Improvements to the performance and applicability of dependency parsing

Daniel FERNÁNDEZ-GONZÁLEZ

DOCTORAL THESIS BY PUBLICATION UDC / 2015

Supervisors

Dr. Manuel VILARES FERRO

Dr. Carlos GÓMEZ-RODRÍGUEZ

PH.D. IN COMPUTATIONAL SCIENCE

UNIVERSIDADE DA CORUÑA

Dr. Manuel Vilares Ferro, Catedrático de Universidad del Departamento de Informática de la Universidade de Vigo,

Dr. Carlos Gómez Rodríguez, Profesor Contratado Doctor del Departamento de Computación de la Universidade da Coruña,

CERTIFICAN

Que la presente memoria titulada **Improvements to the performance and applicability of dependency parsing** ha sido realizada bajo su dirección y constituye la Tesis presentada por **Daniel Fernández González** para optar al grado de Doctor con Mención Internacional por la Universidade da Coruña.

Fdo. Manuel Vilares Ferro
Director de la Tesis Doctoral

Fdo. Carlos Gómez Rodríguez
Codirector de la Tesis Doctoral

Acknowledgements

This thesis is the result of the help of many people who deserve my infinite gratitude.

First of all, I would like to thank my advisors. *Manuel* provided the necessary guidance and advice to meet all goals I set, especially in those moments when the darkness did not let me see the way. He also contributed with the material and financial support essential to have a successful research career. *Carlos* was the main responsible for the thesis subject. He introduced me in the thrilling field of dependency parsing, taught me everything he knew about transition-based systems and helped me every time I asked for it. I really appreciate his commitment during all my research career and, above all, during the elaboration of the articles that compound this thesis. Both, *Manuel* and *Carlos*, formed a perfect team as supervisors and I always will be grateful.

I was fortunate to do two research visits during my thesis period. I could not imagine better supervisors in Uppsala and Lisbon. It was in the charming Uppsala where I had the opportunity to meet and work with one of the fathers of transition-based parsing: *Joakim Nivre*. He was a friendly and an always available host. His passion and devotion for his work is contagious. After a talk with him, it is difficult not to feel motivated to work in dependency parsing. In Lisbon, I was under the supervision of *André F. T. Martins*. I have never met a natural language processing researcher that had such an extended knowledge in machine learning as him. It is impossible not to learn something when you work with a person like *André*. Every discussion with him ends up in valuable knowledge and helpful advice. I cannot thank them enough for bringing me the opportunity to collaborate with them in two different articles that are already part of this thesis.

On a personal note, I am deeply grateful to my loving family. They did not participate directly in this work, but their support was crucial. My parents, *José* and *Dosinda*, give me the education and confidence necessary to address any project no matter how hard it was. And, in particular, *Berta*, who has been patiently by my side throughout all this process, encouraging me to go ahead and finish the work I started four years ago.

Finally, it is worth mentioning that this thesis has been partially funded by the Ministry of Education under the FPU Grant Program, Ministry of Economy and Competitiveness/ERDF (grants TIN2010-18552-C03-01 and FFI2014-51978-C2-1-R) and Xunta de Galicia (grants CN2012/317, CN2012/319, R2014/029 and R2014/034).

Computers are incredibly fast, accurate and stupid. Human beings are incredibly slow, inaccurate and brilliant. Together they are powerful beyond imagination.

Albert Einstein (attributed)

Abstract

Dependency parsers have attracted a remarkable interest in the last two decades due to their usefulness in a wide range of natural language processing tasks. They employ a dependency graph to define the syntactic structure of a given sentence. In particular, transition-based algorithms provide accurate and efficient dependency syntactic analyses. However, the main drawback of these techniques is that they tend to suffer from error propagation. So, an early erroneous decision may place the parser into an incorrect state, causing more errors in future decisions.

This thesis focuses on improving the accuracy of transition-based parsers by reducing the effect of error propagation, while preserving their speed and efficiency. Concretely, we propose five different approaches that proved to be beneficial for their performance, mitigating the presence of error propagation and boosting its accuracy.

We also extend the usefulness of dependency parsers beyond building dependency graphs. We present a novel technique that allows these to build constituent representations. This meets the necessity of the natural language processing community to have an efficient parser able to provide constituent trees to represent the syntactic structure of sentences.

Resumen

Los analizadores de dependencias han generado un gran interés en las últimas décadas debido a su utilidad en un amplio rango de tareas de procesamiento de lenguaje natural. Estos utilizan grafos de dependencias para definir la estructura sintáctica de una oración dada. En particular, los algoritmos basados en transiciones proveen un análisis sintáctico de dependencias eficiente y preciso. Sin embargo, su principal inconveniente es que tienden a sufrir propagación de errores. Así, una decisión temprana tomada erróneamente podría posicionar el analizador en un estado incorrecto, causando más errores en futuras decisiones.

Esta tesis se centra en mejorar la precisión de los analizadores basados en transiciones mediante la reducción del efecto de la propagación de errores, mientras mantienen su velocidad y eficiencia. Concretamente, proponemos cinco enfoques diferentes que han demostrado ser beneficiosos para su rendimiento, al aliviar la propagación de errores e incrementar su precisión.

Además, hemos ampliado la utilidad de los analizadores de dependencias más allá de la construcción de grafos de dependencias. Presentamos una novedosa técnica que permite que estos sean capaces de construir representaciones de constituyentes. Esto cubriría la necesidad de la comunidad de procesamiento de lenguaje natural de disponer de un analizador eficiente capaz de proveer un árbol de constituyentes para representar la estructura sintáctica de las oraciones.

Resumo

Os analizadores de dependencias xeraron gran interese nas últimas décadas debido á súa utilidade nun amplo rango de tarefas de procesamento da linguaxe natural. Estes utilizan grafos de dependencias para definir a estrutura sintáctica dunha oración dada. En particular, os algoritmos baseados en transicións provén un análise sintáctico de dependencias eficiente e preciso. Sen embargo, o seu principal inconveniente é que tenden a sufrir propagación de erros. Así, unha decisión temprana tomada erroneamente podería posicionar o analizador nun estado incorrecto, causando máis erros en futuras decisións.

Esta tese centrase en mellorar a precisión dos analizadores baseados en transicións mediante a redución do efecto da propagación de erros, mentres manteñen a súa velocidade e eficiencia. Concretamente, propomos cinco diferentes enfoques que demostraron ser beneficiosos para o seu rendemento, ó aliviar a propagación de erros e incrementar a súa precisión.

Ademais, ampliámo-la utilidade dos analizadores de dependencias máis alá da construción de grafos de dependencias. Presentamos unha novidosa técnica que permite que estes sexan capaces de construir representacións de constituíntes. Isto cubriría a necesidade da comunidade de procesamento da linguaxe natural de dispor dun analizador eficiente capaz de prover unha árbore de constituíntes para representar a estrutura sintáctica das oracións.

Contents

List of Figures

CHAPTER I

Introduction

The final goal pursued by *natural language processing* (NLP) is to transform unrestricted natural language text into representations tractable by machines. In that way, raw textual information can be feasibly used by computers to undertake more complex tasks such as automatic translation, information extraction or question answering.

Syntactic parsing is one of the most ubiquitous and useful NLP processes. It consists of determining the grammatical structure of sentences in natural language: given an input sentence, a parser will map it into its syntactic representation. This underlying structure can be represented in different formats depending on the particular syntactic theory followed by the parser.

There are two different widely-used syntactic formalisms for this purpose: *constituent* or *phrase-structure* representations [2, 7] and *dependency* representations [44]. In the first case, sentences are analyzed by decomposing them into meaningful parts called *constituents* or *phrases*, and creating relationships between them and the words to finally build a *phrase-structure tree*, as shown in Figure 1.1. In dependency parsing, the syntactic structure of a sentence is represented by means of a *dependency graph*. This consists of a set of binary relations called *dependencies* that link pairs of words of a given sentence to describe a syntactic relation between them, where one word acts as the *head* and the other as the *dependent*. We call it a *dependency tree* if every word of the sentence has only one incoming arc (single-head constraint), the structure is acyclic and only has one root, as the one described in Figure 1.2.

In the last two decades, dependency parsing has become very popular in the NLP community, in detriment of its counterpart, constituent parsing. This is mainly because its representations have some irrefutable advantages over the phrase-stucture formalism. The lack of intermediate nodes in dependency trees have granted them the simplicity necessary to represent more complex linguistic phenomena, such as discontinuities caused by free word order, and has allowed the construction of more efficient parsing

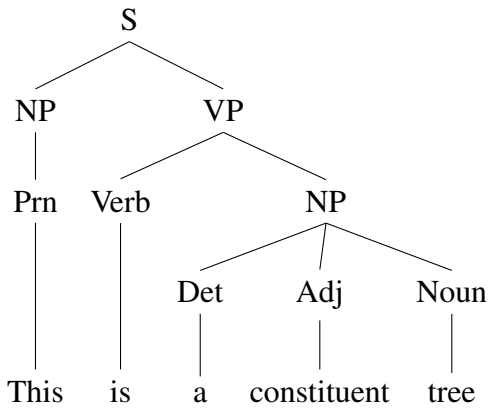


Figure 1.1: Constituent tree for an English sentence.

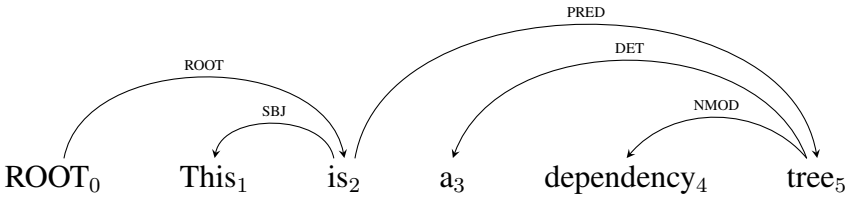


Figure 1.2: Dependency tree for an English sentence.

algorithms. As a consequence, dependency parsers have been successfully applied in a wide variety of problems ranging from machine translation [12, 24, 41, 49] and relation extraction [11, 14, 23, 33] to question answering [8, 10] and opinion mining [22, 46].

In contrast to *grammar-driven* dependency parsers, a new paradigm centered in data has emerged in the last twenty years. The explicit representation of knowledge by formal grammatical rules [27, 43] has been replaced by a *data-driven* paradigm based on *machine learning* and the massive amount of data available. The increasing availability of human annotated resources, such as the Penn Treebank [28] or those corpora provided by shared tasks such as CoNLL-X [5] and CoNLL-XI [37], made it possible to apply machine learning techniques, which automatically extract statistical models from the data without the need of an explicit grammar, and implement dependency parsers that produce accurate analyses very efficiently.

Data-driven dependency parsers have been a very fruitful field in NLP, resulting in some of the most accurate and efficient systems as those by Nivre et al. [38], McDonald et al. [32], Titov and Henderson [45], Martins et al. [29], Huang and Sagae [20], Koo and Collins [26], Zhang and Nivre [51], Bohnet and Nivre [3] or Goldberg and Nivre [15]. Practically all these systems can be classified into two families, commonly called *graph-based* and *transition-based* parsers [31, 50].

Graph-based parsers learn a global model for scoring possible dependency graphs for a given sentence and, then, the parsing process consists of searching for the highest-scoring graph. The proposals by McDonald et al. [32] and Martins et al. [29] are well-known graph-based systems. The main drawback of this approach is that their parsing process is accomplished in at least a quadratic time complexity.

On the other hand, transition-based parsers proved to be faster and more efficient, since many of them are able to undertake the analysis of a sentence in linear execution time; while still providing state-of-the-art accuracies. Given a sentence, the parser incrementally builds a dependency graph from left to right by greedily choosing the highest-scoring permissible transition at each state. In that way, the input sentence is analyzed by a sequence of transitions compounded by the highest-scoring actions. The set of transitions that the parser can use are individually scored by a previously trained statistical model called *oracle*. In addition, there exists a variant in transition-based parsing that includes *beam search* for choosing the best sequence of transitions [50], instead of undertaking it greedily. In fact, one of the state-of-the-art transition-based parsers is the beam-search-based system by Zhang and Nivre [51].

