projective edges of non-positive and negative level types. All above properties are also enumerated in percentages relative to total numbers of non-projective edges. We also give the total numbers of non-projective edges and their percentages relative to all edges in the evaluated languages.

All but one of the tree and edge properties listed above have been thoroughly described in the theoretical part of the thesis. The only exception is level signature complemented with ancestry information: for each component level we indicate by a superscript whether the corresponding component root is an ancestor of the particular non-projective edge—“a” means that it is its ancestor, “n” means that it is not.

Note that a component level less than 2 cannot represent an ancestor component root; and a non-ancestor component root with any component level implies ill-nestedness (because it implies a non-empty ill-nested set of the corresponding non-projective edge). We hope that by including this property in the empirical evaluation we provide to the reader an even more detailed insight into what non-projective configurations occur in natural language treebanks.

Both level signatures and level signatures complemented with ancestry information are provided primarily to give a more detailed insight into non-projective structures occurring in different languages. We think that com-

bined with lexical information for nodes they can serve as a well-grounded basis for a linguistic inquiry into non-projective constructions in natural languages.

For all properties of non-projective edges taking several values, the values are always listed according to decreasing frequency; values with the same frequency are ordered according to increasing value of the particular property. Level signatures complemented with ancestry information are presented similarly to level signatures: component levels are ordered non-decreasingly, components differing only in ancestry information are ordered ancestors first, non-ancestors second.

Since the numbers of different values for level signatures and level signatures complemented with ancestry information are too large, for most languages we give only values whose counts are at least 5 and which at the same time amount to at least 0.1% of non-projective edges. Exceptions are Latin and Spanish; for both languages all values fit into the tables.

10.1.3 Note on computing the tree and edge properties

By conjoining dependency analyses of parts of a sentence under one artificial root node, we let all their edges interact. Since the artificial root comes before the sentence itself, it does not turn any edge non-projective. Edges from artificial roots may, however, introduce non-planarity. From Proposition 5.2.4 we know that if we considered these edges when examining properties of dependency trees, planarity and projectivity would get conflated.

Therefore, in our empirical evaluation we treat in a special way all edges from artificial root nodes. They are excluded from non-planar sets of non-projective edges (also from ill-nested sets of non-projective edges, but this does not affect them in any way). We also exclude them from the total numbers of edges.

Since we determine planarity and well-nestedness of whole dependency trees using non-planar and ill-nested sets, respectively, this in particular affects the counts of trees conforming to the planarity constraint; counts of well-nested trees are not affected. All other edge properties of non-projective edge are defined through their gaps, and so are left unaffected too.

Figure 10.1 exemplifies how this may affect counts of non-planar trees and non-planar sets of non-projective edges. The sample tree is non-planar according to Definition 5.1.1, however we do not consider it as such, because the pair of "crossing edges" involve an edge from the artificial root.

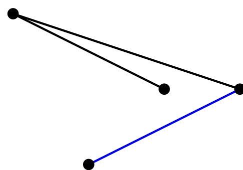


Figure 10.1: Sample non-projective dependency tree that we consider planar in empirical evaluation; edge in blue is considered non-projective, however its non-planar set is considered to be empty.

10.1.4 Program tools

All reported tree and edge properties were obtained using a dedicated Perl module for computing properties of non-projective edges. The module was used within the tree editor TrEd [Pajas, 2007].

The module provides functions for computing gaps of non-projective edges as well as other mathematical structures derived from them, such as non-planar and ill-nested sets. We report only a small subset of properties that it can compute.

Petr Pajas provided conversion tools from the CoNLL data format and the Latin Treebank data format into PML, the native data format of the tree editor TrEd.

The module for computing properties of non-projective edges has also been utilized in joint work on dependency parsing [Hall et al., 2007].

10.2 Empirical results

In this section, we present the counts for tree and edges properties. The section for each language starts on a new page and contains two tables: one with tree properties, the second one with edge properties. In the case of Latin, four subsections corresponding to four different authors (and four different works) are given.

10.2.1 Arabic

Table 10.1: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	1 (0.07%)
non-planar	150 (10.27%)
non-projective	163 (11.16%)
all	1460

Table 10.2: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 200 (94.79%), 2 / 10 (4.74%), 3 / 1 (0.47%)
ideg	1 / 211 (100.00%)
Type < 0	0 (0.00%)
Type ≤ 0	0 (0.00%)
Type	1 / 101 (47.87%), 2 / 58 (27.49%), 3 / 18 (8.53%), 4 / 10 (4.74%), 5 / 7 (3.32%), 6 / 6 (2.84%), 7 / 4 (1.90%), 9 / 1 (0.47%), 12 / 1 (0.47%), 13 / 1 (0.47%), 15 / 1 (0.47%), 16 / 1 (0.47%), 18 / 1 (0.47%), 21 / 1 (0.47%)
Signature	$\langle 1 \rangle$ / 92 (43.60%), $\langle 2 \rangle$ / 56 (26.54%), $\langle 3 \rangle$ / 18 (8.53%), $\langle 4 \rangle$ / 10 (4.74%), $\langle 1, 1 \rangle$ / 8 (3.79%), $\langle 5 \rangle$ / 7 (3.32%), $\langle 6 \rangle$ / 6 (2.84%), ...
Signature$^{\text{ancestry}}$	$\langle 1^{\text{n}} \rangle$ / 92 (43.60%), $\langle 2^{\text{a}} \rangle$ / 30 (14.22%), $\langle 2^{\text{n}} \rangle$ / 26 (12.32%), $\langle 3^{\text{n}} \rangle$ / 12 (5.69%), $\langle 4^{\text{n}} \rangle$ / 9 (4.27%), $\langle 1^{\text{n}}, 1^{\text{n}} \rangle$ / 8 (3.79%), $\langle 3^{\text{a}} \rangle$ / 6 (2.84%), $\langle 6^{\text{n}} \rangle$ / 6 (2.84%), $\langle 5^{\text{n}} \rangle$ / 5 (2.37%), ...
In $\neq \emptyset$	2 (0.95%)
Np $\neq \emptyset$	192 (91.00%)
non-projective	**211** (0.42% of all edges)
all edges	50097

10.2.2 Basque

Table 10.3: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	78 (2.45%)
non-planar	448 (14.04%)
non-projective	836 (26.21%)
all	3190

Table 10.4: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 1251 (84.41%), 2 / 177 (11.94%), 3 / 32 (2.16%), 4 / 12 (0.81%), 5 / 3 (0.20%), 6 / 3 (0.20%), 8 / 1 (0.07%), 10 / 1 (0.07%), 12 / 1 (0.07%), 14 / 1 (0.07%)
ideg	1 / 1383 (93.32%), 2 / 85 (5.74%), 3 / 7 (0.47%), 4 / 2 (0.13%), 6 / 1 (0.07%), 8 / 1 (0.07%), 10 / 1 (0.07%), 12 / 1 (0.07%), 14 / 1 (0.07%)
Type < 0	59 (3.98%)
Type ≤ 0	96 (6.48%)
Type	2 / 638 (43.05%), 1 / 471 (31.78%), 3 / 160 (10.80%), 4 / 77 (5.20%), 0 / 37 (2.50%), -1 / 33 (2.23%), -2 / 21 (1.42%), 5 / 20 (1.35%), 6 / 10 (0.67%), 7 / 5 (0.34%), -3 / 3 (0.20%), 8 / 3 (0.20%), 9 / 2 (0.13%), -5 / 1 (0.07%), -4 / 1 (0.07%)
Signature	$\langle 2\rangle$ / 561 (37.85%), $\langle 1\rangle$ / 415 (28.00%), $\langle 3\rangle$ / 120 (8.10%), $\langle 2,2\rangle$ / 50 (3.37%), $\langle 4\rangle$ / 46 (3.10%), $\langle 1,1\rangle$ / 40 (2.70%), $\langle 0\rangle$ / 34 (2.29%), $\langle -1\rangle$ / 31 (2.09%), $\langle -2\rangle$ / 20 (1.35%), $\langle 5\rangle$ / 12 (0.81%), $\langle 2,4\rangle$ / 10 (0.67%), $\langle 3,3\rangle$ / 10 (0.67%), $\langle 1,2\rangle$ / 9 (0.61%), $\langle 0,2\rangle$ / 7 (0.47%), $\langle 2,3\rangle$ / 7 (0.47%), $\langle 1,3\rangle$ / 5 (0.34%), $\langle 1,4\rangle$ / 5 (0.34%), $\langle 4,4\rangle$ / 5 (0.34%), $\langle 6,6\rangle$ / 5 (0.34%), $\langle 1,1,1\rangle$ / 5 (0.34%), $\langle 2,2,2\rangle$ / 5 (0.34%), ...

continued on next page

Property	value / count *or* count (proportion of non-projective edges)
Signature$^{\text{ancestry}}$	$\langle 1^{\text{n}} \rangle$ / 415 (28.00%), $\langle 2^{\text{a}} \rangle$ / 380 (25.64%), $\langle 2^{\text{n}} \rangle$ / 181 (12.21%), $\langle 3^{\text{n}} \rangle$ / 60 (4.05%), $\langle 3^{\text{a}} \rangle$ / 60 (4.05%), $\langle 1^{\text{n}}, 1^{\text{n}} \rangle$ / 40 (2.70%), $\langle 0^{\text{n}} \rangle$ / 34 (2.29%), $\langle 4^{\text{n}} \rangle$ / 32 (2.16%), $\langle -1^{\text{n}} \rangle$ / 31 (2.09%), $\langle 2^{\text{n}}, 2^{\text{n}} \rangle$ / 25 (1.69%), $\langle 2^{\text{a}}, 2^{\text{n}} \rangle$ / 25 (1.69%), $\langle -2^{\text{n}} \rangle$ / 20 (1.35%), $\langle 4^{\text{a}} \rangle$ / 14 (0.94%), $\langle 2^{\text{a}}, 3^{\text{n}} \rangle$ / 7 (0.47%), $\langle 0^{\text{n}}, 2^{\text{a}} \rangle$ / 6 (0.40%), $\langle 5^{\text{n}} \rangle$ / 6 (0.40%), $\langle 5^{\text{a}} \rangle$ / 6 (0.40%), $\langle 1^{\text{n}}, 1^{\text{n}}, 1^{\text{n}} \rangle$ / 5 (0.34%), $\langle 1^{\text{n}}, 2^{\text{n}} \rangle$ / 5 (0.34%), ...
$\mathsf{In} \neq \emptyset$	223 (15.05%)
$\mathsf{Np} \neq \emptyset$	717 (48.38%)
non-projective	**1482** (3.25% of all edges)
all edges	45630

