Transforming Dependency Structures to LTAG Derivation Trees

Caio Corro **Joseph Le Roux**
Laboratoire d'Informatique de Paris Nord,
Université Paris 13 – SPC, CNRS UMR 7030,
F-93430, Villetaneuse, France
{corro,leroux}@lipn.fr

Abstract

We propose a new algorithm for parsing Lexicalized Tree Adjoining Grammars (LTAGs) which uses pre-assigned bilexical dependency relations as a filter. That is, given a sentence and its corresponding well-formed dependency structure, the parser assigns elementary trees to words of the sentence and return attachment sites compatible with these elementary trees and predefined dependencies. Moreover, we prove that this algorithm has a linear-time complexity in the input length. This algorithm returns all compatible derivation trees as a packed forest. This result is of practical interest to the development of efficient weighted LTAG parsers based on derivation tree decoding.

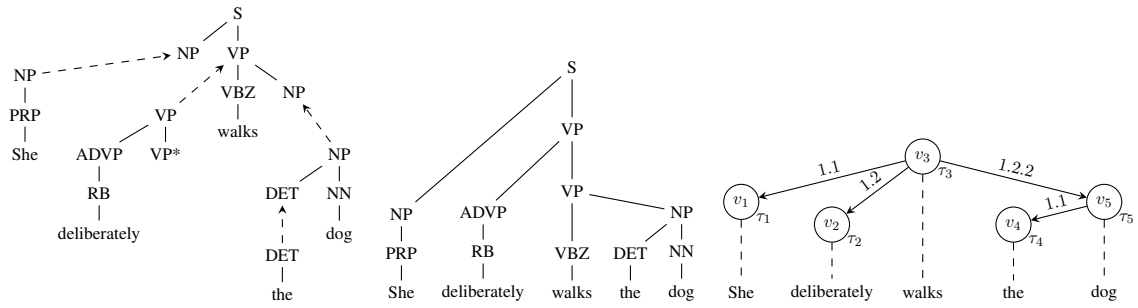
1 Introduction

Lexicalized Tree Adjoining Grammars (LTAGs), that is TAGs where each elementary tree contains exactly one lexical anchor, have been proposed as an attractive formalism to model the phrase-structure construction in natural languages (Schabes et al., 1988; Abeille et al., 1990). An important property of lexicalized grammars is their ability to directly encode semantic information in combination operations. Borrowing the example provided by Eisner and Satta (2000), in the sentence "`She deliberately walks the dog`", the tree anchored with "`dog`" is combined to the tree anchored with "`walks`", see Figure 1a.[1] Thus, the object associated with a transitive realization "`walks`" can be restricted to a subset of allowed words, including "`dog`" but not "`river`".

Unfortunately, parsing with a LTAG is hardly tractable. Eisner and Satta (2000) proposed the best-known parsing strategy with a $\mathcal{O}(n^7)$ worst case time complexity and $\mathcal{O}(n^5)$ space complexity where n is the length of the input sentence. This is a major drawback for downstream applications where speed and low memory use is important. Moreover, by reducing boolean matrix multiplication to TAG parsing, Satta (1994) argued that obtaining a lower complexity bound for the latter problem is unlikely to be straightforward. Hence, parsing with weighted TAGs (Resnik, 1992) has received too little attention even though Chiang (2000) experimentally demonstrated their usefulness in the Penn Treebank parsing task. In order to bypass this major bottleneck, two main strategies have been explored. On the one hand, splittable grammars (Schabes and C. Waters, 1995; Carreras et al., 2008) are interesting because they have a lower asymptotic complexity than LTAGs. However, they cannot directly encode several properties that make TAGs linguistically plausible, such as cross-serial dependencies. In fact, they are restricted to context-free languages. On the other hand, a popular approach to speed up LTAG parsing is to include a preliminary step called supertagging: only a subset of tree fragments per word are retained as candidates, or exactly one in the most aggressive form (Chen and Bangalore, 1999). However, this pruning does not improve the asymptotic complexity of the parser.[2] Moreover, it is usually performed via local methods that can hardly take into account long distance relationships since lexical dependencies are unknown at this step. For instance, in the sentence "`She deliberately walks, despite her hatred for quadruped mammals, the dog`", capturing the transitive nature of the first verb is difficult without further analysis (Bonfante et al., 2009).

[1] We use a simplified grammar to ease the presentation.

[2] Intuitively, even when lexicon assigns only one elementary tree per word, a sentence can have several derivations which exhibit different tree structures.



(a) Sentence construction with a LTAG (b) Resulting phrase-structure (c) Associated derivation tree

Figure 1: Phrase-structure of the sentence "`She deliberately walks the dog`", its construction thanks to a LTAG and the associated derivation tree. Dashed arrows in the left figure represent combination operations. Note that the derivation tree is a a labelled dependency structure.

In this work, we explore a component for a new alternative in fast and efficient LTAG parsing. Dependency structures can be interpreted as sets of derivation trees which share lexical composition operations. This property has already been investigated in order to propose efficient phrase-structure parsers under several formalisms (Section 2). The derivation tree induced by an LTAG analysis, as shown in Figure 1c, is a dependency structure exposing bilexical dependencies and labelled with elementary trees and operation sites (Section 3). As such, LTAG parsing can be seen as dependency parsing where both valid tree structures and valid labellings are constrained. Following the pipeline used in most labeled dependency parsers:

1. a parser starts by assigning a single head to each lexical item, without taking into account the grammar;

2. then, a parse labeler assigns elementary trees and operation sites.

