How to embed noncrossing trees in Universal Dependencies treebanks in a low-complexity regular language

Anssi Yli-Jyrä
University of Helsinki, Finland

ABSTRACT

A recently proposed balanced-bracket encoding (Yli-Jyrä and Gómez-Rodríguez 2017) has given us a way to embed all noncrossing dependency graphs into the string space and to formulate their exact arc-factored inference problem (Kuhlmann and Johnsson 2015) as the best string problem in a dynamically constructed and weighted unambiguous context-free grammar. The current work improves the encoding and makes it shallower by omitting redundant brackets from it. The streamlined encoding gives rise to a *bounded-depth subset approximation* that is represented by a *small finite-state automaton*. When bounded to 7 levels of balanced brackets, the automaton has 762 states and represents a strict superset of more than 99.9999% of the noncrossing trees available in Universal Dependencies 2.4 (Nivre *et al.* 2019). In addition, it strictly contains all 15-vertex noncrossing digraphs. When bounded to 4 levels and 90 states, the automaton still captures 99.2% of all noncrossing trees in the reference dataset. The approach is flexible and extensible towards unrestricted graphs, and it suggests tight finite-state bounds for dependency parsing, and for the main existing parsing methods.

Keywords: bounded stack, coding morphisms, context-free grammars, dependencies, finite-state, parsing, state complexity, treebanks

1 INTRODUCTION

Dependency structures – rooted trees and more general digraphs – have tremendous importance in multilingual syntactic analysis and in the related semantic analysis, and its applicability to the world's

languages have been demonstrated recently very strongly by the Universal Dependencies (UD) initiative.[1] The main approaches to produce syntactic dependency structures include *graph-based parsers* (Eisner and Satta 1999; McDonald *et al.* 2005) that usually aim at exact inference, and *transition-based parsers* (Nivre 2008) that treat parsing as beam search that runs in linear time with a small risk of missing the best analysis. *Neural network-based parsers*, such as Libovický (2016), Ma and Hovy (2017) and many more, provide additional flexibility and high accuracy. In the present work, we advance the long-term development of a new, *code-theoretic parsing* approach (Yli-Jyrä and Gómez-Rodríguez 2017) that may lend itself to unforeseen combinations with the existing approaches.

Parsing that leads to *noncrossing trees and graphs* (Kuhlmann and Johnsson 2015) is a simplification of more general approaches that produce nonprojective trees and ordered graphs with crossing edges. Although such parsing is limited in coverage, it is a very important, well-understood core for some more general parsing algorithms. Recently, Yli-Jyrä and Gómez-Rodríguez (2017) have explored an approach that embeds[2] *the set of noncrossing digraphs* (NXDIGRAPHS) into the string space Σ^* using an injective encoding morphism between the noncrossing digraphs and the corresponding set of code strings ($L_{\texttt{NXDIGRAPHS}}$) that form an unambiguous context-free language:

$$\texttt{NXDIGRAPHS} \rightarrow L_{\texttt{NXDIGRAPHS}}, L_{\texttt{NXDIGRAPHS}} \subseteq \Sigma^*.$$

The embedding can be used to turn the finite, sentence-specific search space of noncrossing graphs dynamically into a finite string set where each string corresponds to a distinct element in the search space. This gives us a *code-theoretic parsing approach* that has five advantages:

1. **Flexibility:** Several subfamilies of noncrossing digraphs can be treated as alternative search spaces that are treated uniformly by

[1] `http://universaldependencies.org/`

[2] In mathematics, when some object X is said to be embedded in another object Y, the obtained embedding is given by some injective and structure-preserving map $f : X \rightarrow Y$. In this work, embedding of graphs is based on code strings over a code alphabet and should not be confused with continuous vector space representations, although such an embedding is commonly used in natural language processing and in modern neural network architectures.

a generic parser whose search space can be restricted to these subfamilies (Yli-Jyrä and Gómez-Rodríguez 2017).

2. **Context-freeness:** The search space can be represented compactly with a context-free grammar that can also have weights (ibid.).
3. **Decidability:** These grammars are unambiguous and can be related to a rich calculus of tree automata. These are then connected to monadic second-order logic whose formulas define linear-time decidable properties over ordered trees and tree-decompositions of graphs (Bojańczyk and Pilipczuk 2016).
4. **Compatibility:** It is probable that the approach can be combined with existing parsing frameworks such as graph-based parsing (McDonald *et al.* 2005; Kiperwasser and Goldberg 2016; Zheng 2017), transition-based parsing (Dyer *et al.* 2015; Kiperwasser and Goldberg 2016), encoder-decoder parsing (Vinyals *et al.* 2015), parsing as sequence labeling (Strzyz *et al.* 2019) and parsing with recurrent neural network grammars (Dyer *et al.* 2016; Kuncoro *et al.* 2017).
5. **Extensibility:** There is follow-up work that extends the encoding developed in this paper to all ordered digraphs (Yli-Jyrä 2019).

In the code-theoretic arc-factored parsing approach (Yli-Jyrä 2012; Yli-Jyrä and Gómez-Rodríguez 2017), the construction of the compact representation of the complete distribution of potential parses takes cubic time. The construction involves building, dynamically, a weighted context-free grammar for the complete parse forest. The exact decoding of the optimal parse is then carried out in time that is linear to the size of the dynamic grammar. Since the combined complexity of these tasks remains in $O(n^3)$, the complexity hits the previously known worst-case bound for parsing whose output is restricted to some families of noncrossing graphs (Kuhlmann and Johnsson 2015). But in today's terms, parsing through a cubic time procedure is often considered too expensive as real-time data applications demand low latency and high throughput. More efficient parsing algorithms are already available in the established parsing frameworks. Especially transition-based parsing is a very successful and efficient parsing framework (Nivre 2008, 2009; Bohnet *et al.* 2016) that has inspired recent work

on transition-based neural network parsing (Dyer *et al.* 2015; Kiperwasser and Goldberg 2016).

According to Covington (2001, 101–102), it is important to study the *constrained computational complexity* of parsing algorithms when they are applied to natural language data:

> An important principle of linguistics seems to be that *the worst case does not occur*, i.e., people do not actually utter sentences that put any reasonable parsing algorithm into a worst-case situation. Human language does not use unconstrained phrase-structure or dependency grammar; it is constrained in ways that are still being discovered.

The code-theoretic parsing approach has a special advantage in the study of the constrained computational complexity of parsing algorithms because there the constraints are reflected immediately in the complexity of the search space embedding. The unknown complexity of the sufficiently constrained search space embedding for natural language gives rise to the following hypothesis:

> **Hypothesis**
> The practically occurring (noncrossing) dependency digraphs can be embedded into a subset approximation that has a very compact finite-state representation.

The concrete aim of this article is to investigate the existence of *a practical, very compact finite-state representation* for the search space of noncrossing trees and digraphs in dependency syntactic parsing. Given an encoding morphism and a depth bound that limits the maximal complexity of dependency digraphs, the corresponding set of digraphs will be recognized by a minimal deterministic finite automaton, where each state has a constant number of transitions. The *state complexity* of the minimal automaton depends only on the language it recognizes. Thus, the only way to reduce the state complexity is to improve the embedding of the digraphs into a regular, i.e. finite-state language. The hypothesis is valid, if a very compact finite-state representation for the practically occurring dependency digraphs exists.

Going from the cubic-time algorithms for noncrossing graphs to the linear-bounded state complexity of a depth-bounded search space means that that we are slightly closer to linear-time inference over

arc-factored weighted parses. However, the scope of the current work does not allow us to study whether also the representation of the weighted search space with the arc-factored weights actually remains linear bounded and compactly represented as a finite-state network. If the number of possible distinct arc weights in the statistical model is not bounded by a constant, the dynamic finite-state representation of the weighted search space is super-linear. But since there are techniques for pruning parse forests (Roark and Hollingshead 2008; Zhang and McDonald 2014; Zheng 2017), and the weights can be also simplified, e.g. by quantisation, we may avoid such super-linearity. It is, thus, conceivable that the dynamically weighted search space could also have a good finite-state approximation if there is a dynamic finite-state representation for the corresponding unweighted search space.

The structure of this article is as follows. Sections 2.1 and 2.2 contain the definitions and basic results required to understand how the sets of noncrossing graphs and digraphs are embedded into a context-free string language. Since a finite bound for the bracketing depth is desirable, Section 3 seeks a streamlined encoding that would improve on the proposal of Yli-Jyrä and Gómez-Rodríguez (2017) by radically reducing the bracketing depth of an average parse. A proposal for such a streamlined encoding is presented and formally analysed in Section 4. Finally, the prior and the streamlined encoding are evaluated in Section 5 from the point of view of state complexity and coverage. Section 6 concludes the article and identifies some questions that remain open after the current work.

2 DEFINITIONS

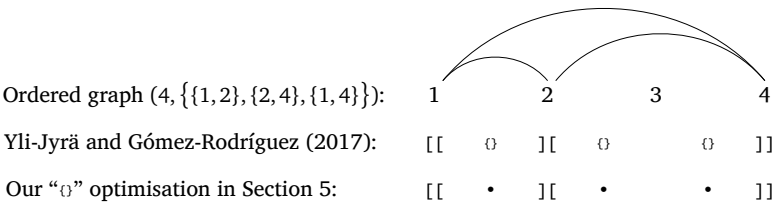
We assume that the reader is familiar with the basics of formal language theory and especially the theory of context-free grammars, finite-state automata and finite-state transducers. Algorithms will be written in a pseudo-formal language that mixes Python-like syntax with mathematical notation. In the following, we give definitions for noncrossing graphs and digraphs (Kuhlmann 2015) and the corresponding encoding that we will call *strong bracketing*, **S**. Strong bracketing for graphs is defined in Section 2.1, and Section 2.2 defines strong bracketing for digraphs and relates these two classes of structures.

2.1 *Strong bracketing for noncrossing graphs*

A (nonempty) *graph* is a pair (V,E) where V is a finite, nonempty set of vertices and $E \subseteq \{\{i,j\} \subseteq V \mid i \neq j\}$ is a set of edges. Each edge in a graph may have a label or even a multiset of labels. The *complete graph* (V,E) has all possible edges $E = \{\{i,j\} \subseteq V \mid i \neq j\}$. The vertices in graphs are usually an ordered set $V = [1,\ldots,n]$ with a linear order $\leq$. Such an *ordered graph* (V,E) is given more simply as the pair (n,E). By working on ordered graphs, we avoid the usual difficulty of defining equality of graphs through isomorphism: graph $(3,\{\{1,2\}\})$ is *not* equivalent to graph $(3,\{\{2,3\}\})$ although these graphs are isomorphic. In an ordered graph, the edge $\{i,j\} \in E$ can be viewed as an ordered pair $(\min\{i,j\},\max\{i,j\})$. Two edges (i,j), (k,l) where $i < k$ are said to be *crossing* if $k < j < l$. The *concatenation* of two ordered graphs (n,E_1) and (m,E_2), denoted by $(n,E_1)\cdot(m,E_2)$, is $(n+m-1,E)$ where $E = E_1 \cup \{\{i+n-1,j+n-1\} \mid \{i,j\} \in E_2\}$. An ordered graph is *noncrossing* if it has no crossing edges. The set of (nonempty) noncrossing graphs is denoted as NXGRAPH. Together with the trivial graph $(1,\{\})$ and concatenation, this set has the structure of a monoid.

Yli-Jyrä and Gómez-Rodríguez (2017) have proposed an encoding scheme according to which any noncrossing graph (n,E) can be represented as a string of brackets. For example, the ordered noncrossing graph $(4,\{\{1,2\},\{2,4\},\{1,4\}\})$ is encoded as the string "`[[{][{ }]]`", see Figure 1 (the middle row).

Figure 1: An example of an ordered graph



The original reason for using the curly brackets "{}" in Yli-Jyrä and Gómez-Rodríguez (2017) was that, with them, the code strings respect the balanced bracketing and form a subset of a Dyck language. They also encode, intuitively, the successor edges over the vertices. Since these motivations for the curly brackets are less important in the current work, it is plausible to replace "{}" with a single character "-" to optimise the code strings; see Figure 1 (last row). This optimisa-

tion will be discussed later in this paper, in Section 5, but we stick momentarily to the original encoding (Yli-Jyrä and Gómez-Rodríguez 2017) that uses curly brackets. In both encoding schemes, the square brackets "[" and "]" connect the vertices by spanning the gaps that separate them. In particular, each pair of matching square brackets "[...]" correspond to an arc between two vertices. Because the brackets in this encoding always come in pairs, we will call this encoding a *strong bracketing*, **S**.

The encoding function $\mathsf{enc}_\mathbf{S}$ that maps the elements of NXGRAPH to the elements of the monoid $\{[,], \{, \}\}^*$ is implemented by an algorithm that is given in Figure 2. Since the algorithm is not used during parsing, we give a simple, unoptimised version that is designed to illustrate the encoding scheme. This algorithm runs in $O(n^2)$ time, but more efficient algorithms exist.

def $\mathbf{enc_S}$((n,E) ∈ NXGRAPH):

```
1  str = ε
2  for i in [1,2,...,n]:
3      for j in [i-1,i-2,...,1]:
4          if {j,i} in E:
5              str += "]"
6      for j in [n,n-1,...,i+1]:
7          if {i,j} in E:
8              str += "["
9      if i<n:
10         str += "{" + "}"
11 return str
```

def $\mathbf{dec_S}$(str ∈ $L_{\text{NXGRAPH,S}}$):

```
1  (n,E) = (1,{})
2  stk = ε
3  for c in str:
4      if c == "[":
5          stk.push(n)
6      if c == "]":
7          i = stk.pop()
8          E += { {i,n} }
9      if c == "{":
10         n += 1
11 return (n,E)
```

Figure 2: The encoding and decoding algorithms for noncrossing ordered graphs

Lemma 2.1. *The encoding* $\mathsf{enc}_\mathbf{S}$ *maintains an iconic correspondence between the parts of the graph and the string structure.*

Proof. The encoding function, $\mathsf{enc}_\mathbf{S}$, produces a closing square bracket for the right end of the edge, an opening square bracket for the left end of the edge and a pair of curly brackets to indicate that adjacent vertices are in a successor relation with each other. The length of the code string is exactly $2|E| + 2n - 2$ characters when $n > 0$. The empty string ε encodes the unit graph that consists of a single vertex. In other words, the encoding is based on an iconic correspondence between the graph and the bracketing. □

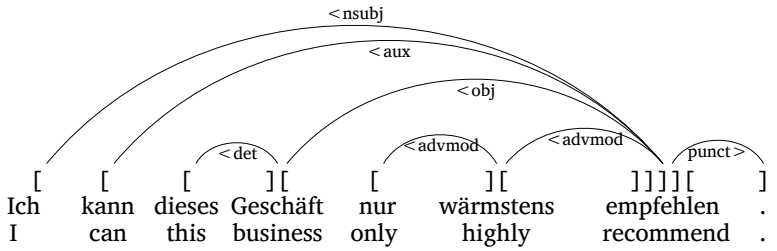


Figure 3: An example sentence with a noncrossing parse

I highly recommend this business.

To see how the encoding applies to a syntactic dependency analysis of a natural language sentence, such an analysis for an example sentence is given in Figure 3. The sentence is in German and it reads *Ich kann dieses Geschäft nur wärmstens empfehlen*. Under the line containing this sentence, there are a word-by-word English translation and a free translation in English. At the top of the figure, there is a diagram of an undirected graph that indicates the dependencies between the vertices, i.e. the tokens that constitute the sentence. The label of each edge specifies the direction of the *dependent* vertex-token and the category of this vertex-token from the perspective of the *head* vertex-token that is at the opposite end of the edge.

Between the graph diagram and the German sentence, there is a line that contains a bracket string. In this bracket string, the iconic correspondence between the brackets and the edge degree of each vertex-token is clearly recognizable, but this string does not show the curly brackets that separate the vertices of the graph. With the curly brackets, the bracket string is "`[{[{[{][{[{][{]}]}]}]}[{]}`". It is also possible to add the edge labels at the corresponding brackets:

```
[ <nsubj {[ <aux {[ <det {][ <obj {[ <advmod {][ <advmod {]}]}]}[{ punct> ].
```

The encoding considered in the current article ignores the edge labels in order to keep the presentation clear. Recall that the current goal is not to develop a front-end descriptive formalism for linguists but to investigate the search space of noncrossing dependency graphs from the perspective of its state complexity.