Unfortunately, the greedy nature that grants them their efficiency also represents their main weakness. McDonald and Nivre [30] show that the main reason for loss of accuracy in transition-based parsing is error propagation: a transition erroneously chosen by the parser may place it in an incorrect state, causing more errors in the transition sequence. As the sentence is parsed in a sequential process, a mistake in an early stage of that process can lead to further mistakes in future stages. In particular, one source of error propagation is the enforcement of the single-head constraint necessary to shape a dependency tree. For instance, if a transition-based system is parsing the sentence in Figure 1.2 and mistakenly applies a transition that builds an arc from a_3 to $tree_5$ instead of the correct dependency from $tree_5$ to a_3; we will not only fail in creating this dependency but also the one from is_2 to $tree_5$, since we will be in a state where the single-head constraint will not allow to create two incoming arcs to node $tree_5$. Moreover, long sentences and, especially, long arcs are more likely to be affected by error propagation, since they need a longer sequence of transitions to be parsed. In this thesis, we focused our efforts on making transition-based parsing more accurate by mitigating error propagation, but without penalizing its advantageous efficiency. In addition, some of the improvements successfully applied in transition-based parsing can also be applied to dependency parsers of any kind.

In spite of the fact that dependency parsing has been in the spotlight in the last twenty years, constituent parsing has recently gained interest among NLP researchers. This was facilitated by the fact that some fields, such as sentiment analysis and opinion mining [1, 13, 21, 48], require syntax formalisms more informative than dependency representations. However, the main problem is that most existing constituent parsers are significantly slow [6, 25, 39], since they need to deal with a heavy grammar constant. Thus, the NLP community demands phrase-structure parsers as efficient as dependency systems. To meet this need, we propose a new approach that mixes the best of both worlds:

building informative constituent representations by efficient dependency parsing.

Therefore, this thesis aims to contribute to both constituent and dependency parsing. We improve the performance of dependency parsers (concretely, transition-based systems) and apply them to undertake phrase-structure parsing in an efficient way.

The remainder of this document is organized as follows: In Chapter 2, we summarize our main contributions. Chapter 3 discusses the conclusions of our work. And, finally, Chapter 4 details the published articles that constitute this thesis and includes an original copy of them for further deepening in the subject. Note that Appendix A contains a Spanish extended abstract of the thesis.

CHAPTER II

Contributions

In this chapter, we first introduce some basic definitions and notations concerning transition-based dependency parsing, which will serve as a basis to present all our contributions. Secondly, we summarize five different approaches that aim to improve the accuracy of transition-based parsing by alleviating the effect of error propagation. Finally, we show how to apply dependency parsing to build the phrase-structure representation of a given sentence.

2.1 | Preliminaries

2.1.1 | Dependency parsing

A *dependency graph* is a labelled directed graph that represents the syntactic structure of a given sentence. More formally, it can be defined as follows:

Definition 1 *Let $w = w_1 \dots w_n$ be an input string. A* dependency graph *for $w_1 \dots w_n$ is a labelled directed graph $G = (V_w, A)$, where $V_w = \{0, \dots, n\}$ is the set of nodes, and $A \subseteq V_w \times L \times V_w$ is the set of labelled directed arcs.*

■

Apart from every word of the sentence with index i such that $1 \leq i \leq n$, the set V_w includes a special node in index 0 called ROOT, which does not correspond to any token of the sentence and which will always be a root of the dependency graph.

Each arc in A encodes a dependency relation between two words. We call an edge (w_i, l, w_j) in a dependency graph G a *dependency link* from w_i to w_j with label l, represented as $w_i \overset{l}{\rightarrow} w_j$. We say that w_i is the *head* of w_j and, conversely, that w_j

is a *dependent* of w_i. The labels on dependency links are typically used to represent their associated syntactic functions, such as SBJ for subject in the dependency link $is_2 \rightarrow This_1$ in Figure 1.2.

For convenience, we write $w_i \rightarrow w_j \in G$ if the link (w_i,w_j) exists (regardless of its label) and $w_i \rightarrow^* w_j \in G$ if there is a (possibly empty) directed path from w_i to w_j.

Most dependency-based syntactic formalisms are typically restricted to acyclic graphs where each node has at most one head. Such dependency graphs are called *dependency forests*.

Definition 2 *A dependency graph G is said to be a* dependency forest *if it satisfies the following:*

1. Single-head constraint*: if $w_i \rightarrow w_j$, then there is no $w_k \neq w_i$ such that $w_k \rightarrow w_j$.*
2. Acyclicity constraint*: if $w_i \rightarrow^* w_j$, then there is no arc $w_j \rightarrow w_i$.*

■

Nodes that have no head in a dependency forest are called *roots*. Apart from the previous two constraints, some dependency formalisms add the additional constraint that a dependency forest can have only one root (or, equivalently, that all the nodes of the graph must be connected). A forest of this form is called a *dependency tree*.

A *dependency parser* is the system in charge of parsing a given sentence producing a dependency graph. In this thesis, we will work with dependency parsers that output dependency trees. This means that they enforce the single-head and acyclicity constraints, as well as link all of the graph's root nodes as dependents of the dummy root node in index 0.

Finally, many dependency parsers are restricted to work with *projective* dependency structures in order to preserve their computational efficiency. These are dependency graphs where the set of nodes reachable by traversing zero or more arcs from any given node k corresponds to a continuous substring of the input, that is, an interval $\{x \in V_w \mid i \leq x \leq j\}$. A graphical identification of projective dependency trees can be done on drawn graphs as that showed in Figure 1.2, where the absence of crossing links confirms the projectivity of the structure. To analyze more complex syntactic phenomena, it is necessary to use non-projective dependency graphs (with crossing links), which allow the representation of discontinuities caused by free word order.

2.1.2 | Transition systems

The framework proposed by Nivre [36] provides the components necessary for developing a transition-based dependency parser. According to this, a transition-based parser is a

deterministic dependency parser defined by a non-deterministic *transition system*. This transition system specifies a set of elementary operations that are deterministically applied by an *oracle* at each choice point of the parsing process. More formally, they are defined as follows:

Definition 3 *A* transition system *for dependency parsing is a tuple* $S = (C, T, c_s, C_t)$*, where*

1. C *is a set of possible parser* configurations*,*
2. T *is a finite set of* transitions*, which are partial functions* $t : C \rightarrow C$*,*
3. c_s *is a total initialization function that maps each input string* w *to a unique* initial configuration $c_s(w)$*, and*
4. $C_t \subseteq C$ *is a set of* terminal configurations*.*

■

Definition 4 *An* oracle *for a transition system is a function* $o : C \rightarrow T$*.*

■

An input sentence w can be parsed using a transition system $S = (C, T, c_s, C_t)$ and an oracle o by starting in the initial configuration $c_s(w)$, calling the oracle function on the current configuration c, and moving to the next configuration by applying the transition returned by the oracle. This process is repeated until a terminal configuration is reached. Each sequence of configurations that the parser can traverse from an initial configuration to a terminal one for some input w is called a *transition sequence*.

The oracle for practical parsers is implemented by a statistical model previously trained on *treebank data* [38] and it is in charge of returning the highest-scoring transition from the set T to apply on each configuration. This treebank data consists of a huge amount of sentences manually annotated with their associated dependency graph. In particular, in this thesis we work with the English Penn Treebank [28] and datasets from the CoNLL-X [5] and CoNLL-XI [37] shared tasks.

2.1.3 | The arc-eager parser

Nivre's *arc-eager* dependency parser [35] is one of the most widely known and used transition-based parsers. Concretely, the arc-eager transition system (C, T, c_s, C_t) is defined as follows:

Transition	Stack (σ)	Buffer (β)	Added Arc
	[ROOT$_0$]	[This$_1$, ... , tree$_5$]	
SHIFT	[ROOT$_0$, This$_1$]	[is$_2$, ... , tree$_5$]	
LA$_{SBJ}$	[ROOT$_0$]	[is$_2$, ... , tree$_5$]	(2, SBJ, 1)
RA$_{ROOT}$	[ROOT$_0$, is$_2$]	[a$_3$, ... , tree$_5$]	(0, ROOT, 2)
SHIFT	[ROOT$_0$, is$_2$, a$_3$]	[dependency$_4$, tree$_5$]	
SHIFT	[ROOT$_0$, is$_2$, a$_3$, dependency$_4$]	[tree$_5$]	
LA$_{NMOD}$	[ROOT$_0$, is$_2$, a$_3$]	[tree$_5$]	(5, NMOD, 4)
LA$_{DET}$	[ROOT$_0$, is$_2$]	[tree$_5$]	(5, DET, 3)
RA$_{PRED}$	[ROOT$_0$, is$_2$, tree$_5$]	[ ]	(2, PRED, 5)

Figure 2.1: Transition sequence for parsing the sentence in Figure 1.2 using the arc-eager parser (LA=LEFT-ARC, RA=RIGHT-ARC).

1. C is the set of all configurations of the form $c = \langle \sigma, \beta, A \rangle$, where σ and β are disjoint lists of nodes from V_w (for some input w), and A is a set of dependency arcs over V_w. The list β, called the *buffer*, is used to hold nodes corresponding to input words that have not yet been read. The list σ, called the *stack*, contains nodes for words that have already been read, but still have dependency links pending to be created. For convenience, we will use the notation $\sigma|w_i$ to denote a stack with top w_i and tail σ, and the notation $w_j|\beta$ to denote a buffer with top w_j and tail β. The set A of dependency arcs contains the part of the output parse that the system has constructed at each point.

2. The initial configuration is $c_s(w_1 \dots w_n) = \langle [], [w_1 \dots w_n], \emptyset \rangle$, where the buffer initially holds the whole input string while the stack is empty (or $c_s(w_1 \dots w_n) = \langle [ROOT_0], [w_1 \dots w_n], \emptyset \rangle$, if the stack contains the dummy root node).

3. The set of terminal configurations is $C_t = \{\langle \sigma, [], A \rangle \in C\}$, where final configurations are those with an empty buffer, regardless of the contents of the stack.

4. The set T has the following transitions:

   - SHIFT : $\langle \sigma, w_i|\beta, A \rangle \Rightarrow \langle \sigma|w_i, \beta, A \rangle$
   - REDUCE : $\langle \sigma|w_i, \beta, A \rangle \Rightarrow \langle \sigma, \beta, A \rangle$
   - LEFT-ARC$_l$:
     $\langle \sigma|w_i, w_j|\beta, A \rangle \Rightarrow \langle \sigma, w_j|\beta, A \cup \{w_j \xrightarrow{l} w_i\} \rangle$

only if $\nexists w_k \mid w_k \rightarrow w_i \in A$ (single-head)

- RIGHT-ARC$_l$:
  $\langle \sigma|w_i, w_j|\beta, A\rangle \Rightarrow \langle \sigma|w_i|w_j, \beta, A \cup \{w_i \xrightarrow{l} w_j\}\rangle$
  only if $\nexists w_k \mid w_k \rightarrow w_j \in A$ (single-head)

The SHIFT transition is used to read words from the input string, by moving the next node in the buffer to the top of the stack. The LEFT-ARC$_l$ transition creates a leftward dependency arc labelled with l from the first node on the buffer to the topmost node on the stack and pops the stack. Conversely to this, the RIGHT-ARC$_l$ transition builds a rightward dependency arc labelled with l from the topmost node on the stack to the first node on the buffer and pushes the first node on the buffer onto the stack. Finally, the REDUCE transition is used to pop the topmost node from the stack when we have finished building arcs to or from it. Figure 2.1 shows a transition sequence in the arc-eager transition system which derives the labelled dependency graph in Figure 1.2.