Regarding the first step, seminal work of Bodirsky et al. (2005) showed that, ignoring labels, the derivation tree is an arborescence[3] that can be characterized thanks to two structural properties: 2-bounded block degree and well-nestedness. Gómez-Rodríguez et al. (2009) introduced an algorithm with a $\mathcal{O}(n^7)$ time complexity for decoding this type of structures. Still, this is a dependency parsing problem hardly tractable for sentences longer than $15 \sim 20$ words. More recently, Corro et al. (2016) proposed an experimentally fast alternative based on combinatorial optimization and the maximum spanning arborescence problem. Either way, this means that it is now possible to obtain dependency trees that are compatible with LTAG parsing.

Still, we are left with the second step: the parse labeler. Our contribution is a novel algorithm for this second step that can infer elementary trees and operation sites (Sections 4 and 5) as a post-processing step to build all compatible derivation trees, from which derived trees are completely specified. The time complexity in the length of the sentence is linear (Section 6).

2 Related work

Syntactic content must not be confused with the type of representation, as clearly argued by Rambow (2010). As such, phrase structures can be encoded into dependency-like structures as long as the transformation is correctly formalized. One example of this fact is the correspondence between derivation trees constructed from lexicalized grammar and bilexical dependency relations. However, the equivalence with dependency-based linguistic theories may not be as straightforward as the similarity in the type of representation suggests (Rambow and Joshi, 1997; Kallmeyer and Kuhlmann, 2012). Following this line of thought, we reduce LTAG parsing to dependency parsing where unannotated bilexical dependencies represents abstract attachment operations and dependency labels specify the type of attachment and the site of the operation on the head elementary tree.

Historically, many syntactic dependency treebanks have been built by transforming phrase-structures using head-percolation tables (Collins, 2003; Yamada and Matsumoto, 2003), and thus one could learn a dependency parser from a

[3] In the rest of the paper, we use *tree* to denote the linguistic object and *arborescence* to denote the graph-theoretic object. The term *tree* may be confusing because in graph theory it refers to an undirected type of graph.

phrase-structure treebank. More Recently, following this observation, Fernández-González and Martins (2015) proposed to encode constituents as bilexical dependency labels without relying on a grammar, solving the two problems jointly. In this setting, constituent parse tree is recovered from a labeled dependency parse, which makes it possible to use tools developed for dependency parsing. This is appealing because dependency parsing has received a considerable amount of attention recently and is now well understood. Moreover, dependency parsing is usually less complex than phrase-structure parsing, at least in practice if not asymptotically. This technique has been showed to be experimentally efficient but it may be argued that the lack of an explicit grammar may result in non-interpretable structures. For instance, the valency of a verbal predicate may be incorrect.

Another line of work used predicted bilexical dependencies as a filter for a standard grammar parser. Recently, Kong et al. (2015) applied this technique to Context-Free Grammars with lexical projections, ie. head-projections are specified in right-hand sides of production rules (Collins, 2003). The resulting algorithm is certified to have quadratic worst time-complexity but, in practice, they observed a linear running-time with respect to the sentence length. Our method is very close but with one major difference. In an LTAG derivation each word is assigned a single elementary tree, while in CFGs with lexical projections a word can anchor several rules. This observation provides us with way to improve the worst-case time complexity of our algorithm from cubic to linear with regards to the input length. We believe that this observation could also be adapted to the framework of Kong et al. (2015). Also, we can interpret the low-order dependency parsing model explored by Carreras et al. (2008) as a similar filtering procedure. However, they did not study the theoretical complexity benefit of their approach. Note that both of these works only handle context-free languages whereas we focus on a mildly-context sensitive formalism.

Guided parsing has been studied for TAGs and related formalisms like RCGs (Barthélemy et al., 2001). Our approach differs because the predefined dependency tree cannot be considered as a guide, nor an oracle, since it is not required to be a superset of the set of possible derivations. Also, we could see our method as an algorithm to compute the intersection of the tree language defined by an LTAG and the tree language consisting of a single dependency tree. In this way our method can be seen as an instance of the framework defined in (Nederhof, 2009).

Finally, we follow the common idea that derivation trees should be built directly and that a derived tree is a by-product of a derivation tree. See for instance Debusmann et al. (2004) which introduced LTAG parsing as a constraint satisfaction problem.

3 Lexicalized Tree Adjoining Grammar

In this section, we describe notions related to LTAG parsing that we will use to expose the parsing algorithm in Section 4. We assume the reader is familiar with the TAG formalism (Joshi, 1987; Joshi and Schabes, 1992). We start by defining TAGs, then we introduce the structure of derivation trees and finally we give an overview of parsing with LTAGs.

3.1 Definition

We define a LTAG as a tuple $\langle N, T, \Gamma^I, \Gamma^A, \Gamma^S, f_{OA}, f_{SA}, f_{SS} \rangle$ where:

- N and T are disjoint finite sets of non-terminals (constituents) and terminals (words), respectively;
- Γ^I and Γ^A are the finite sets of initial and auxiliary trees, respectively, built upon N and T and we define the set of elementary trees as $\Gamma \triangleq \Gamma^I \cup \Gamma^A$;
- $\Gamma^S \subseteq \Gamma^I$ is the set of start trees.

We use Gorn addresses[4] to index nodes and $p \in \tau$ is true if and only if address p exists in tree τ. We use the term *site* to refer to both the Gorn address and the corresponding node.

