On Dependency Analysis via Contractions and Weighted FSTs

Anssi Yli-Jyrä

Abstract Arc contractions in syntactic dependency graphs can be used to decide which graphs are trees. The paper observes that these contractions can be expressed with weighted finite-state transducers (weighted FST) that operate on string-encoded trees. The observation gives rise to a finite-state parsing algorithm that computes the parse forest and extracts the best parses from it. The algorithm is customizable to functional and bilexical dependency parsing, and it can be extended to non-projective parsing via a multi-planar encoding with prior results on high recall. Our experiments support an analysis of projective parsing according to which the worst-case time complexity of the algorithm is *quadratic* to the sentence length, and *linear* to the overlapping arcs and the number of functional categories of the arcs. The results suggest several interesting directions towards efficient and high-precision dependency parsing that takes advantage of the flexibility and the demonstrated ambiguity-packing capacity of such a parser.

1 Introduction

Finite-state transducers (FSTs) – and their underlying string relations – specify elegant but general parsing algorithms. In this contribution, the methodology of weighted FSTs is applied to efficient dependency grammar verification and search for the globally best parse in a dependency-based forest. The solution would not be as practical without memoizing composition and a simple implementation of arc contractions in dependency analyses, which are perhaps the most original aspects of this work.

Anssi Yli-Jyrä
Department of Modern Languages, University of Helsinki, Finland,
e-mail: anssi.yli-jyra@helsinki.fi

Dependency grammar (Tesnière 1959) is typically implemented in computational linguistics by parsing algorithms that compromise between efficiency and the linguistic accuracy in different ways:

- Many practical dependency parsers are based on *deterministic parsing algorithms* (Nivre 2008) that can produce all kinds of trees but depend on heuristics that may not always find the globally optimal parse. This compromises the accuracy or recall of the analysis, but yields practically fast parsers.
- *Projective dependency parsing* gives higher accuracy because the globally optimal parses among all the projective parses can be found. The time complexity of projective parsers is comparable with context-free parsers: for the sentences of length n, it is in $O(n^3)$ in the case of functional and bilexical dependency grammars (Lombardo and Lesmo 1996, Eisner 1997). However, the projectivity condition for the parses restricts the admissible analyses to the subset of dependency trees that do not contain dependencies that cross one other in the drawings of the trees. The condition is fully explained in Yli-Jyrä (2005) and Kuhlmann (2010).
- *Non-projective dependency parsers* relax the projectivity condition by allowing crossing dependencies. The admissible parses thus include all possible dependency trees, some of which are non-projective. Non-projective trees are common in treebanks for major European languages (Kuhlmann 2010). However, unrestricted non-projective parsing is intractable (Neuhaus and Bröker 1997).
- *Parameterized non-projective dependency grammars* (Yli-Jyrä and Nykänen 2004, Nivre 2006, Kuhlmann 2010) have been proposed in order to address the precision, recall and efficiency considerations. For example, well-nested dependency trees with bounded gap-degree can be parsed in polynomial time (Gómez-Rodríguez et al. 2009, Kuhlmann and Satta 2009). The time complexity is in $O(n^7)$ – quite much in comparison to deterministic parsing. More research is thus needed in order to make parameterized non-projective parsers practical.

My objective is to describe a practical parsing algorithm (in fact a family of algorithms) that takes advantage of partial projectivity and a performance-motivated parameter, t, for overlapping dependencies. In the case of unrooted projective trees with a fixed bound for t, the current analysis of the worst-case time complexity of the final algorithm (in Sect. 5.4) is based on evidence of $O(n^2)$ space and $O(n^2)$ time complexities that are measured using an efficient finite-state library. I am arguing that the approach is extendible towards non-projective parsing (then the actual complexity bound depends on subtle properties of the grammar, being, in the worst case, exponential to the number of dependencies that overlap but are not nested).

My secondary aim is to demonstrate the relevance of string-based finite-state methods in packing and processing syntactic parse forests. Therefore, I will describe the algorithm using weighted rational relations whose operations can be translated to operations on weighted finite-state transducers. In order to process trees with string automata, the work develops three new techniques:

1. The first new technique is to check the treeness of dependency graphs via *arc contractions*. Under the contractions, the notions of the bottom and the top in

the reduction tree are no more based on the dominance or precedence relations but on the minorization relation.[1] Contractions exhibit also a surprising ability to make the crossing dependencies local.

2. The second technique is the *dependency bracketing* for various kinds of dependency trees. Dependency bracketing with assigned planes is expressive enough for non-projective parsing and it admits finite-state processing of contractions in dependency forests. Dependency bracketing can be customized for functional and bilexical dependency grammars.
3. The third technique is the *memoization* of intermediate results in finite-state cascades. Memoization allows for combining bottom-up and top-down filtering in order to extract only the best parses. Memoization "tabulates" the found arcs via epsilon removal after contractions in finite-state transducers.

The resulting method can be seen as an upgrade to a constraint-based dependency parsing method (Yli-Jyrä 2005) that I developed during my Ph.D. studies.[2] The predecessor was based on a conjunctive decomposition of finite-state constraints that restricted the set of lexicon-generated candidate parses. The new algorithm does not only layerize the constraints (Oflazer 2003, Yli-Jyrä 2004, Yli-Jyrä and Koskenniemi 2004) but it also packs the local ambiguity and shares the subtrees.

2 The Input and Output Representations

In this section, I describe the graph representations manipulated by the parser and give the general principles for validating dependency trees.

2.1 Functional Dependency Parsing

In terms of the outcome, the task of the dependency parser is to take a tokenized orthographical string, such as (1a) and annotate it with one or more dependency trees[3] as in (1b).

---

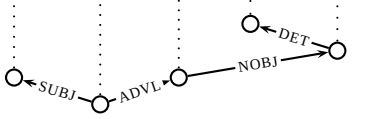
[1] It would be interesting to study how the minorization relation compares with the derivation relation of tree adjoining grammars. In both cases, the derived tree is manipulated from inside.

[2] This article is published on the occasion of Professor Lauri Carlson's birthday. As he co-supervised my Ph.D. research together with Kimmo Koskenniemi a decade ago, it is now a great privilege for me to write about these new advances in the research area where we started together.

[3] The tree is drawn with the `xdag.sty` package written by Denys Duchier, Ralph Debusmann and Robert Grabowski. For convenience, the orientation of the tree is flipped in the context of the linguistic example that is typeset with `expex.sty`.

(1) a. ⟨*time* *flies* *like* *an* *arrow*⟩

b. ⟨*time* *flies* *like* *an* *arrow*⟩



The dependency trees of this work visualize *syntactic* dependency relations, in contrast to deeper, semantic dependencies. A dependency is a link between a *dependent* word (Tesnière 1959: "subordonné") and its *head* (Tesnière 1959: "régissant"). By convention, the arrowhead of each arc in the tree points, in this paper, to the dependent node. The arc has a label that indicates what syntactic function is played by the dependent word under the head. For example, the word *an* in (1) is a determiner (DET) for the word *arrow*. Since it is quite common to add several uncoordinated modifier words as dependents, the syntactic functions should not be confused with mathematical functions.

The finite-state parser will encode the actual drawing via special markup that is associated with the input string, as in the example (2).

(2) ⟨*time* *flies* *like* *an* *arrow*⟩

<SUBJ # SUBJ\/ADVL # ADVL> /NOBJ # <DET # DET\NOBJ>

The markup is based on *balanced dependency bracketing* (Table 1) that gives information on the dependency orientation, the syntactic function, and the lexeme. The brackets are viewed as tags that annotate the tokenized string. The order of the tags under each token mirrors the proximity of the connections in order to ensure that nested brackets match neatly, and different kinds of brackets are chosen for different kinds of trees, as demonstrated later in this paper.

Table 1 The dependency brackets

Left bracket	Right bracket	Head	Corresponding arc	Arc label
<SUBJ	SUBJ\	on the right	functional arc	SUBJ
/OBJ	OBJ>	on the left	functional arc	OBJ
(SUBJ	SUBJ)	(by convention)	undirected functional arc	SUBJ
(OBJ	OBJ)	(by convention)	undirected functional arc	OBJ
time<	\flies	on the right	bilexical arc	–
time/	>flies	on the left	bilexical arc	–
time(	)flies	(not specified)	undirected bilexical arc	–
an(	)arrow	(not specified)	undirected bilexical arc	–

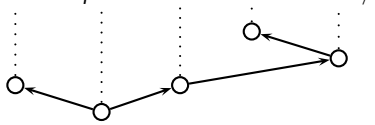
In addition to the balanced brackets, the encoding includes a separator, #, that is used to bound the nodes in the encoded dependency graph. The set of brackets and

the separator form the grammar alphabet, Γ. In addition, there is a corresponding alphabet, $\overline{\Gamma} = \{\, \bar{a} \mid a \in \Gamma\}$, that consists of the overlined variants of these symbols.