Note that the arc-eager parser is a linear-time algorithm, since it is guaranteed to terminate after $2n$ transitions (with n being the length of a given input string), and is restricted to projective dependency trees.

Other well-known transition systems are: the *arc-standard* and *Covington* parsers [9, 36], as well as, the *planar* and *two-planar* parsers [18].

2.2 | Improving dependency parsing performance

In this section we address the error-propagation problem and present different techniques to improve the accuracy of transition-based parsing.

2.2.1 | Undirected dependency parsing

Our first attempt to reduce the effect of error propagation introduces a novel approach: *undirected* dependency parsing. Until now, all existing dependency systems build directed graphs (Figure 1.2), where the dependencies have a direction from the head word to the dependent one. We develop transition-based parsers able to work with undirected dependency graphs as the one presented in Figure 2.2. This means that the single-head constraint need not be observed during the parsing process, since the directed notions of head and dependent (or of incoming and outgoing arcs) are lost in undirected graphs. Therefore, this gives the parser more freedom, and can prevent situations where enforcing the constraint leads to error propagation.

After the undirected parsing process, we will need a post-processing step to recover the direction of the dependencies, generating a valid dependency structure. Thus, some

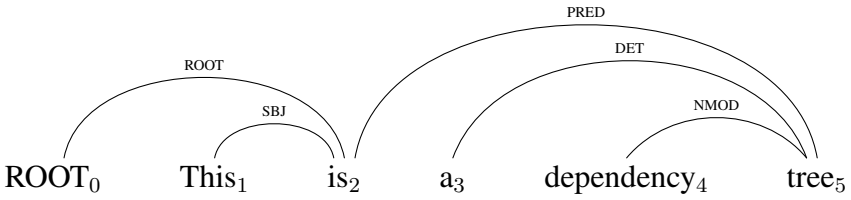


Figure 2.2: Undirected dependency graph for the English sentence in Figure 1.2.

complexity is moved from the transition-based parsing process to this post-processing step. This simplification leads the undirected parser to avoid more errors than the original directed version, achieving an increase in accuracy in most of the experiments tested.

Concretely, we implement the undirected version of the planar and two-planar [18] and Covington [9, 36] transition-based parsers and test them on the English Penn Treebank [28] and datasets from the CoNLL-X shared task [5]. The results proved the usefulness of undirected dependency parsing and a posterior analysis showed that, indeed, this approach mitigates the effect of error propagation.

This approach can be applied to dependency parsers of any kind and further information of this technique can be found in Articles 4.1 and 4.2. Please note that Article 4.2 is an extended version of Article 4.1, which adds further content such as new experiments and a thorough error analysis.

2.2.2 | Adding buffer transitions

Since transition-based parsing works by following a sequence of transitions, the longer the transition sequence is, the more the error propagation will affect the later stages of the process. In other words, it is more likely to make more mistakes (and propagate them) if the number of decisions that the parser has to make to parse a sentence is greater. This was the initial intuition we followed to propose this approach: we intend to reduce the number of transitions that a parser needs to analyze a given sentence. To achieve that, we had to design new transitions whose effect replaces two or more original transitions. It was also desirable that these new transitions were applied in simple scenarios where the parser can easily use them without overloading the classifier's job.

In particular, we develop four different transitions for the well-known arc-eager parser [35] detailed in Section 2.1.2, called *buffer transitions*. They are defined as follows:

- LEFT-BUFFER-ARC$_l$:
  $(\sigma, w_i|w_j|\beta, A) \Rightarrow (\sigma, w_j|\beta, A \cup \{w_j \xrightarrow{l} w_i\})$.
- RIGHT-BUFFER-ARC$_l$:
  $(\sigma, w_i|w_j|\beta, A) \Rightarrow (\sigma, w_i|\beta, A \cup \{w_i \xrightarrow{l} w_j\})$.

- LEFT-NONPROJ-BUFFER-ARC$_l$:
  $(\sigma|w_i, w_j|w_k|\beta, A) \Rightarrow (\sigma, w_j|w_k|\beta, A \cup \{(w_k \xrightarrow{l} w_i)\})$. Preconditions: $i \neq 0$ and $\nexists w_m, l' \mid (w_m, l', w_i) \in A$ (single-head constraint)

- RIGHT-NONPROJ-BUFFER-ARC$_l$:
  $(\sigma|w_i, w_j|w_k|\beta, A) \Rightarrow (\sigma|w_i, w_j|\beta, A \cup \{w_i \xrightarrow{l} w_k\})$.

These are transitions that create a dependency arc involving some node in the buffer, which would typically be considered unavailable for linking by the standard transitions. In that way, these buffer transitions construct some *easy* dependency arcs in advance, before the involved nodes reach the stack, so that the classifier's job when choosing among standard transitions is simplified. In addition, LEFT-NONPROJ-BUFFER-ARC and RIGHT-NONPROJ-BUFFER-ARC extend the coverage of the original arc-eager parser. These two transitions allow the creation of a limited set of non-projective dependency arcs, since they act on non-contiguous nodes in the stack and buffer.

Note as well that LEFT-BUFFER-ARC and RIGHT-BUFFER-ARC transitions are equivalent to applying the sequence of standard transitions SHIFT + LEFT-ARC and SHIFT + RIGHT-ARC + REDUCE, respectively, resulting in a shortening of the final transition sequence. On the other hand, the effect of LEFT-NONPROJ-BUFFER-ARC and RIGHT-NONPROJ-BUFFER-ARC transitions cannot be expressed with standard transitions, since they add a new functionality not present in the arc-eager parser, but they also involve a shortening of the transition sequence.

Experiments on datasets from the CoNLL-X shared task [5] back our hypothesis and show that by adding a buffer transition to the arc-eager parser its accuracy is boosted in practically all cases.

This approach is detailed in Article 4.3 and can be used in any transition-based dependency parser that uses a buffer and a stack in its configurations.

2.2.3 | Using reverse parsing

Transition-based parsers analyze a sentence from left to right. Due to error propagation, the probability of choosing an incorrect transition tends to be higher as we approach the end of a given sentence. As a result, the arcs situated in the rightmost side of the dependency graph suffer a higher loss in accuracy. It seems reasonable that applying a transition-based system that parses the sentence from right to left might be more accurate on the rightmost arcs of the graph. In fact, this was the main idea that led us to propose the use of *reverse* parsing. In particular, we present a combinative system, where a left-to-right parser is combined with its reverse right-to-left variant.

It has been proven that analyzing a sentence in reverse order does not improve the global accuracy of transition-based parsers [34]. However, the reverse parser is able

to build correctly some arcs of the dependency graph that the original version creates erroneously. Concretely, we found out that, in addition to having a better performance in rightmost arcs, the right-to-left parser is able to achieve higher accuracy in arcs with certain lengths. To take advantage of that, we present an efficient combination system that obtains a new dependency graph by joining the dependency graph created by each parser. This system uses two different strategies to accomplish the combination: a strategy based on the position of the arcs and another based on the length of the arcs.

We tested this novel technique in different transition-based parsers on the English Penn Treebank [28] and datasets from the CoNLL-X shared task [5]. The results obtained show that this approach with any of both strategies produces improvements in the accuracy of the transition-based parsers tested on all of the datasets used in the experiments (see more details in Article 4.4).

2.2.4 | Arc-eager parsing with the tree constraint

The research work described in Article 4.5 concerns only the arc-eager transition-based parser [35] described in Section 2.1.2.

The arc-eager algorithm might reach a terminal configuration without completely emptying the stack (regardless the dummy root). Due to errors produced during the parsing process, some words left in the stack, might have not been connected to the built dependency graph (or, equivalently, might have no head). This causes a fragmented dependency graph as output of the parsing process. The ideal output should be a connected, one-rooted, single-head constrained and acyclic dependency tree. To accomplish this, the standard solution is to convert this fragmented graph into a dependency tree by connecting all words without head in the stack to the artificial dummy root at the end of the parsing process. This heuristic achieves a well-formed dependency tree, but does not mitigate the loss in accuracy caused by errors committed during the analysis.

We define a modification of the arc-eager algorithm which guarantees that, after the analysis, the resulting graph is a dependency tree. Concretely, we add a deterministic UNSHIFT transition that, if the buffer is empty, pops words without head in the stack and pushes them back into the buffer, so that they can be processed again. We also set a new terminal configuration that entails not only an empty buffer, but also an empty stack (regardless the dummy root). In that way, by disabling the standard SHIFT transition, we force the arc-eager parser to create arcs on these words until reach the new terminal configuration, since the parser is only allowed to apply LEFT-ARC, RIGHT-ARC and REDUCE transitions (the latter is only allowed on words with a head and is used deterministically when the buffer is empty). As a result, the arc-eager parser will have the opportunity to rectify some mistakes made during the parsing process.

Empirical evaluations on all datasets from the CoNLL-X shared task [5] concluded

that the tree-constrained arc-eager parser consistently outperforms the old system with the standard heuristic of attaching all unattached tokens to the artificial root node.

2.2.5 | Non-projective dynamic oracle

In order to address the error-propagation problem in transition-based parsing, Goldberg and Nivre [15] developed a new strategy called *dynamic oracles*. It consists of endowing the oracles of transition-based systems with the capacity to tolerate errors produced during the parsing process, mitigating their impact in the resulting dependency graph. These new oracles are designed to recover from the presence of mistakes made in previous decisions and miss the minimum number of arcs in subsequent decisions.

Unlike standard oracles that are trained with the correct transition sequence necessary to analyse a given sentence, dynamic oracles are trained with a non-optimal sequence. During training, some transitions are randomly selected to simulate the errors committed during parsing time. In that way, dynamic oracles will be ready to deal with errors produced in the real parsing process.

Different research works provided dynamic oracles for the arc-eager and other well-known projective transition-based algorithms [15, 16, 17], as well as, for the Attardi parser [19], which supports a restricted set of non-projective arcs. However, the lack of an unrestricted non-projective dynamic oracle, gave us the motivation to propose this strategy on the Covington parser [9, 36]. This is an algorithm with full coverage on non-projective arcs and is considered one of the fastest transition-based systems in practice [47].

We implemented an efficient dynamic oracle specifically adapted to the Covington parser and tested its performance on datasets from the CoNLL-X [5] and CoNLL-XI [37] shared tasks. The results proved that dynamic oracles are also beneficial for the non-projective Covington algorithm, increasing its accuracy significantly.

In Article 4.6, we present the details of this contribution.

2.3 | Reduction of constituent parsing to dependency parsing

As a final goal of this thesis, we intended to use efficient dependency parsing to build informative constituent representations. To tackle this task, it was necessary to develop an intermediate formalism that allows to reduce constituent parsing to dependency parsing. In that way, an off-the-shelf dependency parser would be enough to build an efficient and accurate constituent parser.

We base our approach in the novel notion of *head-ordered* dependency trees. After determining the *head words* of each phrase, we encode the structure of a constituent tree into a dependency one as we can see in Figure 2.3. In particular, we use the outgoing

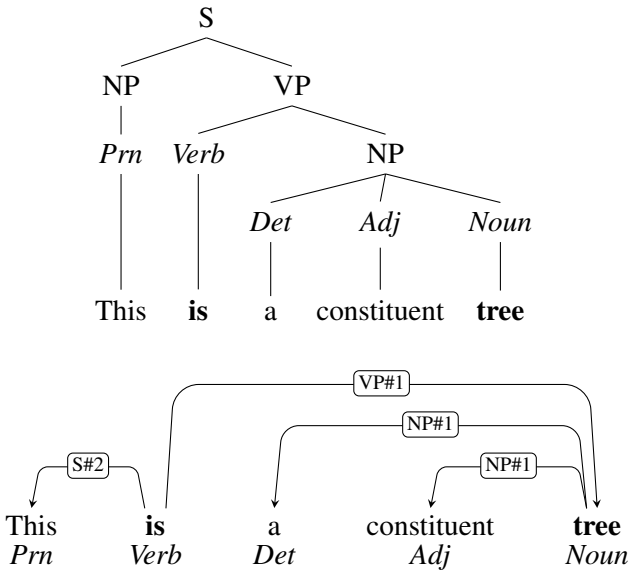


Figure 2.3: A constituent tree (top) and its corresponding head-ordered dependency tree (bottom). The head words and part-of-speech tags are marked in bold and italics, respectively.

dependency labels of the head words to save each constituent node (concretely, the parent node of each phrase where those words act as the head) together with an index that indicates the order of attachment in the phrase-structure tree. For instance, in Figure 2.3 the word *is* is the head of the phrases with parent nodes `VP` and `S`, therefore, we encode both constituent nodes with the dependency labels `VP#1` and `S#2`, respectively, where indexes `#1` and `#2` indicate that `VP` is attached first in the constituent tree with respect to `S`. If we ignore the unary node `NP` (dropped during the conversion) and the part-of-speech tags, we can see that both trees are isomorphic. This enables a dependency-to-constituent conversion necessary to produce constituent trees with dependency parsers. In addition, the unary nodes can be efficiently recovered in a post-processing step.