Each node of the elementary tree $\tau \in \Gamma$ is labelled with a non-terminal except exactly one leaf labelled with the terminal whose address is denoted $lex(\tau)$, called the *lexical anchor*. For any auxiliary tree $\tau \in \Gamma^A$, the address of its mandatory foot node is written $foot(\tau)$. Without loss of generality, we restrict Γ to binary trees only.

In order to control combination operations over nodes in Γ, we use the following functions:

- $f_{OA} : \Gamma \times \mathbb{Z}^+ \rightarrow \mathbb{B}$ specifies whether adjunction is obligatory at a site in a tree or not;[5]

[4] A Gorn address for a node is a sequence of integers from $\mathbb{Z}^+$ indicating a path from the root to the addressed node.

[5] $\mathbb{B}$ is the set containing boolean values true and false.

- $f_{SA} : \Gamma \times \mathbb{Z}^+ \times \Gamma^A \rightarrow \mathbb{B}$ specifies the trees that can be adjoined at a site;
- $f_{SS} : \Gamma \times \mathbb{Z}^+ \times \Gamma^I \rightarrow \mathbb{B}$ is a similar function for substitution.

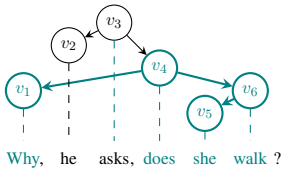
To simply notation, we also use function f_{SA} and f_{SS} in order to check if a site is an adjunction or substitution site, and thus assume these functions check if the given site is a leaf or not.

The derived tree is built by combining elementary trees thanks to combination operations: substitution and adjunction. In this paper, we have to distinguish three different types of adjunction. A *wrapping adjunction* is the attachment of a subanalysis which has lexical anchors on both sides of the adjoined auxiliary tree's foot node. Similarly, *left adjunction* and *right adjunction* restrict lexical anchors to the left and right of the foot node, respectively. [6]

3.2 Derivation tree

Given a sentence $s = s_1 \dots s_n$ and a LTAG, a derivation tree is a dependency structure representing a valid parse of the sentence by means of elementary trees and combination operations. It is a directed graph $G = (V, A)$ with $V = \{v_1 \dots v_n\}$ the set of vertices, v_i corresponding to word s_i. The set of arcs $A \subset V \times V$ describes a spanning arborescence: A contains $n-1$ arcs with no circuit and each vertex has at most one incoming arc. The unique vertex v_r without incoming arc is called the root vertex. The set of children of a vertex is defined as $children(v_h) = \{v_m | v_h \rightarrow v_m \in A\}$. Moreover, vertices are labelled with elementary trees and arcs with Gorn addresses. Vertex labels indicate supertag assignment, while arc labels specify adjunction or substitution sites. The derivation tree contains all necessary information to construct the derived tree. See Figure 1b and 1c for an example. Different derivation trees can induce the same derived tree.

We borrow notation and definitions from Corro et al. (2016). The *predecessor* of a vertex $v_i \in V$ is v_{i-1}. We refer to the vertex with the smaller (resp. larger) index in a set as the *leftmost* (resp. *rightmost*) vertex. The *yield* of a vertex $v_k \in V$ is the set of vertices reachable from v_k with respect to A, including itself. We note $(v_k)_\Leftarrow$ and $(v_k)_\Rightarrow$ the leftmost and rightmost vertices in the yield of v_k respectively. The *span* of v_k corresponds to the set of vertices delimited by $(v_k)_\Leftarrow$ and $(v_k)_\Rightarrow$, possibly including vertices which are not in the yield of v_k. The *block degree* of a vertex set $W \subseteq V$ is the number of vertices of W without a predecessor inside W. The block degree of a vertex is the block degree of its yield and the block degree of an arborescence is the maximum block degree of its incident vertices.



Notation	Value
$(v_4)_\Leftarrow$	v_1
$(v_4)_\Rightarrow$	v_6
$(v_4)_\leftarrow$	v_2
$(v_4)_\rightarrow$	v_3
$(v_4)_\uparrow$	v_3

Figure 2: Example of a 2-bounded block degree and well-nested dependency structure. The right table exposes notation we use for information we can extract about vertex v_4.

Traditionally, dependency structures have been characterized as either projective (all vertices have a yield equal to their span) or non-projective (any arborescence). Previous work of Bodirsky et al. (2005) showed that the structure of G is highly constrained and can be characterized thanks to two finer-grained structural properties. First, G has a *2-bounded block degree*, meaning the block degree of G is less or equal to two. Given a vertex $v_k \in V$ with a block degree of 2, its *gap* is the vertex set of block degree 1 including a vertex with its predecessor and one with its successor in the yield of v_k. We note $(v_k)_\leftarrow$ and $(v_k)_\rightarrow$ the leftmost and rightmost nodes in the gap of v_k, respectively, $(v_k)_\leftarrow = (v_k)_\rightarrow = -$ if there is none (ie. v_k as a block degree of 1, thus its yield is equal to its span and contains no gap). An example is illustrated in Figure 2. Second, G is well-nested, that is two distinct sub-trees may not interleave. We will not explicitly use this property in the following and refer readers interested dependency structures characterization to Kuhlmann and Nivre (2006). However, we assume dependency structures to be well-nested.[7]

3.3 Parsing

So far we defined LTAGs and discussed the structure of derivation trees. We will now briefly focus

[6] These should not be confused with wrapping, left and right auxiliary trees which only describe the structure of auxiliary trees.

[7] More precisely, the algorithm proposed in Section 4 cannot parse ill-nested arborescences. Well-nestedness is a required property of the input.

on the parsing problem.