10.2.3 Bulgarian

Table 10.5: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	0 (0.00%)
non-planar	677 (5.28%)
non-projective	690 (5.38%)
all	12823

Table 10.6: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 723 (99.72%), 2 / 1 (0.14%), 3 / 1 (0.14%)
ideg	1 / 724 (99.86%), 2 / 1 (0.14%)
Type < 0	0 (0.00%)
Type ≤ 0	0 (0.00%)
Type	2 / 674 (92.97%), 3 / 32 (4.41%), 1 / 12 (1.66%), 4 / 5 (0.69%), 5 / 2 (0.28%)
Signature	$\langle 2 \rangle$ / 674 (92.97%), $\langle 3 \rangle$ / 32 (4.41%), $\langle 1 \rangle$ / 10 (1.38%), $\langle 4 \rangle$ / 5 (0.69%), ...
Signature$^{\text{ancestry}}$	$\langle 2^{\text{a}} \rangle$ / 672 (92.69%), $\langle 3^{\text{a}} \rangle$ / 32 (4.41%), $\langle 1^{\text{n}} \rangle$ / 10 (1.38%), $\langle 4^{\text{a}} \rangle$ / 5 (0.69%), ...
In $\neq \emptyset$	0 (0.00%)
Np $\neq \emptyset$	712 (98.21%)
non-projective	**725** (0.41% of all edges)
all edges	177394

10.2.4 Catalan

Table 10.7: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	9 (0.06%)
non-planar	440 (2.94%)
non-projective	440 (2.94%)
all	14958

Table 10.8: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 448 (94.32%), 2 / 23 (4.84%), 3 / 3 (0.63%), 4 / 1 (0.21%)
ideg	1 / 469 (98.74%), 2 / 6 (1.26%)
Type < 0	3 (0.63%)
Type ≤ 0	7 (1.47%)
Type	1 / 204 (42.95%), 2 / 200 (42.11%), 3 / 45 (9.47%), 4 / 17 (3.58%), 0 / 4 (0.84%), 6 / 2 (0.42%), -4 / 1 (0.21%), -3 / 1 (0.21%), -1 / 1 (0.21%)
Signature	$\langle 2\rangle$ / 196 (41.26%), $\langle 1\rangle$ / 190 (40.00%), $\langle 3\rangle$ / 42 (8.84%), $\langle 4\rangle$ / 12 (2.53%), $\langle 1,1\rangle$ / 12 (2.53%), ...
Signature$^{\text{ancestry}}$	$\langle 1^{n}\rangle$ / 190 (40.00%), $\langle 2^{a}\rangle$ / 151 (31.79%), $\langle 2^{n}\rangle$ / 45 (9.47%), $\langle 3^{a}\rangle$ / 27 (5.68%), $\langle 3^{n}\rangle$ / 15 (3.16%), $\langle 1^{n}, 1^{n}\rangle$ / 12 (2.53%), $\langle 4^{a}\rangle$ / 8 (1.68%), ...
In $\neq \emptyset$	20 (4.21%)
Np $\neq \emptyset$	475 (100.00%)
non-projective	**475** (0.11% of all edges)
all edges	415884

10.2.5 Czech

Table 10.9: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	79 (0.11%)
non-planar	13783 (18.96%)
non-projective	16831 (23.15%)
all	72703

Table 10.10: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 23190 (98.39%), 2 / 292 (1.24%), 3 / 66 (0.28%), 4 / 11 (0.05%), 6 / 5 (0.02%), 5 / 2 (0.01%), 9 / 2 (0.01%), 12 / 2 (0.01%)
ideg	1 / 23376 (99.18%), 2 / 189 (0.80%), 3 / 3 (0.01%), 4 / 2 (0.01%)
Type < 0	4 (0.02%)
Type ≤ 0	75 (0.32%)
Type	2 / 18594 (78.89%), 1 / 3061 (12.99%), 3 / 1570 (6.66%), 4 / 203 (0.86%), 0 / 71 (0.30%), 5 / 40 (0.17%), 6 / 11 (0.05%), 7 / 6 (0.03%), -1 / 4 (0.02%), 9 / 4 (0.02%), 8 / 3 (0.01%), 11 / 2 (0.01%), 10 / 1 (0.00%)
Signature	$\langle 2\rangle$ / 18507 (78.52%), $\langle 1\rangle$ / 2886 (12.24%), $\langle 3\rangle$ / 1515 (6.43%), $\langle 4\rangle$ / 154 (0.65%), $\langle 1,1\rangle$ / 115 (0.49%), $\langle 0\rangle$ / 70 (0.30%), $\langle 2,2\rangle$ / 58 (0.25%), $\langle 1,1,1\rangle$ / 48 (0.20%), $\langle 2,4\rangle$ / 44 (0.19%), $\langle 1,3\rangle$ / 32 (0.14%), $\langle 5\rangle$ / 29 (0.12%), ...
Signature$^{\text{ancestry}}$	$\langle 2^a\rangle$ / 18292 (77.61%), $\langle 1^n\rangle$ / 2886 (12.24%), $\langle 3^a\rangle$ / 1438 (6.10%), $\langle 2^n\rangle$ / 215 (0.91%), $\langle 4^a\rangle$ / 119 (0.50%), $\langle 1^n,1^n\rangle$ / 115 (0.49%), $\langle 3^n\rangle$ / 77 (0.33%), $\langle 0^n\rangle$ / 70 (0.30%), $\langle 2^a,2^n\rangle$ / 50 (0.21%), $\langle 1^n,1^n,1^n\rangle$ / 48 (0.20%), $\langle 2^a,4^a\rangle$ / 41 (0.17%), $\langle 4^n\rangle$ / 35 (0.15%), $\langle 1^n,3^a\rangle$ / 32 (0.14%), ...
In $\neq \emptyset$	171 (0.73%)

continued on next page

Property	value / count *or* count (proportion of non-projective edges)
$\mathsf{Np} \neq \emptyset$	18758 (79.58%)
non-projective	**23570** (2.13% of all edges)
all edges	1105437

10.2.6 Danish

Table 10.11: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	6 (0.12%)
non-planar	787 (15.16%)
non-projective	811 (15.63%)
all	5190

Table 10.12: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 842 (89.10%), 2 / 78 (8.25%), 3 / 22 (2.33%), 4 / 3 (0.32%)
ideg	1 / 940 (99.47%), 2 / 5 (0.53%)
Type < 0	0 (0.00%)
Type ≤ 0	3 (0.32%)
Type	2 / 570 (60.32%), 1 / 197 (20.85%), 3 / 103 (10.90%), 4 / 43 (4.55%), 5 / 16 (1.69%), 6 / 7 (0.74%), 0 / 3 (0.32%), 7 / 3 (0.32%), 8 / 1 (0.11%), 10 / 1 (0.11%), 15 / 1 (0.11%)
Signature	$\langle 2\rangle$ / 555 (58.73%), $\langle 1\rangle$ / 115 (12.17%), $\langle 3\rangle$ / 100 (10.58%), $\langle 1,1\rangle$ / 63 (6.67%), $\langle 4\rangle$ / 41 (4.34%), $\langle 5\rangle$ / 16 (1.69%), $\langle 1,1,1\rangle$ / 16 (1.69%), $\langle 2,2\rangle$ / 7 (0.74%), ...
Signature$^{\text{ancestry}}$	$\langle 2^a\rangle$ / 537 (56.83%), $\langle 1^n\rangle$ / 115 (12.17%), $\langle 3^a\rangle$ / 92 (9.74%), $\langle 1^n,1^n\rangle$ / 63 (6.67%), $\langle 4^a\rangle$ / 39 (4.13%), $\langle 2^n\rangle$ / 18 (1.90%), $\langle 1^n,1^n,1^n\rangle$ / 16 (1.69%), $\langle 5^a\rangle$ / 10 (1.06%), $\langle 3^n\rangle$ / 8 (0.85%), $\langle 2^a,2^n\rangle$ / 7 (0.74%), $\langle 2^a,2^n,2^n\rangle$ / 6 (0.63%), $\langle 5^n\rangle$ / 6 (0.63%), ...
In $\neq \emptyset$	13 (1.38%)
Np $\neq \emptyset$	888 (93.97%)
non-projective	**945** (1.06% of all edges)
all edges	89171

10.2.7 Dutch

Table 10.13: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	15 (0.11%)
non-planar	4115 (30.83%)
non-projective	4865 (36.44%)
all	13349

Table 10.14: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 10264 (97.14%), 2 / 238 (2.25%), 3 / 47 (0.44%), 4 / 8 (0.08%), 5 / 6 (0.06%), 7 / 3 (0.03%)
ideg	1 / 10209 (96.62%), 2 / 349 (3.30%), 3 / 8 (0.08%)
Type < 0	0 (0.00%)
Type ≤ 0	2 (0.02%)
Type	2 / 8129 (76.94%), 3 / 1509 (14.28%), 1 / 660 (6.25%), 4 / 228 (2.16%), 5 / 29 (0.27%), 6 / 5 (0.05%), 0 / 2 (0.02%), 7 / 2 (0.02%), 8 / 2 (0.02%)
Signature	$\langle 2\rangle$ / 8061 (76.29%), $\langle 3\rangle$ / 1461 (13.83%), $\langle 1\rangle$ / 512 (4.85%), $\langle 4\rangle$ / 201 (1.90%), $\langle 1,1\rangle$ / 118 (1.12%), $\langle 2,2\rangle$ / 52 (0.49%), $\langle 1,1,1\rangle$ / 25 (0.24%), $\langle 5\rangle$ / 23 (0.22%), $\langle 1,3\rangle$ / 16 (0.15%), $\langle 3,3\rangle$ / 15 (0.14%), $\langle 2,4\rangle$ / 12 (0.11%), ...
Signature$^{\text{ancestry}}$	$\langle 2^a\rangle$ / 8002 (75.73%), $\langle 3^a\rangle$ / 1452 (13.74%), $\langle 1^n\rangle$ / 512 (4.85%), $\langle 4^a\rangle$ / 200 (1.89%), $\langle 1^n,1^n\rangle$ / 118 (1.12%), $\langle 2^n\rangle$ / 59 (0.56%), $\langle 2^a,2^n\rangle$ / 33 (0.31%), $\langle 1^n,1^n,1^n\rangle$ / 25 (0.24%), $\langle 5^a\rangle$ / 22 (0.21%), $\langle 2^n,2^n\rangle$ / 19 (0.18%), $\langle 1^n,3^a\rangle$ / 15 (0.14%), $\langle 3^n,3^n\rangle$ / 13 (0.12%), $\langle 2^a,4^a\rangle$ / 12 (0.11%), ...
In $\neq \emptyset$	33 (0.31%)
Np $\neq \emptyset$	9054 (85.69%)
non-projective	**10566** (5.90% of all edges)

continued on next page

Property	value / count *or* count (proportion of non-projective edges)
all edges	179063