Lemma 2.2. *The encoding function* enc_S *is a bijection whose inverse can be computed in linear time.*

Proof. The right side of Figure 2 presents a decoding algorithm, $\texttt{dec}_\texttt{S}$, that maps bracket strings to noncrossing graphs. The obtained function can be easily seen to be the inverse of $\texttt{enc}_\texttt{S}$. Thus, the encoding function is a bijection between its domain and the range.

Since the for-loop in the decoding algorithm $\texttt{dec}_\texttt{S}$ needs only as many iterations as there are characters in the argument `str`, it computes the inverse of $\texttt{enc}_\texttt{S}$ in linear time. □

According to Lemma 2.2, any noncrossing graph is in a 1-to-1 relationship with the corresponding string that encodes the graph. These strings constitute a subset $L_{\mathsf{NXGRAPH,S}}$ of the free monoid $\{\texttt{[}, \texttt{]}, \texttt{\{}, \texttt{\}}\}^*$. Since the input of the decoding algorithm in Figure 2 is restricted to the outputs of $\texttt{enc}_\texttt{S}$, the algorithm ignores the right curly bracket "`}`" in code strings.

Lemma 2.3. *The range of* $\texttt{enc}_\texttt{S}$, $L_{\mathsf{NXGRAPH,S}}$, *is an unambiguous context-free language.*

Proof. There is an unambiguous grammar that describes the range of the encoding function.

$$\begin{aligned} &S \to Q\,S \mid \texttt{\{\}}\,S \mid \varepsilon \quad S' \to Q\,S'' \mid \texttt{\{\}}\,S \quad S'' \to Q\,S \mid \texttt{\{\}}\,S \\ &Q \to \texttt{[}S'\texttt{]}. \end{aligned} \tag{1}$$

We make three observations of the grammar:

Firstly, this grammar produces balanced bracketing over $\{\texttt{[}, \texttt{]}, \texttt{\{}, \texttt{\}}\}$ where the opening and the closing curly brackets are always adjacent, like in line 10 of the $\texttt{enc}_\texttt{S}$ algorithm.

Secondly, by the productions for the phrases S' and S'', each level of square brackets "`[]`" contains one pair of curly bracket of its own or two or more nested square brackets. Thus we may have substrings "`[{}]`", "`[{}[{}]]`", and "`[[{}][{}]]`" but not substrings "`[]`" or "`[[]]`". This principle avoids connecting a pair of vertices more than once and corresponds to the fact that lines 3–8 in the encoding algorithm (Figure 2) produce exactly one pair of square brackets per edge.

Thirdly, whenever two balanced substrings correspond to two subgraphs, they can be concatenated without adding any curly brackets or vertices between them. Concatenation corresponds to the S rule(s) in the grammar and the immediate succession between lines 3–5 and 5–8 in the algorithm.

These observations can be extended to an inductive formal proof over well-formed substrings and the corresponding graphs. □

Graph concatenation generates the monoid $(\mathsf{NXGRAPH}, \cdot, (1, \{\}))$ of noncrossing ordered graphs where the trivial graph $(1, \{\})$ is the identity element. The string concatenation of the encoded noncrossing graphs generates the monoid $(L_{\mathsf{NXGRAPH},\mathbf{S}}, \cdot, \varepsilon)$ where the empty string ε is the identity element.

Lemma 2.4. *The encoding function* $\mathsf{enc}_{\mathbf{S}}$ *is a homomorphism between the concatenation monoid of noncrossing graphs* NXGRAPH *and the concatenation monoid of code strings* $L_{\mathsf{NXGRAPH},\mathbf{S}}$.

Proof. It is is easy to verify that the encoding $\mathsf{enc}_{\mathbf{S}}$ respects the monoid structure: firstly, the encoding is compositional in the sense that $\mathsf{enc}_{\mathbf{S}}(n, E_1) \cdot \mathsf{enc}_{\mathbf{S}}(m, E_2) = \mathsf{enc}_{\mathbf{S}}((n, E_1) \cdot (m, E_2))$. Secondly, the identity element of the first monoid is the trivial graph $(1, \{\})$ that is encoded as the empty string ε, the identity element of the second monoid. Thus the encoding is a homomorphism. □

In grammars for bracketed graphs, it is often handy to use production schemas that are more expressive than the standard context-free productions. *Extended context-free grammars (ECFG)* (Salomaa 1973) extend grammar productions to production schemas whose right-hand sides are regular languages over the nonterminal and terminal symbols. ECFGs are weakly equivalent to context-free grammars but more succinct and flexible. In particular, any right linear grammar is equivalent to a ECFG that has just one rule schema and whose derivations have only one rewriting step. This expressivity of ECFG is very nice when we do not need too fine-grained derivation trees but rather want to reduce the height of derivation trees during the recognition of strings.

Lemma 2.5. *There is an extended context-free grammar that generates the language* $L_{\mathsf{NXGRAPH},\mathbf{S}}$ *with derivation steps that correspond 1–1 to the pairs of brackets (except the topmost step).*

Proof. The original grammar of Lemma 2.3 can be written as an extended context-free grammar that removes some recursion and uses

regular expressions instead:

(2) $S \rightarrow \big(Q \mid \{\}\big)^*$

(3) $Q \rightarrow [\,S'\,] \qquad S' \rightarrow \big(Q \mid \{\}\big)\, S'' \mid \{\} \qquad S'' \rightarrow \big(Q \mid \{\}\big)^+$

By substitution of S' and S'', we replace (3) to obtain:

(4) $Q \rightarrow [\,\big(\,\{\} \mid \big(Q \mid \{\}\big)\big(Q \mid \{\}\big)^+\big)\,]$

Each production schema consists of a left-hand-side (lhs) and a right-hand side (rhs) – a nonterminal and a regular language. □

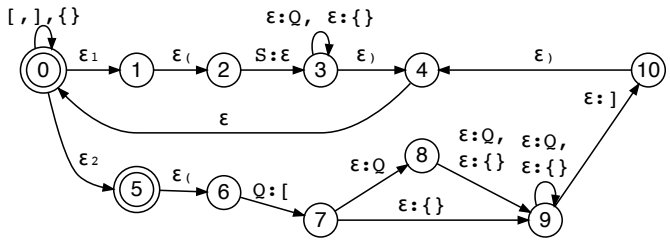
ECFGs reduce beautifully to the iterated application of finite-state transducers (FSTs). We prove just the following special case.

Lemma 2.6. *There is a finite-state transducer that represents the grammar of the proof of Lemma* 2.3. *Its transitive closure generates the language* $L_{NXGRAPH,S}$ *when restricted to the start symbol* S *in the input side and the terminal symbol string in the output side.*

Proof. Starting from the grammar of Lemma 2.5, we will construct one possible transducer representation. First, the two production schemas (2) and (3) compile into two finite-state transducers. Each transducer maps the lhs of the corresponding production to the corresponding rhs. A larger transducer T_G is constructed from these two subtransducers with additional epsilon transitions and self-loop transitions that accept any terminal symbols that have been produced in the earlier stages of the derivation.

The constructed grammar transducer T_G is shown in (5).

(5)



In this transducer, an edge label with a colon indicates that an input string is replaced with some other factor in the output. For example, ε:{} indicates that the empty string ε is replaced with the string "{}".

There are also some edges that do not have labels with a colon. Such labels denote a transition that copies its input to the output. Note that the production schemas (2) and (4) appear as subtransducers in the whole. The first corresponds to the transducer between states 2 and 3, and the second corresponds to the transducer between states 6 and 10. The epsilon symbols $\varepsilon_1, \varepsilon_2, \varepsilon_(, \varepsilon_)$ denote empty strings like ε. The first two avoid cluttering the diagram and the latter two mark the beginning and end of a rule application. From the perspective of the current proof, these epsilons could have been replaced with ε and removed from the transducer all together (Mohri 1997).

The transitive closure of this transducer maps the start symbol to all the intermediate ("sentential form") strings that can be derived from S with the production schemas. When these strings are restricted to the terminal alphabet $\{[,],\{,\}\}$, we obtain the string language generated by the original grammar. □

2.2 *Strong bracketing for noncrossing digraphs*

A (nonempty) digraph is a pair (V,A) where V is a nonempty set of vertices and $A \subseteq \{(i,j) \in V \times V \mid i \neq j\}$ is a set of *arcs*. Each arc (u,v), written as $u \to v$, is a directed edge from vertex u to vertex v. A digraph (V,A) is *inverted* if $(i,j) \in A$ implies $(j,i) \in A$. The *complete digraph* (V,A) has all possible arcs $A = \{(i,j) \in V \times V \mid i \neq j\}$. The *underlying graph* of a digraph (V,A) is the graph (V,E_A) where $E_A = \{\{u,v\} \mid (u,v) \in A\}$ is the set of underlying edges. Note that the cardinality $|E_A|$ of the set of underlying edges can be smaller than the cardinality $|A|$ of the set of arcs. An *ordered digraph* (n,A) is a digraph (V,A) with $\leq$-ordered vertices $V = [1,\ldots,n]$ and a *noncrossing digraph* (n,A) is an ordered digraph whose underlying ordered graph (n,E_A) is noncrossing. The set of (nonempty) noncrossing digraphs is denoted as NXDIGRAPH. This set extends to a concatenation monoid in the same way as the carrying set of the concatenation monoid of noncrossing graphs.

Lemma 2.7. *There is a bijection between ordered digraphs and ordered graphs with 3 labels for edges.*

Proof. Let $C = \{\boxed{\leftarrow}, \boxed{\rightarrow}, \boxed{\leftrightarrow}\}$ be the set of three edge labels. Let f be the trivial function that maps each ordered digraph (n,A) to its underlying graph (n,E_A) and a labelling function $\lambda : E_A \to C$ such that $\lambda(\{i,j\}_{i<j}) = \boxed{\leftrightarrow}$ if $(i,j),(j,i) \in A$, $\lambda(\{i,j\}_{i<j}) = \boxed{\rightarrow}$ if $(i,j) \in A, (j,i) \notin A$

and $\lambda(\{i,j\}_{i<j}) = \boxed{\leftarrow}$ if $(i,j) \notin A, (j,i) \in A$. Conversely, the inverse of f maps the pair of (n, E_A) and a labelling function $\lambda : E_A \to C$ back to a digraph (n,A) where $A = \{(i,j),(j,i) \mid \lambda(\{i,j\}) = \boxed{\leftrightarrow}\} \cup \{(i,j) \mid i < j, \lambda(\{i,j\}) = \boxed{\leftarrow}\} \cup \{(j,i) \mid i < j, \lambda(\{i,j\}) = \boxed{\rightarrow}\}$. Since the inverse f^{-1} is also a function, f is a bijection. □

By Lemma 2.7, an encoding for digraphs is available if we can add labels to edges. In Yli-Jyrä and Gómez-Rodríguez (2017), edge labels are used to signal the type of their larger configuration context.

Lemma 2.8. *There is an iconic, invertible and homomorphic encoding for digraphs.*

Proof. Any noncrossing ordered digraph (n,A) can be encoded with slight modifications to the encoding algorithm $\mathsf{enc_S}$ for noncrossing graphs: instead of printing "`[...]`" for an edge $\{i,j\} \in E_A$, $i \leq j$, the algorithm should now print

$$\begin{array}{ll} \text{“}/\cdots>\text{”} & \text{if } (i,j) \in A, (j,i) \notin A; \\ \text{“}<\ldots\backslash\text{”} & \text{if } (i,j) \notin A, (j,i) \in A; \\ \text{“ [...] ”} & \text{if } (i,j), (j,i) \in A. \end{array}$$

This extends the image of the encoding function to the language of encoded noncrossing digraphs, $L_{\mathtt{NXDIGRAPH},\mathtt{S}}$. The output of the changed encoding function respects the concatenation of digraphs. Corresponding changes are introduced to $\mathsf{dec_S}$ to obtain an inverse function for the encoding function. □

By Lemma 2.8, we encode the ordered digraph $(4, \{(4,1),(1,2),(4,2)\})$ as the string "`</{}><{}{}\\`". Similarly, the digraph in Figure 3 is encoded as the string "`<{}<{}<{}\<{}<{}\<{}\\\\/{}>`".

It is not necessary to have two separate encoding functions $\mathsf{enc_S}$ for noncrossing graphs and noncrossing digraphs. Lemma 2.9 effectively states that we can embed `NXGRAPH` into `NXDIGRAPH`.

Lemma 2.9. *There is a bijection between graphs and inverted digraphs.*

Proof. Let f be a function that maps each inverted digraph (V,A) to a graph (V,E) where $E = \{\{i,j\} \mid (i,j),(j,i) \in A\}$. The definition of f is straightforward. The inverse f^{-1} of f relates each graph (V,E) to an

inverted digraph (V,A) where $A = \{(i,j),(j,i) \mid \{i,j\} \in E\}$. Since this is also a function, f is a bijection. □

By Lemma 2.9, graphs can be treated as special cases of digraphs. Yli-Jyrä and Gómez-Rodríguez (2017) employ the encoding of noncrossing digraphs and Lemma 2.9 (implicitly) to construct unambiguous context-free languages that encode important families of digraphs and graphs. Such context-free languages correspond to the rooted noncrossing trees, the projective trees, the noncrossing dags, the noncrossing weakly connected dags, unoriented noncrossing trees and many other families of noncrossing digraphs and graphs.

Although digraph bracketing is more general and expressive than the graph bracketing, it has two practical disadvantages due to which we prefer to focus, in the rest of this article, on the encoding of (unlabelled) noncrossing graphs whenever possible.

- Firstly, the ordered digraph bracketing is more difficult to interpret than square brackets that contain less information. To get a possibly more readable notation, the direction of the edges can be encoded using subscripted square brackets: plain square brackets would indicate inverted or undirected edges, but a specific orientation of the corresponding edges is indicated with subscripts as in "$[_{<}{}[_{<}{}[_{<}{}]\,[_{<}{}[_{<}{}]\,[_{<}{}]\,]\,]\,]\,[_{/}{}]_{>}$".
- Secondly, since the different types of left and right brackets must match each other, bracketing of digraphs require more states in the finite-state approximation. The increased complexity is needed to keep track of the open brackets. This consideration in the encoding complexity may be addressed with one-sided labelling, e.g., by dropping the subcripts of the right square brackets:

  $$[_{<}{}[_{<}{}[_{<}{}][_{<}{}[_{<}{}][_{<}{}]]]][_{/}{}],$$

  the left square brackets:

  $$[{}[{}[{}]\,[{}[{}]\,[{}]\,]\,]\,]\,[{}]_{>},$$

  or, for example, the head side brackets:

  $$[_{<}{}[_{<}{}[_{<}{}][_{<}{}[_{<}{}][_{<}{}]]]][{}]_{>}.$$

The families of noncrossing trees and noncrossing rooted trees can be treated as restrictions of the sets of noncrossing graphs and digraphs. These families are not treated separately in this article (except the observation on the state complexity of the search space of projective trees on page 205). In fact, the present work on noncrossing graphs generalizes to all 50 subfamilies of noncrossing graphs described in Yli-Jyrä and Gómez-Rodríguez (2017), although the current discussion of these is restricted to the most general case.

3 THE PROBLEM OF UNBOUNDED DEPTH

Normal-looking natural language sentences may give rise to a surprisingly high complexity when measured in terms of the depth of nested brackets and overlapping edges.

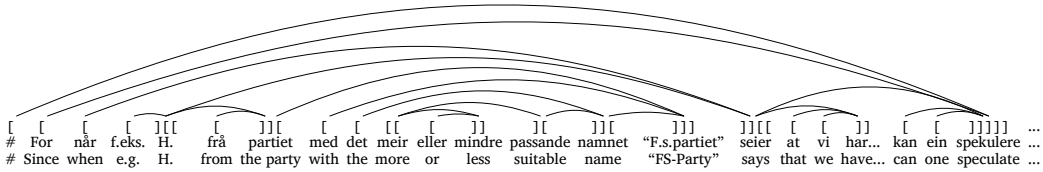


Figure 4: The underlying dependency tree of a Nynorsk (Norwegian) sentence

Figure 4 shows the parse or analysis of a sentence found in a treebank that follows the Universal Dependencies annotation scheme. The figure does not show all the details of the edge orientation and labels, but it reveals that the underlying graph of the parse is a noncrossing tree. The exceptional complexity of this ordered graph comes from its multiple levels of overlapping edges. These overlapping edges correspond to nested brackets. Due to the overlapping, the original encoding (Yli-Jyrä and Gómez-Rodríguez 2017) requires, in fact, up to 10 levels of square brackets. The sentence demonstrates that natural language sentences may involve many levels of overlapping dependency edges even though no clausal center embedding is clearly present.