2.2 Bilexical Dependency Parsing

In fully data-driven parsing, the syntactic functions of dependencies are often unknown. Therefore, a bilexical dependency tree (3) focuses on the dependencies between two lexical entries.

(3) ⟨*time* *flies* *like* *an* *arrow*⟩



Although the tree in (3) is very elegant without arc labels, the currently described parser needs bracket labels in order to know about the syntactic properties of the linked tokens. Provided that the possible lexical types are fixed, the internal tag alphabet of the parser can be expanded with the brackets that indicate the lexical types (4). The expansion temporarily increases the redundancy in the encoding.

(4)

⟨*time*		*flies*		*like*		*an*		*arrow*⟩
time<	#	\flies flies/	#	>like like/	#	an<	#	\arrow >arrow

Although the linguistic aspects of dependency analyses would be an interesting topic for further discussions, the rest of the paper will focus on the computational properties of dependency tree processing.

2.3 The General Properties of Dependency Trees

Syntactic dependency trees have a number of crucial properties that we will need in order to distinguish a valid parse from invalid parses.

- Every syntactic dependency tree is a *labeled directed graph* $G = (V, \Gamma', E)$ where
  - V is the set of *nodes* (vertices) that correspond to the tokens in the sentence;
  - $\Gamma' = \Gamma - \{\#\}$ is the set of *arc labels*, and
  - $E \subseteq V \times \Gamma' \times V$ is the set of *labeled arcs* (aka directed labeled edges).

  In the dependency tree drawings, the arc (d, x, h) is drawn as $d \overset{x}{\leftarrow} h$. The arc indicates that node d depends on node h that is a head for d.
- Every syntactic dependency tree G is a labeled directed graph where every node has at most one head. That is, the set of arcs E can be seen as a partial function $E : V \to (\Gamma' \times V)$. Under this condition, we say that G has the *head property*.

- Every syntactic dependency tree is connected and acyclic. These properties are not local graph properties and, therefore, their definitions require additional machinery. In this paper, the machinery consists of *contractions* and *minors*:

  Definition 1. Let $G = (V, \Gamma', E)$ be a labeled directed graph with the head property. If there is an arc $(d, x, h) \in E$, then $(d, x, h') \in E$ implies $h = h'$. The *contraction* of arc (d, x, h) produces a new graph $H = (V', \Gamma', E')$ with

  $$V' = V - \{d\}, \tag{1}$$
  $$E' = (V' \times \Gamma' \times V') \cap (E \cup \{(d', y, h) \mid (d', y, d) \in E\}). \tag{2}$$

  The orthographical content corresponding to node h includes now implicitly the content of node d.

  Definition 2. In the current sense, a graph H is a *minor* of a directed graph G if a copy of H can be obtained from G via arc contractions.[4]

  Now we can test the connectedness and acyclicity as follows:

  – A labeled directed graph G with the head property is *connected* if and only if it has a trivial minor $H = (V', \Gamma', \emptyset)$ where $|V'| = 1$.
  – A labeled directed graph G with the head property is *acyclic* if and only if it has no minor graph $H = (V', \Gamma', E')$ with a loop $(d, x, d) \in E'$.

- Every syntactic dependency tree is a *rooted tree*. A connected labeled directed graph with the head property is a rooted tree if there is exactly one independent node, called a *root*, and all the arcs point away from the root. In the example (1), the root word is '*flies*'. All the arcs point away from this node.

It can be shown that a labeled directed graph G with the head property is a rooted tree if and only if G is connected and acyclic.

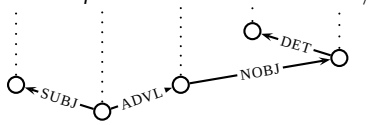
2.4 Validating Syntactic Dependency Trees

The relevant set of dependency trees are now characterized as acyclic and connected labeled directed graphs with the head property. This characterization does not directly involve testing for the existence of a root. Instead, we must (i) check that no word has two heads and (ii) prove the acyclicity and connectedness by contracting non-loop arcs until a trivial graph is reached.

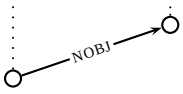
Some contractions can be performed in parallel. For example, (1) can be validated by two layers of contractions:

[4] This definition excludes arc deletion that is normally included in the definition of graph minors.

(5) a. ⟨*time flies like an arrow*⟩



b. ⟨*time flies like an arrow*⟩



c. ⟨*time flies like an arrow*⟩

An important observation of the current contribution is that the validation can be implemented directly on bracketed dependency trees. In (6), the tags affected by each contraction are indicated with an overline. A contraction of an arc (d,x,h) is an *internal contraction* if h has some other connections and d is a head for some other node. The contraction of the ADVL arc in (6a) is an internal contraction if performed before the SUBJ arc has been contracted.

(6) a. $\langle \overline{\textit{time}\ {<}\textsc{subj}}\ \overline{\#}\ \overline{\textit{flies}\ \textsc{subj}\backslash\ /\textsc{advl}}\ \overline{\#}\ \overline{\textit{like}\ \textsc{advl}{>}\ /\textsc{nobj}}\ \#\ \overline{\textit{an}\ {<}\textsc{det}}\ \overline{\#}\ \overline{\textit{arrow}\rangle\ \textsc{det}\backslash\ \textsc{nobj}{>}}$

b. $\langle \overline{\textit{time flies like}\ /\textsc{nobj}}\ \overline{\#}\ \overline{\textit{an arrow}\rangle\ \textsc{nobj}{>}}$

c. ⟨*time flies like an arrow*⟩

The validation of bracketed trees is based on three principles:

1. *Decodability.* For each label $\alpha \in \Gamma - \{\#\}$, the left brackets ${<}\alpha$ and $\alpha\backslash$ are matched with the corresponding right brackets $\alpha\backslash$ and $\alpha{>}$, respectively. Each pair of matching brackets corresponds to an arc in the labeled directed graph.
2. *Equicardinality.* There is the same number of arcs and word boundaries. Every boundary between two adjacent words is indicated with a hash symbol (#). A hash symbol is eliminated at the same time as the brackets. This ensures that a loop cannot be eliminated because the left and right brackets are not separated by any hash symbol. Thus, a cyclic dependency graph cannot be fully reduced to a trivial graph. If the graph is not connected, there remains a word boundary that is not eliminated, and the graph does not have a trivial graph as a minor.
3. *Contiguity.* The internal contraction of an arc (d,x,h) is allowed only if node h corresponds to a contiguous string of brackets in the resulting graph. This principle ensures that the resulting minor graph can be encoded with dependency bracketing.

2.4.1 Ensuring Decodability

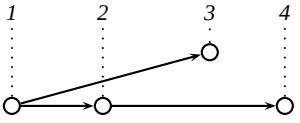
The bracket labels play a crucial role in non-projective dependency trees and in minors obtained from them.

Any non-projective dependency tree can be bracketed when we adopt a multiplanar decomposition for the arcs and corresponding brackets (Yli-Jyrä 2003). This means that there is no limit for the complexity of non-projective trees, provided that the number of available planes is not fixed. In bracketed encodings of bilexical dependency trees, multiplanarity seems presently to be the only way to encode crossing brackets.

A 2-plane encoding is already enough to achieve very high coverage (Gómez-Rodríguez and Nivre 2010). Syntactic functions could further extend the set of non-projective trees that can be encoded by allocating each function to a plane of its own.

Example (7) shows that matching brackets of crossing arcs are distinguished using two planes, I and II.

(7) a.



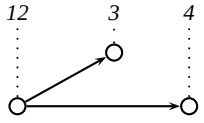
b.

2.4.2 Ensuring Contiguity

A typical non-projective dependency tree contains a large subgraph that does not contain crossing links. Therefore, it is often possible to reduce many non-crossing arcs before it is necessary to contract any crossing arc.

An interesting observation is that contractions of non-crossing arcs often transform a non-projective tree into a projective tree as demonstrated by Example (8) that is obtained from the non-projective tree in (7).

(8) a.

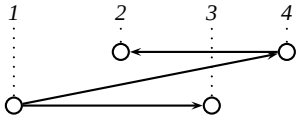


b.