10.2.8 English

Table 10.15: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	116 (0.62%)
non-planar	1248 (6.72%)
non-projective	1248 (6.72%)
all	18577

Table 10.16: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 1170 (79.43%), 2 / 242 (16.43%), 3 / 37 (2.51%), 4 / 9 (0.61%), 5 / 4 (0.27%), 6 / 4 (0.27%), 7 / 2 (0.14%), 9 / 2 (0.14%), 8 / 1 (0.07%), 11 / 1 (0.07%), 12 / 1 (0.07%)
ideg	1 / 1432 (97.22%), 2 / 25 (1.70%), 3 / 6 (0.41%), 4 / 5 (0.34%), 5 / 5 (0.34%)
Type < 0	0 (0.00%)
Type ≤ 0	5 (0.34%)
Type	2 / 868 (58.93%), 3 / 338 (22.95%), 1 / 210 (14.26%), 4 / 41 (2.78%), 5 / 9 (0.61%), 0 / 5 (0.34%), 6 / 1 (0.07%), 7 / 1 (0.07%)
Signature	$\langle 2\rangle$ / 824 (55.94%), $\langle 3\rangle$ / 173 (11.74%), $\langle 1\rangle$ / 140 (9.50%), $\langle 0,3\rangle$ / 70 (4.75%), $\langle 1,3\rangle$ / 67 (4.55%), $\langle 1,1\rangle$ / 51 (3.46%), $\langle 4\rangle$ / 25 (1.70%), $\langle 2,2\rangle$ / 13 (0.88%), $\langle 1,1,1\rangle$ / 9 (0.61%), $\langle 0,1\rangle$ / 6 (0.41%), $\langle 0,0,3\rangle$ / 6 (0.41%), $\langle 1,4\rangle$ / 6 (0.41%), $\langle 2,4\rangle$ / 6 (0.41%), $\langle 1,1,3\rangle$ / 6 (0.41%), $\langle -1,3\rangle$ / 5 (0.34%), $\langle 0\rangle$ / 5 (0.34%), ...

continued on next page

Property	value / count *or* count (proportion of non-projective edges)
Signature$^{\text{ancestry}}$	$\langle 2^{a} \rangle$ / 817 (55.47%), $\langle 3^{a} \rangle$ / 169 (11.47%), $\langle 1^{n} \rangle$ / 140 (9.50%), $\langle 0^{n}, 3^{a} \rangle$ / 70 (4.75%), $\langle 1^{n}, 3^{a} \rangle$ / 67 (4.55%), $\langle 1^{n}, 1^{n} \rangle$ / 51 (3.46%), $\langle 4^{a} \rangle$ / 24 (1.63%), $\langle 1^{n}, 1^{n}, 1^{n} \rangle$ / 9 (0.61%), $\langle 2^{n} \rangle$ / 7 (0.48%), $\langle 2^{n}, 2^{n} \rangle$ / 7 (0.48%), $\langle 0^{n}, 0^{n}, 3^{a} \rangle$ / 6 (0.41%), $\langle 0^{n}, 1^{n} \rangle$ / 6 (0.41%), $\langle 1^{n}, 1^{n}, 3^{a} \rangle$ / 6 (0.41%), $\langle 1^{n}, 4^{a} \rangle$ / 6 (0.41%), $\langle 2^{a}, 2^{n} \rangle$ / 6 (0.41%), $\langle 0^{n} \rangle$ / 5 (0.34%), $\langle 2^{n}, 4^{a} \rangle$ / 5 (0.34%), ...
$\mathsf{In} \neq \emptyset$	278 (18.87%)
$\mathsf{Np} \neq \emptyset$	1473 (100.00%)
non-projective	**1473** (0.34% of all edges)
all edges	427946

10.2.9 German

Table 10.17: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	416 (1.06%)
non-planar	10865 (27.71%)
non-projective	10883 (27.75%)
all	39216

Table 10.18: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 13107 (82.73%), 2 / 2206 (13.92%), 3 / 434 (2.74%), 4 / 77 (0.49%), 5 / 13 (0.08%), 6 / 5 (0.03%), 7 / 1 (0.01%), 8 / 1 (0.01%)
ideg	1 / 14605 (92.18%), 2 / 1198 (7.56%), 3 / 37 (0.23%), 4 / 4 (0.03%)
Type < 0	0 (0.00%)
Type ≤ 0	41 (0.26%)
Type	2 / 9018 (56.92%), 1 / 5018 (31.67%), 3 / 1566 (9.88%), 4 / 172 (1.09%), 0 / 41 (0.26%), 5 / 24 (0.15%), 6 / 5 (0.03%)
Signature	$\langle 2\rangle$ / 8407 (53.06%), $\langle 1\rangle$ / 3112 (19.64%), $\langle 1,1\rangle$ / 1503 (9.49%), $\langle 3\rangle$ / 1397 (8.82%), $\langle 2,2\rangle$ / 476 (3.00%), $\langle 1,1,1\rangle$ / 312 (1.97%), $\langle 4\rangle$ / 136 (0.86%), $\langle 3,3\rangle$ / 98 (0.62%), $\langle 2,2,2\rangle$ / 69 (0.44%), $\langle 1,1,1,1\rangle$ / 59 (0.37%), $\langle 1,3\rangle$ / 47 (0.30%), $\langle 0\rangle$ / 38 (0.24%), $\langle 0,2\rangle$ / 22 (0.14%), ...
Signature$^{\text{ancestry}}$	$\langle 2^a\rangle$ / 7970 (50.30%), $\langle 1^n\rangle$ / 3112 (19.64%), $\langle 1^n,1^n\rangle$ / 1503 (9.49%), $\langle 3^a\rangle$ / 1389 (8.77%), $\langle 2^n\rangle$ / 437 (2.76%), $\langle 2^a,2^n\rangle$ / 331 (2.09%), $\langle 1^n,1^n,1^n\rangle$ / 312 (1.97%), $\langle 2^n,2^n\rangle$ / 145 (0.92%), $\langle 4^a\rangle$ / 136 (0.86%), $\langle 3^a,3^n\rangle$ / 84 (0.53%), $\langle 1^n,1^n,1^n,1^n\rangle$ / 59 (0.37%), $\langle 2^a,2^n,2^n\rangle$ / 49 (0.31%), $\langle 1^n,3^a\rangle$ / 47 (0.30%), $\langle 0^n\rangle$ / 38 (0.24%), $\langle 0^n,2^a\rangle$ / 22 (0.14%), $\langle 2^n,2^n,2^n\rangle$ / 20 (0.13%), ...

continued on next page

Property	value / count *or* count (proportion of non-projective edges)
$\mathsf{In} \neq \emptyset$	1013 (6.39%)
$\mathsf{Np} \neq \emptyset$	15824 (99.87%)
non-projective	**15844** (2.40% of all edges)
all edges	660394

10.2.10 Greek

Table 10.19: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	0 (0.00%)
non-planar	454 (16.78%)
non-projective	549 (20.30%)
all	2705

Table 10.20: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 738 (98.40%), 2 / 10 (1.33%), 3 / 2 (0.27%)
ideg	1 / 741 (98.80%), 2 / 9 (1.20%)
Type < 0	0 (0.00%)
Type ≤ 0	0 (0.00%)
Type	2 / 460 (61.33%), 1 / 177 (23.60%), 3 / 87 (11.60%), 4 / 11 (1.47%), 5 / 9 (1.20%), 6 / 3 (0.40%), 7 / 1 (0.13%), 8 / 1 (0.13%), 10 / 1 (0.13%)
Signature	$\langle 2 \rangle$ / 456 (60.80%), $\langle 1 \rangle$ / 169 (22.53%), $\langle 3 \rangle$ / 87 (11.60%), $\langle 4 \rangle$ / 11 (1.47%), $\langle 5 \rangle$ / 9 (1.20%), $\langle 1, 1 \rangle$ / 6 (0.80%), ...
Signature$^{\text{ancestry}}$	$\langle 2^{a} \rangle$ / 407 (54.27%), $\langle 1^{n} \rangle$ / 169 (22.53%), $\langle 3^{a} \rangle$ / 60 (8.00%), $\langle 2^{n} \rangle$ / 49 (6.53%), $\langle 3^{n} \rangle$ / 27 (3.60%), $\langle 4^{n} \rangle$ / 7 (0.93%), $\langle 5^{n} \rangle$ / 7 (0.93%), $\langle 1^{n}, 1^{n} \rangle$ / 6 (0.80%), ...
In $\neq \emptyset$	0 (0.00%)
Np $\neq \emptyset$	621 (82.80%)
non-projective	**750** (1.25% of all edges)
all edges	59983

10.2.11 Hungarian

Table 10.21: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	4 (0.07%)
non-planar	1590 (26.35%)
non-projective	1590 (26.35%)
all	6034

Table 10.22: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 3013 (79.00%), 2 / 515 (13.50%), 3 / 201 (5.27%), 4 / 50 (1.31%), 5 / 17 (0.45%), 6 / 8 (0.21%), 7 / 6 (0.16%), 8 / 3 (0.08%), 10 / 1 (0.03%)
ideg	1 / 3483 (91.32%), 2 / 279 (7.32%), 3 / 43 (1.13%), 4 / 8 (0.21%), 5 / 1 (0.03%)
Type < 0	0 (0.00%)
Type ≤ 0	0 (0.00%)
Type	2 / 2553 (66.94%), 1 / 702 (18.41%), 3 / 461 (12.09%), 4 / 84 (2.20%), 5 / 10 (0.26%), 6 / 4 (0.10%)
Signature	$\langle 2\rangle$ / 2228 (58.42%), $\langle 3\rangle$ / 406 (10.64%), $\langle 1,1\rangle$ / 312 (8.18%), $\langle 1\rangle$ / 297 (7.79%), $\langle 2,2\rangle$ / 166 (4.35%), $\langle 2,2,2\rangle$ / 118 (3.09%), $\langle 4\rangle$ / 72 (1.89%), $\langle 1,1,1\rangle$ / 58 (1.52%), $\langle 2,2,2,2\rangle$ / 30 (0.79%), $\langle 2,3\rangle$ / 18 (0.47%), $\langle 1,1,1,1,1\rangle$ / 12 (0.31%), $\langle 3,3,3\rangle$ / 11 (0.29%), $\langle 1,1,1,1\rangle$ / 11 (0.29%), $\langle 2,3,3\rangle$ / 8 (0.21%), $\langle 5\rangle$ / 7 (0.18%), $\langle 3,3\rangle$ / 6 (0.16%), $\langle 4,4\rangle$ / 6 (0.16%), ...