Given a sentence, this term may typically refer to three different but strongly related tasks via the notion of semi-ring parsing (Goodman, 1999). First, recognition: can this sentence be generated by a given grammar? Second, derivation forest parsing: decoding the set of all possible derivation trees. Third, weighted disambiguation: what is the best parse in the derivation forest given a scoring model? Our work was motivated by the weighted framework, but can generalize to others with some limitation.

Thus, a parser takes as input a grammar and a sentence. Because the LTAG chart-based parser has a high complexity, a standard approach is to use a pipeline system. First, a labeler assigns a single elementary tree to each lexical item. Then, the chart-based parser is run. This is merely a beam approach but it does not impact the assymptotic complexity of the second step. We propose to reverse this standard pipeline. In the first stage of the pipeline, which we consider already performed, a dependency parser assigns bilexical relations that we interpret as an abstract, ie. unlabelled, derivation tree. Contrary to the standard pipeline, this first step do not take into account the grammar, which lead to the development of an efficient parser based on combinatorial optimization (Corro et al., 2016). Next, the linear-time algorithm exposed in Section 4 can be used to assign elementary trees and operation sites.

4 Parsing with given bilexical relations

Not every labelled graph G describing an arborescence is a valid dependency tree. Indeed, many constraints must be satisfied in order to transform the derivation tree to into a derived tree. Since an arc $v_h \rightarrow v_m$ represents a combination operation of the tree of v_m into the tree of v_h, non-terminals at attachment sites must be equal and if the destination site is a leaf (resp. internal) node, the tree of v_m must be an initial (resp. auxiliary) tree, among others.

In this section, we propose a new algorithm for LTAG parsing with given bilexical relations in the form of a deduction system (Pereira and Warren, 1983). We formalize the algorithm as a recognizer: the **Goal** item can only be obtained if the input can be generated by the grammar. As usual, it can be implemented as a chart-based dynamic program with back-pointers in order to retrieve every allowed derivation tree. Moreover, rules can be augmented with scores to build a weighted parser (Goodman, 1999).

Intuitively, our algorithm is an adaptation of the standard CYK variant for TAG parsing (Vijay-Shankar and K. Joshi, 1986; Vijay-Shanker and J. Weir, 1993), extended to handle constraints of lexicalized grammars. It is bottom-up in two ways. First, vertices of the dependency structure are traversed from leaves to root: a vertex is considered only after all its children. Second, given a vertex of the dependency structure, its corresponding elementary tree is visited in a similar fashion as in the CYK-like algorithm. The resulting algorithm has a linear time complexity. This is due to the fact that the maximum number of operations on a given elementary tree is bounded by its size. Thus, given a word, its maximum number of modifiers is not bounded by the sentence length but by the grammar, which leads to a tighter asymptotic complexity.

4.1 Item definition

Given a LTAG $\langle N, T, \Gamma^I, \Gamma^A, \Gamma^S, f_{OA}, f_{SA}, f_{SS} \rangle$, a sentence $s = s_1 \dots s_n$ and the corresponding dependency structure $G = (V, A)$, items are 6-tuples of the form $[v_h, \tau, p, c, b_l, b_r]$ with:

1. **considered vertex** $v_h \in V$ in the dependency structure, corresponding to word s_h;
2. **elementary tree** $\tau \in \Gamma$, indicating the association of the anchor of τ with word s_h;
3. **Gorn address** $p \in \tau$ of a node in the elementary tree;
4. **combination flag** $c \in \{\bot, \top\}$ specifying if a combination operation has already been investigated $\top$ or not $\bot$ at node p;
5. **left boundary** $b_l \in V \cup \{l_h, g_h, \overleftarrow{g_h}, \overrightarrow{g_h}\}$ defines the left boundary of the yield of the item, which is discussed in more details below;
6. **right boundary** $b_r \in V \cup \{l_h, g_h, \overleftarrow{g_h}, \overrightarrow{g_h}\}$ defines its right boundary.

In most approaches, boundaries of the yield are given using integer indices. Instead, we use either vertices V or special values l_h, g_h, $\overleftarrow{g_h}$ and $\overrightarrow{g_h}$.[8]

[8] In the literature, the yield is often defined as a pair $\langle i, j \rangle$ meaning words from s_{i+1} to s_j are covered. We do not follow this convention: a yield $\langle i, j \rangle$ indicates that words from s_i to s_j are covered.

If the left boundary of an item is node $v_m \in V$, the left (resp. right) boundary *position* is given by $(v_m)_\Leftarrow$ (resp. $(v_m)_\Rightarrow$). Special value l_h is used to indicate that the lexical anchor is used as a boundary, thus we define $(l_h)_\Leftarrow \triangleq h$ and $(l_h)_\Rightarrow \triangleq h$. The remaining values are used to indicate that the boundary is determined by the foot node span. The boundary is set to special value g_h if and only if the span of v_h has a gap, ie. $(v_h)_\leftarrow \neq -$. Simply, we set $(g_h)_\Leftarrow \triangleq (v_h)_\leftarrow$ and $(g_h)_\Rightarrow \triangleq (v_h)_\rightarrow$. Finally, note that vertices which do not have a gap in their span, can still be combined through left or right adjunction (see Figure 1c). Thus, we use $\overleftarrow{g_h}$ (resp. $\overrightarrow{g_h}$) in order to qualify the boundary of an sub-analysis which targets to be combined through a left (resp. right) adjunction. We set $(\overleftarrow{g_h})_\Leftarrow \triangleq (v_h)_\Rightarrow + 1$ and $(\overrightarrow{g_h})_\Rightarrow \triangleq (v_h)_\Leftarrow - 1$. Note that $(\overleftarrow{g_h})_\Rightarrow$ and $(\overrightarrow{g_h})_\Leftarrow$ are undefined, meaning that rules which use these values can not be applied.