Another observation from Figure 4 is that the current visualisation of overlapping edges is not very readable, and the corresponding brackets are stacked up to form almost meaningless sequences. It is, thus, obvious that this kind of bracketing requires many states in a finite-state approximation. In this section, our objective is to find a

new, simpler way to encode and draw diagrams of dependency analysis, and to reduce the state complexity of the encoding.

3.1 *Deep nesting outside dependency graphs*

Balanced bracketing has been used in many contexts that include but are not restricted to Generative Grammar, programming languages and document markup systems. In Generative Grammar, so-called *P-markers* have been used to describe phrase-structure trees. In the Lisp programming language (Teitelman 1978), a large number of parentheses are typically needed in lists that constitute the fundamental data structure of the programming language. The XML markup language and its predecessor, SGML (Goldfarb 1999), use brackets to indicate trees.

Deep nesting of brackets is a standard source of difficulties in applications of balanced bracketing. For example, it is well known that adding P-markers to context-free grammars changes their tail recursion into center-embedding (Langendoen 1975). The change converts regular, right- or left-linear context-free grammars into grammars that generate non-regular languages (except if the grammar is completely recursion-free and generates a finite language). Also, in Lisp programs and structured SGML and XML documents, brackets can be nested arbitrarily, and specialized markup editors are needed to keep track of the open brackets while editing them. Often the problem is in left- or right-linear recursion whose balanced bracketing is inconvenient due to the unbounded nesting.

To overcome the challenges of deep nesting in *strong* balanced bracketing, there are several approaches and techniques that are closely related to each other. The techniques make the bracketing unbalanced in a controlled and reversible ways. As to Generative Grammar, Chomsky (1963) already proposed omitting left or right brackets of P-markers in contexts where the original bracketing can be recovered without ambiguity. This idea of “semibrackets” was used to turn context-free grammars into grammars that produce *weak* bracketing, and to turn any non-self-embedding grammar as a whole into a finite-state transducer (Langendoen 1975; Krauwer and des Tombe 1981; Langendoen and Langsam 1984; Yli-Jyrä 2003c; Hulden and Silfverberg 2014). A complementary idea appears in InterLISP (Teitelman 1978, Section 2: Using Interlisp, page 2.4) where the pro-

grammer could close any unfinished round brackets with just one square bracket (`]`), a.k.a. "super-parentheses": "a right square bracket automatically supplies enough right parentheses to match back to the last left square bracket (in the expression being read), or if none has appeared, to match the first left parentheses". For example, this gives the following short-hand notations:

(6)	short-hand		expansion
	$(A(B(C]$	$=$	$(A(B(C)))$
	$(A[B(C(D]E)$	$=$	$(A(B(C(D)))E)$

The role of "super-parentheses" (Teitelman 1978), or "superbrackets" in the sequel, is complementary to that of "semibrackets" that indicate the location of an initial or a final embedding (Chomsky 1963; Langendoen 1975): they close arbitrarily many one-sided brackets

It is not always easy to take advantage of weak bracketing that is based on superbrackets and semibrackets. The SGML standard (ISO 8879:1986) allowed the omission of redundant brackets, but this capacity of the standard made SGML-documents difficult to validate and parse, and contributed to the abandoning of the standard, in favour of XML. Elsewhere, a version of weak bracketing in the framework of Finite-State Intersection Grammar (Koskenniemi 1990) was used in an encoding scheme where an unbalanced clause-boundary marker "`@/`" indicated left or right recursion of clauses, leaving some unresolved ambiguity in the encoding on purpose: in (7), the sentence contains two levels of final clausal embedding, and in (8), there is an initial clause embedding. Thus, the markup used in the grammar framework did not indicate which clause is a subordinate clause and which is the main clause.

(7) `It was a dog @/ that ate the mouse @/ that chased the cat.`

(8) `If the rats ate the cat @/ we were surprised.`

Our present discussion does not try to advocate weak bracketing as a markup formalism for annotated data, because the benefits of weak bracketing for human-computer interaction are controversial. Instead, the focus of the research is on possible benefits for the state complexity when the subfamilies of graphs or the corresponding

search spaces are represented as string languages. The prior experiences with weak bracketing, in fact, suggest that its main advantage is related to the more natural treatment of left- and right-linear recursion and to the computational benefits of such a treatment.

3.2 *Tail recursion in dependency bracketing*

The idea of weak bracketing has gone almost unrecognised in the context of the dependency or edge bracketing of Yli-Jyrä and Gómez-Rodríguez (2017) since such bracketing is historically unrelated to P-markers and their recursion problems. In the dependency bracketing, right-linear embedding corresponds to local bracketing that does not introduce any recursive center-embedding. For example, tail recursion in (9) does increase the depth of dependency bracketing.

(9) `It was a dog[ that[]]ate the mouse[that[]]chased the cat.`

Close to the earliest use of dependency bracketing is due to Greibach (1973) who used brackets to mark "phrase-subphrase" dependencies and to represent context-free languages via specifically bracketed Greibach Normal-Form (GNF) grammars. This bracketing maintains the regularity of the language although it contains balanced brackets: since a right-linear grammar (10) is already in a GNF, adding "phrase-subphrase" brackets converts it to another non-self-embedding context-free grammar (11) that generates a regular language. The same is not true for P-markers, which produce a grammar (12) that generates a non-regular language.

(10) $S \to aS_b \qquad S_b \to bS_b \qquad S_b \to \varepsilon$

(11) $S \to a\ [_{S_b} S_b \qquad S_b \to]_{S_b}\, b\ [_{S_b} S_b \qquad S_b \to]_{S_b}$

(12) $S \to aS_b \qquad S_b \to [_{S_b}\ bS_b\ ]_{S_b} \qquad S_b \to [_{S_b}\ ]_{S_b}$

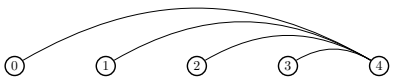
Bracketed "phrase-subphrase" dependencies have been rediscovered in projective dependency parsing by Oflazer (2003) and in nonprojective dependency parsing by Yli-Jyrä (2003b). The bracketing in projective dependency parsing has been developed further to obtain a Chomsky-Schützenberger representation for the string set and the set of structures generated by a projective dependency grammar (Yli-Jyrä 2005a) and a Link Grammar (Ginter *et al.* 2006), to obtain a cubic-time projective dependency parsing algorithm (Yli-Jyrä 2012), and

finally to obtain a synthesis where the representation and the algorithm are combined and generalised to noncrossing graphs (Yli-Jyrä and Gómez-Rodríguez 2017). There has also been some research on bracketing schemes that apply to nonprojective dependency parsing (Yli-Jyrä 2003b, 2004; Gómez-Rodríguez and Nivre 2010; Yli-Jyrä and Nykänen 2014).

3.3 *Unbounded branching in dependency graphs*

It now comes as a surprise that the encoding scheme for noncrossing (di)graphs generates deeply nested brackets when the scheme is applied to syntactic analysis of natural language. The key observation is that multiple sibling edges give rise to adjacent copies of similar brackets:

(13)



(14) `[ {} [ {} [ {} [ {} ]]]]`

The encoding of siblings create nested brackets that are similar to what one obtains in tail recursion. So, if there is no bound for the edge-degree of vertices, the encoded graphs can require a self-embedding grammar even if the graphs would be as simple as star-graphs (trees where one vertex has vertex-degree $n-1$ and all other nodes have vertex-degree 1).

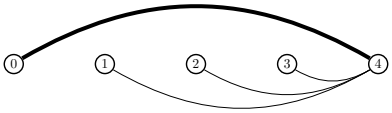
4 NEW ENCODING AND VISUALISATION

Interestingly, it turns out that we can use “superbrackets” and “semibrackets”, introduced for Lisp, P-markers and SGML, when we encode dependency graphs. The intuitive idea is simple: an outermost edge is replaced with *superbrackets* “⟦” and “⟧” that mark the incident vertices. If the left incident vertex has more edges on the right, their respective end vertices are marked with “`]`”. If the right incident vertex has more edges on the left, their respective end vertices are marked with “`[`”. The brackets “`]`” and “`[`” are called *semibrackets*. The process is repeated until all outermost edges and their shorter siblings have been converted in this way. As a whole, we call this encoding *weak (dependency) bracketing*. The prototypical example (13) is encoded with weak bracketing (15).

(15) ⟦ {} [{} [{} [{} ⟧

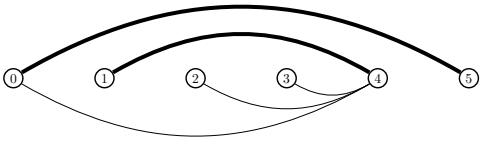
The weak bracketing separates edges into three categories: The *superbracketed edges*, the *left (inner) siblings* of superbracketed edges, and the *right (inner) siblings* of the superbracketed edge. In the following, we use this classification to reduce the visual clutter of dependency diagrams: the inner siblings of the superbracketed edges are drawn below the line of vertices:

(16)



The classification of edges is based barely on the graph structure and is, therefore, not dependent on processing order. However, the category of an edge is not a local property: the category of an edge alternates between a superbracketed edge and a sibling edge. Such alternation starts from the outermost edge and proceeds transitively towards inner edges:

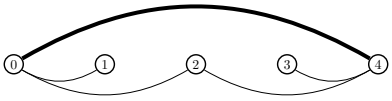
(17)



(18) ⟦ {} ⟦ {} [{} [{} ⟧] {} ⟧

The technique extends to situations where one vertex is is connected to both ends of the outermost edge with sibling edges:

(19)



(20) ⟦ {}] {}][{} [{} ⟧

Figure 5 shows how the improved encoding is applied to a real dependency tree. The obtained graphical representation is immediately more readable in a very systematic way.

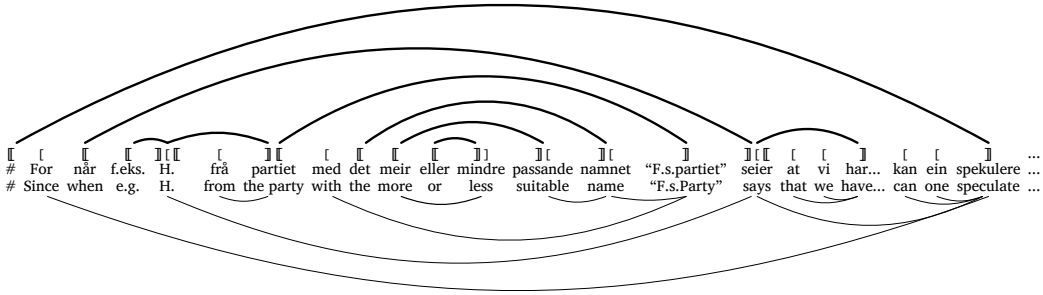


Figure 5: The application of the weak bracketing to the Nynorsk parse tree

Lemma 4.1. *There is a context-free grammar that generates the relation between the strong and the weak bracketing for noncrossing graphs.*

Proof. In the grammar of Lemma 2.6, we can distinguish three kinds of occurrences of the nonterminal Q: the initial Q_I, the central Q_C and the final Q_F. Each of these will be bracketed differently by the following grammar whose terminal alphabet is a pair alphabet $\Sigma = \{$`{:{,}:}`, `[:`ε`,]:`ε, `[:[,]:]`, `[:⟦,]:⟧`$\}$. The pair symbols in this alphabet are constructed from the empty string ε and the input and output symbols, and from the colon that separates them.

(21) $S \rightarrow \big(Q_C \mid$ `{:{}:}` $\big)^*$

(22) $Q_I \rightarrow$ `[:`ε S' `]:]`

(23) $Q_C \rightarrow$ `[:⟦` S' `]:⟧`

(24) $Q_F \rightarrow$ `[:[` S' `]:`ε

(25) $S' \rightarrow \big(Q_I \mid$ `{:{}:}` $\big)$ $S'' \mid$ `{:{}:}`

(26) $S'' \rightarrow \big(Q_C \mid$ `{:{}:}` $\big)^* \big(Q_F \mid$ `{:{}:}` $\big)$

Each symbol in the terminal alphabet of this grammar is a pair $a{:}b$ where $a \in \Sigma_1^*$ is the input factor and $b \in \Sigma_2^*$ is the output factor. The alphabet of the input strings is $\Sigma_1 = \{$`{`, `}`, `[`, `]`$\}$ and the alphabet of the output strings is $\Sigma_2 = \{$`{`, `}`, `[`, `]`, `⟦`, `⟧`$\}$. The factor ε, in particular, is the empty string. Let $w = (a_1{:}b_1)\ldots(a_n{:}b_n) \in \Sigma^*$ be a string generated by the grammar. The concatenation of the input factors $a_1, \ldots, a_n$ constitutes the input string $a_1 \ldots a_n$ and the concatenation of

the output factors $b_1, \ldots, b_n$ constitutes the output string $b_1 \ldots b_n$. Together, the input and the output constitute a pair $(a_1 \ldots a_n, b_1 \ldots b_n) \in \Sigma_1^* \times \Sigma_2^*$. In this way, the grammar defines a relation between the input strings Σ_1^* and the output strings Σ_2^*. For example, the grammar relates the input "`[[{}]{}]`" to the output "⟦{}]{}⟧". □

In contrast to the language of strong bracketing for noncrossing graphs, $L_{\text{NXGRAPH,S}}$, we denote the language of weak bracketing for noncrossing graphs with $L_{\text{NXGRAPH,W}}$.

Lemma 4.2. *There is an extended context-free grammar that generates the language of weak bracketing ($L_{\text{NXGRAPH,W}}$) with derivation steps that correspond 1–1 to the pairs of superbrackets (except the topmost step).*

Proof. The language $L_{\text{NXGRAPH,W}}$ of the encoded noncrossing graphs is generated by an extended context-free grammar:

$$S \rightarrow \left(\{\} \mid Q \right)^*$$

$$Q \rightarrow [\!] \; S_{]} \; E \; S_{[} \; [\!] \mid [\!] \; S_{!} \; [\!] \tag{27}$$

$$E_{]} \rightarrow \varepsilon \mid] \qquad E_{[} \rightarrow \varepsilon \mid [\qquad E \rightarrow [\mid][$$

$$S_{!} \rightarrow \left(\{\} \left(E_{]} \; Q \right)^* E_{]} \right)^* \{\} \tag{28}$$

$$S_{]} \rightarrow \left(\{\} \left(E_{]} \; Q \right)^* E_{]} \right)^* \{\} \left(E_{]} \; Q \right)^* \tag{29}$$

$$S_{[} \rightarrow \left(Q \; E_{[} \right)^* \{\} \left(E_{[} \left(Q \; E_{[} \right)^* \{\} \right)^* \tag{30}$$

To expand the right-hand side of the production schema (27), we substitute the nonterminal symbols $S_{!}$, $S_{]}$, $S_{[}$, $E_{]}$, $E_{[}$, and E with the right-hand sides of the corresponding production schemas. One application of the expanded production schema then corresponds to exactly one level of superbrackets. □

Lemma 4.3. *There is a finite-state transducer whose transitive closure maps the start symbol S to the language $L_{\text{NXGRAPH,W}}$.*

Proof. Figure 6 shows a transducer that represents the grammar of Lemma 4.2. In this transducer, the transitions that copy the input factor to the output are indicated with simple labels that do not contain a colon. Starting from the input string S, the transitive closure of this transducer generates exactly the language of the grammar when output strings of the closure are restricted to the terminal strings. □

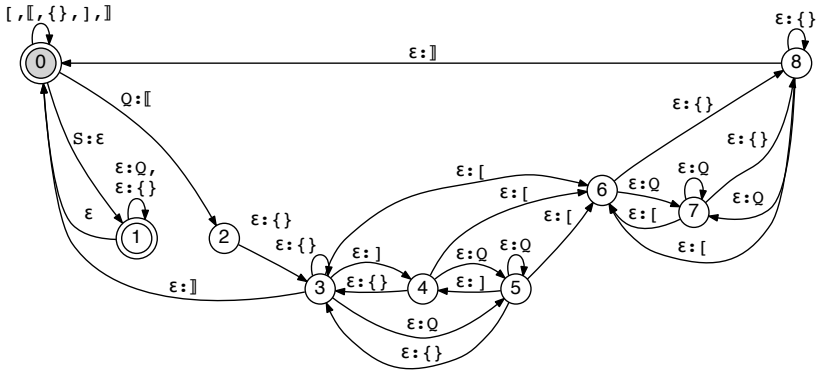


Figure 6:
A finite-state transducer that represents the grammar of Lemma 4.2

Lemma 4.4. *There is a bijective encoding morphism from the monoid of noncrossing graphs to the language* $L_{\text{NXGRAPH,W}}$.