The proposed system obtained very competitive results on par with the phrase-structure Berkeley parser [39] on the English Penn Treebank [28], and with the best single system in the recent SPMRL14 shared task [40]. We also performed experiments on discontinuous German treebanks such as Tiger [4] and Negra [42], surpassing the current state of the art by a wide margin.

A thorough description of this technique is shown in Article 4.7.

CHAPTER III

Conclusion

Throughout this thesis we presented different novel techniques to improve the accuracy of transition-based dependency parsing. In particular, the proposed approaches tackled the main source of loss in accuracy in this kind of techniques: error propagation.

Experiments in different languages from the English Penn Treebank [28], the CoNLL-X [5] and CoNLL-XI [37] shared tasks proved that all contributions were beneficial for the performance of transition-based systems. In all cases, we achieved an increment in accuracy without penalizing their efficiency. In our research work, we also report detailed error analyses that show how our techniques mitigate error propagation.

Note as well that the approaches described here are perfectly compatible with beam-search-based transition-based parsers and any technique that improves parsing accuracy. In fact, our five contributions could be applied together under the same system. In addition, some of the presented strategies can be extended to other dependency parsers: for instance, undirected parsing might be also beneficial in graph-based dependency approaches.

Moreover, we expand the applicability of dependency parsing beyond the dependency formalism itself. We present a novel technique to perform accurate constituent parsing with an off-the-shelf dependency parser. In that way, syntax analyses can be efficiently undertaken by a dependency parser in either of the two widely-used formalisms. Of course, the dependency parser can incorporate our improvements to accuracy, so that they extend to constituents as well.

In conclusion, we can affirm that the contributions developed during this thesis have enriched the transition-based parsing field with new techniques, as well as have provided a novel approach to build constituent structures efficiently.

CHAPTER IV

Published Articles

This thesis is presented as a collection of the research articles detailed below:

1. CARLOS GÓMEZ-RODRÍGUEZ and **DANIEL FERNÁNDEZ-GONZÁLEZ**, Dependency Parsing with Undirected Graphs. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2012)*, pp. 66-76, Avignon, France, 2012. ISBN 978-1-937284-19-0.

2. CARLOS GÓMEZ-RODRÍGUEZ, **DANIEL FERNÁNDEZ-GONZÁLEZ** and VÍCTOR M. DARRIBA, Undirected Dependency Parsing. *Computational Intelligence*, 31(2):348-384, 2015. ISSN 1467-8640.

3. **DANIEL FERNÁNDEZ-GONZÁLEZ** and CARLOS GÓMEZ-RODRÍGUEZ, Improving Transition-Based Dependency Parsing with Buffer Transitions. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL 2012)*, pp. 308-319, Jeju, Korea, 2012. ISBN 978-1-937284-43-5.

4. **DANIEL FERNÁNDEZ-GONZÁLEZ**, CARLOS GÓMEZ-RODRÍGUEZ AND DAVID VILARES, Improving the Arc-Eager Model with Reverse Parsing. To appear in *Computing and Informatics*, submission number 1711, 2016. ISSN 1335-9150.

5. JOAKIM NIVRE and **DANIEL FERNÁNDEZ-GONZÁLEZ** , Arc-Eager Parsing with the Tree Constraint. *Computational Linguistics*, 40(2):259-267, 2014. ISSN 0891-2017. DOI 10.1162/COLI_a_00185

6. CARLOS GÓMEZ-RODRÍGUEZ and **DANIEL FERNÁNDEZ-GONZÁLEZ** An Efficient Dynamic Oracle for Unrestricted Non-Projective Parsing. To appear in *Proceedings of the Joint Conference of the 53rd Annual Meeting of the Association*

for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing (ACL-IJCNLP 2015), submission number 168, Beijing, China, 2015.

7. **DANIEL FERNÁNDEZ-GONZÁLEZ** and ANDRÉ F. T. MARTINS, Parsing as Reduction. To appear in *Proceedings of the Joint Conference of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing (ACL-IJCNLP 2015)*, submission number 478, Beijing, China, 2015.

Dependency Parsing with Undirected Graphs

Carlos Gómez-Rodríguez
Departamento de Computación
Universidade da Coruña
Campus de Elviña, 15071
A Coruña, Spain
carlos.gomez@udc.es

Daniel Fernández-González
Departamento de Informática
Universidade de Vigo
Campus As Lagoas, 32004
Ourense, Spain
danifg@uvigo.es

Abstract

We introduce a new approach to transition-based dependency parsing in which the parser does not directly construct a dependency structure, but rather an undirected graph, which is then converted into a directed dependency tree in a post-processing step. This alleviates error propagation, since undirected parsers do not need to observe the single-head constraint.

Undirected parsers can be obtained by simplifying existing transition-based parsers satisfying certain conditions. We apply this approach to obtain undirected variants of the planar and 2-planar parsers and of Covington's non-projective parser. We perform experiments on several datasets from the CoNLL-X shared task, showing that these variants outperform the original directed algorithms in most of the cases.

1 Introduction

Dependency parsing has proven to be very useful for natural language processing tasks. Data-driven dependency parsers such as those by Nivre et al. (2004), McDonald et al. (2005), Titov and Henderson (2007), Martins et al. (2009) or Huang and Sagae (2010) are accurate and efficient, they can be trained from annotated data without the need for a grammar, and they provide a simple representation of syntax that maps to predicate-argument structure in a straightforward way.

In particular, **transition-based** dependency parsers (Nivre, 2008) are a type of dependency parsing algorithms which use a model that scores transitions between parser states. Greedy deterministic search can be used to select the transition to be taken at each state, thus achieving linear or quadratic time complexity.

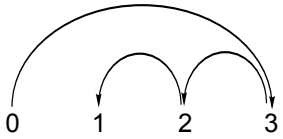


Figure 1: An example dependency structure where transition-based parsers enforcing the single-head constraint will incur in error propagation if they mistakenly build a dependency link $1 \rightarrow 2$ instead of $2 \rightarrow 1$ (dependency links are represented as arrows going from head to dependent).

It has been shown by McDonald and Nivre (2007) that such parsers suffer from **error propagation**: an early erroneous choice can place the parser in an incorrect state that will in turn lead to more errors. For instance, suppose that a sentence whose correct analysis is the dependency graph in Figure 1 is analyzed by any bottom-up or left-to-right transition-based parser that outputs dependency trees, therefore obeying the **single-head constraint** (only one incoming arc is allowed per node). If the parser chooses an erroneous transition that leads it to build a dependency link from 1 to 2 instead of the correct link from 2 to 1, this will lead it to a state where the single-head constraint makes it illegal to create the link from 3 to 2. Therefore, a single erroneous choice will cause two attachment errors in the output tree.

With the goal of minimizing these sources of errors, we obtain novel *undirected* variants of several parsers; namely, of the planar and 2-planar parsers by Gómez-Rodríguez and Nivre (2010) and the non-projective list-based parser described by Nivre (2008), which is based on Covington's algorithm (Covington, 2001). These variants work by collapsing the LEFT-ARC and

19
Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics, pages 66–76,
Avignon, France, April 23 - 27 2012.

RIGHT-ARC transitions in the original parsers, which create right-to-left and left-to-right dependency links, into a single ARC transition creating an undirected link. This has the advantage that the single-head constraint need not be observed during the parsing process, since the directed notions of head and dependent are lost in undirected graphs. This gives the parser more freedom and can prevent situations where enforcing the constraint leads to error propagation, as in Figure 1.

On the other hand, these new algorithms have the disadvantage that their output is an undirected graph, which has to be post-processed to recover the direction of the dependency links and generate a valid dependency tree. Thus, some complexity is moved from the parsing process to this post-processing step; and each undirected parser will outperform the directed version only if the simplification of the parsing phase is able to avoid more errors than are generated by the post-processing. As will be seen in latter sections, experimental results indicate that this is in fact the case.

The rest of this paper is organized as follows: Section 2 introduces some notation and concepts that we will use throughout the paper. In Section 3, we present the undirected versions of the parsers by Gómez-Rodríguez and Nivre (2010) and Nivre (2008), as well as some considerations about the feature models suitable to train them. In Section 4, we discuss post-processing techniques that can be used to recover dependency trees from undirected graphs. Section 5 presents an empirical study of the performance obtained by these parsers, and Section 6 contains a final discussion.

2 Preliminaries

2.1 Dependency Graphs

Let $w = w_1 \dots w_n$ be an input string. A **dependency graph** for w is a directed graph $G = (V_w, E)$, where $V_w = \{0, \dots, n\}$ is the set of nodes, and $E \subseteq V_w \times V_w$ is the set of directed arcs. Each node in V_w encodes the position of a token in w, and each arc in E encodes a dependency relation between two tokens. We write $i \rightarrow j$ to denote a directed arc (i, j), which will also be called a **dependency link** from i to j.[1] We say that i is the **head** of j and, conversely, that j is a syntactic **dependent** of i.

Given a dependency graph $G = (V_w, E)$, we write $i \rightarrow^\star j \in E$ if there is a (possibly empty) directed path from i to j; and $i \leftrightarrow^\star j \in E$ if there is a (possibly empty) path between i and j in the undirected graph underlying G (omitting the references to E when clear from the context).

Most dependency-based representations of syntax do not allow arbitrary dependency graphs, instead, they are restricted to acyclic graphs that have at most one head per node. Dependency graphs satisfying these constraints are called dependency forests.

Definition 1 *A dependency graph G is said to be a **forest** iff it satisfies:*

1. *Acyclicity constraint: if $i \rightarrow^\star j$, then not $j \rightarrow i$.*
2. *Single-head constraint: if $j \rightarrow i$, then there is no $k \neq j$ such that $k \rightarrow i$.*

A node that has no head in a dependency forest is called a **root**. Some dependency frameworks add the additional constraint that dependency forests have only one root (or, equivalently, that they are connected). Such a forest is called a **dependency tree**. A dependency tree can be obtained from any dependency forest by linking all of its root nodes as dependents of a dummy root node, conventionally located in position 0 of the input.

2.2 Transition Systems

In the framework of Nivre (2008), transition-based parsers are described by means of a non-deterministic state machine called a **transition system**.

Definition 2 *A **transition system** for dependency parsing is a tuple $S = (C, T, c_s, C_t)$, where*

1. *C is a set of possible parser **configurations**,*
2. *T is a finite set of **transitions**, which are partial functions $t : C \rightarrow C$,*
3. *c_s is a total initialization function mapping each input string to a unique **initial configuration**, and*
4. *$C_t \subseteq C$ is a set of **terminal configurations**.*

To obtain a deterministic parser from a non-deterministic transition system, an **oracle** is used to deterministically select a single transition at

[1] In practice, dependency links are usually **labeled**, but to simplify the presentation we will ignore labels throughout most of the paper. However, all the results and algorithms presented can be applied to labeled dependency graphs and will be so applied in the experimental evaluation.

each configuration. An oracle for a transition system $S = (C, T, c_s, C_t)$ is a function $o : C \rightarrow T$. Suitable oracles can be obtained in practice by training classifiers on treebank data (Nivre et al., 2004).