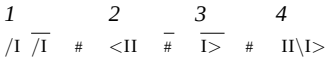
Another interesting observation is that a non-internal contraction does not need to merge adjacent positions in a bracketed tree. This produces a significant extension to simple contractions that can be used to transform a non-projective tree into a

projective one. The power of non-internal contractions is illustrated by Example (9) that reduces to (10).

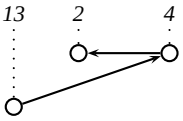
(9) a.



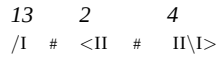
b.



(10) a.



b.



Many of the non-projective trees discussed in Yli-Jyrä (2003) and Kuhlmann (2010) can be reduced to the trivial tree via contractions of non-crossing arcs. The remaining non-projective trees can be reduced with the aid of non-internal contractions, because every nontrivial tree admits at least one such contraction.

3 Computing Weighted Minors

This section describes a mechanical, finite-state implementable deterministic method whose purpose is to perform at least one contraction in any nonempty string. The reader is referred to Mohri (2009) for a detailed exposition of algorithms on weighted transducers.

In this article, the algorithms are specified with weighted rational relations whose operations can be implemented through manipulation of finite-state transducers.

3.1 The Formalism of Weighted Rational Relations

In this paper, weights are nonnegative real numbers ($\mathbb{R}_{\geq 0} \cup \{\infty\}$) with the usual multiplication operation and the *maximum* (max) as the additive operation (i.e., $+$ and $\sum$ denote the max operation). This set of weights gives us an easily understandable starting point and supports Viterbi-decoding of the best parses.

Let Σ be an alphabet. The free monoid generated by Σ is denoted by Σ^*. The neutral element of this monoid is the empty string, ε. The set of rational (i.e., regular) languages includes the finite subsets of Σ^* and is closed, for any two elements L, M,

under the rational operations such as concatenation $L \cdot M$, star (L^*), the Boolean operations ($L \cup M$, $L \cap M$, $L - M$, etc.).

The set of (binary) rational relations over Σ^* is also closed under concatenation, star and union and includes rational relations such as

$$\mathrm{Id}(L) = \{(x,x) \mid x \in L\}, \tag{3}$$
$$L \times M = \{(x,y) \mid x \in L, y \in M\}. \tag{4}$$

Let $R \subseteq \Sigma^* \times \Sigma^*$ be a rational relation. In a pair $(x,y) \in R$, x is called the *input* string and y is called the *output* string. Define the characteristic *weighted rational relations* $\mathbb{1}(R), \mathbb{1}_\varepsilon : (\Sigma^* \times \Sigma^*) \to (\mathbb{R}_{\geq 0} \cup \{\infty\})$ by

$$\mathbb{1}(R)(x,y) = \begin{cases} 1 & (x,y) \in R \\ 0 & \text{otherwise,} \end{cases}, \tag{5}$$
$$\mathbb{1}_\varepsilon = \mathbb{1}(\mathrm{Id}(\varepsilon)). \tag{6}$$

Simple weighted rational relations can be defined with the comprehension notation, but the notation itself does not guarantee that the defined set is a weighted rational relation. Instead, the set of weighted rational relations over the alphabet Σ^* and the weights $\mathbb{R}_{\geq 0} \cup \{\infty\}$ (with max and multiplication) is closed under certain operations. Let T and U be weighted rational relations $(\Sigma^* \times \Sigma^*) \to (\mathbb{R}_{\geq 0} \cup \{\infty\})$ and let $w \in \mathbb{R}_{\geq 0} \cup \{\infty\}$. Define the *left product*, *union*, *composition*, *concatenation*, *star* and the *projection* operations by

$$w \cdot T = \{((x,y), w \cdot T(x,y)) \mid x,y \in \Sigma^*\}, \tag{7}$$
$$T \cup U = \{((x,y), T(x,y) + U(x,y)) \mid x,y \in \Sigma^*\}, \tag{8}$$
$$T \circ U = \{((x,z), \sum_{y \in \Sigma^*} T(x,y)U(y,z)) \mid x,z \in \Sigma^*\}, \tag{9}$$
$$T \cdot U = \{((x,y), \sum_{x=x_0x_1, y=y_0y_1} T(x_0,y_0)U(x_1,y_1)) \mid x,y \in \Sigma^*\}, \tag{10}$$
$$T^* = \{((x,y), \sum_{\substack{n \in \mathbb{N} \\ x=x_1 \ldots x_n \\ y=y_1 \ldots y_n}} T(x_1,y_1) \ldots T(x_n,y_n)) \mid x,y \in \Sigma^*\}, \tag{11}$$
$$\mathrm{Proj}_1(T) = \{((x_1,x_1), \sum_{x_2 \in \Sigma^*} T(x_1,x_2)) \mid x_1 \in \Sigma^*\}, \tag{12}$$
$$\mathrm{Proj}_2(T) = \{((x_2,x_2), \sum_{x_1 \in \Sigma^*} T(x_1,x_2)) \mid x_2 \in \Sigma^*\}. \tag{13}$$

Note that if $T(\varepsilon,\varepsilon) \neq 0$ and $T(x,y) \neq 0$, $T^*(x,y) = \infty$.

For a weighted rational relation T, define its *image* and *support* by

$$\mathrm{Im}(T) = \{T(x,y) \mid x,y \in \Sigma^*\}, \tag{14}$$
$$\mathrm{Supp}(T) = \{(x,y) \mid x,y \in \Sigma^*, T(x,y) \neq 0\}. \tag{15}$$

By convention, T can be viewed as a weighted rational relation $(\Sigma_1^* \times \Sigma_2^*) \to (\mathbb{R}_{\geq 0} \cup \{\infty\})$ where $\Sigma_1, \Sigma_2 \subseteq \Sigma$ if T is a weighted rational relation $(\Sigma^* \times \Sigma^*) \to (\mathbb{R}_{\geq 0} \cup \{\infty\})$ with $\mathrm{Supp}(T) \subseteq \Sigma_1^* \times \Sigma_2^*$, and vice versa.

Let T be a weighted rational relation with a finite support and $p \in \{1,2\}$. Let w be the maximal value in $\mathrm{Im}(T)$. Let the sequence $\langle x_1, \ldots, x_j \rangle$ contain the elements of the set $\{x \mid x \in \Sigma^*, \mathrm{Proj}_p(T)(x,x) = w\}$ in the lexicographical order. Define the k-bounded *best restriction* of $\mathrm{Proj}_p(T)$ as

$$\mathrm{BestProj}_p(T,k) = \{((x_i,x_i),w) \mid i \in \{1,\ldots,\min\{j,k\}\}\}. \tag{16}$$

3.2 Weighted Contractions

In the parser, the rule component of the grammar defines a weighted rational relation, Contraction: $\mathrm{Id}((\Gamma \cup \overline{\Gamma})^*) \to (\mathbb{R}_{\geq 0} \cup \{\infty\})$. The support language of this relation is $\mathrm{Id}^{-1}(\mathrm{Supp}(\mathrm{Contraction}))$ and it is a subset of $\Gamma^*\overline{\Gamma}\Gamma^* \; \overline{\#} \; \Gamma^*\overline{\Gamma}\Gamma^*$.

A convenient way to specify Contraction is through a finite set of contraction rules $\alpha \mapsto w$ where the expression α gives a rational (i.e., regular) subset of the language $\Gamma^*\overline{\Gamma}\Gamma^* \; \overline{\#} \; \Gamma^*\overline{\Gamma}\Gamma^*$ and w is a non-negative real number. The examples of contraction rules include projective functional rules (11a–b), projective bilexical rules (11c–d), and non-projective bilexical rules (11e).

(11) a. $(\overline{<\textsc{subj}} \;\; \overline{\#} \;\; \overline{\textsc{subj}\backslash}) \;\mapsto\; .97,$

b. $(\overline{/\textsc{advl}} \;\; \overline{\#} \;\; \overline{\textsc{advl}>}) \;\mapsto\; .47,$

c. $(\overline{\mathrm{like}/} \;\; \overline{\#} \;\; \overline{>\mathrm{arrow}}) \;\mapsto\; .00127,$

d. $(\overline{\mathrm{an}<} \;\; \overline{\#} \;\; \overline{\backslash\mathrm{arrow}}) \;\mapsto\; .42,$

e. $(\# \; \overline{<\textsc{subj}} \;\; \overline{\#} \;\; (\Gamma - \{<\textsc{subj}, \textsc{subj}\backslash\})^* \;\; \overline{\textsc{subj}\backslash}) \;\mapsto\; .97,$

When applied by the finite-state implementation, each contraction rule removes a pair of dependency brackets and a respective node separator ($\#$). The overlining of some symbols indicates which three tags in the strings disappear when a contraction is performed. When a rule with weight w is applied, the total weight of the string is multiplied by w. In (11e), there are potentially some symbols that do not disappear.