continued on next page

Property	value / count *or* count (proportion of non-projective edges)
Signature$^{\text{ancestry}}$	$\langle 2^a \rangle$ / 2208 (57.89%), $\langle 3^a \rangle$ / 405 (10.62%), $\langle 1^n, 1^n \rangle$ / 312 (8.18%), $\langle 1^n \rangle$ / 297 (7.79%), $\langle 2^a, 2^n \rangle$ / 147 (3.85%), $\langle 2^a, 2^n, 2^n \rangle$ / 107 (2.81%), $\langle 4^a \rangle$ / 72 (1.89%), $\langle 1^n, 1^n, 1^n \rangle$ / 58 (1.52%), $\langle 2^a, 2^n, 2^n, 2^n \rangle$ / 29 (0.76%), $\langle 2^n \rangle$ / 20 (0.52%), $\langle 2^n, 2^n \rangle$ / 19 (0.50%), $\langle 2^a, 3^n \rangle$ / 18 (0.47%), $\langle 1^n, 1^n, 1^n, 1^n, 1^n \rangle$ / 12 (0.31%), $\langle 1^n, 1^n, 1^n, 1^n \rangle$ / 11 (0.29%), $\langle 2^n, 2^n, 2^n \rangle$ / 11 (0.29%), $\langle 3^a, 3^n, 3^n \rangle$ / 11 (0.29%), $\langle 2^a, 3^n, 3^n \rangle$ / 8 (0.21%), $\langle 5^a \rangle$ / 7 (0.18%), $\langle 1^n, 1^n, 1^n, 1^n, 1^n, 1^n \rangle$ / 6 (0.16%), $\langle 3^a, 3^n, 3^n, 3^n \rangle$ / 6 (0.16%), $\langle 4^a, 4^n \rangle$ / 6 (0.16%), ...
$\mathsf{In} \neq \emptyset$	8 (0.21%)
$\mathsf{Np} \neq \emptyset$	3814 (100.00%)
non-projective	**3814** (3.03% of all edges)
all edges	125765

10.2.12 Italian

Table 10.23: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	6 (0.19%)
non-planar	139 (4.47%)
non-projective	229 (7.36%)
all	3110

Table 10.24: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 304 (90.75%), 2 / 27 (8.06%), 3 / 2 (0.60%), 4 / 1 (0.30%), 5 / 1 (0.30%)
ideg	1 / 325 (97.01%), 2 / 10 (2.99%)
Type < 0	0 (0.00%)
Type ≤ 0	6 (1.79%)
Type	2 / 159 (47.46%), 1 / 114 (34.03%), 3 / 24 (7.16%), 4 / 13 (3.88%), 9 / 9 (2.69%), 0 / 6 (1.79%), 6 / 4 (1.19%), 5 / 2 (0.60%), 7 / 2 (0.60%), 8 / 2 (0.60%)
Signature	$\langle 2\rangle$ / 152 (45.37%), $\langle 1\rangle$ / 97 (28.96%), $\langle 3\rangle$ / 21 (6.27%), $\langle 1,1\rangle$ / 14 (4.18%), $\langle 4\rangle$ / 12 (3.58%), $\langle 9\rangle$ / 9 (2.69%), $\langle 2,2\rangle$ / 6 (1.79%), $\langle 0\rangle$ / 5 (1.49%), ...
Signature$^{\text{ancestry}}$	$\langle 2^{a}\rangle$ / 125 (37.31%), $\langle 1^{n}\rangle$ / 97 (28.96%), $\langle 2^{n}\rangle$ / 27 (8.06%), $\langle 3^{a}\rangle$ / 17 (5.07%), $\langle 1^{n}, 1^{n}\rangle$ / 14 (4.18%), $\langle 4^{n}\rangle$ / 12 (3.58%), $\langle 9^{n}\rangle$ / 9 (2.69%), $\langle 0^{n}\rangle$ / 5 (1.49%), ...
In $\neq \emptyset$	14 (4.18%)
Np $\neq \emptyset$	197 (58.81%)
non-projective	**335** (0.50% of all edges)
all edges	67360

10.2.13 Japanese

Table 10.25: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	0 (0.00%)
non-planar	1 (0.01%)
non-projective	902 (5.29%)
all	17044

Table 10.26: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 1484 (89.02%), 2 / 143 (8.58%), 3 / 26 (1.56%), 4 / 10 (0.60%), 5 / 4 (0.24%)
ideg	1 / 1570 (94.18%), 2 / 81 (4.86%), 3 / 12 (0.72%), 4 / 3 (0.18%), 5 / 1 (0.06%)
Type < 0	0 (0.00%)
Type ≤ 0	0 (0.00%)
Type	1 / 550 (32.99%), 2 / 231 (13.86%), 3 / 205 (12.30%), 4 / 203 (12.18%), 5 / 139 (8.34%), 6 / 122 (7.32%), 7 / 87 (5.22%), 8 / 54 (3.24%), 9 / 35 (2.10%), 10 / 20 (1.20%), 11 / 9 (0.54%), 13 / 6 (0.36%), 12 / 4 (0.24%), 14 / 1 (0.06%), 17 / 1 (0.06%)
Signature	$\langle 1\rangle$ / 466 (27.95%), $\langle 2\rangle$ / 209 (12.54%), $\langle 4\rangle$ / 186 (11.16%), $\langle 3\rangle$ / 183 (10.98%), $\langle 5\rangle$ / 126 (7.56%), $\langle 6\rangle$ / 113 (6.78%), $\langle 7\rangle$ / 78 (4.68%), $\langle 1,1\rangle$ / 63 (3.78%), $\langle 8\rangle$ / 49 (2.94%), $\langle 9\rangle$ / 35 (2.10%), $\langle 10\rangle$ / 20 (1.20%), $\langle 3,3\rangle$ / 19 (1.14%), $\langle 4,4\rangle$ / 16 (0.96%), $\langle 2,2\rangle$ / 15 (0.90%), $\langle 5,5\rangle$ / 13 (0.78%), $\langle 1,1,1\rangle$ / 10 (0.60%), $\langle 6,6\rangle$ / 8 (0.48%), $\langle 11\rangle$ / 7 (0.42%), $\langle 1,1,1,1\rangle$ / 7 (0.42%), $\langle 13\rangle$ / 6 (0.36%), $\langle 2,2,2\rangle$ / 6 (0.36%), ...

continued on next page

Property	value / count *or* count (proportion of non-projective edges)
Signature$^{\text{ancestry}}$	$\langle 1^n \rangle$ / 466 (27.95%), $\langle 2^n \rangle$ / 209 (12.54%), $\langle 4^n \rangle$ / 186 (11.16%), $\langle 3^n \rangle$ / 183 (10.98%), $\langle 5^n \rangle$ / 126 (7.56%), $\langle 6^n \rangle$ / 113 (6.78%), $\langle 7^n \rangle$ / 78 (4.68%), $\langle 1^n, 1^n \rangle$ / 63 (3.78%), $\langle 8^n \rangle$ / 49 (2.94%), $\langle 9^n \rangle$ / 35 (2.10%), $\langle 10^n \rangle$ / 20 (1.20%), $\langle 3^n, 3^n \rangle$ / 19 (1.14%), $\langle 4^n, 4^n \rangle$ / 16 (0.96%), $\langle 2^n, 2^n \rangle$ / 15 (0.90%), $\langle 5^n, 5^n \rangle$ / 13 (0.78%), $\langle 1^n, 1^n, 1^n \rangle$ / 10 (0.60%), $\langle 6^n, 6^n \rangle$ / 8 (0.48%), $\langle 1^n, 1^n, 1^n, 1^n \rangle$ / 7 (0.42%), $\langle 11^n \rangle$ / 7 (0.42%), $\langle 2^n, 2^n, 2^n \rangle$ / 6 (0.36%), $\langle 13^n \rangle$ / 6 (0.36%), $\langle 7^n, 7^n, 7^n \rangle$ / 5 (0.30%), ...
$\mathsf{In} \neq \emptyset$	0 (0.00%)
$\mathsf{Np} \neq \emptyset$	1 (0.06%)
non-projective	**1667** (1.32% of all edges)
all edges	126511

10.2.14 Latin

Cicero – *Oratio in Catilinam*

Table 10.27: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	2 (2.99%)
non-planar	40 (59.70%)
non-projective	41 (61.19%)
all	67

Table 10.28: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 61 (82.43%), 2 / 11 (14.86%), 3 / 2 (2.70%)
ideg	1 / 73 (98.65%), 2 / 1 (1.35%)
Type < 0	0 (0.00%)
Type ≤ 0	0 (0.00%)
Type	2 / 43 (58.11%), 1 / 22 (29.73%), 3 / 8 (10.81%), 4 / 1 (1.35%)
Signature	$\langle 2\rangle$ / 40 (54.05%), $\langle 1\rangle$ / 14 (18.92%), $\langle 3\rangle$ / 7 (9.46%), $\langle 1,1\rangle$ / 5 (6.76%), $\langle 0,1\rangle$ / 2 (2.70%), $\langle 2,2\rangle$ / 2 (2.70%), $\langle 1,3\rangle$ / 1 (1.35%), $\langle 2,4\rangle$ / 1 (1.35%), $\langle 1,1,1\rangle$ / 1 (1.35%), $\langle 1,2,2\rangle$ / 1 (1.35%)
Signature$^{\text{ancestry}}$	$\langle 2^{a}\rangle$ / 39 (52.70%), $\langle 1^{n}\rangle$ / 14 (18.92%), $\langle 1^{n},1^{n}\rangle$ / 5 (6.76%), $\langle 3^{a}\rangle$ / 4 (5.41%), $\langle 3^{n}\rangle$ / 3 (4.05%), $\langle 0^{n},1^{n}\rangle$ / 2 (2.70%), $\langle 2^{a},2^{n}\rangle$ / 2 (2.70%), $\langle 1^{n},1^{n},1^{n}\rangle$ / 1 (1.35%), $\langle 1^{n},2^{n},2^{n}\rangle$ / 1 (1.35%), $\langle 1^{n},3^{a}\rangle$ / 1 (1.35%), $\langle 2^{n}\rangle$ / 1 (1.35%), $\langle 2^{a},4^{a}\rangle$ / 1 (1.35%)
In $\neq \emptyset$	4 (5.41%)
Np $\neq \emptyset$	71 (95.95%)
non-projective	**74** (6.53% of all edges)
all edges	1133