Tree τ is a candidate for word represented by vertex v_h and its dependants if we can go up at its root node with boundaries equals to the span of v_h. In order to simplify notation, we use intermediate items to represent them. If the item has both boundaries defined, then:

Full:

$$\frac{[v_h, \tau, 1, \top, b_l, b_r]}{[v_h, \tau]} \begin{array}{l} (b_l)_\Leftarrow = (v_h)_\Leftarrow, \\ (b_r)_\Rightarrow = (v_h)_\Rightarrow \end{array}$$

If $h \neq r$, this item will be a candidate to combination through substitution (resp. adjunction) if $\tau \in T^I$ (resp. $\tau \in T^A$), or similarly, if v_h has no gap (resp. has a gap). Two other intermediate items are used specifically to indicate that they are meaning to be combined through left and right adjunction:

Full left:

$$\frac{[v_h, \tau, 1, \top, b_l, \overleftarrow{g_h}]}{[v_h, \tau, \leftarrow]} \; (b_l)_\Leftarrow = (v_h)_\Leftarrow$$

Full right:

$$\frac{[v_h, \tau, 1, \top, \overrightarrow{g_h}, b_r]}{[v_h, \tau, \rightarrow]} \; (b_r)_\Rightarrow = (v_h)_\Rightarrow$$

A triplet item ending with the $\leftarrow$ (resp. $\rightarrow$) symbol is a candidate for left (resp. right) adjunction. Obviously, τ in the antecedent of both rules must be an auxiliary tree. This is constrained by the **Foot scan** rule introduced in the following subsection.

4.2 Axioms and goal

The first axiom is:

Lex scan:

$$\frac{}{[v_h, \tau, p, \top, l_h, l_h]} \begin{array}{l} lex(\tau) = p, \\ \tau(p) = s_h \end{array}$$

meaning, for each vertex v_h and elementary tree τ, we instantiate items with compatible elementary trees, starting at the lexical anchor address. Moreover, we create items at the foot position of an auxiliary tree for vertices with a gap in their span:

Foot scan:

$$\frac{}{[v_h, \tau, p, \top, g_h, g_h]} \begin{array}{l} \tau \in \Gamma^A, \\ foot(\tau) = p, \\ (v_h)_\leftarrow \neq - \end{array}$$

Finally, the two last axioms are used to predict possible trees, on vertices without gap, that will be combined through left or right adjunction:

Foot scan left:

$$\frac{}{[v_h, \tau, p\top, \overleftarrow{g}_h, \overleftarrow{g}_h]} \begin{array}{l} \tau \in \Gamma^A, \\ foot(\tau) = p, \\ (v_h)_\leftarrow = - \end{array}$$

Foot scan right:

$$\frac{}{[v_h, \tau, p\top, \overrightarrow{g}_h, \overrightarrow{g}_h]} \begin{array}{l} \tau \in \Gamma^A, \\ foot(\tau) = p, \\ (v_h)_\leftarrow = - \end{array}$$

A proof completes if any tree $\tau \in S$ is a candidate for the root vertex v_r of the dependency structure:

Goal:

$$\frac{[v_r, \tau]}{\;} \; \tau \in \tau^S$$

In the rest of this section, we describe rules governing allowed deductions.

4.3 Traversal rules

We start with tree traversal.[9] Obviously, the premise of any move operation is that we already checked any potential operation, marked by the $\top$ flag. Given address $p \cdot 1$ in tree τ, we consider two cases. First, if node $p \cdot 1$ do not have any sibling, ie. $p \cdot 2 \notin \tau$:

Move unary:

$$\frac{[v_h, \tau, p \cdot 1, \top, b_l, b_r]}{[v_h, \tau, p, \bot, b_l, b_r]} \; (p \cdot 2) \notin \tau$$

Secondly, if $p \cdot 2$ exists, the siblings must share the same frontier:

[9] We assume binary elementary trees in the following presentation, but this can be generalized to other tree structures.

Move binary:

$$\frac{[v_h, \tau, p \cdot 1, \top, b_{l1}, b_{r1}] \quad [v_h, \tau, p \cdot 2, \top, b_{l2}, b_{r2}]}{[v_h, \tau, p, \bot, b_{l1}, b_{r2}]} \; (b_{r1})_{\Rightarrow} + 1 = (b_{l2})_{\Leftarrow}$$

4.4 Combination rules

Finally, let us concentrate on combination operations. The simplest, substitution, can only happen at substitution nodes and both attachment sites must be labelled with the same non-terminal. We assume that these conditions are checked by the f_{SS} function:

Substitute:

$$\frac{[v_m, \tau']}{[v_h, \tau, p, \top, v_m, v_m]} \; (v_m)_{\leftarrow} = -, \; f_{SS}(\tau, p, \tau')$$

The wrapping adjunction combines a modifier with a gap:

Wrapping adjoin:

$$\frac{[v_m, \tau'] \quad [v_h, \tau, p, \bot, b_l, b_r]}{[v_h, \tau, p, \top, v_m, v_m]} \; (v_m)_{\leftarrow} = (b_l)_{\Leftarrow}, \; (v_m)_{\rightarrow} = (b_r)_{\Rightarrow}, \; f_{SA}(\tau, p, \tau')$$