3.3 Applying Weighted Contractions Deterministically

FreeReduce is a weighted rational relation that reduces bracketed trees by applying a specified set of contractions freely to the strings. It is constructed as follows:

$$\text{Hesitate} = \mathbb{1}((\text{Id}(\Gamma) \cup \{(a, \overline{a}) \mid a \in \Gamma\})^*), \tag{17}$$

$$\text{FreeMark} = (\mathbb{1}(\text{Id}(\Gamma)) \cup \text{Contraction})^*, \tag{18}$$

$$\text{Perform} = \mathbb{1}((\text{Id}(\Gamma) \cup \{(a, \varepsilon) \mid a \in \overline{\Gamma}\})^*), \tag{19}$$

$$\text{FreeReduce} = \text{Hesitate} \circ \text{FreeMark} \circ \text{Perform}. \tag{20}$$

When an input bracketing is reduced with FreeReduce, each possible place for contractions optionally either undergoes the contraction or is left intact as indicated by the weighted pairs (12a–d) belonging to FreeMark. The optionality generates spurious ambiguity. Spurious ambiguity complicates the extraction of the top k best parses as any optimal parse itself may be reduced in more than k different ways. Furthermore, it restricts the possibilities for generalizing the parsing algorithm and changing the system of weights: if the additive operation over the weights were non-idempotent (*maximum* is idempotent), we would have a danger that spurious ambiguity invalidates the weights of parses.

(12) a. (Id(<SUBJ # SUBJ\/ADVL # ADVL>), 1)

b. (Id(<SUBJ # SUBJ\ $\overline{/\text{ADVL}}$ $\overline{\#}$ $\overline{\text{ADVL}>}$), .47)

c. (Id($\overline{<\text{SUBJ}}$ $\overline{\#}$ $\overline{\text{SUBJ}\backslash}$ /ADVL # ADVL>), .97)

d. (Id($\overline{<\text{SUBJ}}$ $\overline{\#}$ $\overline{\text{SUBJ}\backslash}$ $\overline{/\text{ADVL}}$ $\overline{\#}$ $\overline{\text{ADVL}>}$, $.97 \times .47$)

The spurious ambiguity can be avoided by restricting the support of (Hesitate $\circ$ FreeReduce) in such a way that it is a function from inputs to outputs.

In order to make the restriction, the contraction rules are applied deterministically from left to right. This modification can be implemented with a technique (Yli-Jyrä 2008) that is based on earlier ideas of G. van Noord and D. Gerdemann. To apply this technique, define a rational relation Prefer : $(\Gamma \cup \overline{\Gamma})^* \times (\Gamma \cup \overline{\Gamma})^*$ that relates a pair (x,y) of two overline marked copies of the same string if the first copy, x, contains earlier overlines than y. For example, (12b) is preferred over (12a), (12c) is preferred over (12a) and (12b), and (12d) is preferred over (12a–c).

$$\text{Prefer} = \text{Id}(\Gamma^*) \cdot \{(\,\overline{a}, a) \mid a \in \Gamma\} \cdot \{(x,y) \mid x,y \in \{a, \overline{a}\}, a \in \Gamma\}^*. \tag{21}$$

Now we extract from FreeMark the set of strings, Dispreferred(FreeMark), for which there are preferred alternatives, and construct its complement NotDispreferred(FreeMark).

$$\text{Dispreferred}(S) = \text{Id}^{-1}(\text{Supp}(\text{Proj}_2(S \circ \mathbb{1}(\text{Prefer}) \circ S))), \tag{22}$$

$$\text{NotDispreferred}(S) = (\Gamma \cup \overline{\Gamma})^* - \text{Dispreferred}(S). \tag{23}$$

By filtering the identity pairs in FreeMark with NotDispreferred(FreeMark), we refine FreeMark and obtain DefiniteMark. We also want to reject all nonempty strings without any overlined symbols ($\Gamma\Gamma^*$). In the end, we obtain a weighted rational relation that "performs" a deterministic, non-empty set of contractions in all non-empty inputs.

$$\text{DefiniteMark} = \text{FreeMark} \circ \mathbb{1}(\text{Id}(\text{NotDispreferred}(\text{FreeMark}) - \Gamma\Gamma^*)), \quad (24)$$

$$\text{Reduce} = \text{Hesitate} \circ \text{DefiniteMark} \circ \text{Perform}. \quad (25)$$

We have thus defined a weighted rational relation, Reduce, that maps the input strings deterministically to strings that are strictly shorter unless the input is already the empty string.

Reduce can be viewed as a function $\Gamma^* \to (\Gamma^* \times (\mathbb{R}_{\geq 0} \cup \{\infty\}))$. The existence of this alternative structure implies that Reduce can be implemented very efficiently with a deterministic finite-state device.

4 The Structure of the Grammar and the Parser

The purpose of this section is to define the grammar and the respective parser in terms of weighted rational relations.

4.1 The Grammar Relation

In a high level, the grammar can be seen as a composition (26) of four (weighted) rational relations of type $(\Sigma^* \times \Sigma^*) \to (\mathbb{R}_{\geq 0} \cup \{\infty\})$.

$$\text{Grammar} = \text{Lexicon} \circ \text{Abstract} \circ \text{HasMinor}_t \circ \mathbb{1}_\varepsilon. \quad (26)$$

In the composition, Lexicon does tokenization and morphological analysis and then retrieves arguments and functions, Abstract is a relation (27) that deletes all but syntactic symbols in strings, HasMinor_t performs t levels of reductions, being thus a finite composition (28) of t identical Reduce relations, and $\mathbb{1}_\varepsilon$ ensures that we finally obtain a trivial minor graph.

$$\text{Abstract} = \mathbb{1}((\text{Id}(\Gamma) \cup \{(x, \varepsilon) \mid x \in \Sigma - \Gamma\})^*), \quad (27)$$

$$\text{HasMinor}_t = \underbrace{\text{Reduce} \circ \cdots \circ \text{Reduce}}_{t}. \quad (28)$$

The component relations of the grammar link four representations:

$$\text{Ortho} \xrightarrow{\text{Lexicon}} \text{MorphoSyn} \xrightarrow{\text{Abstract}} \text{Syn} \xrightarrow{\text{HasMinor}_t} \{\varepsilon\}. \quad (29)$$

In this system, Ortho is the set of orthographical strings over the set of orthographical symbols Ω, MorphoSyn is the set of morpho-syntactic strings that consist of morphological symbols M and grammatical symbols Γ, and Syn is the set of syntactic strings over the alphabet Γ.

To be precise, Grammar is a weighted rational relation that maps the pairs $(x, \varepsilon) \in \Omega^* \times \{\varepsilon\}$ to the set of weights. The precise interpretation of the weights

remains intentionally open. The Grammar relation also characterizes a string set, Grammatical $\subseteq \Omega^*$, that is defined by

$$\text{Grammatical} = \text{Id}^{-1}(\text{Supp}(\text{Proj}_1(\text{Grammar}))). \tag{30}$$

4.2 The Parser Relation

In order to *parse* an orthographical string $x \in \text{Ortho}$, we need to extract the corresponding morpho-syntactic strings $y \in \text{MorphoSyn}$ from the internals of the system (29). The extraction process (31) defines a weighted rational relation, $\text{Parser} : (\text{Ortho} \times \text{MorphoSyn}) \to (\mathbb{R}_{\geq 0} \cup \{\infty\})$.

$$\text{Parser} = \text{Lexicon} \circ \text{Proj}_1(\text{Abstract} \circ \text{HasMinor}_t \circ \mathbb{1}_\varepsilon). \tag{31}$$

Let $x \in \text{Ortho}$ be an orthographical string. If (x,y) is a pair in $\text{Supp}(\text{Parser})$, we say that y is a *parse* for x. The set of all parses for x is denoted by

$$\text{Parses}(x) = \text{Id}^{-1}(\text{Supp}(\text{Proj}_2(\mathbb{1}(\text{Id}(x)) \circ \text{Parser}))). \tag{32}$$

The weight of each parse $y \in \text{Parses}(x)$ is Parser(x,y). A k-bounded set of best parses is given by

$$\text{BestParses}(x,k) = \text{Id}^{-1}(\text{Supp}(\text{BestProj}_2(\mathbb{1}(\text{Id}(x)) \circ \text{Parser}, k))). \tag{33}$$

4.3 The Grammar Constant

The parameter t limits the number of iterations of Reduce in HasMinor_t. In the dependency trees, the parameter limits the number of overlapping arcs that can be contracted. The parameter can be fixed to a relatively small integer without any observable loss in recall. This makes HasMinor_t a fixed weighted rational relation. The Grammar and Parser relations are thus applicable in linear time, at least according to the asymptotic complexity analysis (as $n \to \infty$).