2.3 The Planar, 2-Planar and Covington Transition Systems

Our undirected dependency parsers are based on the planar and 2-planar transition systems by Gómez-Rodríguez and Nivre (2010) and the version of the Covington (2001) non-projective parser defined by Nivre (2008). We now outline these directed parsers briefly, a more detailed description can be found in the above references.

2.3.1 Planar

The planar transition system by Gómez-Rodríguez and Nivre (2010) is a linear-time transition-based parser for **planar** dependency forests, i.e., forests whose dependency arcs do not cross when drawn above the words. The set of planar dependency structures is a very mild extension of that of projective structures (Kuhlmann and Nivre, 2006).

Configurations in this system are of the form $c = \langle \Sigma, B, A \rangle$ where Σ and B are disjoint lists of nodes from V_w (for some input w), and A is a set of dependency links over V_w. The list B, called the **buffer**, holds the input words that are still to be read. The list Σ, called the **stack**, is initially empty and is used to hold words that have dependency links pending to be created. The system is shown at the top in Figure 2, where the notation $\Sigma \mid i$ is used for a stack with top i and tail Σ, and we invert the notation for the buffer for clarity (i.e., $i \mid B$ as a buffer with top i and tail B).

The system reads the input sentence and creates links in a left-to-right order by executing its four transitions, until it gets to a terminal configuration. A SHIFT transition moves the first (leftmost) node in the buffer to the top of the stack. Transitions LEFT-ARC and RIGHT-ARC create leftward or rightward link, respectively, involving the first node in the buffer and the topmost node in the stack. Finally, REDUCE transition is used to pop the top word from the stack when we have finished building arcs to or from it.

2.3.2 2-Planar

The 2-planar transition system by Gómez-Rodríguez and Nivre (2010) is an extension of the planar system that uses two stacks, allowing it to recognize 2-planar structures, a larger set of dependency structures that has been shown to cover the vast majority of non-projective structures in a number of treebanks (Gómez-Rodríguez and Nivre, 2010).

This transition system, shown in Figure 2, has configurations of the form $c = \langle \Sigma_0, \Sigma_1, B, A \rangle$, where we call Σ_0 the **active stack** and Σ_1 the **inactive stack**. Its SHIFT, LEFT-ARC, RIGHT-ARC and REDUCE transitions work similarly to those in the planar parser, but while SHIFT pushes the first word in the buffer to *both* stacks; the other three transitions only work with the top of the active stack, ignoring the inactive one. Finally, a SWITCH transition is added that makes the active stack inactive and vice versa.

2.3.3 Covington Non-Projective

Covington (2001) proposes several incremental parsing strategies for dependency representations and one of them can recover non-projective dependency graphs. Nivre (2008) implements a variant of this strategy as a transition system with configurations of the form $c = \langle \lambda_1, \lambda_2, B, A \rangle$, where λ_1 and λ_2 are **lists** containing partially processed words and B is the **buffer** list of unprocessed words.

The Covington non-projective transition system is shown at the bottom in Figure 2. At each configuration $c = \langle \lambda_1, \lambda_2, B, A \rangle$, the parser has to consider whether any dependency arc should be created involving the top of the buffer and the words in λ_1. A LEFT-ARC transition adds a link from the first node j in the buffer to the node in the head of the list λ_1, which is moved to the list λ_2 to signify that we have finished considering it as a possible head or dependent of j. The RIGHT-ARC transition does the same manipulation, but creating the symmetric link. A NO-ARC transition removes the head of the list λ_1 and inserts it at the head of the list λ_2 without creating any arcs: this transition is to be used where there is no dependency relation between the top node in the buffer and the head of λ_1, but we still may want to create an arc involving the top of the buffer and other nodes in λ_1. Finally, if we do not want to create any such arcs at all, we can execute a SHIFT transition, which advances the parsing process by removing the first node in the buffer B and inserting it at the head of a list obtained by concatenating

λ_1 and λ_2. This list becomes the new λ_1, whereas λ_2 is empty in the resulting configuration.

Note that the Covington parser has quadratic complexity with respect to input length, while the planar and 2-planar parsers run in linear time.

3 The Undirected Parsers

The transition systems defined in Section 2.3 share the common property that their LEFT-ARC and RIGHT-ARC have exactly the same effects except for the direction of the links that they create. We can take advantage of this property to define undirected versions of these transition systems, by transforming them as follows:

- Configurations are changed so that the arc set A is a set of undirected arcs, instead of directed arcs.
- The LEFT-ARC and RIGHT-ARC transitions in each parser are collapsed into a single ARC transition that creates an undirected arc.
- The preconditions of transitions that guarantee the single-head constraint are removed, since the notions of head and dependent are lost in undirected graphs.

By performing these transformations and leaving the systems otherwise unchanged, we obtain the undirected variants of the planar, 2-planar and Covington algorithms that are shown in Figure 3.

Note that the transformation can be applied to any transition system having LEFT-ARC and RIGHT-ARC transitions that are equal except for the direction of the created link, and thus collapsable into one. The above three transition systems fulfill this property, but not every transition system does. For example, the well-known arc-eager parser of Nivre (2003) pops a node from the stack when creating left arcs, and pushes a node to the stack when creating right arcs, so the transformation cannot be applied to it.[2]

[2]One might think that the arc-eager algorithm could still be transformed by converting each of its arc transitions into an undirected transition, without collapsing them into one. However, this would result into a parser that violates the acyclicity constraint, since the algorithm is designed in such a way that acyclicity is only guaranteed if the single-head constraint is kept. It is easy to see that this problem cannot happen in parsers where LEFT-ARC and RIGHT-ARC transitions have the same effect: in these, if a directed graph is not parsable in the original algorithm, its underlying undirected graph cannot not be parsable in the undirected variant.

3.1 Feature models

Some of the features that are typically used to train transition-based dependency parsers depend on the direction of the arcs that have been built up to a certain point. For example, two such features for the planar parser could be the POS tag associated with the head of the topmost stack node, or the label of the arc going from the first node in the buffer to its leftmost dependent.[3]

As the notion of head and dependent is lost in undirected graphs, this kind of features cannot be used to train undirected parsers. Instead, we use features based on undirected relations between nodes. We found that the following kinds of features worked well in practice as a replacement for features depending on arc direction:

- Information about the ith node linked to a given node (topmost stack node, topmost buffer node, etc.) on the left or on the right, and about the associated undirected arc, typically for $i = 1, 2, 3$,
- Information about whether two nodes are linked or not in the undirected graph, and about the label of the arc between them,
- Information about the first left and right "undirected siblings" of a given node, i.e., the first node q located to the left of the given node p such that p and q are linked to some common node r located to the right of both, and vice versa. Note that this notion of undirected siblings does not correspond exclusively to siblings in the directed graph, since it can also capture other second-order interactions, such as grandparents.

4 Reconstructing the dependency forest

The modified transition systems presented in the previous section generate undirected graphs. To obtain complete dependency parsers that are able to produce directed dependency forests, we will need a reconstruction step that will assign a direction to the arcs in such a way that the single-head constraint is obeyed. This reconstruction step can be implemented by building a directed graph with weighted arcs corresponding to both possible directions of each undirected edge, and then finding an optimum branching to reduce it to a directed

[3]These example features are taken from the default model for the planar parser in version 1.5 of MaltParser (Nivre et al., 2006).

Planar initial/terminal configurations:		$c_s(w_1 \dots w_n) = \langle [], [1 \dots n], \emptyset \rangle$, $C_f = \{\langle \Sigma, [], A \rangle \in C\}$
Transitions:	SHIFT	$\langle \Sigma, i\|B, A \rangle \Rightarrow \langle \Sigma\|i, B, A \rangle$
	REDUCE	$\langle \Sigma\|i, B, A \rangle \Rightarrow \langle \Sigma, B, A \rangle$
	LEFT-ARC	$\langle \Sigma\|i, j\|B, A \rangle \Rightarrow \langle \Sigma\|i, j\|B, A \cup \{(j, i)\} \rangle$ only if $\nexists k \mid (k, i) \in A$ (single-head) and $i \hookrightarrow^* j \notin A$ (acyclicity).
	RIGHT-ARC	$\langle \Sigma\|i, j\|B, A \rangle \Rightarrow \langle \Sigma\|i, j\|B, A \cup \{(i, j)\} \rangle$ only if $\nexists k \mid (k, j) \in A$ (single-head) and $i \hookrightarrow^* j \notin A$ (acyclicity).
2-Planar initial/terminal configurations:		$c_s(w_1 \dots w_n) = \langle [], [], [1 \dots n], \emptyset \rangle$, $C_f = \{\langle \Sigma_0, \Sigma_1, [], A \rangle \in C\}$
Transitions:	SHIFT	$\langle \Sigma_0, \Sigma_1, i\|B, A \rangle \Rightarrow \langle \Sigma_0\|i, \Sigma_1\|i, B, A \rangle$
	REDUCE	$\langle \Sigma_0\|i, \Sigma_1, B, A \rangle \Rightarrow \langle \Sigma_0, \Sigma_1, B, A \rangle$
	LEFT-ARC	$\langle \Sigma_0\|i, \Sigma_1, j\|B, A \rangle \Rightarrow \langle \Sigma_0\|i, \Sigma_1, j\|B, A \cup \{j, i)\} \rangle$ only if $\nexists k \mid (k, i) \in A$ (single-head) and $i \hookrightarrow^* j \notin A$ (acyclicity).
	RIGHT-ARC	$\langle \Sigma_0\|i, \Sigma_1, j\|B, A \rangle \Rightarrow \langle \Sigma_0\|i, \Sigma_1, j\|B, A \cup \{(i, j)\} \rangle$ only if $\nexists k \mid (k, j) \in A$ (single-head) and $i \hookrightarrow^* j \notin A$ (acyclicity).
	SWITCH	$\langle \Sigma_0, \Sigma_1, B, A \rangle \Rightarrow \langle \Sigma_1, \Sigma_0, B, A \rangle$
Covington initial/term. configurations:		$c_s(w_1 \dots w_n) = \langle [], [], [1 \dots n], \emptyset \rangle$, $C_f = \{\langle \lambda_1, \lambda_2, [], A \rangle \in C\}$
Transitions:	SHIFT	$\langle \lambda_1, \lambda_2, i\|B, A \rangle \Rightarrow \langle \lambda_1 \cdot \lambda_2\|i, [], B, A \rangle$
	NO-ARC	$\langle \lambda_1\|i, \lambda_2, B, A \rangle \Rightarrow \langle \lambda_1, i\|\lambda_2, B, A \rangle$
	LEFT-ARC	$\langle \lambda_1\|i, \lambda_2, j\|B, A \rangle \Rightarrow \langle \lambda_1, i\|\lambda_2, j\|B, A \cup \{(j, i)\} \rangle$ only if $\nexists k \mid (k, i) \in A$ (single-head) and $i \hookrightarrow^* j \notin A$ (acyclicity).
	RIGHT-ARC	$\langle \lambda_1\|i, \lambda_2, j\|B, A \rangle \Rightarrow \langle \lambda_1, i\|\lambda_2, j\|B, A \cup \{(i, j)\} \rangle$ only if $\nexists k \mid (k, j) \in A$ (single-head) and $i \hookrightarrow^* j \notin A$ (acyclicity).

Figure 2: Transition systems for planar, 2-planar and Covington non-projective dependency parsing.