Caesar – *Commentarii de Bello Gallico*

Table 10.29: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	0 (0.00%)
non-planar	32 (45.07%)
non-projective	33 (46.48%)
all	71

Table 10.30: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 46 (97.87%), 3 / 1 (2.13%)
ideg	1 / 47 (100.00%)
Type < 0	0 (0.00%)
Type ≤ 0	0 (0.00%)
Type	2 / 26 (55.32%), 1 / 14 (29.79%), 3 / 7 (14.89%)
Signature	$\langle 2 \rangle$ / 26 (55.32%), $\langle 1 \rangle$ / 13 (27.66%), $\langle 3 \rangle$ / 7 (14.89%), $\langle 1, 1, 1 \rangle$ / 1 (2.13%)
Signature$^{\text{ancestry}}$	$\langle 2^{\text{a}} \rangle$ / 26 (55.32%), $\langle 1^{\text{n}} \rangle$ / 13 (27.66%), $\langle 3^{\text{a}} \rangle$ / 7 (14.89%), $\langle 1^{\text{n}}, 1^{\text{n}}, 1^{\text{n}} \rangle$ / 1 (2.13%)
In $\neq \emptyset$	0 (0.00%)
Np $\neq \emptyset$	45 (95.74%)
non-projective	**47** (3.32% of all edges)
all edges	1414

Vergil – *Aeneid*

Table 10.31: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	13 (7.30%)
non-planar	122 (68.54%)
non-projective	126 (70.79%)
all	178

Table 10.32: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 270 (90.91%), 2 / 24 (8.08%), 3 / 3 (1.01%)
ideg	1 / 281 (94.61%), 2 / 16 (5.39%)
Type < 0	1 (0.34%)
Type ≤ 0	12 (4.04%)
Type	2 / 181 (60.94%), 1 / 56 (18.86%), 3 / 36 (12.12%), 0 / 11 (3.70%), 4 / 11 (3.70%), -1 / 1 (0.34%), 5 / 1 (0.34%)
Signature	$\langle 2\rangle$ / 176 (59.26%), $\langle 1\rangle$ / 43 (14.48%), $\langle 3\rangle$ / 34 (11.45%), $\langle 0\rangle$ / 11 (3.70%), $\langle 1,1\rangle$ / 10 (3.37%), $\langle 4\rangle$ / 4 (1.35%), $\langle 2,4\rangle$ / 4 (1.35%), $\langle 2,2\rangle$ / 3 (1.01%), $\langle 0,2\rangle$ / 2 (0.67%), $\langle 2,2,4\rangle$ / 2 (0.67%), $\langle -2,1\rangle$ / 1 (0.34%), $\langle -1\rangle$ / 1 (0.34%), $\langle 0,1\rangle$ / 1 (0.34%), $\langle 5\rangle$ / 1 (0.34%), $\langle 1,4\rangle$ / 1 (0.34%), $\langle 2,3\rangle$ / 1 (0.34%), $\langle 3,3\rangle$ / 1 (0.34%), $\langle 1,1,1\rangle$ / 1 (0.34%)
Signature$^{\text{ancestry}}$	$\langle 2^a\rangle$ / 172 (57.91%), $\langle 1^n\rangle$ / 43 (14.48%), $\langle 3^a\rangle$ / 34 (11.45%), $\langle 0^n\rangle$ / 11 (3.70%), $\langle 1^n,1^n\rangle$ / 10 (3.37%), $\langle 2^n\rangle$ / 4 (1.35%), $\langle 4^a\rangle$ / 4 (1.35%), $\langle 2^a,2^n\rangle$ / 3 (1.01%), $\langle 2^a,4^a\rangle$ / 3 (1.01%), $\langle 0^n,2^a\rangle$ / 2 (0.67%), $\langle 2^a,2^n,4^a\rangle$ / 2 (0.67%), $\langle -2^n,1^n\rangle$ / 1 (0.34%), $\langle -1^n\rangle$ / 1 (0.34%), $\langle 0^n,1^n\rangle$ / 1 (0.34%), $\langle 1^n,1^n,1^n\rangle$ / 1 (0.34%), $\langle 1^n,4^a\rangle$ / 1 (0.34%), $\langle 2^n,4^a\rangle$ / 1 (0.34%), $\langle 2^a,3^n\rangle$ / 1 (0.34%), $\langle 3^a,3^n\rangle$ / 1 (0.34%), $\langle 5^a\rangle$ / 1 (0.34%)
In $\neq \emptyset$	28 (9.43%)
Np $\neq \emptyset$	285 (95.96%)
non-projective	**297** (12.25% of all edges)

continued on next page

Property	value / count *or* count (proportion of non-projective edges)
all edges	2424

Jerome – *Vulgate*

Table 10.33: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	0 (0.00%)
non-planar	104 (25.68%)
non-projective	110 (27.16%)
all	405

Table 10.34: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 149 (98.03%), 2 / 2 (1.32%), 3 / 1 (0.66%)
ideg	1 / 150 (98.68%), 2 / 2 (1.32%)
Type < 0	0 (0.00%)
Type ≤ 0	0 (0.00%)
Type	2 / 104 (68.42%), 1 / 34 (22.37%), 3 / 11 (7.24%), 4 / 2 (1.32%), 5 / 1 (0.66%)
Signature	$\langle 2\rangle$ / 103 (67.76%), $\langle 1\rangle$ / 32 (21.05%), $\langle 3\rangle$ / 11 (7.24%), $\langle 4\rangle$ / 2 (1.32%), $\langle 5\rangle$ / 1 (0.66%), $\langle 1,1\rangle$ / 1 (0.66%), $\langle 2,2\rangle$ / 1 (0.66%), $\langle 1,1,1\rangle$ / 1 (0.66%)
Signature$^{\text{ancestry}}$	$\langle 2^{a}\rangle$ / 101 (66.45%), $\langle 1^{n}\rangle$ / 32 (21.05%), $\langle 3^{a}\rangle$ / 11 (7.24%), $\langle 2^{n}\rangle$ / 2 (1.32%), $\langle 4^{a}\rangle$ / 2 (1.32%), $\langle 1^{n},1^{n}\rangle$ / 1 (0.66%), $\langle 1^{n},1^{n},1^{n}\rangle$ / 1 (0.66%), $\langle 2^{a},2^{n}\rangle$ / 1 (0.66%), $\langle 5^{a}\rangle$ / 1 (0.66%)
In $\neq \emptyset$	0 (0.00%)
Np $\neq \emptyset$	141 (92.76%)
non-projective	**152** (1.92% of all edges)
all edges	7899

10.2.15 Portuguese

Table 10.35: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	7 (0.08%)
non-planar	1713 (18.88%)
non-projective	1718 (18.94%)
all	9071

Table 10.36: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 2466 (91.27%), 2 / 151 (5.59%), 3 / 64 (2.37%), 4 / 14 (0.52%), 5 / 7 (0.26%)
ideg	1 / 2398 (88.75%), 2 / 272 (10.07%), 3 / 24 (0.89%), 4 / 7 (0.26%), 5 / 1 (0.04%)
Type < 0	0 (0.00%)
Type ≤ 0	3 (0.11%)
Type	2 / 1721 (63.69%), 1 / 742 (27.46%), 3 / 219 (8.11%), 4 / 12 (0.44%), 0 / 3 (0.11%), 6 / 3 (0.11%), 5 / 1 (0.04%), 7 / 1 (0.04%)
Signature	$\langle 2\rangle$ / 1670 (61.81%), $\langle 1\rangle$ / 571 (21.13%), $\langle 3\rangle$ / 208 (7.70%), $\langle 1,1\rangle$ / 113 (4.18%), $\langle 1,1,1\rangle$ / 44 (1.63%), $\langle 2,2\rangle$ / 29 (1.07%), $\langle 2,2,2\rangle$ / 13 (0.48%), $\langle 4\rangle$ / 12 (0.44%), $\langle 1,1,1,1\rangle$ / 7 (0.26%), $\langle 1,1,1,1,1\rangle$ / 6 (0.22%), ...
Signature$^{\text{ancestry}}$	$\langle 2^{a}\rangle$ / 1604 (59.36%), $\langle 1^{n}\rangle$ / 571 (21.13%), $\langle 3^{a}\rangle$ / 201 (7.44%), $\langle 1^{n},1^{n}\rangle$ / 113 (4.18%), $\langle 2^{n}\rangle$ / 66 (2.44%), $\langle 1^{n},1^{n},1^{n}\rangle$ / 44 (1.63%), $\langle 2^{n},2^{n}\rangle$ / 20 (0.74%), $\langle 2^{n},2^{n},2^{n}\rangle$ / 13 (0.48%), $\langle 4^{a}\rangle$ / 10 (0.37%), $\langle 2^{a},2^{n}\rangle$ / 9 (0.33%), $\langle 1^{n},1^{n},1^{n},1^{n}\rangle$ / 7 (0.26%), $\langle 3^{n}\rangle$ / 7 (0.26%), $\langle 1^{n},1^{n},1^{n},1^{n},1^{n}\rangle$ / 6 (0.22%), ...
In $\neq \emptyset$	25 (0.93%)
Np $\neq \emptyset$	2695 (99.74%)
non-projective	**2702** (1.37% of all edges)

continued on next page

Property	value / count *or* count (proportion of non-projective edges)
all edges	197607

10.2.16 Slovene

Table 10.37: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	3 (0.20%)
non-planar	283 (18.45%)
non-projective	340 (22.16%)
all	1534