The asymptotic analysis ignores the fact that the application of the grammar to the input involves a large coefficient, the *grammar constant*, that is bounded from the above by the product of the sizes of the finite-state transducers for Lexicon, Abstract and HasMinor_t. As to their sizes, Lexicon and Abstract are just ordinary kinds of weighted rational relations used in natural-language processing. Their implementation does not require our attention now.

In contrast to Lexicon and Abstract, the finite-state implementation of HasMinor_t is of an impractical size. To see this, assume that $\text{Supp}(\text{Contract}) = \{(i \# i) \mid i \in \{1,\ldots,c\}\}$ where c is the number of arc types. Table 2 shows experimental results on how the size of HasMinor_t grows as a function of c and t. From these I gather

that the number of states in the finite-state implementation of $\mathrm{HasMinor}_t \circ \mathbb{1}_\varepsilon$ is

$$(2c)^t + (2c)^{t-1} + \cdots + 1 = \sum_{k=0}^{t} (2c)^k = \frac{(1-(2c)^{t+1})}{1-(2c)} = O((2c)^t). \qquad (34)$$

Table 2 The growth of $\mathrm{HasMinor}_t$ as a function of t and the number of functional bracket pairs

	$c=1$			$c=2$			$c=3$		
t	states	trans.	explanation	states	trans.	explanation	states	trans.	explanation
1	3	3	$2+1$	5	6	$4+1$	7	9	$6+1$
2	7	9	$4+2+1$	21	30	$16+4+1$	43	63	$36+6+1$
3	15	21	$8+\cdots+1$	85	126	$64+\cdots+1$	259	387	$216+\cdots+1$
4	31	45	$16+\cdots+1$	341	510	$256+\cdots+1$	1555	2331	$1296+\cdots+1$
5	63	93	$32+\cdots+1$	1365	2046	$1024+\cdots+1$	9331	13995	$7776+\cdots+1$
6	127	189	$64+\cdots+1$	5461	8190	$4096+\cdots+1$	55987	83979	$46656+\cdots+1$

4.4 An On-Demand Construction

A slight improvement to the precomputation of (28) is obtained by the on-demand computation of $\mathrm{HasMinor}_t$. This idea is used in (36), where Grammar is restricted to the pair of the orthographical string x and the empty string ε.

$$\mathrm{Bot(x,y)} = \mathbb{1}(\{(x,y)\}) \circ \mathrm{Lexicon}, \qquad (35)$$

$$\mathrm{Grammar}_{|\{(x,\varepsilon)\}} = \underbrace{(\ldots(}_{t}(\mathrm{Bot}(x,x) \circ \mathrm{Abstract}) \underbrace{\circ \mathrm{Reduce}) \cdots \circ \mathrm{Reduce})}_{t} \circ \mathbb{1}_\varepsilon. \qquad (36)$$

The worst-case size complexity of the finite-state representation of $\mathrm{Grammar}_{|\{(x,\varepsilon)\}}$ is still exponential to t, but the average-case complexity of (36) can be much smaller than the complexity of the constant grammar (28). This admits practical applicability on similar grounds as some previous parsing approaches that iteratively verify labeled bracketing (Roche 1997, Oflazer 2003).

5 A Non-Linear but Efficient Approach

In the above, we have seen that although the parser can be represented as a rational relation that can be applied in linear time to the input string, the hidden grammar constant does not guarantee that the relation could always be restricted efficiently to an orthographical string. There are situations where we need guarantees for the worst-case complexity.

This section describes algorithms that do not fully elaborate the composition (28) of the relations. Instead, the algorithms compute the composition indirectly through intermediate languages. The space complexity of each intermediate representation is not linear to the length of the sentence because their epsilons are removed. Since we never compute the composition as a whole, the algorithms are still more practical than the naive approaches that construct $\mathrm{HasMinor}_t$ in one way or another.

5.1 Forgetting Composition

If $\mathrm{Grammar}_{|\{(x,\varepsilon)\}}$ is immediately applied to the pair (x,ε), we may replace, in (37), the input side of the composition with the empty string and still compute the same weight for (x,ε).

$$\mathrm{Grammar}(x,\varepsilon) = \mathrm{Im}(\underbrace{(\ldots(}_{t}(\mathrm{Bot}(\varepsilon,x)\circ\mathrm{Abstract})\underbrace{\circ\mathrm{Reduce})\cdots\circ\mathrm{Reduce})}_{t}\circ\mathbb{1}_{\varepsilon}). \tag{37}$$

The effect of the modification is significant. It basically makes the composition to forget everything that is contracted. Since the matching pairs of brackets are forgotten, the details of the contracted brackets are not complicating the further processing. The forgetting effect can be implemented also via projections as in (38).

$$\mathrm{Grammar}(x,\varepsilon) = \mathrm{Im}(\underbrace{[\ldots[}_{t}[\mathrm{Bot(x,x)}\circ\mathrm{Abstract}]\underbrace{\circ\,\mathrm{Reduce}\ldots]\circ\mathrm{Reduce}]}_{t}\circ\mathbb{1}_{\varepsilon}), \tag{38}$$
$$\text{where } [X] = \mathrm{Proj}_2(X).$$

The time complexity of this composition-projection method is linear to t and to the worst-case time complexity of the iteration rounds.

5.2 A Preliminary Complexity Analysis

In order to analyze the space complexity of the minimized sizes of the projections, I carried out some experiments. In these experiments, the number of tokens was $n \leq 80$ and the number of iterations $t \leq n-1$, which is sufficient for obtaining all parses. A highly ambiguous lexicon was modeled by replacing $\mathrm{Bot}(x,x)$ either with model (13a), where Γ_{d} contains dependent-side brackets and Γ_{h} contains head-side brackets, or with model (13b), where Γ_{u} consists of brackets that encode undirected arcs.

(13) a. $\mathbb{1}(\mathrm{Id}(\ \Gamma_{\mathrm{d}}^*(\varepsilon\cup\Gamma_{\mathrm{h}})\Gamma_{\mathrm{d}}^*\ \ (\#\,\Gamma_{\mathrm{d}}^*(\varepsilon\cup\Gamma_{\mathrm{h}})\Gamma_{\mathrm{d}}^*)^{n-1}\ ))$

b. $\mathbb{1}(\mathrm{Id}(\ \Gamma_{\mathrm{u}}^*\ \ (\#\,\Gamma_{\mathrm{u}}^*)^{n-1}))$

The first model (13a) gives rise to rooted trees and (13b) to unrooted ones. Bilexical brackets were modeled by adding token numbers to the respective brackets. The contraction rules of the grammar are restricted to those of the shape $\overline{\alpha}\ \overline{\ }_{\#}\ \overline{\beta} \mapsto 1$, where $\overline{\alpha}, \overline{\beta} \subseteq \overline{\Gamma}$.

In the experiments, I measured the size (number of states and number of transitions) of minimal (unweighted) finite-state transducers that correspond to the first intermediate result, $[[\mathrm{Bot}(x,x)] \circ \mathrm{Reduce}]$, and the subsequent composition-projections in (38). To reduce the number of necessary experiments, I eliminated some dimensions with simple tests. These tests gave the following useful results:

- The sizes of intermediate results grow only by a constant factor when we switch from unrooted trees to rooted trees.
- If the lexical differences were reduced, the size of the largest intermediate result would be become smaller. Thus, the bilexical bracketing presents the maximal complexity.
- If the number of functional categories of the dependencies doubles, the number of transitions will double too, but the number of states does not change.

My main experiment focused on unrooted bilexical bracketing without dependency functions. The models of inputs consisting of $n = 20, \ldots, 80$ tokens were compared in order to see how the sizes of the intermediate results in bilexical parsing grow as a function of n. For all sentence lengths, the $(n/2-1)^{\text{th}}$ iteration produced the largest result (Table 3).

In Table 3, the number of transitions in minimized projections is almost quadratic (the exponent is between 1.60 and 1.87) to the number of states. This motivates the observation that the complexity of the algorithm is not linear to n. In each intermediate result, the contractions shorten the strings, which gives, in the finite-state representations, rise to epsilon removal and a quadratic number of transitions. Besides the epsilon removal, the finite-state library automatically performs determinization and minimization of the finite-state representations of the projections.