Undirected Planar initial/term. conf.:		$c_s(w_1 \dots w_n) = \langle [], [1 \dots n], \emptyset \rangle$, $C_f = \{\langle \Sigma, [], A \rangle \in C\}$
Transitions:	SHIFT	$\langle \Sigma, i\|B, A \rangle \Rightarrow \langle \Sigma\|i, B, A \rangle$
	REDUCE	$\langle \Sigma\|i, B, A \rangle \Rightarrow \langle \Sigma, B, A \rangle$
	ARC	$\langle \Sigma\|i, j\|B, A \rangle \Rightarrow \langle \Sigma\|i, j\|B, A \cup \{\{i, j\}\} \rangle$ only if $i \hookrightarrow^* j \notin A$ (acyclicity).
Undirected 2-Planar initial/term. conf.:		$c_s(w_1 \dots w_n) = \langle [], [], [1 \dots n], \emptyset \rangle$, $C_f = \{\langle \Sigma_0, \Sigma_1, [], A \rangle \in C\}$
Transitions:	SHIFT	$\langle \Sigma_0, \Sigma_1, i\|B, A \rangle \Rightarrow \langle \Sigma_0\|i, \Sigma_1\|i, B, A \rangle$
	REDUCE	$\langle \Sigma_0\|i, \Sigma_1, B, A \rangle \Rightarrow \langle \Sigma_0, \Sigma_1, B, A \rangle$
	ARC	$\langle \Sigma_0\|i, \Sigma_1, j\|B, A \rangle \Rightarrow \langle \Sigma_0\|i, \Sigma_1, j\|B, A \cup \{\{i, j\}\} \rangle$ only if $i \hookrightarrow^* j \notin A$ (acyclicity).
	SWITCH	$\langle \Sigma_0, \Sigma_1, B, A \rangle \Rightarrow \langle \Sigma_1, \Sigma_0, B, A \rangle$
Undirected Covington init./term. conf.:		$c_s(w_1 \dots w_n) = \langle [], [], [1 \dots n], \emptyset \rangle$, $C_f = \{\langle \lambda_1, \lambda_2, [], A \rangle \in C\}$
Transitions:	SHIFT	$\langle \lambda_1, \lambda_2, i\|B, A \rangle \Rightarrow \langle \lambda_1 \cdot \lambda_2\|i, [], B, A \rangle$
	NO-ARC	$\langle \lambda_1\|i, \lambda_2, B, A \rangle \Rightarrow \langle \lambda_1, i\|\lambda_2, B, A \rangle$
	ARC	$\langle \lambda_1\|i, \lambda_2, j\|B, A \rangle \Rightarrow \langle \lambda_1, i\|\lambda_2, j\|B, A \cup \{\{i, j\}\} \rangle$ only if $i \hookrightarrow^* j \notin A$ (acyclicity).

Figure 3: Transition systems for undirected planar, 2-planar and Covington non-projective dependency parsing.

tree. Different criteria for assigning weights to arcs provide different variants of the reconstruction technique.

To describe these variants, we first introduce preliminary definitions. Let $U = (V_w, E)$ be an undirected graph produced by an undirected parser for some string w. We define the following sets of arcs:

$$A_1(U) = \{(i,j) \mid j \neq 0 \wedge \{i,j\} \in E\},$$
$$A_2(U) = \{(0,i) \mid i \in V_w\}.$$

Note that $A_1(U)$ represents the set of arcs obtained from assigning an orientation to an edge in U, except arcs whose dependent is the dummy root, which are disallowed. On the other hand, $A_2(U)$ contains all the possible arcs originating from the dummy root node, regardless of whether their underlying undirected edges are in U or not; this is so that reconstructions are allowed to link unattached tokens to the dummy root.

The reconstruction process consists of finding a minimum branching (i.e. a directed minimum spanning tree) for a weighted directed graph obtained from assigning a cost $c(i,j)$ to each arc (i,j) of the following directed graph:

$$D(U) = \{V_w, A(U) = A_1(U) \cup A_2(U)\}.$$

That is, we will find a dependency tree $T = (V_w, A_T \subseteq A(U))$ such that the sum of costs of the arcs in A_T is minimal. In general, such a minimum branching can be calculated with the Chu-Liu-Edmonds algorithm (Chu and Liu, 1965; Edmonds, 1967). Since the graph $D(U)$ has $O(n)$ nodes and $O(n)$ arcs for a string of length n, this can be done in $O(n \log n)$ if implemented as described by Tarjan (1977).

However, applying these generic techniques is not necessary in this case: since our graph U is acyclic, the problem of reconstructing the forest can be reduced to choosing a root word for each connected component in the graph, linking it as a dependent of the dummy root and directing the other arcs in the component in the (unique) way that makes them point away from the root.

It remains to see how to assign the costs $c(i,j)$ to the arcs of $D(U)$: different criteria for assigning scores will lead to different reconstructions.

4.1 Naive reconstruction

A first, very simple reconstruction technique can be obtained by assigning arc costs to the arcs in $A(U)$ as follows:

$$c(i,j) \begin{cases} 1 & \text{if } (i,j) \in A_1(U), \\ 2 & \text{if } (i,j) \in A_2(U) \wedge (i,j) \notin A_1(U). \end{cases}$$

This approach gives the same cost to all arcs obtained from the undirected graph U, while also allowing (at a higher cost) to attach any node to the dummy root. To obtain satisfactory results with this technique, we must train our parser to explicitly build undirected arcs from the dummy root node to the root word(s) of each sentence using arc transitions (note that this implies that we need to represent forests as trees, in the manner described at the end of Section 2.1). Under this assumption, it is easy to see that we can obtain the correct directed tree T for a sentence if it is provided with its underlying undirected tree U: the tree is obtained in $O(n)$ as the unique orientation of U that makes each of its edges point away from the dummy root.

This approach to reconstruction has the advantage of being very simple and not adding any complications to the parsing process, while guaranteeing that the correct directed tree will be recovered if the undirected tree for a sentence is generated correctly. However, it is not very robust, since the direction of all the arcs in the output depends on which node is chosen as sentence head and linked to the dummy root. Therefore, a parsing error affecting the undirected edge involving the dummy root may result in many dependency links being erroneous.

4.2 Label-based reconstruction

To achieve a more robust reconstruction, we use labels to encode a preferred direction for dependency arcs. To do so, for each pre-existing label X in the training set, we create two labels X_l and X_r. The parser is then trained on a modified version of the training set where leftward links originally labelled X are labelled X_l, and rightward links originally labelled X are labelled X_r. Thus, the output of the parser on a new sentence will be an undirected graph where each edge has a label with an annotation indicating whether the reconstruction process should prefer to link the pair of nodes with a leftward or a rightward arc. We can then assign costs to our minimum branching algorithm so that it will return a tree agreeing with as many such annotations as possible.

To do this, we call $A_{1+}(U) \subseteq A_1(U)$ the set of arcs in $A_1(U)$ that agree with the annotations, i.e., arcs $(i,j) \in A_1(U)$ where either $i < j$ and i, j is labelled X_r in U, or $i > j$ and i, j is labelled X_l in U. We call $A_{1-}(U)$ the set of arcs in $A_1(U)$ that disagree with the annotations, i.e., $A_{1-}(U) = A_1(U) \setminus A_{1+}(U)$. And we assign costs as follows:

$$c(i,j) \begin{cases} 1 & \text{if } (i,j) \in A_{1+}(U), \\ 2 & \text{if } (i,j) \in A_{1-}(U), \\ 2n & \text{if } (i,j) \in A_2(U) \wedge (i,j) \notin A_1(U). \end{cases}$$

where n is the length of the string.

With these costs, the minimum branching algorithm will find a tree which agrees with as many annotations as possible. Additional arcs from the root not corresponding to any edge in the output of the parser (i.e. arcs in $A_2(U)$ but not in $A_1(U)$) will be used only if strictly necessary to guarantee connectedness, this is implemented by the high cost for these arcs.

While this may be the simplest cost assignment to implement label-based reconstruction, we have found that better empirical results are obtained if we give the algorithm more freedom to create new arcs from the root, as follows:

$$c(i,j) \begin{cases} 1 & \text{if } (i,j) \in A_{1+}(U) \wedge (i,j) \notin A_2(U), \\ 2 & \text{if } (i,j) \in A_{1-}(U) \wedge (i,j) \notin A_2(U), \\ 2n & \text{if } (i,j) \in A_2(U). \end{cases}$$

While the cost of arcs from the dummy root is still $2n$, this is now so even for arcs that are in the output of the undirected parser, which had cost 1 before. Informally, this means that with this configuration the postprocessor does not "trust" the links from the dummy root created by the parser, and may choose to change them if it is convenient to get a better agreement with label annotations (see Figure 4 for an example of the difference between both cost assignments). We believe that the better accuracy obtained with this criterion probably stems from the fact that it is biased towards changing links from the root, which tend to be more problematic for transition-based parsers, while respecting the parser output for links located deeper in the dependency structure, for which transition-based parsers tend to be more accurate (McDonald and Nivre, 2007).

Note that both variants of label-based reconstruction have the property that, if the undirected parser produces the correct edges and labels for a given sentence, then the obtained directed tree is guaranteed to be correct (as it will simply be the tree obtained by decoding the label annotations).

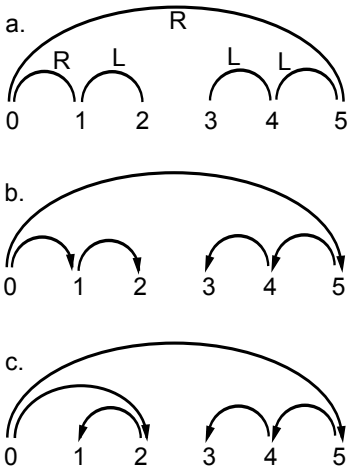


Figure 4: a) An undirected graph obtained by the parser with the label-based transformation, b) and c) The dependency graph obtained by each of the variants of the label-based reconstruction (note how the second variant moves an arc from the root).

5 Experiments

In this section, we evaluate the performance of the undirected planar, 2-planar and Covington parsers on eight datasets from the CoNLL-X shared task (Buchholz and Marsi, 2006).

Tables 1, 2 and 3 compare the accuracy of the undirected versions with naive and label-based reconstruction to that of the directed versions of the planar, 2-planar and Covington parsers, respectively. In addition, we provide a comparison to well-known state-of-the-art projective and non-projective parsers: the planar parsers are compared to the arc-eager projective parser by Nivre (2003), which is also restricted to planar structures; and the 2-planar parsers are compared with the arc-eager parser with pseudo-projective transformation of Nivre and Nilsson (2005), capable of handling non-planar dependencies.

We use SVM classifiers from the LIBSVM package (Chang and Lin, 2001) for all the languages except Chinese, Czech and German. In these, we use the LIBLINEAR package (Fan et al., 2008) for classification, which reduces training time for these larger datasets; and feature models adapted to this system which, in the case of German, result in higher accuracy than published results using LIBSVM.

The LIBSVM feature models for the arc-eager projective and pseudo-projective parsers are the same used by these parsers in the CoNLL-X shared task, where the pseudo-projective version of MaltParser was one of the two top performing systems (Buchholz and Marsi, 2006). For the 2-planar parser, we took the feature models from Gómez-Rodríguez and Nivre (2010) for the languages included in that paper. For all the algorithms and datasets, the feature models used for the undirected parsers were adapted from those of the directed parsers as described in Section 3.1.[4]

The results show that the use of undirected parsing with label-based reconstruction clearly improves the performance in the vast majority of the datasets for the planar and Covington algorithms, where in many cases it also improves upon the corresponding projective and non-projective state-of-the-art parsers provided for comparison. In the case of the 2-planar parser the results are less conclusive, with improvements over the directed versions in five out of the eight languages.

The improvements in LAS obtained with label-based reconstruction over directed parsing are statistically significant at the .05 level[5] for Danish, German and Portuguese in the case of the planar parser; and Czech, Danish and Turkish for Covington's parser. No statistically significant decrease in accuracy was detected in any of the algorithm/dataset combinations.