Proof. The left column of Figure 7 contains the algorithm enc_W that maps a noncrossing graph (n, E) to an encoding that is based on weak bracketing. This algorithm works by iterating the vertex index i over the ordered vertices $[1, \ldots, n]$.

- On lines 7–10, the algorithm adds a closing superbracket "⟧" if it is time to remove the corresponding edge from the stack (`stk`).
- On lines 11–12, the algorithm adds a semibracket "]" if vertex i ends a shorter sibling of the topmost edge in the stack.

def enc_W((n,E) $\in$ NXGRAPH):

```
1 stk = [(0,n+1)]
2 str = ""
3 for i in [1,2,...,n]:
4     if i > 1:
5        str += "{}";
6     (l,r) = stk.top()
7     if i == r:
8        str += "⟧"
9        stk.pop()
10       (l,r) = stk.top()
11    if {l,i} in E:
12       str += "]"
13    if {i,r} in E:
14       str += "["
15    for j in [r-1,r-2,...,i+1]:
16        if {i,j} in E:
17            str += "⟦"
18            stk.push( (i,j) )
19            break
20 return str
```

def dec_W(str $\in L_{\text{NXGRAPH,W}}$):

```
1 n   = 1
2 E   = {}
3 stk = []
4 for a in str:
5     if a == "⟦":
6         stk.push([n])
7     elif a == "{":
8         n += 1
9     elif a == "]":
10        E += { {stk.top[0],n} }
11    elif a == "[":
12        stk.top() = stk.top() + [n]
13    elif a == "⟧":
14        for j in stk.pop():
15            E += { {j,n} }
16 return (n,E)
```

Figure 7:
Functions that encode/decode noncrossing graphs using weak bracketing

- On lines 13–14, the algorithm adds a semibracket "`[`" if vertex i starts a shorter sibling of the topmost edge in the stack.
- On lines 15–19, the algorithm adds an opening superbracket "`[[`" if vertex i starts a superbracketed edge that is not a shorter sibling of the topmost edge in the stack.
- Between iterations, on lines 4–5, the algorithm adds a vertex boundary "`{}`".

It is easy to see that the algorithm $\mathsf{enc}_{\mathsf{W}}$ always terminates, and it implements a mapping from all noncrossing graphs to strings in $L_{\text{NXGRAPH},\mathsf{W}}$. It is also easy to see that the algorithm respects concatenation: $\mathsf{enc}_{\mathsf{W}}((n,E_1)) \cdot \mathsf{enc}_{\mathsf{W}}((m,E_2)) = \mathsf{enc}_{\mathsf{W}}((n,E_1) \cdot (m,E_2))$.

Conversely, the right column of Figure 7 contains the algorithm $\mathsf{dec}_{\mathsf{W}}$ that maps strings in $L_{\text{NXGRAPH},\mathsf{W}}$ to noncrossing graphs. The algorithm reads its input string from left to right.

- On lines 5–6, when the opening superbracket "`[[`" is read, the algorithm pushes to the stack a list that just contains the current vertex n.
- On lines 7–8, when the left curly bracket "`{`" is read, the algorithm starts a new vertex by incrementing n. The right curly bracket "`}`" is just ignored in the well-formed input.
- On lines 9–10, when the right semibracket "`]`" is read, the algorithm looks for the first vertex number in the topmost list in the stack and adds an edge between it and the current vertex.
- On lines 11–12, when the left semibracket "`[`" is read, the algorithm adds the current vertex to the topmost list in the stack.
- On lines 13–15, when the closing superbracket "`]]`" is read, the algorithm pops the topmost list from the stack and adds an edge between the current vertex and the vertices in this list.

It is easy to verify that the decoding algorithm $\mathsf{dec}_{\mathsf{W}}$ runs in linear time to the length of the input string. □

Lemma 4.5. *There is an algorithm to compute the bracketing depth of graphs in weak bracketing.*

Proof. The algorithm for computing the weak bracketing depth of a graph (n,E) is given in Figure 8. □

def depth$_{\mathsf{W}}$((n,E)):

```
d = maxd = 0
for c in enc_W(n,E):
    if c == "⟦":
        d += 1
        maxd = max(maxd,d)
    elif c == "⟧":
        d -= 1
return maxd
```

Figure 8:
An algorithm for measuring the depth of the balanced brackets in weak edge bracketing

For example, the algorithm depth$_{\mathsf{W}}$ returns value 0 for graph $(2, \{\})$ and 1 for graph $(3, \{\{1,3\},\{2,3\}\})$ because these graphs encode as "$\langle\rangle$" and "$\llbracket\langle\rangle\rangle\rrbracket$". For $(4, \{\{1,4\},\{2,3\}\})$, the algorithm returns a depth of 2 because its code string "$\llbracket\langle\rangle\llbracket\langle\rangle\rrbracket\langle\rangle\rrbracket$" contains two levels of superbrackets.

Lemma 4.6. *There is a conventional unambiguous context-free grammar for the streamlined encoding of noncrossing graphs,* $L_{\mathsf{NXGRAPH,W}}$.

Proof. The grammar of Lemma 4.2 is turned into a conventional context-free grammar

$$S \to \langle\rangle\, S \mid Q\, S \mid \varepsilon$$

$$Q \to \llbracket\, S_{]}\, E\, S_{[}\, \rrbracket \mid \llbracket\, S_{!}\, \rrbracket \tag{31}$$

$$E_{]} \to \varepsilon \mid] \qquad E_{[} \to \varepsilon \mid [\qquad E \to [\ \mid\ ][$$

$$S_{!} \to \langle\rangle \mid \langle\rangle\, T\ ;\ T \to E_{]}\, \langle\rangle\, T \mid E_{]}\, \langle\rangle \mid E_{]}\, Q\, T \tag{32}$$

$$S_{]} \to \langle\rangle \mid \langle\rangle\, U\ ;\ U \to E_{]}\, \langle\rangle\, U \mid E_{]}\, \langle\rangle \mid E_{]}\, Q\, U \mid E_{]}\, Q \tag{33}$$

$$S_{[} \to W\ ;\ W \to Q\, E_{[}\, W \mid \langle\rangle\, E_{[}\, W \mid \langle\rangle \tag{34}$$

To obtain this grammar, we take each right-hand-side that describes a regular language over an alphabet $\Sigma \cup V$ and replace it with a context-free subgrammar that generates this regular language. In this way, the production schemas (28)–(30) expand, respectively, to the subgrammars (32)–(34). It is now easy to verify that the resulting grammar, as a whole, is unambiguous. Especially, the production schema (31) is unambiguous, since $S_{!}$ does not generate [outside an embedded Q while E always generates [. □

5 EVALUATION OF THE ENCODING SCHEMES

5.1 *Variants of the two encoding schemes*

Until now, we have given ECFG grammars for two different encodings for graphs. When these grammars are extended to brackets that indicate the direction of an edge on both sides, we obtain grammars for two different encodings for digraphs. For example, in the ECFG grammar of Lemma 2.6, the grammar is extended with two additional rules

(35) $Q \to /S'>$ $\qquad$ $Q \to <S'\backslash$

We call the encoding of Yli-Jyrä and Gómez-Rodríguez (2017) the *strong bracketing (*S*)* and the currently proposed encoding *the weak bracketing (*W*)*. In addition to these, we identify three optimisations that are available to both strong and weak bracketing:

1. The first optimisation ("$\{\}$") simplifies the pair of curly brackets by replacing it with an atomic symbol: a bullet dot "$\bullet$":

   (36) $\{\} \to \bullet$

2. The second optimisation ("1") is to eliminate the difference between the symbols used as a left bracket:

   (37) $[\![_< \to [\![\qquad [\![_/ \to [\![\qquad < \to [\qquad / \to [$

3. The third optimisation ("$[\{\}]$") introduces new vertex boundaries "$\underset{[]}{\bullet}$", "$\underset{<\backslash}{\bullet}$", and "$\underset{/>}{\bullet}$" in order to compress the edges between adjacent vertices as follows:

   (38) $[\bullet] \to \underset{[]}{\bullet} \qquad [_<\bullet]_\backslash \to \underset{<\backslash}{\bullet} \qquad [_/\bullet]_> \to \underset{/>}{\bullet}$

   (39) $[\![\bullet]\!] \to \underset{[]}{\bullet} \qquad [\![_<\bullet]\!]_\backslash \to \underset{<\backslash}{\bullet} \qquad [\![_/\bullet]\!]_> \to \underset{/>}{\bullet}$

   (40) $[\![_\alpha\bullet] \to [\![_\alpha \underset{[]}{\bullet} \qquad [\![_\alpha\bullet]_\backslash \to [\![_\alpha \underset{<\backslash}{\bullet} \qquad [\![_\alpha\bullet]_> \to [\![_\alpha \underset{/>}{\bullet}$

   (41) $[\bullet]\!]_\beta \to \underset{[]}{\bullet}]\!]_\beta \qquad [_<\bullet]\!]_\beta \to \underset{<\backslash}{\bullet}]\!]_\beta \qquad [_/\bullet]\!]_\beta \to \underset{/>}{\bullet}]\!]_\beta$

   where $\alpha \in \{\varepsilon, <, /\}$ and $\beta \in \{\varepsilon, >, \backslash\}$.

   This optimisation implies the "$\{\}$"-optimisation.

These optimisations are meant to optimise the state complexity of finite-state automata, but they also come with some trade-offs. The main disadvantage of the "1"-optimisation is that the information about the direction is no longer locally present at the bracket where one might need it. The "[◊]"-optimisation suffers from an increased alphabet size.

The Cartesian product of two encoding schemes and three optimisations gives us twelve different bracketing schemes:

$$\{\mathbf{S}, \mathbf{W}\} \times \{\varepsilon, \text{“1”}\} \times \{\varepsilon, \text{“◊”}, \text{“[◊]”}\}.$$

We will not compare all of these schemes in detail, but we will include some of them in experiments to get an idea of their relative efficiency. The corresponding bracket alphabets for digraph encoding schemes are summarised in Table 1.

Table 1: The alphabets of different encoding schemes and their variants

Strong bracketing (Lemma 2.6)		Weak bracketing (Lemma 4.2)	
S	[,<,/,],\,>,◊,◊	**W**	⟦,⟦<,⟦/,⟧,⟧\,⟦>,[,<,/,],\,>,◊,◊
S◊	[,<,/,],\,>,•	**W**◊	⟦,⟦<,⟦/,⟧,⟧\,⟦>,[,<,/,],\,>,•
S[◊]	[,<,/,],\,>,•, •[],•<\,•/>	**W**[◊]	⟦,⟦<,⟦/,⟧,⟧\,⟦>,[,<,/,],\,>,•, •[],•<\,•/>
S1	[,],\,>,◊,◊	**W**1	⟦,⟧,⟧\,⟦>,[,<,/,],\,>,◊,◊
S1◊	[,],\,>,•	**W**1◊	⟦,⟧,⟧\,⟦>,[,<,/,],\,>,•
S1[◊]	[,],\,>,•, •[],•<\,•/>	**W**1[◊]	⟦,⟧,⟧\,⟦>,[,<,/,],\,>,•, •[],•<\,•/>

5.2 *State complexity of finite search spaces*

Table 2 reports the size and the state complexity of the search spaces as the function of the number of vertices, n. The first two columns indicate, for example, that there are 1,792 noncrossing 4-vertex digraphs. The deterministic state complexity of this "4-vertex" search space is 490, 334, 30, 106, 19, or 10 states, depending on the encoding (**S** or **W**) and the additional optimisations ("◊", "1◊", "1[◊]").

We learn from Table 2, firstly, that the state complexity of the search space grows exponentially with the number of vertices in the digraphs, regardless of the encoding scheme. The context-free representation of Yli-Jyrä and Gómez-Rodríguez (2017) is immune to the state complexity concerns, but a straightforward depth-bounded finite-state approximation of the **S** scheme explodes immediately.

Table 2: The DFA state complexity of finite search spaces

n	Digraphs	**S**	**S**$_{\{\}}$	**S1**$_{\{\}}$	**W**	**W1**$_{\{\}}$	**W1**$_{[\{\}]}$
1	1	1	1	1	1	1	1
2	4	12	8	4	12	4	2
3	64	80	54	12	26	8	6
4	**1,792**	**490**	**334**	**30**	**106**	**19**	**10**
5	62,464	2,952	2,018	68	207	33	24
6	2,437,120	27,040	12,126	146	704	61	38
7	**101,859,328**	**106,372**	**72,778**	**304**	**1,327**	**95**	**72**

The rapid growth of the state complexity is explained by the fact that larger digraphs involve more open brackets and both encodings (**S** and **W**) keep a record of the type of the open brackets as well as of the intermediate constituent structure of each level, corresponding to the right-hand-side of the production schemas (4) and (27) that expand the nonterminal symbol Q in each grammar. In addition, the state space must keep track of the total number of vertices produced so far. Overall, the state complexity of the strong bracketing compares poorly against the superset approximations of context-free phrase structure grammars (Nederhof 2000).

Secondly, we learn from Table 2 that weak bracketing gives a clear advantage over the strong bracketing. The state complexity of the original encoding (**S**) blows up after 3 vertices and reaches 106,372 DFA states when there are 7 vertices. The state complexity of the weak bracketing scheme is substantially lower than the strong bracketing (**S**). With 7 vertices, **W** requires only 1,327 states, which is an 80-times improvement over the strong bracketing scheme. Moreover, it seems that **S** simply grows faster and faster in comparison to **W** when n increases. The lower growth rate of the complexity of **W** is explained by the fact that **W** does not open more than one superbracket "⟦" per every two vertices whereas **S** opens one pair of brackets per edge.

Similar results are obtained for subfamilies of noncrossing graphs. Figure 9 compares the state complexity of the search spaces of three different families of noncrossing graphs as a function of the number of vertices or words in the sentence. The figure indicates that the advantage of weak bracketing (**W**) in contrast to strong bracketing (**S**) is relatively robust across different families of noncrossing graphs: digraphs, projective trees and graphs are all more compactly presented with weak bracketing than with strong bracketing.

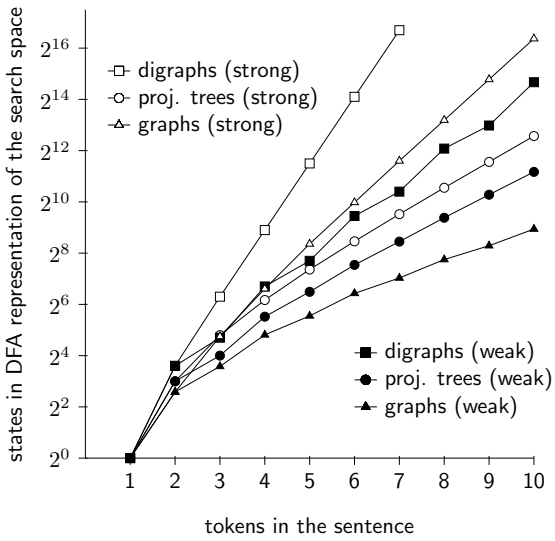


Figure 9:
Weak bracketing brings exponential savings in the size of the DFA representing the search space of a sentence when the search space consists of digraphs, projective trees, or graphs

In sum, the exponential growth of the state space seems to be unavoidable for the set of noncrossing graphs, which reflects the fact that bijective encoding is more difficult than an a superset approximation of the search space. But the state space complexity improves dramatically with our new techniques: weak bracketing, search space restrictions to noncrossing subfamilies, and optimisations in the encoding scheme. The columns for **S**`{}`, **S**1`{}`, **W**1`{}`, and **W**1`[{}]` in Table 2, on page 204, show the improved state complexity of some combined optimisations. These indicate that the state complexity drops, quite dramatically, from **S** and **S**`{}` to **S**1`{}` (from 106,372 and 72,778 to 304 states for n=7) and from **W** to **W**1`{}` (from 1,327 to 95 states). A further improvement is given by the "`[{}]`"-optimisation (from 95 to 72 states for 7 vertices).

Along with the weak bracketing, another big improvement in the state complexity of unweighted search space is due to the "1"-optimisation: thanks to these two improvements, we are able to build complete search spaces for relatively large ordered graphs with at least 30-vertices. Table 3 gives an idea of the implications of these improvements. In short, they can now be expressed as follows:

- The complete search space of all 10-vertex noncrossing graphs and digraphs is represented by a deterministic automaton that has 254 states.