Table 3 The sizes of the projections of the first, the fifth and the $(n/2-1)^{\text{th}}$ applications

	1^{th} iteration				5^{th} iteration				$(n/2-1)^{\text{th}}$ iteration			
n	states	trans.	exp.	$2n^2$	states	trans.	$6n^2$	secs	$(n/2-1)$	states	trans.	exp.
20	38	834	1.85	800	90	1974	2400	.06	9	110	2410	1.66
30	58	1854	1.85	1800	150	4794	5400	.10	14	240	7665	1.63
40	78	3274	1.86	3200	210	8814	9600	.19	19	420	17620	1.62
50	98	5094	1.86	5000	270	14034	1500	.34	24	650	33775	1.61
60	118	7314	1.87	7200	330	20454	21600	.58	29	930	57630	1.60
70	138	9934	1.87	9800	390	28074	29400	.94	34	1260	90685	1.60
80	158	12954	1.87	12800	450	36894	38400	1.45	39	1640	134440	1.60

In Table 3, the number of states in the first intermediate result is $2(n-1)$ and in the largest intermediate result the number of states coincided with the function

$(n/2)(n/2+1)$. The number of transitions in the largest intermediate result coincides with the function $n(n/2+1)(n/2+1)-(n/2)$.

Usually, however, t is fixed and much smaller than n. The table indicates that when n doubles from 20 to 40 and 80, the number of transitions in the first intermediate result grows by the factors $2^{1.97}$ and $2^{1.98}$ and the fifth intermediate result by the factors $2^{2.15}$ and $2^{2.07}$. This indicates that the number of transitions in a fixed intermediate result, such as the first and the fifth one, is actually $O(n^s)$ where s is close to 2. The number of iterations does not have any drastic effect on the space complexity, since the 5th intermediate result, for example, has less than $6n^2$ transitions. As the number of compositions is bounded by t, we actually compute only a fixed number of intermediate projections. The transition complexity of each minimized intermediate result seems to be in $O(t\,n^2)$.

Assuming that the required time would be linear to the size of the results, the total time complexity of computing the value of Grammar(x,ε) would be $O(t^2n^2)$. But Table 3 displays the running times for the fifth iteration round using an unweighted finite-state library (foma). The measured running time appears to be in $O(n^3)$ since, e.g., $\log_2(1.45/.19)=2.93$. The experiment does not allow us, however, to conclude that an implementation with a quadratic time complexity would be impossible. The contributions of the determinization and minimization steps and the actual library implementation have not been analyzed yet.

The current experimental analysis has assumed that the maximally ambiguous sentences and grammars are asymptotically at least as difficult as practical sentences and grammars. I have currently no complete proof for this assumption, but I believe that the complexity of a realistic situation differs only linearly from the current artificial situation. Clearly, the assumption prompts for further study.

The current analysis does not fully apply to the weighted case. Since weighted determinization and minimization (Mohri 2009) can move the weights from the original places, there is a danger that the intermediate results grow more than necessary. The detailed analysis of the weighted case is postponed to further work.

5.3 Memoizing Composition

An efficient decision method for grammatical strings in (38) leads us halfway to obtaining some if not all parses efficiently. This requires reusing the computations done during the decision process. Therefore, the intermediate results are memoized inductively to variables $\mathrm{Up}_0,\ldots,\mathrm{Up}_t$ by setting

$$\mathrm{Up}_0 = [\mathrm{Bot}(x,x)\circ\mathrm{Abstract}], \tag{39}$$

$$\mathrm{Up}_i = \underbrace{[\ldots[}_{i}\,\mathrm{Up}_0\,\underbrace{\circ\,\mathrm{Reduce}\ldots]\circ\mathrm{Reduce}]}_{i} = [\mathrm{Up}_{i-1}\circ\mathrm{Reduce}]. \tag{40}$$

In the end, $\mathrm{Up}_t(\varepsilon,\varepsilon)$ tells the weight of the best parse.

Now we could compute k-bounded best restrictions iteratively in order to obtain (at most k) best parses. This is achieved by processing the intermediate levels i from the top level, t, back to the lowermost level 0 and by filtering the lower level with the information on the partial parses of the best parses.[5]

If x is a grammatical string, the support of the first downward level, Dn_t, contains the encoded trivial graph, ε, whose top-down weight is 1:

$$\mathrm{Dn}_t = \mathbb{1}(\mathrm{Supp}(\mathrm{Up}_t \circ \mathbb{1}_\varepsilon)). \tag{41}$$

For each level $i \in \{t-1,\ldots,2,1\}$, we first compute $\mathrm{DnSupport}_i$ that contains partial parses of the best parses. The best parses are selected on the basis of their total weight, whose factors come from the Up_i, Reduce and Dn_{i+1} components. After this, we compute Dn_i, which contains the same strings with the top-down weights only.

$$\mathrm{DnSupport}_i = \mathrm{Supp}(\mathrm{BestProj}_1(\mathrm{Up}_i \circ \mathrm{Reduce} \circ \mathrm{Dn}_{i+1}, k)), \tag{42}$$

$$\mathrm{Dn}_i = \mathrm{Proj}_1(\mathbb{1}(\mathrm{DnSupport}_i) \circ \mathrm{Reduce} \circ \mathrm{Dn}_{i+1}). \tag{43}$$

The last level, Dn_0, is computed differently:

$$\mathrm{Dn}_0 = \mathrm{BestProj}_1(\mathrm{Proj}_2(\mathrm{Bot}(x,x)) \circ \mathrm{Abstract} \circ \mathrm{Dn}_1, k). \tag{44}$$

In the end, the support and the image of Dn_0 contains up to k parses and the best weight, respectively. We can now define the selection of k best parses by

$$\mathrm{BestParses}'(x,k) = \mathrm{Id}^{-1}(\mathrm{Supp}(\mathrm{Dn}_0)). \tag{45}$$

The previously defined set $\mathrm{BestParses}(x,k)$ in (33) does not necessarily coincide with $\mathrm{BestParses}'(x,k)$ in (45), because the different methods may pick a different selection from the best parses.

The total time complexity of this best-parse algorithm is dominated by the bottom-up phase, because extracting the best k parses from the memoized cascade $\mathrm{Up}_0,\ldots,\mathrm{Up}_{t-1}$ takes only linear time to the size of the memoized finite-state transducers. This result makes use of the linear time complexity of the shortest-distance algorithm for acyclic weighted automata (Mohri 2009).

The same parser algorithm is applicable with non-projective contraction rules. However, the time complexity of the resulting non-projective parser depends on the specifics of the rule component and remains open for the time being.

[5] These minorization and "majorization" phases could be compared to the forward and backward procedures used in trellis algorithms for Hidden Markov Models.

5.4 Allowing Spurious Ambiguity

Since we use *maximum* as the additive operation for the weights, the spurious ambiguity does not actually affect the weights of the parses. This observation allows us to avoid Reduce and use FreeReduce instead. That is, the grammar semantics will be retained even if we replace $\mathrm{HasMinor}_t$ with $\mathrm{HasMinor}'_t$, defined by

$$\mathrm{HasMinor}'_t = \underbrace{\mathrm{FreeReduce} \circ \cdots \circ \mathrm{FreeReduce}}_{t}. \tag{46}$$

Similarly, the use of Reduce could be replaced with FreeReduce in the forgetting composition. In memoizing composition, the change applies too, provided that we then extract only the best parse ($k = 1$).

In practice, FreeReduce is much easier to construct than Reduce. It also induces smaller projections (Table 4) and provides much faster application to long sentences. This is explained by the fact that the states in the composition with FreeReduce do not keep track of the number of applied contractions. On the contrary, the obligatory contractions in Reduce expand the state space of the compositions and the projections, which also complicates the subsequent epsilon removal, determinization and minimization steps.

Table 4 The sizes of the projections after applying FreeReduce, and of Parses(x) (now avoided)

	1st iteration		5th iteration				t iterations with $t = n-1$		Parses(x)
n	states	trans.	states	trans.	(n^2+2n-1)	tot.secs	tot.secs	parses	states in fsa
10	10	119	10	119	119	.04	.04	246675	2036
20	20	439	20	439	439	.05	.10	16332922290300	2097130
40	40	1679	40	1679	1679	.08	.34	2.1×10^{29}	2.2×10^{12}
60	60	3719	60	3719	3719	.14	1.80	4.5×10^{45}	2.3×10^{18}
80	80	6559	80	6559	6659	.27	4.50	1.1×10^{62}	2.4×10^{24}

I experimented with forgetting composition that uses FreeReduce. By comparing the lines for $n = 40$ and $n = 80$ in Table 4, the total time complexity of the projective parser (unrooted trees, $t = n-1$) appears to be in $O(n^{3.73})$ since $\log_2(4.5/.34) = 3.73$. However, if we fix $t = 5$, the total time complexity appears to be quadratic to n since $\log_2(.27/.08) = 1.75 \approx 2$. Since this is linear to the transitions in each projection, it appears that the worst-case complexity is in $O(t\,n^2)$ in general.