As expected, the good results obtained by the undirected parsers with label-based reconstruction contrast with those obtained by the variants with root-based reconstruction, which performed worse in all the experiments.

6 Discussion

We have presented novel variants of the planar and 2-planar transition-based parsers by Gómez-Rodríguez and Nivre (2010) and of Covington's non-projective parser (Covington, 2001; Nivre, 2008) which ignore the direction of dependency links, and reconstruction techniques that can be used to recover the direction of the arcs thus produced. The results obtained show that this idea of undirected parsing, together with the label-based reconstruction technique of Section 4.2, improves parsing accuracy on most of the tested dataset/algorithm combinations, and it can outperform state-of-the-art transition-based parsers.

The accuracy improvements achieved by relaxing the single-head constraint to mitigate error propagation were able to overcome the errors generated in the reconstruction phase, which were few: we observed empirically that the differences between the undirected LAS obtained from the undirected graph before the reconstruction and the final directed LAS are typically below 0.20%. This is true both for the naive and label-based transformations, indicating that both techniques are able to recover arc directions accurately, and the accuracy differences between them come mainly from the differences in training (e.g. having tentative arc direction as part of feature information in the label-based reconstruction and not in the naive one) rather than from the differences in the reconstruction methods themselves.

The reason why we can apply the undirected simplification to the three parsers that we have used in this paper is that their LEFT-ARC and RIGHT-ARC transitions have the same effect except for the direction of the links they create. The same transformation and reconstruction techniques could be applied to any other transition-based dependency parsers sharing this property. The reconstruction techniques alone could potentially be applied to any dependency parser (transition-based or not) as long as it can be somehow converted to output undirected graphs.

The idea of parsing with undirected relations between words has been applied before in the work on Link Grammar (Sleator and Temperley, 1991), but in that case the formalism itself works with undirected graphs, which are the final output of the parser. To our knowledge, the idea of using an undirected graph as an intermediate step towards obtaining a dependency structure has not been explored before.

Acknowledgments

This research has been partially funded by the Spanish Ministry of Economy and Competitiveness and FEDER (projects TIN2010-18552-C03-01 and TIN2010-18552-C03-02), Ministry of Education (FPU Grant Program) and Xunta de Galicia (Rede Galega de Recursos Lingüísticos para unha Soc. do Coñec.). The experiments were conducted with the help of computing resources provided by the Supercomputing Center of Galicia (CESGA). We thank Joakim Nivre for helpful input in the early stages of this work.

[4] All the experimental settings and feature models used are included in the supplementary material and also available at `http://www.grupolys.org/~cgomezr/exp/`.

[5] Statistical significance was assessed using Dan Bikel's randomized comparator: `http://www.cis.upenn.edu/~dbikel/software.html`

	Planar		UPlanarN		UPlanarL		MaltP	
Lang.	LAS(p)	UAS(p)	LAS(p)	UAS(p)	LAS(p)	UAS(p)	LAS(p)	UAS(p)
Arabic	**66.93** (**67.34**)	**77.56** (**77.22**)	65.91 (66.33)	77.03 (76.75)	66.75 (67.19)	77.45 (**77.22**)	66.43 (66.74)	77.19 (76.83)
Chinese	84.23 (84.20)	88.37 (88.33)	83.14 (83.10)	87.00 (86.95)	84.51* (84.50*)	88.37 (88.35*)	**86.42** (**86.39**)	**90.06** (**90.02**)
Czech	77.24 (77.70)	83.46 (83.24)	75.08 (75.60)	81.14 (81.14)	**77.60*** (**77.93***)	**83.56*** (**83.41***)	77.24 (77.57)	83.40 (83.19)
Danish	83.31 (82.60)	88.02 (86.64)	82.65 (82.45)	87.58 (86.67*)	**83.87*** (**83.83***)	**88.94*** (**88.17***)	83.31 (82.64)	88.30 (86.91)
German	84.66 (83.60)	87.02 (85.67)	83.33 (82.77)	85.78 (84.93)	**86.32*** (**85.67***)	**88.62*** (**87.69***)	86.12 (85.48)	88.52 (87.58)
Portug.	86.22 (83.82)	89.80 (86.88)	85.89 (83.82)	89.68 (87.06*)	86.52* (**84.83***)	**90.28*** (**88.03***)	**86.60** (84.66)	90.20 (87.73)
Swedish	**83.01** (82.44)	88.53 (87.36)	81.20 (81.10)	86.50 (85.86)	82.95 (**82.66***)	88.29 (87.45*)	82.89 (82.44)	**88.61** (**87.55**)
Turkish	62.70 (71.27)	73.67 (78.57)	59.83 (68.31)	70.15 (75.17)	**63.27*** (**71.63***)	**73.93*** (**78.72***)	62.58 (70.96)	73.09 (77.95)

Table 1: Parsing accuracy of the undirected planar parser with naive (UPlanarN) and label-based (UPlanarL) postprocessing in comparison to the directed planar (Planar) and the MaltParser arc-eager projective (MaltP) algorithms, on eight datasets from the CoNLL-X shared task (Buchholz and Marsi, 2006): Arabic (Hajič et al., 2004), Chinese (Chen et al., 2003), Czech (Hajič et al., 2006), Danish (Kromann, 2003), German (Brants et al., 2002), Portuguese (Afonso et al., 2002), Swedish (Nilsson et al., 2005) and Turkish (Oflazer et al., 2003; Atalay et al., 2003). We show labelled (LAS) and unlabelled (UAS) attachment score excluding and including punctuation tokens in the scoring (the latter in brackets). Best results for each language are shown in boldface, and results where the undirected parser outperforms the directed version are marked with an asterisk.

	2Planar		U2PlanarN		U2PlanarL		MaltPP	
Lang.	LAS(p)	UAS(p)	LAS(p)	UAS(p)	LAS(p)	UAS(p)	LAS(p)	UAS(p)
Arabic	**66.73** (**67.19**)	**77.33** (**77.11**)	66.37 (66.93)	77.15 (77.09)	66.13 (66.52)	76.97 (76.70)	65.93 (66.02)	76.79 (76.14)
Chinese	84.35 (84.32)	88.31 (88.27)	83.02 (82.98)	86.86 (86.81)	84.45* (84.42*)	88.29 (88.25)	86.42 (86.39)	90.06 (90.02)
Czech	77.72 (77.91)	83.76 (83.32)	74.44 (75.19)	80.68 (80.80)	78.00* (**78.59***)	84.22* (**84.21***)	**78.86** (78.47)	**84.54** (83.89)
Danish	**83.81** (83.61)	88.50 (87.63)	82.00 (81.63)	86.87 (85.80)	83.75 (**83.65***)	**88.62*** (**87.82***)	83.67 (83.54)	88.52 (87.70)
German	86.28 (85.76)	88.68 (87.86)	82.93 (82.53)	85.52 (84.81)	86.52* (85.99*)	88.72* (87.92*)	**86.94** (**86.62**)	**89.30** (**88.69**)
Portug.	87.04 (**84.92**)	**90.82** (**88.14**)	85.61 (83.45)	89.36 (86.65)	86.70 (84.75)	90.38 (87.88)	**87.08** (84.90)	90.66 (87.95)
Swedish	83.13 (**82.71**)	88.57 (**87.59**)	81.00 (80.71)	86.54 (85.68)	82.59 (82.25)	88.19 (87.29)	**83.39** (82.67)	**88.59** (87.38)
Turkish	61.80 (70.09)	72.75 (77.39)	58.10 (67.44)	68.03 (74.06)	61.92* (70.64*)	72.18 (77.46*)	**62.80** (**71.33**)	**73.49** (**78.44**)

Table 2: Parsing accuracy of the undirected 2-planar parser with naive (U2PlanarN) and label-based (U2PlanarL) postprocessing in comparison to the directed 2-planar (2Planar) and MaltParser arc-eager pseudo-projective (MaltPP) algorithms. The meaning of the scores shown is as in Table 1.

	Covington		UCovingtonN		UCovingtonL	
Lang.	LAS(p)	UAS(p)	LAS(p)	UAS(p)	LAS(p)	UAS(p)
Arabic	65.17 (65.49)	75.99 (**75.69**)	63.49 (63.93)	74.41 (74.20)	**65.61*** (**65.81***)	**76.11*** (75.66)
Chinese	85.61 (85.61)	89.64 (89.62)	84.12 (84.02)	87.85 (87.73)	**86.28*** (**86.17***)	**90.16*** (**90.04***)
Czech	78.26 (77.43)	84.04 (83.15)	74.02 (74.78)	79.80 (79.92)	**78.42*** (**78.69***)	**84.50*** (**84.16***)
Danish	83.63 (82.89)	88.50 (87.06)	82.00 (81.61)	86.55 (85.51)	**84.27*** (**83.85***)	**88.82*** (**87.75***)
German	**86.70** (85.69)	**89.08** (87.78)	84.03 (83.51)	86.16 (85.39)	86.50 (**85.90***)	88.84 (**87.95***)
Portug.	84.73 (82.56)	89.10 (86.30)	83.83 (81.71)	87.88 (85.17)	**84.95*** (**82.70***)	**89.18*** (**86.31***)
Swedish	**83.53** (**82.76**)	**88.91** (**87.61**)	81.78 (81.47)	86.78 (85.96)	83.09 (82.73)	88.11 (87.23)
Turkish	64.25 (72.70)	74.85 (79.75)	63.51 (72.08)	74.07 (79.10)	**64.91*** (**73.38***)	**75.46*** (**80.40***)

Table 3: Parsing accuracy of the undirected Covington non-projective parser with naive (UCovingtonN) and label-based (UCovingtonL) postprocessing in comparison to the directed algorithm (Covington). The meaning of the scores shown is as in Table 1.

References

Susana Afonso, Eckhard Bick, Renato Haber, and Diana Santos. 2002. "Floresta sintá(c)tica": a treebank for Portuguese. In *Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC 2002)*, pages 1968–1703, Paris, France. ELRA.

Nart B. Atalay, Kemal Oflazer, and Bilge Say. 2003. The annotation process in the Turkish treebank. In *Proceedings of EACL Workshop on Linguistically Interpreted Corpora (LINC-03)*, pages 243–246, Morristown, NJ, USA. Association for Computational Linguistics.

Sabine Brants, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius, and George Smith. 2002. The tiger treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories, September 20-21*, Sozopol, Bulgaria.

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pages 149–164.

Chih-Chung Chang and Chih-Jen Lin, 2001. *LIBSVM: A Library for Support Vector Machines*. Software available at http://www.csie.ntu.edu.tw/∼cjlin/libsvm.

K. Chen, C. Luo, M. Chang, F. Chen, C. Chen, C. Huang, and Z. Gao. 2003. Sinica treebank: Design criteria, representational issues and implementation. In Anne Abeillé, editor, *Treebanks: Building and Using Parsed Corpora*, chapter 13, pages 231–248. Kluwer.

Y. J. Chu and T. H. Liu. 1965. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400.

Michael A. Covington. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102.

Jack Edmonds. 1967. Optimum branchings. *Journal of Research of the National Bureau of Standards*, 71B:233–240.

R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. 2008. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874.

Carlos Gómez-Rodríguez and Joakim Nivre. 2010. A transition-based parser for 2-planar dependency structures. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 1492–1501, Stroudsburg, PA, USA. Association for Computational Linguistics.

Jan Hajič, Otakar Smrž, Petr Zemánek, Jan Šnaidauf, and Emanuel Beška. 2004. Prague Arabic Dependency Treebank: Development in data and tools. In *Proceedings of the NEMLAR International Conference on Arabic Language Resources and Tools*.

Jan Hajič, Jarmila Panevová, Eva Hajičová, Jarmila Panevová, Petr Sgall, Petr Pajas, Jan Štěpánek, Jiří Havelka, and Marie Mikulová. 2006. Prague Dependency Treebank 2.0. CDROM CAT: LDC2006T01, ISBN 1-58563-370-4. Linguistic Data Consortium.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 1077–1086, Stroudsburg, PA, USA. Association for Computational Linguistics.