Table 3: State complexity of larger search spaces

n	Encoded graphs	**W1**	**W1**$\{\}$	**W1**$[\{\}]$
10	21,292,032	492	363	**254**
⋮	⋮	⋮	⋮	⋮
19	29,312,424,612,462,592	12,395	9,347	7,569
20	314,739,971,287,154 688	18,276	13,693	10,114
21	3,393,951,437,605,044,224	24,925	18,807	15,228
⋮	⋮	⋮	⋮	⋮
30	$\approx 7.681 \cdot 10^{27}$	589,598	442,177	**327,494**

- The complete search space of all 30-vertex noncrossing graphs and digraphs is represented by a deterministic automaton that has 327,494 states.

5.3 *State complexity of bounded-depth grammars*

Bounding the bracketing depth in the encoding schemes turns the respective grammars into subset approximations. Each approximation is equivalent to a cyclic finite-state automaton that recognises a regular language. We now turn our focus to the state complexity of such bounded-depth grammars and their languages.

Table 4 illustrates the relative parsimony of **W1**$\{\}$ against **S1**$\{\}$ when the depth of balanced bracketing, d, grows. The table shows that **W1**$\{\}$ captures the complete 7-vertex search space of 101,859,328 digraphs already with 3 levels of balanced brackets, while **S1**$\{\}$ needs 6 levels of brackets to capture the same search space. When we increase

Table 4: The state complexity and the largest contained complete search space of depth-bounded grammars

d	n-Digraphs	n	**S1**$\{\}$	n-Digraphs	n	**W1**$\{\}$
0	1	1	1	1	1	1
1	4	2	3	4	2	6
2	64	3	8	62,464	5	18
3	**1,792**	**4**	18	**101,859,328**	**7**	42
4	62,464	5	38	**201,889,939,456**	**9**	90
5	2,437,120	6	78	443,939,433,742,336	11	186
6	**101,859,328**	**7**	158	1,041,383,605,688,860,672	13	378
7	**4,459,528,192**	**8**	318	$\approx 2 \cdot 10^{21}$	15	762

d	$\mathbf{S}_d$	$\mathbf{S}\{\}_d$	$\mathbf{S}1\{\}_d$	$\mathbf{W}\{\}_d$	$\mathbf{W}1_d$	$\mathbf{W}1\{\}_d$	$\mathbf{W}1[\{\}]_d$
0	2	1	1	1	2	1	1
1	11	7	3	2	9	6	7
$d+1$	$6s_d+2$	$6s_d+4$	$2s_d+2$	$6s_d+17$	$2s_d+7$	$2s_d+6$	$2s_d+6$
2	68	46	8	23	25	18	20
3	410	280	18	155	57	42	46
4	2,462	1,684	38	947	121	**90**	98
5	14,774	10,108	78	5,699	249	186	202
6	88,646	**60,652**	**158**	**34,211**	505	**378**	410
7	531,878	363,916	318	205,283	1,017	**762**	826

Table 5: The DFA state complexity of the bounded-depth grammar

both these bounds with one more level, the 8-vertex search space in $\mathbf{S}1\{\}$ is only 44 times larger, whereas the search space captured by $\mathbf{W}1\{\}$ has 9-vertex graphs and grows 1,998 times larger. Thus, the growth of the weakly bracketed search space is roughly quadratic to growth of the strong bracketing. This trend becomes even more striking when the bracketing gets deeper.

Table 5 shows the state complexity of the bounded-depth grammar as the function of the bracketing depth (d) and the used encoding scheme. The first impression is that strong bracketing (**S**) and weak bracketing (**W**) give rise to very similar state complexity of the bounded-depth grammars: while $\mathbf{W}\{\}$ is more compact than $\mathbf{S}\{\}$ with its 34,211 states against 60,652 states, $\mathbf{S}1\{\}$ initially looks more compact than $\mathbf{W}1\{\}$ with its 158 states against 378 states.

We already learned from Table 4 that two levels in **S** compare roughly to one level in **W**. Besides this, the depth of the latter is not sensitive to unbounded branching. Therefore, the complexity of the bounded grammar for weak bracketing is more interesting than the complexity of the bounded grammar for strong bracketing.

We now have an idea about the state complexity of a deterministic finite automaton that recognizes languages of different bounded grammars for strong and weak bracketing. As we from now on talk about the state complexity of bounded grammars, we will focus on the weak bracketing (**W**) and on its one-sided variant (**W**1). Its "$\{\}$"-optimisation is even more succinct, but the "$[\{\}]$"-optimisation appears to be harmful to the state complexity of the bounded grammars.

When the bound d for the depth of bracketing is fixed, the depth of bracketing does not grow arbitrarily when the length of input sentence changes. A grammar with a fixed bound will be applied to short sentences as well as to long sentences. This raises the question of what happens with the coverage and the state complexity of the restricted search space when the sentence is exceptionally long.

When we process a growing number of vertices, the first consequence of using a bounded-depth grammar is that there will be a bound k for the number of vertices beyond which the bounded grammar ceases to capture the complete search space of the noncrossing graphs. Beyond this point, the search space will be limited by the bounded depth. The state complexity of the limited search space will then grow linearly with the number of vertices when the number of vertices continues to grow enough.

For example, let us restrict the depth of bracketing to 6 levels, which gives us a bounded grammar with 505 states. The largest complete search space captured by this grammar consists of 13-vertex graphs (or digraphs, whose state complexity is the same under the **W1** encoding scheme).

If we now extract 21-vertex graphs from the same grammar, we will get only a proper subset of all 21-vertex graphs because graphs that require more than 6 levels of brackets are missing. Capturing the complete search space for 21-vertex graphs requires 10 levels of bracketing. Table 6 shows in detail what happens to the search space of 21-vertex graphs when we decrease the maximum depth of bracket-

Table 6: 98.96% coverage of 21-vertex graphs requires only 6 bracket levels and only 1/8th of the full coverage states

d	$\mathbf{W1}_d$	$\mathbf{W1}_d \cap \mathbf{W1}_{n=21}$		The number/% of 21-vertex graphs	
3	57	929	3.7%	872,294,071,717,330,944	25.70143%
4	121	1,765	7.1%	2,313,578,416,163,258,368	68.16769%
5	249	3,181	12.7%	3,131,655,209,939,369,984	92.27166%
6	**505**	**5,501**	**22.1%**	**3,358,682,892,406,358,016**	**98.96084%**
7	1,017	9,117	36.6%	3,391,549,974,785,294,336	99.92924%
8	2,041	14,301	57.4%	3,393,882,839,790,387,200	99.99798%
9	4,089	20,573	82.5%	3,393,950,914,747,301,888	99.99998%
10	**8,185**	**24,925**	**100.0%**	**3,393,951,437,605,044,224**	**100.00000%**

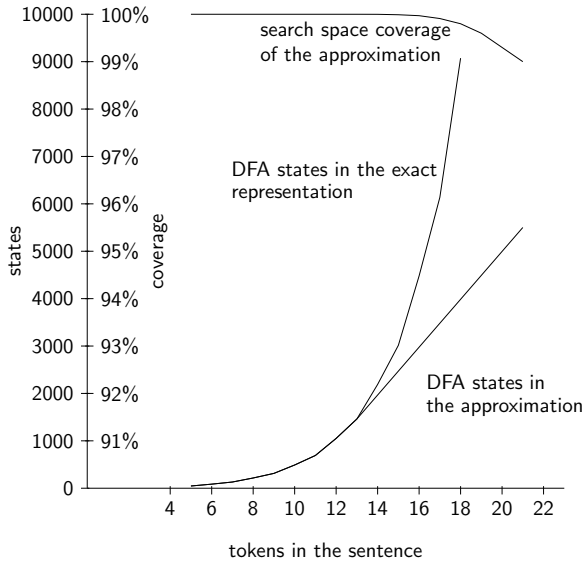


Figure 10:
The state complexity of the limited search space $\mathbf{W1}_{d=6}$ grows only linearly with sentence length after 13 vertices, while the exact search space representation explodes quickly. At the same time, the coverage of the limited search space remains close to 100% for many more vertices

ing. Quite surprisingly, the 6-bounded grammar still contains 98.96% of all 21-vertex graphs!

The linear growth of the state complexity of the limited search space gives a huge advantage over the standard situation where the complete search space requires an exponentially growing number of states as a function of the number of vertices. This point is illustrated by Figure 10. This is also illustrated by Table 6, which shows that the state complexity of this limited search space is only 1/5 of the state complexity of the complete search space and the state complexity of the corresponding bounded grammar is only 6.2% (505 states) of the state complexity of the 10-bounded grammar (8,185 states). Thus, limiting the search space of long sentences by depth is an effective way to reduce the number of states in the grammar and search space representations. This reduction is necessary in practice because there is no fixed limit for the length of natural language sentences: as a challenge for parser developers, the Universal Dependencies treebanks contain a few really long sentences that have more than 500 tokens.

Table 7 describes the state complexity of extremely large limited search spaces that contain 8-, 16-, 32-, …, and 512-vertex noncrossing graphs bounded to 7 levels of brackets. The 7-bounded grammar $\mathbf{W1}_{d=7}$ can be represented, according to Table 5, with 762 states. For

Table 7:
The linear growth of the state complexity of the limited search space with at most 7 levels of balanced brackets

n	Upper bound $((2n-1)\cdot 762)$	Actual states
8	11,430	157
16	23,622	3,029
32	48,006	15,221
64	96,774	39,605
128	194,310	88,373
256	389,382	185,909
512	**779,526**	**380,981**

each integer $n \geq 1$, there is a finite-state automaton whose language $\mathbf{W1}_n$ constrains the graph size to n vertices. The state complexity of $\mathbf{W1}_n$ is exactly $(2n-1)$ states. By multiplying the state complexities of the depth-bounded grammar $\mathbf{W1}_{d=7}$ and the graph size constraint $\mathbf{W1}_n$, we obtain an upper bound for the state complexity of the limited search space of n-vertex noncrossing graphs with a maximum depth of 7 brackets. However, the actual state complexity of the intersection of the two languages is slightly smaller: instead of 779,526 states, we will need only 380,981 states to represent the limited search space of 512-vertex graphs. Thus, with seven levels of brackets, this can be summarised as follows:

- The bounded grammar of noncrossing graphs requires at most 762 DFA states.
- The largest complete search space contained in the bounded grammar consists of noncrossing graphs that have 15 vertices.
- The limited search space for 512-token sentences requires 380,981 DFA states with the **W1** encoding scheme.

5.5 *Empirical coverage of bounded grammars*

An experiment was carried out to apply the bracketing depth measure to the noncrossing trees in the Universal Dependencies (UD) treebanks (Nivre *et al.* 2019). In order to verify that the treebank size does not significantly affect the results, we carried out the same experiment on two different releases of UD treebanks. In the v2.4 release, there are 146 treebanks and 83 languages, while in the v2.0 release, there are 70 treebanks and 50 languages and about half the number of the trees. In

the experiment, we focused on the primary dependencies that define rooted trees. Of the total 1,234,587 rooted trees in the v2.4 data set, 90.5% (1,117,332) are noncrossing, and typically nonprojective. We encoded these trees with the **W** encoding scheme and computed the depth of the bracketing with the algorithm $\texttt{depth}_{\texttt{W}}$ shown previously in Figure 8.

For the purpose of succinct reporting of the results, the number of languages was reduced by grouping some closely related languages into larger buckets. For example, our "Scandinavian" is a group of languages containing Bokmål, Danish, Nynorsk, and Swedish, and Ancient Greek and Old Russian were groupped with their moden variants. However, we did not group Latin with Italian. Although such grouping might remove the sharpest distinctions between languages, it became as a surprise that the depth-based cross-lingual complexity differences decayed so quickly when the depth increased beyond 3 levels. Thus, the row "noncrossing" describes surprisingly well a language independent tendency where the depth of bracketing for noncrossing trees is mostly very low.

The percentage of noncrossing trees and the coverage of the measured complexity levels are shown in Table 8. The first two numerical columns show the percentage and the absolute number of noncrossing trees among all trees for the corresponding language subset. In other columns, the coverage of depth-bounded bracketing is computed against the number of noncrossing trees for the corresponding language subset. The row with the label "noncrossing" corresponds to the set of all languages. Its first two columns tell the percentage and the absolute number of noncrossing trees in the whole UD data set. The mixed data set considered all trees equal in weight regardless of the size of the tree and the size of the containing treebank.

The results in the table are illuminating in two ways. Firstly, the results indicate that a bounded search space with 7 levels of superbrackets is capable of covering 99.999% of the noncrossing trees in the v2.0 and v2.4 versions of the UD dataset. Since 7 levels require only 762 DFA states in "$W1\{\}_d$"-encoding, this result supports our hypothesis according to which *the practically occurring noncrossing dependency digraphs can be embedded in a subset approximation that has a very compact finite-state representation.* Secondly, we observe that the bounded space with 4 levels of superbrackets and 90 states

Table 8: The coverage of depth-bounded W_d grammars measured against UD v2.0 and v2.4 trees that are noncrossing and do not contain *epsilon* nodes

Language	Noncrossing		Depth 1	2	3	4	5	6	7	8
all v2.0 trees	100.0%	**630,518**								
noncrossing	86.7 %	546,492	16.1%	57.7%	91.6%	99.2%	99.95%	99.999%	100.000%	100.000%
all v2.4 trees	100.0%	**1,234,587**								
noncrossing	**90.5%**	**1,117,332**	16.6%	57.8%	91.8%	99.2%	99.95%	99.998%	100.000%	100.000%
Arabic	97.2%	27,608	5.0%	21.1%	64.7%	94.5%	99.70%	99.986%	100.000%	100.000%
Catalan	95.5%	15,931	1.7%	23.0%	75.7%	97.2%	99.78%	100.000%	100.000%	100.000%
Czech	88.3%	112,595	15.0%	52.4%	91.4%	99.2%	99.95%	99.999%	100.000%	100.000%
German	92.9%	178,229	10.9%	52.5%	91.9%	99.4%	99.97%	99.998%	100.000%	100.000%
English	94.7%	32,811	16.5%	57.1%	93.5%	99.5%	99.99%	100.000%	100.000%	100.000%
Spanish	94.5%	32,789	2.3%	29.2%	82.1%	98.2%	99.90%	100.000%	100.000%	100.000%
Finnish	93.5%	32,589	39.3%	80.4%	97.3%	99.7%	99.98%	99.997%	100.000%	100.000%
French	91.6%	57,459	20.0%	59.1%	92.2%	99.3%	99.97%	99.995%	100.000%	100.000%
Greek	54.8%	18,364	31.4%	81.7%	97.7%	99.8%	99.99%	100.000%	100.000%	100.000%
Hebrew	97.0%	6,032	2.3%	28.8%	82.6%	98.9%	99.95%	100.000%	100.000%	100.000%
Hindi	86.2%	16,843	3.4%	50.6%	88.5%	98.7%	99.93%	99.988%	100.000%	100.000%
Croatian	91.0%	8,205	3.8%	39.3%	89.3%	99.3%	99.94%	100.000%	100.000%	100.000%
Hungarian	72.9%	1,312	4.0%	32.9%	79.1%	96.5%	99.70%	99.924%	100.000%	100.000%
Italian	98.1%	33,398	5.7%	50.8%	90.9%	98.9%	99.91%	99.997%	100.000%	100.000%
Japanese	99.8%	66,950	25.5%	75.3%	96.8%	99.8%	99.99%	100.000%	100.000%	100.000%
Korean	88.6%	30,758	17.5%	70.5%	96.4%	99.8%	99.99%	100.000%	100.000%	100.000%
Latin	67.2%	28,002	31.5%	74.3%	95.7%	99.6%	99.98%	100.000%	100.000%	100.000%
Latvian	90.4%	11,772	13.5%	52.1%	90.7%	98.9%	99.92%	100.000%	100.000%	100.000%
Dutch	88.3%	18,481	23.8%	68.7%	95.3%	99.4%	99.95%	99.989%	100.000%	100.000%
Polish	96.1%	38,882	16.8%	71.4%	96.2%	99.7%	99.98%	100.000%	100.000%	100.000%
Portuguese	87.1%	19,553	6.2%	42.1%	89.1%	99.2%	99.96%	100.000%	100.000%	100.000%
Romanian	93.1%	20,275	3.8%	37.6%	88.2%	99.1%	99.95%	100.000%	100.000%	100.000%
Russian	90.0%	79,657	16.3%	57.4%	91.8%	99.2%	99.95%	99.996%	99.999%	100.000%
Slovenian	86.7%	9,699	25.7%	66.5%	96.5%	99.8%	100.00%	100.000%	100.000%	100.000%
Scandinavian	91.5%	54,890	16.9%	62.6%	95.2%	99.7%	99.98%	100.000%	100.000%	100.000%
Turkish	92.8%	8,753	43.7%	85.2%	97.3%	99.7%	99.98%	100.000%	100.000%	100.000%
Chinese	99.5%	18,544	46.3%	73.9%	92.1%	98.7%	99.90%	99.995%	100.000%	100.000%
others	91.7%	136,951	18.2%	63.1%	93.6%	99.4%	99.96%	99.996%	99.999%	100.000%

is so large that it does not necessarily restrict the performance of state-of-the-art statistical parsers if the gold tree is noncrossing: the finite-state search space contains almost 99.2% of the gold noncrossing trees. The related measure – *unlabeled attachment score (UAS)* – of the best dependency parsers is typically below 98%[3] but these parsers and the used benchmarks are not limited to nonprojective gold trees.