Table 4 shows also the total number of unrooted trees (i.e., parses) for sentences of different lengths. In the worst case, the growth in the number of trees is really fast. The resulting sequence coincides with the sequence A001764 in Sloane's On-Line Encyclopedia of Integer Sequences (OEIS, oeis.org).

If all the strings that encode the parses for a 20-word sentence would be stored into one finite-state automaton (fsa), this would require, in the worst case, more than 2 million states (the last column in Table 4). In general, the sequence of the worst-

case state counts, $2^{n+1}-n-2$, for the single-fsa representations coincides with the Eulerian numbers $\langle {1 \atop 1} \rangle, \langle {2 \atop 1} \rangle, \ldots$ (the sequence A000295 in OEIS). This demonstrates that the memoized cascade is much more efficient representation for the parse forest than a single automaton.

A drawback in using FreeReduce is that only one optimal parse can be extracted directly from the memoized cascade, because extracting k parses can actually result in extracting the same parse in k different ways. In order to obtain the next optimal parse, we can "remove" the best parse from $\mathrm{Bot}(x,x)$ and rerun the parser on the remainder set. This may be inefficient in practice.

6 Comparison to the Prior Work

The body of research on dependency parsing is already large and it is impossible to recall all approaches. The most relevant prior work combines dependency parsing and string-based finite-state methods, or at least suggests such a combination.

- *Constraint Grammar (CG)* parsers perform morphological and surface-syntactic disambiguation and dependency linking.
  - The core CG parsers refine the ambiguity classes of words iteratively, according to the contextual conditions and rule application ordering.
  - Mature CG variants (Tapanainen 1999, Didriksen 2010) provide actions for inserting dependency links between two words and for producing a single dependency analysis for each sentence.
  - Finite-state automata are used in some CG implementations (Hulden 2011, Yli-Jyrä 2011).
- *Finite-state intersection grammar (FSIG)* has been used to parse dependency structures of varying specificity and complexity.
  - Koskenniemi et al. (1992) denote the syntactic functions of words with tags that additionally specify the direction of the possible governors, leaving possible attachment ambiguity unresolved.
  - Yli-Jyrä (2005) encodes every dependency link with a pair of brackets between the nodes. With such encoding, every projective dependency grammar is representable with an intersection of a strictly locally testable regular language and a language that balances labeled brackets. The representation has efficient implementations, but the grammar semantics is based on inviolable properties of the parses.
- *Constraint network parsers* combine consistency-enforcing methods with backtracking search in order to resolve ambiguity and to produce parses as search results.
  - Maruyama (1990) presents a constraint network parser that can produce non-projective dependency graphs.

  - Debusmann et al. (2004) implemented a dependency parser whose constraint network can be extended with word order and dominance constraints.

- *Finite-state cascades* are used in deterministic parsing approaches:
  - Joshi (1996) describes retrospectively a parser (from 1959) that used finite-state cascades. Each level in the cascade corresponded to a deterministic finite-state transducer that read the input either left to right or right to left and marked syntactic units with various kinds of brackets.
  - Abney (1996) also applies finite-state cascades to phrase structure analysis.
- *Iterated finite-state transducers* can bind the rule applications with movable markers. The parsing terminates if a fixed point is reached.
  - Roche (1997) iterates finite-state transducers in order to parse context-free grammars, transformation grammars and tree adjoining grammars. The approach does not include hierarchical ambiguity packing, but it demonstrates the computational power of iteration.
  - Elworthy (2000) uses iterated deterministic finite-state transducers that are augmented with instructions that insert links to the read string. Elworthy's deterministic finite-state parser includes an ambiguity-packing mechanism that adds multiple heads to phrases to avoid the attachment ambiguity. Thanks to the deterministic parsing that does not elaborate all ambiguity, the parsing time is $O(n^2)$ for an input of n words.
  - Oflazer (2003) uses an iterated finite-state transducer that implements projective dependency parsing. The approach is robust but does not include hierarchical ambiguity packing.
- *Bilexical dependency parsers* can carry out projective dependency analysis without lexical functions (Eisner 1997).
- *Restarting automata* perform a sequence of monotonic rewrite steps that reduce the length or weight of the input tape. Plátek et al. (2003) motivate restarting automata as a tool for dependency analysis.

6.1 The Distinctive Characteristics of the Current Approach

Although it is partially similar to the prior approaches, the currently presented algorithm has clear distinctive characteristics that make it new as for now.

In comparison with most dependency parsers, the current system differs by assuming a *parametric bound for the number of overlapping arcs*. The time complexity is similar to Elworthy's parser, but the method computes implicitly all *parameterized parses*.

The iterated application of Reduce reminds us of the fixpoint method (Roche 1997) and of finite-state cascades (Abney 1996). A striking difference from them is that the current (bottom-up) cascade produces nothing as its output.

Some of the cascading parsers resolve the ambiguity on the basis of deterministic heuristics and underspecification, while the current system *resolves the ambiguity on the basis of the lexical categories (functional tags or bilexical pairs), the performance constraints, and the weights*.

The analysis-by-elimination approach of the current system reminds of one-level intersection grammars that assume a set of candidate parses as their input. In contrast to the early practice in FSIG (Voutilainen 1994), the dependency bracketing of the current system specifies a *full syntactic tree*.

The author has used a similar encoding for trees in an earlier regular approximation method for dependency grammars (Yli-Jyrä 2005). However, the current work operates on *weighted rational relations* rather than parallel constraints.

The use of rewrites rather than constraints as a means for validating the arcs is familiar from Oflazer's dependency parser (Oflazer 2003). However, the new parser *contracts the validated brackets* and *memoizes the intermediate results* of the cascade, which improves efficiency.

Parsing by contractions is a familiar approach from restarting automata and contextual grammars. It is not yet known if the currently presented *memoization technique* is completely new in the context of restarting automata, but it may prove useful in practice. The current contractions operate *directly* on the encoded dependency trees and there is a performance *limit for overlapping rule applications*.

Deterministic contractions and functional rational relations are also a natural approach to Constraint Grammar parsing. However, the current approach *manages sentence-level ambiguity* and *combines deterministic contractions with full parsing*.

7 Conclusions

The paper has described a new approach to dependency parsing. The presented finite-state approach uses three new techniques: dependency bracketing, bracketed arc contraction and cascade memoization. The paper has presented the final parsing algorithm of Sect. 5.4 via an abstract calculus of weighted rational relations and motivated its efficiency through a series of experiments and design choices. In addition, we provided new interpretations for the integer sequences A001764 and A000295, which might be of interest in applied mathematics.

7.1 Practical Benefits

In the case of projective parsing, the proposed memoizing parsing algorithm produces optimal parses and is efficient: its time complexity appears to be in $O(t\,n^2)$ according to the analysis of the method that uses FreeReduce and $O(t^2 n^3)$ according to the preliminary analysis of the method that uses Reduce.

The proposed parser can be tailored for functional and bilexical dependency parsing. Under the performance-motivated parameter t for the overlapping arcs, the parse forest contains all plausible parses of the projective grammar. The parse forest is extendible to non-projective trees that contain crossing arcs.

The method has a rational design and it is easy to implement with finite-state methods. The packed weighted parse forest is computed through composition and projection, two commonly used high-level finite-state operations, and the memoization of the internals of the cascade allows for efficient retrieval of the parses.

7.2 Further Work

There are several directions for further study. (i) The weight structure could be generalized to arbitrary semirings in order to enable the generality of "semiring parsing". (ii) A statistical parser will have to explicate how the weights in Grammar are set and whether they behave like probabilities or indicate some other kinds of weights. Furthermore, the actual implementation of the current illustrative system would replace the semiring of the nonnegative real numbers with the tropical semiring of their negative logarithms (Mohri 2009) in order to improve the numerical stability of the algorithm. (iii) The use of non-projective contractions should be studied further. There are certainly some strategies to reduce the number of non-projective parses while maintaining high recall. (iv) More insight into the packed forest and the growth of the intermediate results is needed. The current experiments were based on unweighted bilexical grammars where all dependencies were possible. In practice, the possible argument structures are more specific, which makes the average case more interesting than the limited experimental results provided so far. (v) The current method throws away all partial parses. For text parsing purposes, the parser can be modified to allow dependency graphs that consist of unconnected trees.