Similarly, left and right adjunctions deal with vertices without gap:

Left adjoin:

$$\frac{[v_m, \tau', \leftarrow] [v_h, \tau, p, \bot, b_l, b_r]}{[v_h, \tau, p, \top, v_m, b_r]} \; (v_m)_{\Rightarrow} = (b_l)_{\Leftarrow} - 1, \; f_{SA}(\tau, p, \tau')$$

Right adjoin:

$$\frac{[v_m, \tau', \rightarrow] [v_h, \tau, p, \bot, b_l, b_r]}{[v_h, \tau, p, \top, b_l, v_m]} \; (b_r)_{\Rightarrow} = (v_m)_{\Leftarrow} - 1, \; f_{SA}(\tau, p, \tau')$$

Finally, adjunction may be skipped if this is allowed at the current site:

Null adjoin:

$$\frac{[v_h, \tau, p, \bot, b_l, b_r]}{[v_h, \tau, p, \top, b_l, b_r]} \; \neg f_{OA}(\tau, p)$$

5 Correctness

Since we use the deductive parsing framework, proving the correctness of the algorithm is straightforward from the notion of item invariant. Proving that every production rule maintains this invariant gives us soundness. Conversely, completeness can be proven by induction on items. In the following we explain our invariant.

An item $[v_h, \tau, p, c, b_l, b_r]$ can be deduced from the axioms through the application of deduction rules if and only if, with respect to the input dependency parse, $\tau(p)$ can be derived to generate the subsequence of terminals and foot nodes:

- if address p in τ does not dominate a foot node, the subsequence is $s_{(b_l)_{\Leftarrow}} \dots s_{(b_l)_{\Rightarrow}}$;
- if p in τ dominates a foot node and this is not a left nor a right adjunction $b_l, b_r \notin \{\overleftarrow{g_h}, \overrightarrow{g_h}\}$, the subsequence is $s_{(b_l)_{\Leftarrow}} \dots s_{(v_h)_{\leftarrow}-1} X^* s_{(v_h)_{\rightarrow}+1} \dots s_{(b_r)_{\Rightarrow}}$;
- if p in τ dominates a foot node and $b_l = \overleftarrow{g_h}$, the subsequence is $X^* s_{(v_h)_{\Leftarrow}} \dots s_{(b_r)_{\Rightarrow}}$;
- if p in τ dominates a foot node and $b_r = \overrightarrow{g_h}$, the subsequence is $s_{(b_l)_{\Leftarrow}} \dots s_{(v_h)_{\Rightarrow}} X^*$;

with X^* the foot node of τ. A visualization is provided on Figure 3.

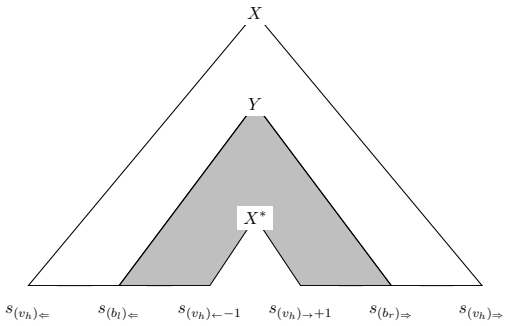


Figure 3: Invariant of an item $[v_h, \tau, p, c, b_l, b_r]$ when vertex v_h has a block degree of 2. X is the root node of τ, X^* its foot node and Y the node at address p. Only the gray area has been parsed.

6 Complexity

It is common practice to directly deduce space and time complexities from item structures and deduction rules, respectively. However, improving bounds in this setting may lead to algorithms difficult to understand. Thus, we decided to propose a rule-based algorithm that is simple to understand but which naively exposes a loose upper bound on its underlying complexity. In this section, we prove that the space and time complexities are linear.

In order to simplify the analysis, we suppose an agenda-based implementation (Kay, 1986). Each deduced item is placed in an agenda. While the agenda is not empty, the main loop pops out an item from it and add it to the chart. Then, the

popped out item is tested as an antecedent, and all deduced consequents are pushed in the agenda if not already present in the chart. See Algorithm 1 for an outline of the algorithm.

Algorithm 1 Outline of the parsing algorithm. Lines 22-27 apply the **Binary move** rule.

1: $A \leftarrow []$ ▷ Empty Agenda
2: **for** $1 \leq m \leq n$ **do** ▷ Init
3: **for** $\tau \in \Gamma$ **do** ▷ Lex. scan
4: **if** $\tau(lex(\tau)) = s_m$ **then**
5: $A.push([\tau, lex(\tau), \top, l_m, l_m])$
6: **end if**
7: **end for**
8: **for** $\tau \in \Gamma^A$ **do** ▷ Foot scan
9: **if** $(v_m)_{\rightarrow} \neq -$ **then**
10: $A.push([\tau, foot(\tau), \top, g_m, g_m])$
11: **else**
12: $A.push([\tau, foot(\tau), \top, \overleftarrow{g_m}, \overleftarrow{g_m}])$
13: $A.push([\tau, foot(\tau), \top, \overrightarrow{g_m}, \overrightarrow{g_m}])$
14: **end if**
15: **end for**
16: **end for**
17: $C \leftarrow []$ ▷ Empty Chart
18: **while** $|A| > 0$ **do**
19: $[\tau, p \cdot 1, \top, b_{l1}, b_{r1}] \leftarrow A.pop()$
20: $C.add([\tau, p \cdot 1, \top, b_{l1}, b_{r1}])$
21: ▷ *apply* **Move binary**
22: Let b_{l2} be the unique boundary with $(b_{r1})_{\Rightarrow} + 1 = (b_{l2})_{\Leftarrow}$
23: **for** $[\tau, p \cdot 2, \top, b_{l2}, b_{r2}] \in C$ **do**
24: **if** $[\tau, p, \top, b_{l1}, b_{r2}] \notin C$ **then**
25: $A.push([\tau, p, \top, b_{l1}, b_{r2}])$
26: **end if**
27: **end for**
28: ...Apply other rules...
29: **end while**