Table 10.38: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 531 (96.55%), 2 / 11 (2.00%), 4 / 4 (0.73%), 3 / 2 (0.36%), 5 / 2 (0.36%)
ideg	1 / 548 (99.64%), 2 / 2 (0.36%)
Type < 0	2 (0.36%)
Type ≤ 0	3 (0.55%)
Type	2 / 385 (70.00%), 1 / 78 (14.18%), 3 / 50 (9.09%), 4 / 13 (2.36%), 5 / 13 (2.36%), 6 / 5 (0.91%), 7 / 2 (0.36%), -5 / 1 (0.18%), -1 / 1 (0.18%), 0 / 1 (0.18%), 8 / 1 (0.18%)
Signature	$\langle 2\rangle$ / 384 (69.82%), $\langle 1\rangle$ / 67 (12.18%), $\langle 3\rangle$ / 45 (8.18%), $\langle 4\rangle$ / 13 (2.36%), $\langle 5\rangle$ / 12 (2.18%), $\langle 1,1\rangle$ / 6 (1.09%), ...
Signature$^{\text{ancestry}}$	$\langle 2^{a}\rangle$ / 346 (62.91%), $\langle 1^{n}\rangle$ / 67 (12.18%), $\langle 2^{n}\rangle$ / 38 (6.91%), $\langle 3^{a}\rangle$ / 35 (6.36%), $\langle 5^{n}\rangle$ / 12 (2.18%), $\langle 3^{n}\rangle$ / 10 (1.82%), $\langle 4^{n}\rangle$ / 8 (1.45%), $\langle 1^{n}, 1^{n}\rangle$ / 6 (1.09%), $\langle 4^{a}\rangle$ / 5 (0.91%), ...
In $\neq \emptyset$	6 (1.09%)
Np $\neq \emptyset$	373 (67.82%)
non-projective	**550** (2.13% of all edges)
all edges	25777

10.2.17 Spanish

Table 10.39: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	0 (0.00%)
non-planar	56 (1.69%)
non-projective	57 (1.72%)
all	3306

Table 10.40: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 59 (100.00%)
ideg	1 / 58 (98.31%), 2 / 1 (1.69%)
Type < 0	0 (0.00%)
Type ≤ 0	0 (0.00%)
Type	2 / 46 (77.97%), 3 / 7 (11.86%), 4 / 4 (6.78%), 1 / 2 (3.39%)
Signature	$\langle 2 \rangle$ / 46 (77.97%), $\langle 3 \rangle$ / 7 (11.86%), $\langle 4 \rangle$ / 4 (6.78%), $\langle 1 \rangle$ / 2 (3.39%)
Signature$^{\text{ancestry}}$	$\langle 2^{a} \rangle$ / 46 (77.97%), $\langle 3^{a} \rangle$ / 7 (11.86%), $\langle 4^{a} \rangle$ / 4 (6.78%), $\langle 1^{n} \rangle$ / 2 (3.39%)
In $\neq \emptyset$	0 (0.00%)
Np $\neq \emptyset$	58 (98.31%)
non-projective	**59** (0.07% of all edges)
all edges	86028

10.2.18 Swedish

Table 10.41: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	71 (0.64%)
non-planar	1076 (9.74%)
non-projective	1079 (9.77%)
all	11042

Table 10.42: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 1546 (81.50%), 2 / 204 (10.75%), 3 / 76 (4.01%), 4 / 27 (1.42%), 5 / 22 (1.16%), 6 / 10 (0.53%), 7 / 8 (0.42%), 8 / 2 (0.11%), 9 / 1 (0.05%), 14 / 1 (0.05%)
ideg	1 / 1829 (96.42%), 2 / 46 (2.42%), 3 / 9 (0.47%), 4 / 5 (0.26%), 6 / 5 (0.26%), 5 / 2 (0.11%), 7 / 1 (0.05%)
Type < 0	15 (0.79%)
Type ≤ 0	50 (2.64%)
Type	2 / 908 (47.87%), 1 / 686 (36.16%), 3 / 182 (9.59%), 4 / 41 (2.16%), 0 / 35 (1.85%), 5 / 17 (0.90%), -1 / 13 (0.69%), 6 / 13 (0.69%), -2 / 2 (0.11%)
Signature	$\langle 2\rangle$ / 823 (43.38%), $\langle 1\rangle$ / 530 (27.94%), $\langle 3\rangle$ / 114 (6.01%), $\langle 1,1\rangle$ / 94 (4.96%), $\langle 0\rangle$ / 31 (1.63%), $\langle 1,3\rangle$ / 27 (1.42%), $\langle 1,1,1\rangle$ / 25 (1.32%), $\langle 4\rangle$ / 21 (1.11%), $\langle 1,2\rangle$ / 19 (1.00%), $\langle 2,2\rangle$ / 16 (0.84%), $\langle 5\rangle$ / 11 (0.58%), $\langle -1\rangle$ / 10 (0.53%), $\langle 0,2\rangle$ / 10 (0.53%), $\langle 2,2,2\rangle$ / 7 (0.37%), $\langle 0,0,1\rangle$ / 6 (0.32%), $\langle 0,1,2\rangle$ / 6 (0.32%), $\langle 0,1\rangle$ / 5 (0.26%), $\langle 6\rangle$ / 5 (0.26%), $\langle 3,3\rangle$ / 5 (0.26%), ...

continued on next page

Property	value / count *or* count (proportion of non-projective edges)
Signature$^{\text{ancestry}}$	$\langle 2^{a} \rangle$ / 738 (38.90%), $\langle 1^{n} \rangle$ / 530 (27.94%), $\langle 3^{a} \rangle$ / 95 (5.01%), $\langle 1^{n}, 1^{n} \rangle$ / 94 (4.96%), $\langle 2^{n} \rangle$ / 85 (4.48%), $\langle 0^{n} \rangle$ / 31 (1.63%), $\langle 1^{n}, 3^{a} \rangle$ / 27 (1.42%), $\langle 1^{n}, 1^{n}, 1^{n} \rangle$ / 25 (1.32%), $\langle 1^{n}, 2^{n} \rangle$ / 19 (1.00%), $\langle 3^{n} \rangle$ / 19 (1.00%), $\langle 4^{a} \rangle$ / 12 (0.63%), $\langle -1^{n} \rangle$ / 10 (0.53%), $\langle 0^{n}, 2^{a} \rangle$ / 9 (0.47%), $\langle 2^{a}, 2^{n} \rangle$ / 9 (0.47%), $\langle 4^{n} \rangle$ / 9 (0.47%), $\langle 5^{a} \rangle$ / 9 (0.47%), $\langle 2^{n}, 2^{n} \rangle$ / 7 (0.37%), $\langle 2^{n}, 2^{n}, 2^{n} \rangle$ / 7 (0.37%), $\langle 0^{n}, 0^{n}, 1^{n} \rangle$ / 6 (0.32%), $\langle 0^{n}, 1^{n}, 2^{n} \rangle$ / 6 (0.32%), $\langle 0^{n}, 1^{n} \rangle$ / 5 (0.26%), ...
$\mathsf{In} \neq \emptyset$	285 (15.02%)
$\mathsf{Np} \neq \emptyset$	1891 (99.68%)
non-projective	**1897** (1.05% of all edges)
all edges	180425

10.2.19 Turkish

Table 10.43: Counts of dependency trees violating global constraints

Class	count (proportion of all trees)
ill-nested	14 (0.28%)
non-planar	556 (11.13%)
non-projective	580 (11.61%)
all	4997

Table 10.44: Counts of properties of non-projective edges

Property	value / count *or* count (proportion of non-projective edges)
cdeg	1 / 623 (74.08%), 2 / 146 (17.36%), 3 / 55 (6.54%), 4 / 13 (1.55%), 5 / 2 (0.24%), 6 / 2 (0.24%)
ideg	1 / 813 (96.67%), 2 / 27 (3.21%), 3 / 1 (0.12%)
Type < 0	2 (0.24%)
Type ≤ 0	8 (0.95%)
Type	2 / 403 (47.92%), 1 / 319 (37.93%), 3 / 67 (7.97%), 4 / 28 (3.33%), 0 / 6 (0.71%), 5 / 5 (0.59%), 7 / 5 (0.59%), 8 / 2 (0.24%), -4 / 1 (0.12%), -1 / 1 (0.12%), 6 / 1 (0.12%), 9 / 1 (0.12%), 10 / 1 (0.12%), 11 / 1 (0.12%)
Signature	$\langle 2\rangle$ / 341 (40.55%), $\langle 1\rangle$ / 189 (22.47%), $\langle 1,1\rangle$ / 91 (10.82%), $\langle 3\rangle$ / 53 (6.30%), $\langle 2,2\rangle$ / 31 (3.69%), $\langle 1,1,1\rangle$ / 29 (3.45%), $\langle 4\rangle$ / 19 (2.26%), $\langle 2,2,2\rangle$ / 10 (1.19%), $\langle 3,3\rangle$ / 6 (0.71%), $\langle 1,1,1,1\rangle$ / 6 (0.71%), $\langle 2,2,2,2\rangle$ / 6 (0.71%), $\langle 5\rangle$ / 5 (0.59%), $\langle 0\rangle$ / 4 (0.48%), $\langle 7\rangle$ / 4 (0.48%), ...
Signature$^{\text{ancestry}}$	$\langle 2^{a}\rangle$ / 281 (33.41%), $\langle 1^{n}\rangle$ / 189 (22.47%), $\langle 1^{n},1^{n}\rangle$ / 91 (10.82%), $\langle 2^{n}\rangle$ / 60 (7.13%), $\langle 3^{a}\rangle$ / 45 (5.35%), $\langle 1^{n},1^{n},1^{n}\rangle$ / 29 (3.45%), $\langle 2^{n},2^{n}\rangle$ / 29 (3.45%), $\langle 4^{a}\rangle$ / 12 (1.43%), $\langle 2^{n},2^{n},2^{n}\rangle$ / 10 (1.19%), $\langle 3^{n}\rangle$ / 8 (0.95%), $\langle 4^{n}\rangle$ / 7 (0.83%), $\langle 1^{n},1^{n},1^{n},1^{n}\rangle$ / 6 (0.71%), $\langle 2^{n},2^{n},2^{n},2^{n}\rangle$ / 6 (0.71%), $\langle 3^{n},3^{n}\rangle$ / 6 (0.71%), ...
In $\neq \emptyset$	34 (4.04%)
Np $\neq \emptyset$	788 (93.70%)

continued on next page

Property	value / count *or* count (proportion of non-projective edges)
non-projective	**841** (1.61% of all edges)
all edges	52273

10.3 Discussion

The empirical results presented in the previous section show that projectivity can hardly be claimed to be a formal constraint that accurately delimits dependency structures in natural languages. Constructions violating projectivity are frequent; on the other hand, it seems that the edge-based tools developed in the theoretical part of this thesis are expressive enough to capture them with a high degree of accuracy.

10.3.1 Tree properties

We see that projectivity is a too restrictive constraint for many languages. The largest proportion of non-projective dependency trees occurs in Latin (the largest deviation from projective word order occurs, unsurprisingly, in the highly literary text of Vergil's *Aeneid*). However, also several modern languages contain between 20 and 30% of non-projective dependency trees (they are Basque, Czech, German, Greek, Hungarian, and Slovene); in Dutch even more than 36% of dependency trees are non-projective!