The possible extensions of the presented method include the intriguing option of combining statistics and linguistic knowledge into the same system. Adding handwritten linguistic constraints to Grammar is technically possible and would allow human interventions to complement statistically estimated parameters and would help us finish the precision and recall of the practical implementation of the approach.

Acknowledgements The research has been made possible by the Academy of Finland grant number 128536 "Open and Language Independent Automata-Based Resource Production Methods for Common Language Research Infrastructure", and, more recently, by the FIN-CLARIN project steered by Krister Lindén. Kimmo Koskenniemi, Pasi Tapanainen, Atro Voutilainen and Lauri Carlson supported my first investigations into contractions in finite-state intersection parsing since 1995. More recently, my thinking has benefitted from several related discussions with Carlos Gómez-Rodríguez, Jason Eisner, Joakim Nivre, Marco Kuhlmann, and John Hale. During the multi-year creative process, I felt several times need for heavenly empowerment. I look gratefully back to every inspired moment.

I am indebted to the prior anonymous reviewers of the CIAA 2011 and FSMNLP 2011 meetings, as well as Aarne Ranta, Wanjiku Nganga, Jussi Piitulainen, and Miikka Silfverberg for their valuable comments pointing out many areas for further study. The remaining imperfections in the text are mine, of course.

References

Abney, Steven. 1996. Partial parsing via finite state cascades. In *Proceedings of the ESSLLI'96 Robust Parsing Workshop*. Prague, Czech.

Debusmann, Ralph, Denys Duchier, and Geert-Jan M. Kruijff. 2004. Extensible dependency grammar: A new methodology. In *Proceedings of the COLING 2004 Workshop of Recent Advances in Dependency Grammar*, ed. Geert-Jan M. Kruijff and Denys Duchier, 78–84. Geneva, Switzerland.

Didriksen, Tino. 2010. *Constraint Grammar Manual: 3rd version of the CG formalism variant*. GrammarSoft ApS, Denmark. http://beta.visl.sdu.dk/cg3/vislcg3.pdf.

Eisner, Jason. 1997. Bilexical grammars and a cubic-time probabilistic parser. In *Proceedings of the 4th International Workshop on Parsing Technologies*, 54–65. MIT, Cambridge, MA.

Elworthy, David. 2000. A finite state parser with dependency structure output. In *Proceedings of Sixth International Workshop on Parsing Technologies (IWPT 2000)*. Trento, Italy: Institute for Scientific and Technological Research.

Gómez-Rodríguez, Carlos, 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)*, 1492–1501. Uppsala, Sweden.

Gómez-Rodríguez, Carlos, David Weir, and John Carroll. 2009. Parsing mildly non-projective dependency structures. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2009)*, 291–299.

Hulden, Mans. 2011. Constraint Grammar parsing with left and right sequential finite transducers. In *Proceedings of the 9th International Workshop on Finite State Methods and Natural Language Processing (FSMNLP 2011)*, 39–47. Blois, France: Association for Computational Linguistics. http://www.aclweb.org/anthology/W11-4406.

Joshi, Aravind K. 1996. A parser from antiquity: an early application of finite state transducers to natural language parsing. In *Extended Finite State Models of Language, Proceedings of the ECAI'96 Workshop*, ed. András Kornai, Studies in Natural Language Processing, 33–34. Cambridge University Press.

Koskenniemi, Kimmo, Pasi Tapanainen, and Atro Voutilainen. 1992. Compiling and using finite-state syntactic rules. In *14th COLING 1992, Proceedings of the Conference*, volume 1, 156–162. Nantes, France.

Kuhlmann, Marco. 2010. *Dependency structures and lexicalized grammars. An algebraic approach*, volume 6270 of *Lecture Notes in Artificial Intelligence, FoLLI Publications on Logic, Language and Information*. Berlin, Heidelberg: Springer.

Kuhlmann, Marco, and Giorgio Satta. 2009. Treebank grammar techniques for non-projective dependency parsing. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics (EACL'09)*, 478–486.

Lombardo, Vincenzo, and Leonardo Lesmo. 1996. An Earley-type recognizer for dependency grammar. In *16th COLING, Proceedings of the Conference*, volume 2, 723–728. Copenhagen, Denmark.

Maruyama, Hiroshi. 1990. Structural disambiguation with constraint propagation. In *28th ACL 1989, Proceedings of the Conference*, 31–38. Pittsburgh, Pennsylvania.

Mohri, Mehryar. 2009. Weighted automata algorithms. In *Handbook of weighted automata*, ed. Manfred Droste, Werner Kuich, and Heiko Vogler, 213–254. Springer.

Neuhaus, Peter, and Norbert Bröker. 1997. The complexity of recognition of linguistically adequate dependency grammars. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and the 8th Conf. of the European Chapter of the Association for Computational Linguistics*, 337–343. Madrid, Spain.

Nivre, Joakim. 2006. Constraints on non-projective dependency parsing. In *Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2006)*, 73–80.

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

Oflazer, Kemal. 2003. Dependency parsing with an extended finite-state approach. *Computational Linguistics* 29:515–544.

Plátek, Martin, Markéta Lopatková, and Karel Oliva. 2003. Restarting automata: motivations and applications. In *Workshop 'Petrinetze' and 13. Theorietag 'Formale Sprachen und Automaten'*, ed. M. Holzer, 90–96. Institut für Informatik, Technische Universität München.

Roche, Emmanuel. 1997. Parsing with finite-state transducers. In *Finite-state language processing*, ed. Emmanuel Roche and Yves Schabes, chapter 8, 241–281. Cambridge, Massachusetts: MIT Press.

Tapanainen, Pasi. 1999. Parsing in two frameworks: finite-state and functional dependency grammar. Doctoral Dissertation, University of Helsinki, Finland.

Tesnière, Lucien. 1959. *Éléments de syntaxe structurale*. Paris: Éditions Klincksieck.

Voutilainen, Atro. 1994. *Designing a parsing grammar*. Number 22 in Publications of the Department of General Linguistics, University of Helsinki. Helsinki, Finland: Yliopistopaino.

Yli-Jyrä, Anssi Mikael. 2003. Multiplanarity – a model for dependency structures in treebanks. In *TLT 2003. Proceedings of the Second Workshop on Treebanks and Linguistic Theories*, ed. Joakim Nivre and Erhard Hinrichs, volume 9 of *Mathematical Modelling in Physics, Engineering and Cognitive Sciences*, 189–200. Växjö, Sweden: Växjö University Press.

Yli-Jyrä, Anssi. 2004. Axiomatization of restricted non-projective dependency trees through finite-state constraints that analyse crossing bracketings. In *Proceedings of the COLING 2004 Workshop of Recent Advances in Dependency Grammar*, 33–40. Geneva, Switzerland.

Yli-Jyrä, Anssi. 2005. Approximating dependency grammars through intersection of star-free regular languages. *International Journal of Foundations of Computer Science* 16:565–579.

Yli-Jyrä, Anssi. 2008. Transducers from parallel replace rules and modes with generalized lenient composition. In *Finite-State Methods and Natural Language Processing, 6th International Workshop, FSMNLP-2007. Revised Papers*, 197–212. Potsdam, Germany: Potsdam University Press.

Yli-Jyrä, Anssi. 2011. An efficient constraint grammar parser based on inward deterministic automata. In *Proceedings of the NODALIDA 2011 Workshop Constraint Grammar Applications*, ed. Eckhard Bick, Kristin Hagen, Kaili Müürisep, and Trond Trosterud, volume 14 of *NEALT Proceedings Series*, 50–60.

Yli-Jyrä, Anssi, and Kimmo Koskenniemi. 2004. Compiling contextual restrictions on strings into finite-state automata. In *Proceedings of the Eindhoven FASTAR Days 2004*, ed. Loek Cleophas and Bruce W. Watson, number 04-40 in Computer Science Reports. Eindhoven, The Netherlands: Technische Universiteit Eindhoven.

Yli-Jyrä, Anssi, and Matti Nykänen. 2004. A hierarchy of mildly context sensitive dependency grammars. In *Proceedings of the 9th conference on Formal Grammar (FGNancy 2004)*, ed. Gerald Penn, Gerhard Jäger, Paola Monachesi, and Shuly Wintner, 151–165.