Before analysing the algorithm complexity, we observe that the first value of an item, the current vertex, is redundant. Indeed, given the value of the left boundary (or right boundary), we can always retrieve the considered vertex in constant time. Obviously, when the boundary is given by a vertex v_m, we have $v_h = (v_m)_{\uparrow}$.[10] For special values allowed for boundaries, all indexed by a word position h, a similar operation is straightforward. For example, if a boundary is given by l_h then the considered vertex is v_h.

[10] We assume that the data structure storing the dependency graph provides such function in constant time.

The algorithm has a maximum of two nested loops: the main while loop and for loops matching the second antecedent of binary rules. We first consider the while loop. An item is added to the agenda if only if it is not present in the chart. Thus, each item is considered exactly once by this loop. We note n the length of the input sentence, t the maximum number of nodes in an elementary tree $\tau \in \Gamma$ and $g \triangleq |\Gamma|$ the number of elementary trees.[11] Naively, the number of items is then bounded by $\mathcal{O}(n^2tg)$. However, the number of combination operations which can be applied on an elementary tree is bounded by its number of nodes, provided we dismiss multiple adjunction sites. In this case, each node of an elementary tree may be adjoined or substituted on at most once. Thus, given an elementary tree and a left boundary, the number of values allowed as a right boundary is limited. This leads to a tighter bound on the number of items: $\mathcal{O}(\min(t, n)ntg)$.

We now investigate time complexity. **Move binary** is the only rule which has two free antecedents. Indeed, it is easy to see that, in the other binary rules, fixing the right antecedent always fixes the left one. For the **Move binary** rule, given the left antecedent, the number of candidates for the right one is naively bounded by $\mathcal{O}(n)$. However, we previously argued that, given a fixed left boundary, the maximum number of right boundary alternatives cannot exceed the number of sites on the current elementary tree. Thus, we can tighten the bound to $\mathcal{O}(\min(t, n))$.

In conclusion, the time complexity of the proposed algorithm is $\mathcal{O}(\min(t, n)^2ntg)$, that is asymptotically linear with respect to the input sentence length.

7 Conclusion

We proposed an algorithm to compute LTAG derivations given a sentence and a dependency structure describing lexical attachments. We showed that under mild assumptions the worst-case complexity is linear in the length of the input.

This work fits in the more general project to interpret LTAG parsing as a dependency structure decoding problem. In this framework, it is common to label the graph structure in a post-processing step. Hence our approach relies on the availability of valid dependency structures for

[11] Alternatively, g can be the number of elementary trees associated with the most ambiguous word of the vocabulary.

LTAGs, more precisely well-nested arborescences with 2-bounded block degree.

Experiments remain to be done in order to validate the practical interest of this approach. Moreover, one limitation of the pipeline system, as in all pipeline approaches, is the possibility of error cascades: the first step may decode a dependency structure that is not recognizable by the grammar while the sentence is grammatical.

We hope that this work will open new perspectives on parsing with lexicalized grammars. Exploring a similar technique, one may develop an efficient parsing algorithm for weighted Lexicalized Linear Context Free Rewriting System.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. First author is supported by a public grant overseen by the French National Research Agency (ANR) as part of the Investissements d'Avenir program (ANR-10-LABX-0083). Second author, supported by a public grant overseen by the French ANR (ANR-16-CE33-0021), completed this work during a CNRS research leave at LIMSI, CNRS / Université Paris Saclay.

References

Anne Abeille, Kathleen Bishop, Sharon Cote, and Yves Schabes. 1990. A lexicalized tree adjoining grammar for english. Technical report, University of Pensylvania.

François Barthélemy, Pierre Boullier, Philippe Deschamp, and Éric Villemonte de la Clergerie. 2001. Guided parsing of range concatenation languages. In *Proceedings of 39th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Toulouse, France, pages 42–49. https://doi.org/10.3115/1073012.1073019.

Manuel Bodirsky, Marco Kuhlmann, and Mathias Möohl. 2005. Well-nested drawings as models of syntactic structure. In *Proceedings of the 10th Conference on Formal Grammar (FG) and Ninth Meeting on Mathematics of Language (MOL)*. Edinburgh, UK, pages 195–203.

Guillaume Bonfante, Bruno Guillaume, and Mathieu Morey. 2009. *Proceedings of the 11th International Conference on Parsing Technologies (IWPT'09)*, Association for Computational Linguistics, chapter Dependency Constraints for Lexical Disambiguation, pages 242–253. http://aclweb.org/anthology/W09-3840.

Xavier Carreras, Michael Collins, and Terry Koo. 2008. *CoNLL 2008: Proceedings of the Twelfth Conference on Computational Natural Language Learning*, Coling 2008 Organizing Committee, chapter TAG, Dynamic Programming, and the Perceptron for Efficient, Feature-Rich Parsing, pages 9–16. http://aclweb.org/anthology/W08-2102.

John Chen and Srinivas Bangalore. 1999. New models for improving supertag disambiguation. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*. http://aclweb.org/anthology/E99-1025.