The results prompt further work on statistical explanations of this phenomenon. It would also seem extremely important to try to develop a more general encoding. If a similar depth limit works well for an encoding that covers nonprojective trees, the corresponding search space would become relevant for parser development in the future. There are already some follow-up results suggesting that this is actually the case (Yli-Jyrä 2019).

5.6 *The contrast between theory and data*

There is a striking contrast between the theoretically limited coverage of depth-bounded grammars and their surprisingly good empirical coverage:

- From the *theoretical* point of view, the bounded grammar with 6 levels (Table 6 and Figure 10) is a finite-state approximation that represents a restricted subspace of parses. The theoretical coverage of this subspace drops rapidly below 99% when the sentence length grows beyond 20 token-vertices.
- From the *empirical* point of view (Table 8), six levels of brackets seem to cover more than 99.998% of the noncrossing trees in the actual linguistic data.

The contrast between theory and practice calls for an explanation: we want to know why the limited bracketing depth gives so much better practical coverage than what we would expect from a flat distribution. The first explanation for the high coverage of the noncrossing trees is that *most trees in the data set are short.* We do not know how representative the UD treebanks actually are, as samples, and it is, indeed, perfectly possible that some treebanks are biased towards short sentences. The solid curve in Figure 11 shows how the average

[3] `http://nlpprogress.com/english/dependency_parsing.html`

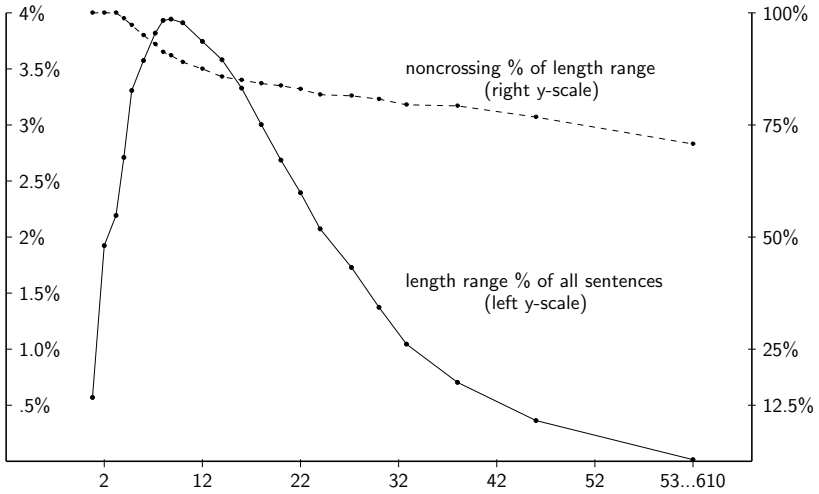


Figure 11: Solid line: the distribution of length ranges; dashed line: the percentage of noncrossing trees per length range in the v2.0 dataset

probability of the length range decreases as a function of sentence length in the data set. The length ranges in this plot were [2,2], …, [9,9], [10,11], …, [24,25], [26, 28], [29, 31], [32, 35], [36, 41], [42, 52], [53, 610], and each of them was represented by the median length, which is 62 for the last length range. The curve roughly indicates that relatively short sentence lengths are more probable than wide length ranges. In fact, although the tail range of lengths is quite long and nonempty, its probability mass is almost invisible in the big picture.

Another explanation for the extremely good coverage of low bracketing depths in Table 8 could be that *longer and more deeply bracketed sentences are more likely to have crossing edges*. The dashed curve in Figure 11 indicates that the percentage of noncrossing parse trees decreases when the sentence length increases. Quickly after the sentence length becomes long enough to have any crossing edges, the probability of noncrossing parses steps down to some 90% on average in the data set. Then, as the sentence length continues to increase further, this probability drops slowly until it goes below 70% of all sentences in the length range that contains the longest sentences in the data set. Thus, the parses of longer sentences are more likely to be excluded from the set of noncrossing trees than the parses of shorter

but more common sentences. As the result, a random sentence in the data set is relatively short and expected to be noncrossing with a high probability (90.5%).

The observation that the distribution of the noncrossing parses contains more shorter sentences does not mean that sentences with crossing edges are otherwise substantially "deeper". In fact, our preliminary experiments on a more general bracketing scheme for all sentences suggest that the bracketing depth of all sentences differs very little from the bracketing depth of noncrossing sentences. Longer sentences may simply be more likely to have complex combinations of edges because they contain more places where a crossing or multiple overlapping can occur. In further work on encoding for unrestricted graphs, it would be possible to test how much crossing edges actually contribute to the local depth of the required bracketing.

The third possible explanation could be based on a psychological model that would predict the tendency to avoid long-distance dependencies (Gibson 1998) and multiple overlapping of edges when the sentence length grows arbitrarily. We could also look for an explanation from bounded memory models (Miller 1956; Kornai and Tuza 1992). With such models, it may be possible to understand why the nesting of superbrackets in the weak bracketing of data is so limited.

The language specific percentages of the noncrossing analyses depend on the choice of the annotation scheme (Havelka 2007). It is very possible that the uniform principles of the UD annotation scheme are not optimal for all languages. But we can perhaps interpret the overall low bracketing depth in the massively multilingual data set as a sign of some kind of cross-lingual uniformity in the complexity scale, which is a surprise because languages differ a lot in their strategies to minimise syntactic complexity.

The v2.0 data set contains seven sentences whose parses require seven levels of superbrackets. In the Appendix, we visualise the dependency structures of these seven noncrossing parses. The first observation from these examples is that their lengths are surprisingly high considering that these sentences are noncrossing: their lengths are between 29 and 106 tokens. This indicates that even long sentences can have noncrossing parses. Secondly, the examples indicate that the new encoding scheme is practically very effective as superbracketed edges have many sibling edges. The weak bracketing scheme divides the

set of edges into two categories, both of which contain a substantial number of overlapping edges. The divided visualisation of noncrossing trees has an advantage that although there are up to 15 overlapping edges and these sentences are pretty long, the paths in the visualised trees are relatively easy to follow from a distance, at a schematic level.[4]

5.7 *On the errors in the data*

It is obvious that treebanks contain a certain number of OCR errors, preprocessing errors and annotation errors. Annotation errors are typically due to the limitations or inconsistencies in the annotation manual or to other human factors that cause inconsistencies and mistakes. Although there is always a reason for annotation errors, we assume that they distribute almost randomly, having non-systematic effects on the depth of the dependency trees.

We had no realistic methods to try to estimate how often annotation errors occur. We just inspected a few most complex trees that we could find and comprehend. In such checking, we found no specific correlation between depth and errors.

6 CONCLUSION

The topic of this paper was to find a regular language that encodes noncrossing dependency graphs in treebanks. Our methodological approach used two different dependency bracketing schemes. The first encoding scheme – *strong bracketing* – has been presented previously and it has been applied to the description of several subfamilies of noncrossing graphs by Yli-Jyrä and Gómez-Rodríguez (2017). This scheme is based on balanced bracketing of edges. It uses three disjoint pairs of brackets to indicate three different orientations of edges. The second scheme – *weak bracketing* – does not properly appear in prior work and it is, therefore, a significant new contribution. In this encoding scheme, sibling edges are encoded with one-sided, weak brackets. We also considered optimisations to both bracketing schemes.

[4] The schematic nature of the Arabic sentence is a gap in the data set used, not a typographical problem.

The main result of this paper is that the new encoding scheme gives rise to a shallower and, in certain sense, less complex balanced bracketing than the previously known encoding scheme.

When we started the current work we did not know if such a shallow approach to dependency bracketing would even be possible and generalizable to noncrossing graphs. Our idea was to reduce the depth of dependency bracketing by omitting brackets when they share the same end of an edge. When this idea was conceived, we did not know if it would bring any practical benefits compared to the first scheme. But the investigation of the idea led to a few important results:

1. **The discovery of a streamlined dependency bracketing**
   In this article, the weak dependency bracketing is presented and evaluated for the first time. Now we know that the scheme exists and corresponds to an unambiguous context-free language (Lemma 4.6), and that it has a deterministic, computable bijection from the set of (di)graphs (Lemma 4.4). This scheme constitutes a unique continuation to the history of ideas that aim at reducing complex balanced bracketing.
2. **A context-free transduction between the two encodings**
   Now we also know that the two bracketing schemes can be related to each other with a context-free (non-deterministic pushdown) transducer (Lemma 4.1). This transducer can be used to convert between the strong and the weak bracketing and to reduce the context-free encodable families of graphs (Yli-Jyrä and Gómez-Rodríguez 2017) to the weak dependency bracketing. This widens the possibilities of both bracketing schemes. Since there is a computable transduction between the strong and weak bracketing, all 50 subfamilies of noncrossing graphs characterized in Yli-Jyrä and Gómez-Rodríguez (2017) can be encoded with context-free languages that describe their weak bracketing. We observe, on page 205, that the search space of projective trees has a smaller state complexity than the noncrossing digraphs, but the state complexity of some specialized search spaces of noncrossing subfamilies may be also slightly higher than the state complexity of the search space that contains all noncrossing graphs.

3. **A low complexity bound with high empirical coverage**
   In dependency bracketing like Yli-Jyrä and Gómez-Rodríguez (2017), bounded-depth bracketing does not make the language finite, but the current work demonstrates that the weak bracketing is still useful because it stabilises the empirically observed depth of dependency bracketing: two levels of superbrackets cover already 58% of the noncrossing trees, three levels cover 92% and five 99.95%. Seven levels of superbrackets give the amazing 99.9998% coverage (with two excluded trees) over the massively multilingual set of dependency treebanks, UD v2.4.

The current work suggests several directions for further developments of the presented framework. We conclude this paper by introducing some of these directions.

6.1 *Fast Parsing and Neural Weighting*

The new empirical bounds open a door to new optimisations towards very efficient dependency parsing of multiple families of noncrossing graphs. An arc-factored, weighted, depth-bounded grammar for the strongly bracketed search space can be constructed in quadratic time. However, it is open whether a similar result is true for weakly bracketed search spaces.

The current work also demonstrated the existence of a high-coverage finite-state representation of a bounded grammar for noncrossing structures. When such a finite-state grammar is matched with a simplified arc weighting model, we would be very close to a linear-time graph-based parsing of bounded families of graphs.

In some state-of-the-art graph-based dependency parsers, the arc weights are computed with neural networks and the statistical inference is based on an algorithm that finds the maximum spanning tree (Libovický 2016; Ma and Hovy 2017) or best path (Rastogi *et al.* 2016). The current work is compatible with such hybrid models. It remains to be seen how the models could then be optimised together and how the search space representation interacts with the training of the weighting model.

Transition-based dependency parsing is mainly based on very expressive transition systems. If the current work could be extended

to nonprojective trees, a finite-state model of the bounded grammar could perhaps be used to handle neural transition systems and to improve nondeterministic strategies in these parsers.

Besides transition-based parsing, there are several other neural network based parsing models to which the current encoding or its unrestricted extension (Yli-Jyrä 2019) could be integrated, as mentioned in the introduction.

6.2 *Generality and Definable Properties of Graphs*

Since the noncrossing graphs have bounded treewidth, it is possible to obtain many efficient algorithms for them. Especially, there is an algorithmic metatheorem (Courcelle 1990) that states that any graph property in monadic second order logic (MSO) can be decided in linear time for bounded-treewidth graphs. Yli-Jyrä and Gómez-Rodríguez (2017) can be seen as a start for a research that reconstructs this metatheorem via dependency bracketing and context-free grammars in the case of noncrossing graphs. It is, indeed, possible to create an algorithm library that implements MSO logic for noncrossing graphs, using the currently explored encoding schemes.

The currently presented encoding is a crucial step towards more comprehensive bracket-based encoding of graphs. It is possible to develop similar encoding schemes for unrestricted ordered graphs. Indeed, we have already worked on an encoding that generalises elegantly to all ordered graphs. The description of the generalised encoding will appear separately (Yli-Jyrä 2019).

6.3 *Learnability of subregular approximations of syntax*

By showing that the positive examples in the training data have a robust bound for the depth of bracketing in the context-free encoding, the dependency structures can be seen as a regular language, with a truncated Chomsky-Schützenberger representation. It has been previously observed that such regular languages are often star-free (Yli-Jyrä 2003a, 2005a,b), but their descriptive complexity depends on the bracketing depth (Yli-Jyrä 2008, 2005c). Thus, they do not belong to any of the basic subregular classes of languages that have been shown to be learnable from positive data (Heinz and Rogers 2013). From the structure of languages in Yli-Jyrä and Gómez-Rodríguez (2017), we can infer that learning non-local properties of noncrossing graphs also

requires learning latent labeling of their bracketing. These challenges put a strain on the research on such subregular language classes that would allow us to learn finite-state approximations of syntax from treebanks. This research could be related to representation learning in neural networks.

ACKNOWLEDGEMENTS

The author has received funding from the Academy of Finland (dec. No 270354/273457/313478 – A Usable Finite-State Model for Adequate Syntactic Complexity) and the Clare Hall Fellowship from the University of Helsinki (dec. RP 137/2013 – SMT based on D-Tree Contraction Grammars and FSTs). The author is also grateful to Kimmo Koskenniemi, Lauri Carlson, Atro Voutilainen, Solomon Marcus, Filip Ginter, Tapio Salakoski, Steven Krauwer, Terence Langendoen, Fred Karlsson, Kemal Oflazer, Mans Hulden, András Kornai, Krister Lindén, James Rogers, Bill Byrne, Gonzalo Iglesias, Mark-Jan Nederhof, Jussi Piitulainen, Alexander Okhotin, Carlos Gómez-Rodríguez, Juha Kontinen, Andreas Maletti, Brian Roark, Jonathan May, Joel Mathew, Ronald Kaplan, Lauri Karttunen, Edward Gibson, Henrik Björklund, Adam Jardine, Galina Jirásková, Elin Ehsani, Eli Shamir, Gary Miles, and Tatu Ylönen for helpful discussions during earlier stages of the research, and to the anonymous reviewers for their valuable feedback on the earlier versions of this paper.