The close relationship between planarity and projectivity shows also here: both constraints are violated by considerable portions of trees in many languages, and usually planarity is almost or completely as restrictive as projectivity for whole dependency trees.

Well-nestedness fits best with natural language data; it covers most trees in all languages. Our results for global properties of dependency trees show that the numbers of ill-nested dependency trees are quite low, but not as low as was claimed by Kuhlmann and Nivre [2006] based on an evaluation on two languages (Czech and Danish).

Some languages exhibit quite large numbers of ill-nested dependency trees, namely Latin and Basque. It is these two languages that show that there are languages in which the proportions of ill-nested dependency trees are not entirely negligible; German, with more than 1%, is another language with relatively many ill-nested dependency trees.

10.3.2 Edge properties

In contrast to global constraints, properties of individual non-projective edges allow us to pinpoint the causes of non-projectivity. Therefore they provide tools for a much more fine-grained classification of non-projective structures occurring in natural language.

Both interval and component degrees take generally low values. On the other hand, in several languages we see edges taking quite large val-

ues for both degrees (e.g., for component degree Basque, Czech, English, and Swedish take values larger than 10). Holan et al. [1998, 2000] show that at least for Czech neither of these two measures of non-projectivity can in principle be bounded. Our results seem to suggest that a natural explanation could be the limited performance capabilities of human beings.

Taking levels of nodes into account seems to bring both better accuracy and expressivity. When compared with the global constraint of well-nestedness, positive level types give an even better fit with real linguistic data (recall that an ill-nested dependency tree need not contain a non-projective edge of non-positive level type; cf. Theorem 7.3.2). For example, in German less than one tenth of ill-nested trees contain an edge of non-positive level types.

Only the following languages contain the very rare non-projective edges of negative level types: Basque, Catalan, Czech, Latin (Vergil), Slovene, Swedish, and Turkish. The lowest value of -5 is achieved by Basque and Slovene, the other languages barely get below 0. Only in Basque, Catalan, and Swedish non-projective edges of negative level types constitute more than 0.5% of all their non-projective edges.

Even for Basque, the language with the highest proportion of ill-nested dependency trees, the counts of non-projective edges of non-positive (and negative) level types are quite low—they amount only to slightly more than 0.2% (0.1%, respectively) of all edges.

Level signatures combine level types and component degrees, and so give an even more detailed picture of the gaps of non-projective edges. Level signatures complemented with ancestry information are meant to provide a detailed insight into possible non-projective structures occurring in natural languages. In some languages the actually occurring signatures are quite limited, in others there is a large variation.

We see that the proportion of non-projective edges with ancestors in their gaps varies among languages. This fact may be to some extent annotation-dependent.

The last two properties of non-projective edges we present are non-empty ill-nested and non-planar sets. Recall that the reported counts exclude edges from artifical root nodes, as this may interact with the annotation schemata for individual languages; cf. Section 10.1.3.

The counts of non-projective edges with non-empty non-planar sets testify that planarity is indeed almost as restrictive as projectivity; for Catalan, English, and Hungarian they become identical. For most other languages, the proportions of-projective edges with non-empty non-planar sets to all non-projective edges are close to 100%. The most notable exception is Japanese; its counts definitely seem to be due to a particular annotation scheme. Some-

what surprisingly Basque has the second lowest proportion of edges with non-empty non-planar sets among its non-projective edges—less than half of them; again, this is most probably due to some peculiarities of the annotation scheme.

As far as the counts of edges with non-empty ill-nested sets are concerned, English, Basque, and Swedish have the largest proportions of them among their non-projective edges (between 15 and 20%). The largest proportion among all edges, however, is achieved by Vergil with slightly more than 1% of all edges having non-empty ill-nested sets.

10.4 Conclusion

Empirical evidence shows that properties of non-projective edges taking into account levels of nodes are capable of describing very accurately natural language data. This is in good accord with the theoretical results presented in this thesis.

We find an edge-based approach to non-projectivity also more appealing linguistically than the traditional approaches based on properties of whole dependency trees or their subtrees. Furthermore, it may prove suitable also for statistical natural language processing, as properties of edges allow machine-learning techniques to model global phenomena locally, resulting in less sparse models.

Our empirical results on nineteen languages can be summarized as follows: Among the considered measures of non-projectivity, both tree-based and edge-based, level types of non-projective edges are best at delimiting non-projective structures in natural languages.

Furthermore, properties of non-projective edges, such as level signatures, combining levels of nodes and gap components, provide both expressive and accurate tools for describing non-projective constructions.

Therefore we hope that the edge-based tools developed in the theoretical part will prove to be instrumental also in linguistic analysis of natural languages. We think that they can serve as a solid theoretical basis for further investigations into natural language syntax.

Index

$\rightarrow$, 13
$\leftrightarrow$, 15
$\preceq$, 14
$\preceq_i$, 31
(i, j), 16
$[i, j]$, 16
$\langle\ \rangle$, 74
$\upharpoonright$, *see* restriction, of binary relation to set
*, *see* closure, reflexive and transitive
$^{\mathrm{tr}}$, *see* reduction, transitive
L_i, *see* tree, local of node
r, *see* node, root
$\mathsf{Anc}_{i\leftrightarrow j}$, 16
Anc_i, 16
$\mathsf{Child}_{i\leftrightarrow j}$, 15
$\mathsf{cdeg}_{i\leftrightarrow j}$, 72
$\mathsf{Gap}_{i\leftrightarrow j}$, 24
$\mathsf{Gap}^{\uparrow}_{i\leftrightarrow j}$, 42
$\mathsf{ideg}_{i\leftrightarrow j}$, 71
$\mathsf{Parent}_{i\leftrightarrow j}$, 15
$\mathsf{Signature}_{i\leftrightarrow j}$, 74
$\mathsf{Subtree}_{i\leftrightarrow j}$, 16
$\mathsf{Subtree}_i$, 16
`field`$[i]$, 19
$\mathrm{Sibl}(j_1, j_2, i)$, 17

closure
    reflexive, 16
    reflexive and transitive, 13
    transitive, 16
component root, *see* node, root of component
convention
    for drawing dependency tree, 17
    for drawing gap of non-projective edge, 24
    for null pointers, 19

data representation
    of dependency tree, 19
    of node, 18
degree of edge
    component, 72
    interval, 71
dependency tree
    ill-nested, 63
    non-planar, 52
    non-projective, 22
    planar, 52
    projective, 22
    smooth, 51
    weakly non-projective, 51
    well-nested, 63

edge
    non-projective, 24
    projective, 24
edge of tree, 13

forest, 14

gap
  of non-projective edge, 24
    upper, 42
gap degree of dependency tree, 72

height of tree, 15

ill-nested set of edge, 67
  upper, 68
interval
  closed, 16
  open, 16

level
  component, 74
level of node, 15
level signature of edge, 74
level type of non-projective edge, 38

model of computation, *see* random-access machine
multiplanarity, 60

node, 13
  ancestor, 15
  ancestor of edge, 15
  child, 14
  child of edge, 14
  descendant, 15
  descendant of edge, 15
  internal, 14
  leaf, 14
  maximal in gap, 25
  parent, 14
  parent of edge, 14
  root, 13
    of component, 72
  sibling, 14
  witnessing level type, 39
non-planar set of edge, 57
  upper, 57

order
  local, 31
  total projective of rooted tree, 33
ordering
  local of rooted tree, 31

pair of edges
  ill-nested, 65
  non-planar, 52
projectivization of dependency tree
  canonical, 30
  general, 31

random-access machine, 13
reduction
  reflexive, 16
  transitive, 16
relation
  dependency, 14
    undirected, 15
  subordination, 14
requirement
  on data representation of dependency tree, *see* data representation, of dependency tree
restriction
  of binary relation to set, 14

set of edge
  ill-nested, *see* ill-nested set of edge
  non-planar, *see* non-planar set of edge
span of edge, 16
subtree
  of dependency tree, 14
  of rooted tree, 14

traversal of rooted tree
  by levels bottom-up, 21
  post-order, 20
    general, 20
  pre-order, 20

tree
  dependency, 14
  local of node, 31
  rooted, 13
    ordered, 17
    totally ordered, *see* tree, dependency

Bibliography

Numbers appearing at the ends of bibliographical entries represent pages where they are referenced.

A. Abeillé, editor. *Treebanks: Building and Using Parsed Corpora*. Kluwer, 2003. 128, 131

I. Aduriz, M. J. Aranzabe, J. M. Arriola, A. Atutxa, A. Diaz de Ilarraza, A. Garmendia, and M. Oronoz. Construction of a Basque dependency treebank. In *Proc. of the 2nd Workshop on Treebanks and Linguistic Theories (TLT)*, pages 201–204, 2003. 84

S. Afonso, E. Bick, R. Haber, and D. Santos. "Floresta sintá(c)tica": a treebank for Portuguese. In *Proc. of the 3rd Intern. Conf. on Language Resources and Evaluation (LREC)*, pages 1698–1703, 2002. 84

N. B. Atalay, K. Oflazer, and B. Say. The annotation process in the Turkish treebank. In *Proc. of the 4th Intern. Workshop on Linguistically Interpreteted Corpora (LINC)*, 2003. 84

David Bamman and Gregory Crane. The Design and Use of a Latin Dependency Treebank. In Jan Hajič and Joakim Nivre, editors, *Proc. of the 5th Workshop on Treebanks and Linguistic Theories (TLT)*, pages 67–78, 2006. 84

David Bamman and Gregory Crane. The Latin Dependency Treebank in a Cultural Heritage Digital Library. In *Proceedings of the Workshop on Language Technology for Cultural Heritage Data (LaTeCH 2007)*, pages 33–40, 2007. 84

Manuel Bodirsky, Marco Kuhlmann, and Matthias Möhl. Well-nested drawings as models of syntactic structure. In *Proceedings of Tenth Conference on Formal Grammar and Ninth Meeting on Mathematics of Language*, 2005. 25, 62, 63

A. Böhmová, J. Hajič, E. Hajičová, and B. Hladká. The PDT: a 3-level annotation scenario. In Abeillé [2003], chapter 7, pages 103–127. 84

S. Brants, S. Dipper, S. Hansen, W. Lezius, and G. Smith. The TIGER treebank. In *Proc. of the 1st Workshop on Treebanks and Linguistic Theories (TLT)*, 2002. 84