David Chiang. 2000. Statistical parsing with an automatically-extracted tree adjoining grammar. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics*. http://aclweb.org/anthology/P00-1058.

Michael Collins. 2003. Head-driven statistical models for natural language parsing. *Computational Linguistics, Volume 29, Number 4, December 2003* http://aclweb.org/anthology/J03-4003.

Caio Corro, Joseph Le Roux, Mathieu Lacroix, Antoine Rozenknop, and Roberto Wolfler Calvo. 2016. Dependency parsing with bounded block degree and well-nestedness via lagrangian relaxation and branch-and-bound. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, pages 355–366. https://doi.org/10.18653/v1/P16-1034.

Ralph Debusmann, Denys Duchier, Marco Kuhlmann, and Stefan Thater. 2004. Tag parsing as model enumeration. In *Proceedings of the 7th International Workshop on Tree Adjoining Grammar and Related Formalisms*. Vancouver, Canada, pages 148–154. http://www.aclweb.org/anthology/W04-3320.

Jason Eisner and Giorgio Satta. 2000. A faster parsing algorithm for lexicalized tree-adjoining grammars. In *Proceedings of the Fifth International Workshop on Tree Adjoining Grammar and Related Frameworks (TAG+5)*. Université Paris 7, pages 79–84. http://www.aclweb.org/anthology/W00-2011.

Daniel Fernández-González and André F. T. Martins. 2015. Parsing as reduction. 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)*. Association for Computational Linguistics, pages 1523–1533. https://doi.org/10.3115/v1/P15-1147.

Carlos Gómez-Rodríguez, David Weir, and John Carroll. 2009. Parsing mildly non-projective dependency structures. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*. Association for Computational Linguistics, pages 291–299. http://aclweb.org/anthology/E09-1034.

Joshua Goodman. 1999. Semiring parsing. *Computational Linguistics, Volume 25, Number 4, December 1999* http://aclweb.org/anthology/J99-4004.

Aravind K Joshi. 1987. An introduction to tree adjoining grammars. *Mathematics of language* 1:87–115.

Aravind K Joshi and Yves Schabes. 1992. Tree-adjoining grammars and lexicalized grammars. *Tree Automata and Languages* .

Laura Kallmeyer and Marco Kuhlmann. 2012. A formal model for plausible dependencies in lexicalized tree adjoining grammar. In *Proceedings of the 11th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+11)*. Paris, France, pages 108–116. http://www.aclweb.org/anthology/W12-4613.

Martin Kay. 1986. *Readings in Natural Language Processing*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, chapter Algorithm Schemata and Data Structures in Syntactic Processing, pages 35–70. http://dl.acm.org/citation.cfm?id=21922.24327.

Lingpeng Kong, M. Alexander Rush, and A. Noah Smith. 2015. Transforming dependencies into phrase structures. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, pages 788–798. https://doi.org/10.3115/v1/N15-1080.

Marco Kuhlmann and Joakim Nivre. 2006. Mildly non-projective dependency structures. In *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*. Association for Computational Linguistics, pages 507–514. http://aclweb.org/anthology/P06-2066.

Mark-Jan Nederhof. 2009. Weighted parsing of trees. In *Proceedings of the 11th International Conference on Parsing Technologies*. Association for Computational Linguistics, Stroudsburg, PA, USA, IWPT '09, pages 13–24. http://dl.acm.org/citation.cfm?id=1697236.1697239.

Fernando C. N. Pereira and David H. D. Warren. 1983. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*. http://aclweb.org/anthology/P83-1021.

Owen Rambow. 2010. The simple truth about dependency and phrase structure representations: An opinion piece. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, pages 337–340. http://aclweb.org/anthology/N10-1049.

Owen Rambow and Aravind Joshi. 1997. A formal look at dependency grammars and phrase-structure grammars, with special consideration of word-order phenomena. *Recent trends in meaning-text theory* 39:167–190.

Philip Resnik. 1992. Probabilistic tree-adjoining grammar as a framework for statistical natural language processing. In *COLING 1992 Volume 2: The 15th International Conference on Computational Linguistics*. http://aclweb.org/anthology/C92-2065.

Giorgio Satta. 1994. Tree-adjoining grammar parsing and boolean matrix multiplication. *Computational Linguistics, Volume 20, Number 2, June 1994* http://aclweb.org/anthology/J94-2002.

Yves Schabes, Anne Abeille, and Aravind K. Joshi. 1988. Parsing strategies with 'lexicalized' grammars: Application to tree adjoining grammars. In *Coling Budapest 1988 Volume 2: International Conference on Computational Linguistics*. http://aclweb.org/anthology/C88-2121.

Yves Schabes and Richard C. Waters. 1995. Tree insertion grammar: A cubic-time, parsable formalism that lexicalizes context-free grammar without changing the trees produced. *Computational Linguistics, Volume 21, Number 4, December 1995* http://aclweb.org/anthology/J95-4002.

K. Vijay-Shankar and Aravind K. Joshi. 1986. Some computational properties of tree adjoining grammars. In *Strategic Computing - Natural Language Workshop: Proceedings of a Workshop Held at Marina del Rey, California, May 1-2, 1986*. http://aclweb.org/anthology/H86-1020.

K. Vijay-Shanker and David J. Weir. 1993. Parsing some constrained grammar formalisms. *Computational Linguistics, Volume 19, Number 4, December 1993* http://aclweb.org/anthology/J93-4002.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of the IWPT (Volume 3)*.