APPENDIX: HIGH-LEVEL VISUALISATION OF THE SEVEN SENTENCES WITH SEVEN LEVELS OF SUPERBRACKETED EDGES IN THE UD V2.0 DATA SET

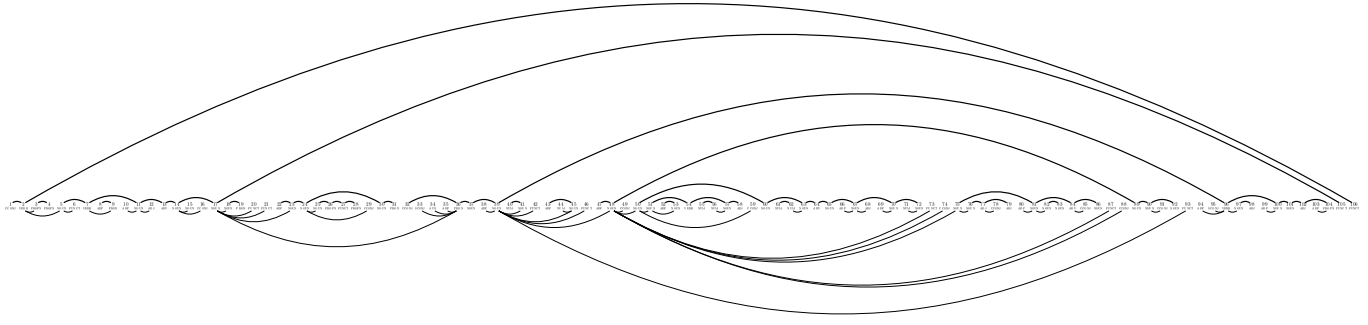


Figure12: Arabic (string of part-of-speech tags)

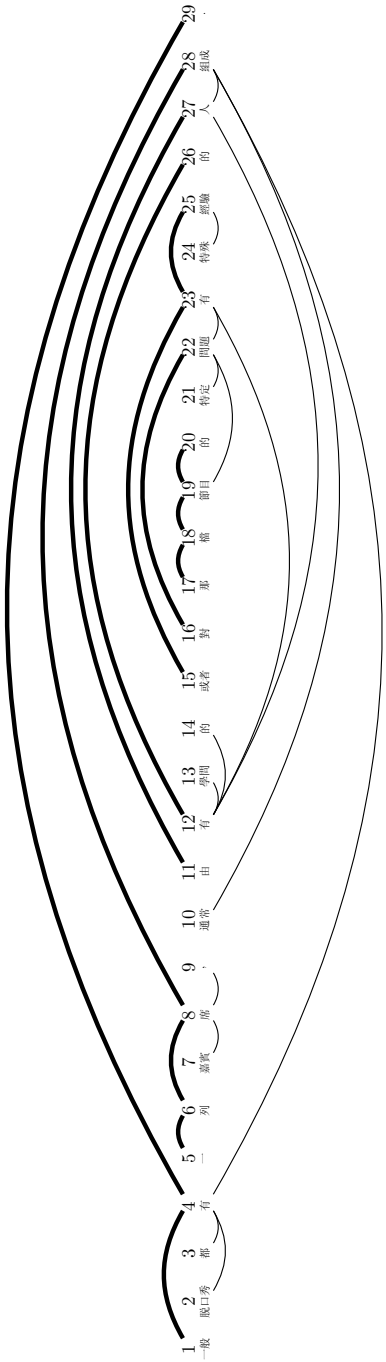


Figure 13: Chinese

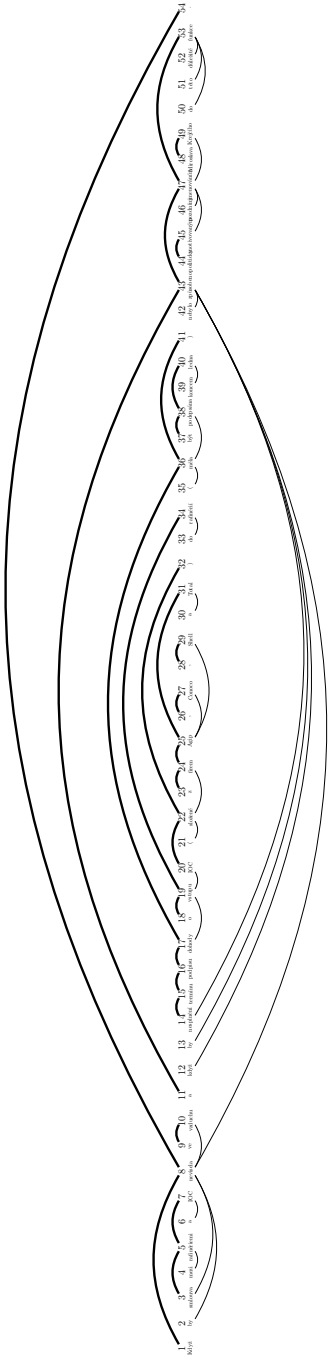


Figure 14: Czech

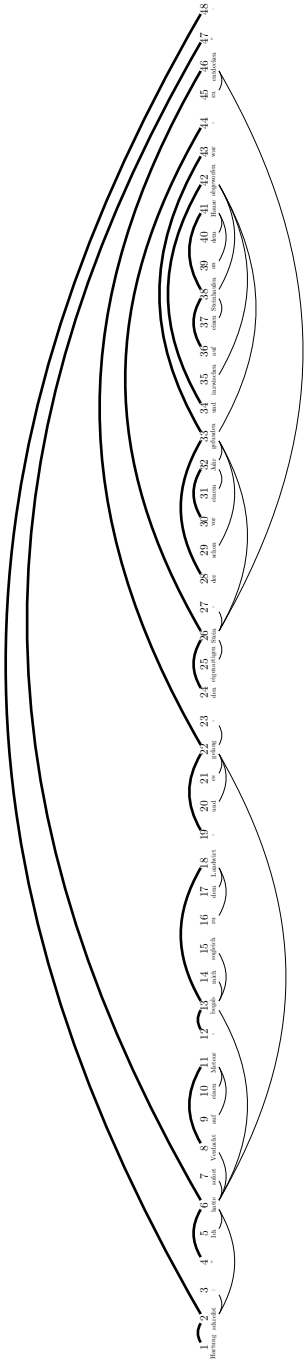


Figure 15: German

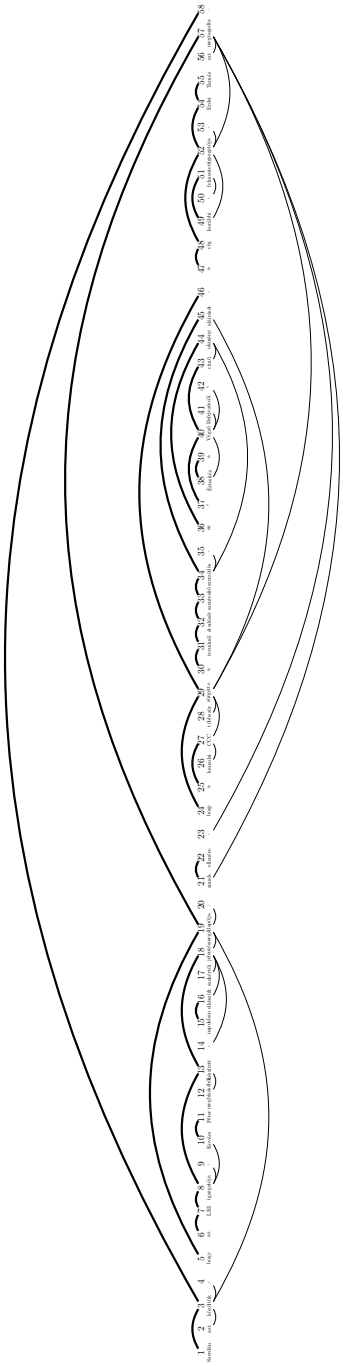


Figure 16: Hungarian

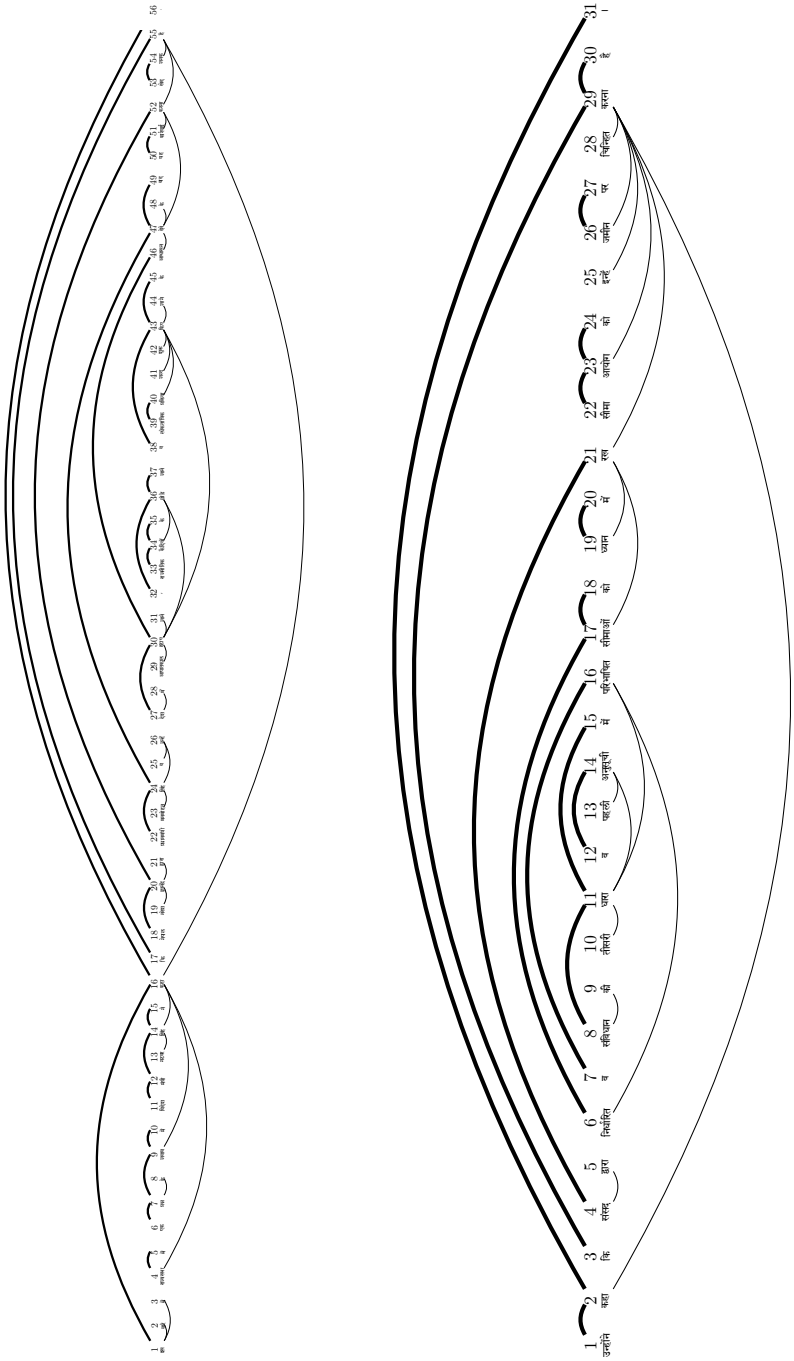


Figure 17: Hindi

REFERENCES

Bernd BOHNET, Ryan MCDONALD, Emily PITLER, and Ji MA (2016), Generalized transition-based dependency parsing via control parameters, in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL 2016)*, pp. 150–160, Association for Computational Linguistics, http://dx.doi.org/10.18653/v1/P16-1015.

Mikołaj BOJAŃCZYK and Michał PILIPCZUK (2016), Definability equals recognizability for graphs of bounded treewidth, in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pp. 407–416, ACM, New York, NY, USA, ISBN 978-1-4503-4391-6, http://dx.doi.org/10.1145/2933575.2934508.

Noam CHOMSKY (1963), Formal properties of grammars, in R. Duncan LUCE, Robert R. BUSH, and Eugene GALANTER, editors, *Handbook of mathematical psychology*, volume 2, pp. 323–418, John Wiley and Sons, New York.

Bruno COURCELLE (1990), The monadic second-order logic of graphs. I. Recognizable sets of finite graphs, *Information and Computation*, 85(1):12 – 75, ISSN 0890-5401, doi:https://doi.org/10.1016/0890-5401(90)90043-H, http://www.sciencedirect.com/science/article/pii/089054019090043H.

Michael A. COVINGTON (2001), A fundamental algorithm for dependency parsing, in John A. MILLER and Jeffrey W. SMITH, editors, *Proceedings of the 39th Annual ACM Southeast Conference*, pp. 95–102, Association for Computing Machinery, http://www.covingtoninnovations.com/mc/dgpacm.pdf.

Chris DYER, Miguel BALLESTEROS, Wang LING, Austin MATTHEWS, and Noah A. SMITH (2015), Transition-based dependency parsing with Stack Long Short-Term Memory, in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 334–343, Association for Computational Linguistics, Beijing, China, http://dx.doi.org/10.3115/v1/P15-1033.

Chris DYER, Adhiguna KUNCORO, Miguel BALLESTEROS, and Noah A. SMITH (2016), Recurrent neural network grammars, in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 199–209, Association for Computational Linguistics, http://www.aclweb.org/anthology/N16-1024.

Jason EISNER and Giorgio SATTA (1999), Efficient parsing for bilexical context-free grammars and Head Automaton Grammars, in *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL 1999)*, pp. 457–464, Association for Computational Linguistics, https://doi.org/10.3115/1034678.1034748.

Edward GIBSON (1998), Linguistic complexity: locality of syntactic dependencies, *Cognition*, 68(1):1–76, http://doi.org/10.1016/S0010-0277(98)00034-1.

Filip GINTER, Sampo PYYSALO, Jorma BOBERG, and Tapio SALAKOSKI (2006), Regular approximation of Link grammar, in *Proceedings of the 5th International Conference on Advances in Natural Language Processing*, FinTAL'06, pp. 564–575, Springer-Verlag, Berlin, Heidelberg, ISBN 3-540-37334-9, 978-3-540-37334-6, http://dx.doi.org/10.1007/11816508_56.

Charles F. GOLDFARB (1999), Future directions in SGML/XML, in Wiebke MÖHR and Ingrid SCHMIDT, editors, *SGML und XML: Anwendungen und Perspektiven*, pp. 3–25, Springer, Berlin, Germany, https://doi.org/10.1007/978-3-642-46881-0_1.

Carlos GÓMEZ-RODRÍGUEZ and Joakim NIVRE (2010), A transition-based parser for 2-planar dependency structures, in *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL 2010)*, pp. 1492–1501, Association for Computational Linguistics, http://dl.acm.org/citation.cfm?id=1858681.1858832.

Sheila GREIBACH (1973), The hardest context-free language, *SIAM Journal on Computing*, 2(4):304–310, http://dx.doi.org/10.1137/0202025.

Jiří HAVELKA (2007), Beyond projectivity: multilingual evaluation of constraints and measures on non-projective structures, in *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics (ACL 2007)*, pp. 608–615, Association for Computational Linguistics, http://www.aclweb.org/anthology/P07-1077.

Jeffrey HEINZ and James ROGERS (2013), Learning subregular classes of languages with factored deterministic automata, in *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, pp. 64–71, Association for Computational Linguistics, Sofia, Bulgaria, https://www.aclweb.org/anthology/W13-3007.

Mans HULDEN and Miikka SILFVERBERG (2014), Finite-state subset approximation of phrase structure, in *Proceedings of the International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, USA, https://www.cs.uic.edu/pub/Isaim2014/WebPreferences/ISAIM2014_NLP_Hulden_Silfverberg.pdf.

Eliyahu KIPERWASSER and Yoav GOLDBERG (2016), Simple and accurate dependency parsing using bidirectional LSTM feature representations, *Transactions of the Association for Computational Linguistics*, 4:313–327, http://dx.doi.org/10.1162/tacl_a_00101.

András KORNAI and Zsolt TUZA (1992), Narrowness, path-width, and their application in natural language processing, *Discrete Applied Mathematics*, 36(1):87–92, https://doi.org/10.1016/0166-218X(92)90208-R.

Kimmo KOSKENNIEMI (1990), Finite-state parsing and disambiguation, in Hans KARLGREN, editor, *13th International Conference on Computational Linguistics, (COLING 1990)*, pp. 229–232, http://aclweb.org/anthology/C90-2040.

Steven KRAUWER and Louis DES TOMBE (1981), Transducers and grammars as theories of language, *Theoretical Linguistics*, 8(1–3):173–202, https://doi.org/10.1515/thli.1981.8.1-3.173.

Marco KUHLMANN (2015), Tabulation of noncrossing acyclic digraphs, arXiv:1504.04993, https://arxiv.org/abs/1504.04993.

Marco KUHLMANN and Peter JOHNSSON (2015), Parsing to noncrossing dependency graphs, *Transactions of the Association for Computational Linguistics*, 3:559–570, http://aclweb.org/anthology/Q/Q15/Q15-1040.pdf.

Adhiguna KUNCORO, Miguel BALLESTEROS, Lingpeng KONG, Chris DYER, Graham NEUBIG, and Noah A. SMITH (2017), What do recurrent neural network grammars learn about syntax?, in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2017)*, pp. 1249–1258, Association for Computational Linguistics, http://aclweb.org/anthology/E17-1117.

D. Terence LANGENDOEN (1975), Finite-state parsing of phrase-structure languages and the status of readjustment rules in grammar, *Linguistic Inquiry*, 6(4):533–554, http://www.jstor.org/stable/4177899.

D. Terence LANGENDOEN and Yedidyah LANGSAM (1984), The representation Of constituent structures for finite-state parsing, in Yorick WILKS, editor, *10th International Conference on Computational Linguistics and 22nd Annual Meeting of the Association for Computational Linguistics, Proceedings of COLING '84, July 2-6, 1984, Stanford University, California, USA.*, pp. 24–27, ACL, http://aclweb.org/anthology/P84-1007.