S. Buchholz and E. Marsi. CoNLL-X shared task on multilingual dependency parsing. In *Proc. of the 10th Conf. on Computational Natural Language Learning (CoNLL-X)*. SIGNLL, 2006. 84

M. Civit Torruella and M[a] A. Martí Antonín. Design principles for a Spanish treebank. In *Proc. of the 1st Workshop on Treebanks and Linguistic Theories (TLT)*, 2002. 84

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, London, England, second edition, 2001. 13

D. Csendes, J. Csirik, T. Gyimóthy, and A. Kocsor. *The Szeged Treebank.* Springer, 2005. 84

Alexander Dikovsky and Larissa Modina. Dependencies on the other side of the Curtain. *Traitement Automatique des Langues (TAL)*, 41(1):67–96, 2000. 51

Vida Dujmović and David R. Wood. On Linear Layouts of Graphs. *Discrete Mathematics and Theoretical Computer Science*, 6:339–358, 2004. 60

S. Džeroski, T. Erjavec, N. Ledinek, P. Pajas, Z. Žabokrtsky, and A. Žele. Towards a Slovene dependency treebank. In *Proc. of the 5th Intern. Conf. on Language Resources and Evaluation (LREC)*, 2006. 84

Haim Gaifman. Dependency systems and phrase-structure systems. *Information and Control*, 8(3):304–337, 1965. 9

M. R. Garey, D. S. Johnson, G. L. Miller, and Papadimitriou C. H. The complexity of coloring circular arcs and chords. *SIAM J. Alg. Disc. Meth.*, 1(2):216–227, 1980. 60, 61

J. Hajič, O. Smrž, P. Zemánek, J. Šnaidauf, and E. Beška. Prague Arabic dependency treebank: Development in data and tools. In *Proc. of the NEMLAR Intern. Conf. on Arabic Language Resources and Tools*, pages 110–117, 2004. 84

Jan Hajič, Jarmila Panevová, Eva Hajičová, Petr Sgall, Petr Pajas, Jan Štěpánek, Jiří Havelka, Marie Mikulová, Zdeněk Žabokrtský, and Magda Ševčíková Razímová. Prague Dependency Treebank 2.0, 2006. Linguistic Data Consortium, Philadelphia, PA, USA, ISBN 1-58563-370-4. 81

Jan Hajič, Petr Pajas, Jarmila Panevová, Eva Hajičová, Petr Sgall, and Barbora Vidová Hladká. Prague Dependency Treebank 1.0, 2001. 84

Eva Hajičová, Jiří Havelka, Petr Sgall, Kateřina Veselá, and Daniel Zeman. Issues of Projectivity in the Prague Dependency Treebank. *The Prague Bulletin of Mathematical Linguistics*, 81:5–22, 2004. ISSN 0032-6585. 8, 29

Keith Hall, Jiří Havelka, and David A. Smith. Log-linear Models of Non-projective Trees, k-best MST Parsing and Tree-ranking. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, 2007. 8, 87

Jiří Havelka. Projectivity in Totally Ordered Rooted Trees: An Alternative Definition of Projectivity and Optimal Algorithms for Detecting Non-Projective Edges and Projectivizing Totally Ordered Rooted Trees. *The Prague Bulletin of Mathematical Linguistics*, 84:13–30, 2005a. ISSN 0032-6585. 8

Jiří Havelka. Projektivita v úplně uspořádaných kořenových stromech: alternativní definice projektivity a optimální algoritmy pro zprojektivnění a nalezení neprojektivních hran. In *MIS 2005*. Matfyzpress, Faculty of Mathematics and Physics, Charles University in Prague, 2005b. 8

Jiří Havelka. Beyond Projectivity: Multilingual Evaluation of Constraints and Measures on Non-Projective Structures. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics*, 2007a. 8, 83

Jiří Havelka. Relationship between Non-Projective Edges, Their Level Types, and Well-Nestedness. In *Proceedings of Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume, Short Papers*, pages 61–64, 2007b. 8

Tomáš Holan, Vladislav Kuboň, Karel Oliva, and Martin Plátek. On Complexity of Word Order. *Traitement Automatique des Langues (TAL)*, 41 (1):273–300, 2000. 25, 122

Tomáš Holan, Vladislav Kuboň, Karel Oliva, and Martin Plátek. Two Useful Measures of Word Order Complexity. In Alain Polguère and Sylvain Kahane, editors, *Proceedings of Dependency-Based Grammars Workshop, COLING/ACL*, pages 21–28, 1998. 25, 72, 122

Josef Jiřička. The Number of Projective Trees with a Given Number of Vertices. *The Prague Bulletin of Mathematical Linguistics*, 24:51–60, 1975. 77

R. Johansson and P. Nugues. Extended constituent-to-dependency conversion for English. In *Proc. of the 16th Nordic Conference on Computational Linguistics (NODALIDA)*, 2007. 84

Y. Kawata and J. Bartels. Stylebook for the Japanese treebank in VERBMOBIL. Verbmobil-Report 240, Seminar für Sprachwissenschaft, Universität Tübingen, 2000. 84

M. T. Kromann. The Danish dependency treebank and the underlying linguistic theory. In *Proc. of the 2nd Workshop on Treebanks and Linguistic Theories (TLT)*, 2003. 84

Marco Kuhlmann and Mathias Möhl. Mildly Context-Sensitive Dependency Languages. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 160–167, Prague, Czech Republic, 2007. Association for Computational Linguistics. 10

Marco Kuhlmann and Joakim Nivre. Mildly Non-Projective Dependency Structures. In *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*, pages 507–514, 2006. 121

M. Marcus, B. Santorini, and M. Marcinkiewicz. Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19(2): 313–330, 1993. 84

Solomon Marcus. Sur la notion de projectivité [On the notion of projectivity]. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 11: 181–192, 1965. 22, 23

Solomon Marcus. *Algebraic Linguistics; Analytical Models.* Academic Press, New York and London, 1967. 9

M. A. Martí, M. Taulé, L. Màrquez, and M. Bertran. CESS-ECE: A multilingual and multilevel annotated corpus. Available for download from: http://www.lsi.upc.edu/∼mbertran/cess-ece/, 2007. 84

Mathias Möhl. Drawings as models of syntactic structure: Theory and algorithms. Diploma thesis, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, 2006. 69

S. Montemagni, F. Barsotti, M. Battista, N. Calzolari, O. Corazzari, A. Lenci, A. Zampolli, F. Fanciulli, M. Massetani, R. Raffaelli, R. Basili, M. T. Pazienza, D. Saracino, F. Zanzotto, N. Nana, F. Pianesi, and R. Delmonte. Building the Italian Syntactic-Semantic Treebank. In Abeillé [2003], chapter 11, pages 189–210. 84

B. Navarro, M. Civit, Mª A. Martí, R. Marcos, and B. Fernández. Syntactic, semantic and pragmatic annotation in Cast3LB. In *Proc. of the Workshop on Shallow Processing of Large Corpora (SProLaC)*, 2003. 84

Ladislav Nebeský. Graph theory and linguistics. In R. J. Wilson and L. W. Beineke, editors, *Applications of Graph Theory*, chapter 12, pages 357–380. Academic Press, 1979. 52

J. Nilsson, J. Hall, and J. Nivre. MAMBA meets TIGER: Reconstructing a Swedish treebank from antiquity. In *Proc. of the NODALIDA Special Session on Treebanks*, 2005. 84

J. Nivre, J. Hall, S. Kübler, R. McDonald, J. Nilsson, S. Riedel, and D. Yuret. The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 915–932, 2007. 84

Joakim Nivre. Constraints on Non-Projective Dependency Parsing. In *Proc. of the 11th Conf. of the European Chapter of the ACL (EACL)*, pages 73–80, 2006. 72

Marc Noy. Enumeration of noncrossing trees on a circle. *Discrete Mathematics*, 180:301–313, 1998. 77

K. Oflazer, B. Say, D. Zeynep Hakkani-Tür, and G. Tür. Building a Turkish treebank. In Abeillé [2003], chapter 15, pages 261–277. 84

Petr Pajas. Tree Editor TrEd, 2007. URL `http://ufal.mff.cuni.cz/~pajas/tred`. 81, 87

P. Prokopidis, E. Desypri, M. Koutsombogera, H. Papageorgiou, and S. Piperidis. Theoretical and practical issues in the construction of a Greek dependency treebank. In *Proc. of the 4th Workshop on Treebanks and Linguistic Theories (TLT)*, pages 149–160, 2005. 84

K. Simov and P. Osenova. Practical annotation scheme for an HPSG treebank of Bulgarian. In *Proc. of the 4th Intern. Workshop on Linguistically Interpreteted Corpora (LINC)*, pages 17–24, 2003. 84

K. Simov, P. Osenova, A. Simov, and M. Kouylekov. Design and implementation of the Bulgarian HPSG-based treebank. In *Journal of Research on Language and Computation – Special Issue*, pages 495–522. Kluwer Academic Publishers, 2005. 84

Neil J. A. Sloane. On-Line Encyclopedia of Integer Sequences, 2007. URL `http://www.research.att.com/~njas/sequences/`. 76, 77, 78

O. Smrž, J. Šnaidauf, and P. Zemánek. Prague dependency treebank for Arabic: Multi-level annotation of Arabic corpus. In *Proc. of the Intern. Symposium on Processing of Arabic*, pages 147–155, 2002. 84

L. van der Beek, G. Bouma, J. Daciuk, T. Gaustad, R. Malouf, G. van Noord, R. Prins, and B. Villada. The Alpino dependency treebank. In *Algorithms for Linguistic Processing*, NWO PIONIER progress report 5. 2002a. 84

L. van der Beek, G. Bouma, R. Malouf, and G. van Noord. The Alpino dependency treebank. In *Computational Linguistics in the Netherlands (CLIN)*, 2002b. 84

Kateřina Veselá and Jiří Havelka. Anotování aktuálního členění věty v Pražském závislostním korpusu [Annotation of Topic-Focus Articulation in the Prague Dependency Treebank]. Technical report, ÚFAL/CKL MFF UK, December 2003. 8, 29

Kateřina Veselá, Jiří Havelka, and Eva Hajičová. Condition of Projectivity in the Underlying Dependency Structures. In *Proceedings of the 20th International Conference on Computational Linguistics*, volume I, pages 289–295, Geneva, Switzerland, August 23-27 2004. Association for Computational Linguistics. ISBN 1-932432-48-5. 8, 29

Anssi Yli-Jyrä. Multiplanarity – a model for dependency structures in treebanks. In *Proc. of the 2nd Workshop on Treebanks and Linguistic Theories (TLT)*, pages 189–200, 2003. 60