Jindřich LIBOVICKÝ (2016), Neural scoring function for MST parser, in Nicoletta CALZOLARI, Khalid CHOUKRI, Thierry DECLERCK, Sara GOGGI, Marko GROBELNIK, Bente MAEGAARD, Joseph MARIANI, Helene MAZO, Asuncion MORENO, Jan ODIJK, and Stelios PIPERIDIS, editors, *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, European Language Resources Association (ELRA), http://www.lrec-conf.org/proceedings/lrec2016/summaries/649.html.

Xuezhe MA and Eduard H. HOVY (2017), Neural probabilistic model for non-projective MST parsing, in *Proceedings of the The 8th International Joint Conference on Natural Language Processing (IJCNLP)*, pp. 59–69, Asian Federation of Natural Language Processing, http://www.aclweb.org/anthology/I17-1007.

Ryan MCDONALD, Fernando PEREIRA, Kiril RIBAROV, and Jan HAJIČ (2005), Non-projective dependency parsing using spanning tree algorithms, in *Proceedings of Human Language Technology Conference and Conference on*

Empirical Methods in Natural Language Processing, pp. 523–530, Association for Computational Linguistics, http://www.aclweb.org/anthology/H/H05/H05-1066.pdf.

George A. Miller (1956), The magical number seven, plus or minus two: some limits on our capacity for processing information, *Psychological Review*, 63(2):81–97, http://dx.doi.org/10.1037/h0043158.

Mehryar Mohri (1997), Finite-state transducers in language and speech processing, *Computational Linguistics*, 23(2):1–42, http://www.aclweb.org/anthology/J97-2003.

Mark-Jan Nederhof (2000), Practical experiments with regular approximation of context-free languages, *Computational Linguistics*, 26(1):17–44, http://dx.doi.org/10.1162/089120100561610.

Joakim Nivre (2008), Algorithms for deterministic incremental dependency parsing, *Computational Linguistics*, 34(4):513–553, https://doi.org/10.1162/coli.07-056-R1-07-027.

Joakim Nivre (2009), Non-projective dependency parsing in expected linear time, in *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing*, pp. 351–359, Association for Computational Linguistics, http://dl.acm.org/citation.cfm?id=1687878.1687929.

Joakim Nivre, Mitchell Abrams, Željko Agić, Lars Ahrenberg, Gabrielė Aleksandravičiūtė, Lene Antonsen, Katya Aplonova, Maria Jesus Aranzabe, Gashaw Arutie, Masayuki Asahara, Luma Ateyah, Mohammed Attia, Aitziber Atutxa, Liesbeth Augustinus, Elena Badmaeva, Miguel Ballesteros, Esha Banerjee, Sebastian Bank, Verginica Barbu Mititelu, Victoria Basmov, John Bauer, Sandra Bellato, Kepa Bengoetxea, Yevgeni Berzak, Irshad Ahmad Bhat, Riyaz Ahmad Bhat, Erica Biagetti, Eckhard Bick, Agnė Bielinskienė, Rogier Blokland, Victoria Bobicev, Loïc Boizou, Emanuel Borges Völker, Carl Börstell, Cristina Bosco, Gosse Bouma, Sam Bowman, Adriane Boyd, Kristina Brokaitė, Aljoscha Burchardt, Marie Candito, Bernard Caron, Gauthier Caron, Gülşen Cebiroğlu Eryiğit, Flavio Massimiliano Cecchini, Giuseppe G. A. Celano, Slavomír Čéplö, Savas Cetin, Fabricio Chalub, Jinho Choi, Yongseok Cho, Jayeol Chun, Silvie Cinková, Aurélie Collomb, Çağrı Çöltekin, Miriam Connor, Marine Courtin, Elizabeth Davidson, Marie-Catherine de Marneffe, Valeria de Paiva, Arantza Diaz de Ilarraza, Carly Dickerson, Bamba Dione, Peter Dirix, Kaja Dobrovoljc, Timothy Dozat, Kira Droganova, Puneet Dwivedi, Hanne Eckhoff, Marhaba Eli, Ali Elkahky, Binyam Ephrem, Tomaž Erjavec, Aline Etienne, Richárd Farkas, Hector Fernandez Alcalde, Jennifer Foster, Cláudia Freitas, Kazunori Fujita, Katarína Gajdošová, Daniel Galbraith, Marcos Garcia,

Moa Gärdenfors, Sebastian Garza, Kim Gerdes, Filip Ginter, Iakes Goenaga, Koldo Gojenola, Memduh Gökirmak, Yoav Goldberg, Xavier Gómez Guinovart, Berta González Saavedra, Matias Grioni, Normunds Grūzītis, Bruno Guillaume, Céline Guillot-Barbance, Nizar Habash, Jan Hajič, Jan Hajič jr., Linh Hà Mỹ, Na-Rae Han, Kim Harris, Dag Haug, Johannes Heinecke, Felix Hennig, Barbora Hladká, Jaroslava Hlaváčová, Florinel Hociung, Petter Hohle, Jena Hwang, Takumi Ikeda, Radu Ion, Elena Irimia, Ọlájídé Ishola, Tomáš Jelínek, Anders Johannsen, Fredrik Jørgensen, Hüner Kaşıkara, Andre Kaasen, Sylvain Kahane, Hiroshi Kanayama, Jenna Kanerva, Boris Katz, Tolga Kayadelen, Jessica Kenney, Václava Kettnerová, Jesse Kirchner, Arne Köhn, Kamil Kopacewicz, Natalia Kotsyba, Jolanta Kovalevskaitė, Simon Krek, Sookyoung Kwak, Veronika Laippala, Lorenzo Lambertino, Lucia Lam, Tatiana Lando, Septina Dian Larasati, Alexei Lavrentiev, John Lee, Phương Lê Hồng, Alessandro Lenci, Saran Lertpradit, Herman Leung, Cheuk Ying Li, Josie Li, Keying Li, KyungTae Lim, Yuan Li, Nikola Ljubešić, Olga Loginova, Olga Lyashevskaya, Teresa Lynn, Vivien Macketanz, Aibek Makazhanov, Michael Mandl, Christopher Manning, Ruli Manurung, Cătălina Mărănduc, David Mareček, Katrin Marheinecke, Héctor Martínez Alonso, André Martins, Jan Mašek, Yuji Matsumoto, Ryan McDonald, Sarah McGuinness, Gustavo Mendonça, Niko Miekka, Margarita Misirpashayeva, Anna Missilä, Cătălin Mititelu, Yusuke Miyao, Simonetta Montemagni, Amir More, Laura Moreno Romero, Keiko Sophie Mori, Tomohiko Morioka, Shinsuke Mori, Shigeki Moro, Bjartur Mortensen, Bohdan Moskalevskyi, Kadri Muischnek, Yugo Murawaki, Kaili Müürisep, Pinkey Nainwani, Juan Ignacio Navarro Horñiacek, Anna Nedoluzhko, Gunta Nešpore-Bērzkalne, Lương Nguyễn Thị, Huyền Nguyễn Thị Minh, Yoshihiro Nikaido, Vitaly Nikolaev, Rattima Nitisaroj, Hanna Nurmi, Stina Ojala, Adédayọ̀ Olúòkun, Mai Omura, Petya Osenova, Robert Östling, Lilja Øvrelid, Niko Partanen, Elena Pascual, Marco Passarotti, Agnieszka Patejuk, Guilherme Paulino-Passos, Angelika Peljak-Łapińska, Siyao Peng, Cenel-Augusto Perez, Guy Perrier, Daria Petrova, Slav Petrov, Jussi Piitulainen, Tommi A Pirinen, Emily Pitler, Barbara Plank, Thierry Poibeau, Martin Popel, Lauma Pretkalniņa, Sophie Prévost, Prokopis Prokopidis, Adam Przepiórkowski, Tiina Puolakainen, Sampo Pyysalo, Andriela Rääbis, Alexandre Rademaker, Loganathan Ramasamy, Taraka Rama, Carlos Ramisch, Vinit Ravishankar, Livy Real, Siva Reddy, Georg Rehm, Michael Rießler, Erika Rimkutė, Larissa Rinaldi, Laura Rituma, Luisa Rocha, Mykhailo Romanenko, Rudolf Rosa, Davide Rovati, Valentin Roșca, Olga Rudina, Jack Rueter, Shoval Sadde, Benoît Sagot, Shadi Saleh, Alessio Salomoni, Tanja Samardžić, Stephanie Samson, Manuela Sanguinetti, Dage Särg, Baiba Saulīte, Yanin Sawanakunanon, Nathan

Schneider, Sebastian Schuster, Djamé Seddah, Wolfgang Seeker, Mojgan Seraji, Mo Shen, Atsuko Shimada, Hiroyuki Shirasu, Muh Shohibussirri, Dmitry Sichinava, Natalia Silveira, Maria Simi, Radu Simionescu, Katalin Simkó, Mária Šimková, Kiril Simov, Aaron Smith, Isabela Soares-Bastos, Carolyn Spadine, Antonio Stella, Milan Straka, Jana Strnadová, Alane Suhr, Umut Sulubacak, Shingo Suzuki, Zsolt Szántó, Dima Taji, Yuta Takahashi, Fabio Tamburini, Takaaki Tanaka, Isabelle Tellier, Guillaume Thomas, Liisi Torga, Trond Trosterud, Anna Trukhina, Reut Tsarfaty, Francis Tyers, Sumire Uematsu, Zdeňka Urešová, Larraitz Uria, Hans Uszkoreit, Sowmya Vajjala, Daniel van Niekerk, Gertjan van Noord, Viktor Varga, Eric Villemonte de la Clergerie, Veronika Vincze, Lars Wallin, Abigail Walsh, Jing Xian Wang, Jonathan North Washington, Maximilan Wendt, Seyi Williams, Mats Wirén, Christian Wittern, Tsegay Woldemariam, Tak-sum Wong, Alina Wróblewska, Mary Yako, Naoki Yamazaki, Chunxiao Yan, Koichi Yasuoka, Marat M. Yavrumyan, Zhuoran Yu, Zdeněk Žabokrtský, Amir Zeldes, Daniel Zeman, Manying Zhang, and Hanzhi Zhu (2019), Universal Dependencies 2.4, `http://hdl.handle.net/11234/1-2988`, LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Kemal Oflazer (2003), Dependency parsing with an extended finite-state approach, *Computational Linguistics*, 29(4):515–544, `http://dx.doi.org/10.1162/089120103322753338`.

Pushpendre Rastogi, Ryan Cotterell, and Jason Eisner (2016), Weighting finite-state transductions with neural context, in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 623–633, Association for Computational Linguistics, `http://dx.doi.org/10.18653/v1/N16-1076`.

Brian Roark and Kristy Hollingshead (2008), Classifying chart cells for quadratic complexity context-free inference, in *Proceedings of the 22nd International Conference on Computational Linguistics (COLING 2008)*, volume 1, pp. 745–752, Association for Computational Linguistics, `http://dl.acm.org/citation.cfm?id=1599081.1599175`.

Arto Salomaa (1973), *Formal languages*, Academic Press, New York, NY.

Michalina Strzyz, David Vilares, and Carlos Gómez-Rodríguez (2019), Viable dependency parsing as sequence labeling, in Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pp. 717–723, Association for Computational Linguistics, ISBN 978-1-950737-13-0, `https://aclweb.org/anthology/papers/N/N19/N19-1077/`.

Warren TEITELMAN (1978), *INTERLISP reference manual*, Xerox Palo Alto Research Center (PARC), Palo Alto, CA.

Oriol VINYALS, Lukasz KAISER, Terry KOO, Slav PETROV, Ilya SUTSKEVER, and Geoffrey E. HINTON (2015), Grammar as a foreign language, in Corinna CORTES, Neil D. LAWRENCE, Daniel D. LEE, Masashi SUGIYAMA, and Roman GARNETT, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pp. 2773–2781, http://papers.nips.cc/paper/5635-grammar-as-a-foreign-language.

Anssi YLI-JYRÄ (2003a), Describing syntax with star-free regular expressions, in *10th Conference of the European Chapter of the Association for Computational Linguistics*, Association for Computational Linguistics, Budapest, Hungary, https://www.aclweb.org/anthology/E03-1031.

Anssi YLI-JYRÄ (2003b), Multiplanarity – a model for dependency structures in treebanks, in Joakim NIVRE and Erhard HINRICHS, editors, *TLT 2003. Proceedings of the Second Workshop on Treebanks and Linguistic Theories*, volume 9 of *Mathematical Modelling in Physics, Engineering and Cognitive Sciences*, pp. 189–200, Växjö University Press, Växjö, Sweden, http://hdl.handle.net/10138/33872.

Anssi YLI-JYRÄ (2003c), Regular approximations through labeled bracketing, in Gerald PENN, editor, *Proceedings of FGVienna: the 8th Conference on Formal Grammar*, pp. 153–166, CSLI Publications, Stanford, CA, http://cslipublications.stanford.edu/FG/2003/ylijyra.pdf.

Anssi YLI-JYRÄ (2004), Axiomatization of restricted non-projective dependency trees through finite-state constraints that analyse crossing bracketings, in Geert-Jan M. KRUIJFF and Denys DUCHIER, editors, *Recent Advances in Dependency Grammar (COLING 2004 workshop)*, pp. 25–32, Association for Computational Linguistics, Geneva, Switzerland, https://www.aclweb.org/anthology/W/W04/W04-1504.pdf.

Anssi YLI-JYRÄ (2005a), Approximating dependency grammars through intersection of star-free regular languages, *International Journal of Foundations of Computer Science*, 16(3):565–579, http://dx.doi.org/10.1142/S0129054105003169.

Anssi YLI-JYRÄ (2005b), *Contributions to the theory of finite-state based grammars*, Ph.D. thesis, University of Helsinki, Faculty of Arts, Department of General Linguistics, http://urn.fi/URN:ISBN:952-10-2510-7.

Anssi YLI-JYRÄ (2005c), Linguistic grammars with very low complexity, in Antti Arppe *et al.*, editor, *Inquiries into Words, Constraints and Contexts: Festschrift in the Honour of Kimmo Koskenniemi on his 60th Birthday*, CSLI Studies in Computational Linguistics ONLINE, pp. 172–183, CSLI Publications, Stanford, CA, https://web.stanford.edu/group/cslipublications/cslipublications/koskenniemi-festschrift/17-yli-jyra.pdf.

Anssi YLI-JYRÄ (2008), Applications of diamonded double negation, in T. HANNEFORTH and K-M. WÜRZNER, editors, *Finite-State Methods and Natural Language Processing, 6th International Workshop, FSMNL-2007, Potsdam, Germany, September 14–16, Revised Papers*, pp. 6–30, Potsdam University Press, Potsdam, https://publishup.uni-potsdam.de/opus4-ubp/frontdoor/index/index/year/2008/docId/2519.

Anssi YLI-JYRÄ (2012), On dependency analysis via contractions and weighted FSTs, in Diana SANTOS, Krister LINDÉN, and Wanjiku NG'ANG'A, editors, *Shall we play the Festschrift game? Essays on the occasion of Lauri Carlson's 60th birthday*, pp. 133–158, Springer, Berlin, Germany, http://dx.doi.org/10.1007/978-3-642-30773-7_10.

Anssi YLI-JYRÄ (2019), Transition-based coding and formal language theory for ordered digraphs, Paper accepted to FSMNLP 2019, The 14th International Conference on Finite-State Methods and Natural Language Processing, September 23-25, Dresden, Germany.

Anssi YLI-JYRÄ and Carlos GÓMEZ-RODRÍGUEZ (2017), Generic axiomatization of families of noncrossing graphs in dependency parsing, in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL 2017)*, pp. 1745–1755, Association for Computational Linguistics, http://aclweb.org/anthology/P17-1160.

Anssi YLI-JYRÄ and Matti NYKÄNEN (2014), A hierarchy of mildly context-sensitive dependency grammars, in Paola MONACHESI, editor, *Proceedings of the Nancy Conference, Formal Grammar 2004*, pp. 155–170, CSLI Publications, Stanford, CA, http://cslipublications.stanford.edu/FG/2004/fg-2004-paper11.pdf.

Hao ZHANG and Ryan T. MCDONALD (2014), Enforcing structural diversity in cube-pruned dependency parsing, in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL 2014)*, pp. 656–661, Association for Computational Linguistics, http://aclweb.org/anthology/P/P14/P14-2107.pdf.

Xiaoqing ZHENG (2017), Incremental graph-based neural dependency parsing, in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 1655–1665, Association for Computational Linguistics, http://dx.doi.org/10.18653/v1/D17-1173.