Matthias T. Kromann. 2003. The Danish dependency treebank and the underlying linguistic theory. In *Proceedings of the 2nd Workshop on Treebanks and Linguistic Theories (TLT)*, pages 217–220, Växjö, Sweden. Växjö University Press.

Marco Kuhlmann and Joakim Nivre. 2006. Mildly non-projective dependency structures. In *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*, pages 507–514.

Andre Martins, Noah Smith, and Eric Xing. 2009. Concise integer linear programming formulations for dependency parsing. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP (ACL-IJCNLP)*, pages 342–350.

Ryan McDonald and Joakim Nivre. 2007. Characterizing the errors of data-driven dependency parsing models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 122–131.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the Human Language Technology Conference and the Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP)*, pages 523–530.

Jens Nilsson, Johan Hall, and Joakim Nivre. 2005. MAMBA meets TIGER: Reconstructing a Swedish treebank from Antiquity. In Peter Juel Henrichsen, editor, *Proceedings of the NODALIDA Special Session on Treebanks*.

Joakim Nivre and Jens Nilsson. 2005. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 99–106.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2004. Memory-based dependency parsing. In *Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL-2004)*, pages 49–56, Morristown, NJ, USA. Association for Computational Linguistics.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006. MaltParser: A data-driven parser-generator for dependency parsing. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC)*, pages 2216–2219.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 149–160.

Joakim Nivre. 2008. Algorithms for Deterministic Incremental Dependency Parsing. *Computational Linguistics*, 34(4):513–553.

Kemal Oflazer, Bilge Say, Dilek Zeynep Hakkani-Tür, and Gökhan Tür. 2003. Building a Turkish treebank. In Anne Abeillé, editor, *Treebanks: Building and Using Parsed Corpora*, pages 261–277. Kluwer.

Daniel Sleator and Davy Temperley. 1991. Parsing English with a link grammar. Technical Report CMU-CS-91-196, Carnegie Mellon University, Computer Science.

R. E. Tarjan. 1977. Finding optimum branchings. *Networks*, 7:25–35.

Ivan Titov and James Henderson. 2007. A latent variable model for generative dependency parsing. In *Proceedings of the 10th International Conference on Parsing Technologies (IWPT)*, pages 144–155.

Computational Intelligence

An International Journal

Computational Intelligence, Volume 31, Number 2, 2015

UNDIRECTED DEPENDENCY PARSING

Carlos Gómez-Rodríguez,[1] Daniel Fernández-González,[2] and Víctor Manuel Darriba Bilbao[2]

[1]*Departamento de Computación, Facultad de Informática, Universidade da Coruña, Campus de Elviña s/n, Coruña, Spain*

[2]*Departamento de Informática, Universidade de Vigo, Campus As Lagoas, Ourense, Spain*

Dependency parsers, which are widely used in natural language processing tasks, employ a representation of syntax in which the structure of sentences is expressed in the form of directed links (dependencies) between their words. In this article, we introduce a new approach to transition-based dependency parsing in which the parsing algorithm does not directly construct dependencies, but rather undirected links, which are then assigned a direction in a postprocessing step. We show that this alleviates error propagation, because undirected parsers do not need to observe the single-head constraint, resulting in better accuracy.

Undirected parsers can be obtained by transforming existing directed transition-based parsers as long as they satisfy certain conditions. We apply this approach to obtain undirected variants of three different parsers (the Planar, 2-Planar, and Covington algorithms) and perform experiments on several data sets from the CoNLL-X shared tasks and on the *Wall Street Journal* portion of the Penn Treebank, showing that our approach is successful in reducing error propagation and produces improvements in parsing accuracy in most of the cases and achieving results competitive with state-of-the-art transition-based parsers.

Received 19 February 2013; Revised 22 November 2013; Accepted 1 December 2013

Key words: natural language processing, dependency parsing, parsing, computational linguistics, automata.

1. INTRODUCTION

Syntactic parsing is the process of determining the grammatical structure of a sentence: Given an input sentence, a parsing algorithm (or parser) will analyze it to output a representation of its underlying structure. The format of this representation and the information it contains depend on the particular syntactic theory used by the parser. In constituency parsers, or phrase structure parsers, sentences are analyzed by breaking them down into meaningful parts called constituents, which are in turn divided into smaller constituents. The result of such an analysis is represented with a constituency tree, such as the one shown in Figure 1. On the other hand, in dependency parsers, the structure of the sentence is represented by a set of directed links (called dependencies) between its words, forming a graph such as the one in Figure 2.

Dependency parsing has gained wide popularity in the natural language processing community, and it has recently been applied to a wide range of problems, such as machine translation (Ding and Palmer 2005; Shen, Xu, and Weischedel 2008; Xu et al. 2009; Katz-Brown et al. 2011), textual entailment recognition (Herrera, Peñas, and Verdejo 2005; Berant, Dagan, and Goldberger 2010), relation extraction (Culotta and Sorensen 2004; Fundel, Küffner, and Zimmer 2006; Miyao et al. 2009; Katrenko, Adriaans, and van Someren 2010), question answering (Cui et al. 2005; Comas, Turmo, and Márquez 2010), opinion mining (Joshi and Penstein-Rosé 2009), or learning for game artificial intelligence agents (Branavan, Silver, and Barzilay 2012).

Address correspondence to Carlos Gómez-Rodríguez, Departamento de Computación, Facultad de Informática, Universidade da Coruña, Campus de Elviña s/n, 15071 A Coruña, Spain; e-mail: cgomezr@udc.es

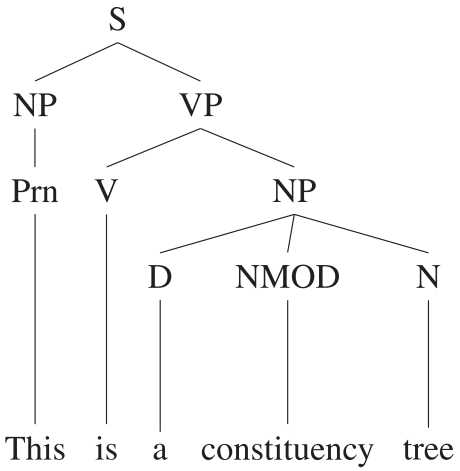


FIGURE 1. Constituency tree for an English sentence.

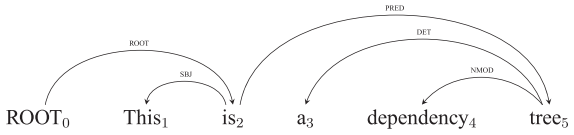


FIGURE 2. Dependency tree for an English sentence.

Some important advantages of dependency representations over constituency trees when applied to natural language processing tasks are that they provide a more explicit representation of the semantic information that is useful for applications (e.g., the subject and object of a sentence in Figure 2 are explicitly represented in its dependency graph), they do not need intermediate nodes (nonterminals) and hence allow for simpler and more efficient parsing algorithms, and they represent discontinuous linguistic phenomena caused by long-range dependencies or free word order in a natural way by using crossing dependencies.

While there has been research in grammar-driven dependency parsing, where formal grammatical rules are used to define the set of dependency structures that can appear in a language (Tapanainen and Järvinen 1997; Lombardo and Lesmo 1996), most current dependency parsers are data-driven, i.e., they use learning techniques to automatically infer linguistic knowledge from annotated corpora, which can then be used to parse new sentences without the need for an explicit grammar.

In particular, the vast majority of data-driven dependency parsers that have been defined in recent years can be described as being either graph-based or transition-based dependency parsers (Zhang and Clark 2008; McDonald and Nivre 2011). Graph-based parsing uses global optimization on models that score dependency graphs (Eisner 1996; McDonald et al. 2005). In transition-based parsing, which is the focus of this article, dependency graphs are built by sequences of actions by an automaton that transitions between parser states, and each action is scored individually. These scores are used to find a suitable sequence of actions for each given sentence, typically by greedy deterministic search or beam search (Yamada and Matsumoto 2003; Nivre, Hall, and Nilsson 2004a; Nivre 2008). Some key advantages of transition-based parsers are their high efficiency (many of them

have linear time complexity, while still providing state-of-the-art accuracy) and the possibility of easily incorporating arbitrarily rich feature models, including nonlocal features (Zhang and Nivre 2011).

However, it has been shown by McDonald and Nivre (2007) that transition-based parsers suffer from error propagation: As the generation of a dependency parse for a sentence is modeled as a sequential process, an early erroneous decision may place the parser into an incorrect state, causing more errors later on. In particular, one source of error propagation in transition-based parsers is the need to enforce the single-head constraint, i.e., the common restriction in dependency syntax that forbids each node from having more than one incoming arc. For instance, if we are parsing the sentence in Figure 2 with a transition-based parser that uses greedy deterministic search and we mistakenly make a decision to build an arc from a_3 to $tree_5$ instead of the correct dependency from $tree_5$ to a_3, we will miss not only this dependency but also the one from is_2 to $tree_5$, because we will be in a parser state where the single-head constraint makes it illegal to create it (because of not allowing node $tree_5$ to have two incoming arcs).

In this article, we introduce a new approach to transition-based parsing that improves accuracy by alleviating this kind of error propagation. To do so, we build novel *undirected* dependency parsers by modifying existing transition-based dependency parsers—namely the Planar and 2-Planar parsers by Gómez-Rodríguez and Nivre (2013) and the non-projective list-based parser by Nivre (2008), which is a variant of the algorithm by Covington (2001).

The obtained undirected parsers are algorithms that build an undirected graph (such as the one in Figure 3) rather than a dependency graph. This means that the single-head constraint need not be observed during the parsing process, because the directed notions of head and dependent (or of incoming and outgoing arcs) are lost in undirected graphs. Therefore, this gives the parser more freedom, and can prevent situations where enforcing the constraint leads to error propagation, such as the previous example.

On the other hand, these new algorithms have the obvious disadvantage that their output is an undirected graph and not a dependency graph. We will need a postprocessing step to recover the direction of the dependencies, generating a valid dependency structure. Thus, some complexity is moved from the transition-based parsing process to this postprocessing step, and each undirected parser will outperform the original directed version only if the simplification of the parsing phase is able to avoid more errors than are generated in the postprocessing step. Fortunately, as we will see in Section 5, this is in fact the case for most of the algorithm-data set combinations that we tried, showing that the undirected parsing approach is useful to improve parsing accuracy.

The remainder of this article is structured as follows: Section 2 introduces common notation and concepts regarding transition-based dependency parsing, which will be used throughout this article. Section 3 describes a technique to transform transition-based dependency parsers satisfying certain conditions into undirected parsers, and applies it to the Planar, 2-Planar, and Covington algorithms. In Section 4, we discuss two different postprocessing techniques that can be used to recover dependency trees from undirected graphs.

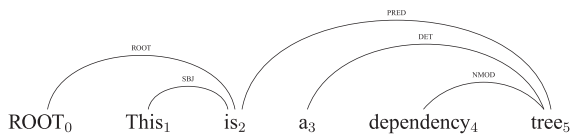


FIGURE 3. Undirected dependency graph for the sentence in Figure 2.

Section 5 puts the ideas in previous sections into practice and presents an empirical study of the accuracy of undirected dependency parsers compared with their directed counterparts, including an evaluation of undirected accuracy measures and direction errors to compare both postprocessing approaches. Section 6 presents an error analysis to see the effect of undirected parsing on error propagation, and Section 7 discusses related work. Finally, Section 8 concludes this article with a discussion of the results.

2. TRANSITION-BASED DEPENDENCY PARSING

We now introduce some definitions and notation concerning transition-based dependency parsing, which will serve as a basis to present our undirected parsing approach.

2.1. Dependency Parsing and Dependency Graphs

A *dependency parser* is a system that analyzes natural language sentences and outputs a representation of their syntactic structure in the form of a dependency graph, such as the one shown in Figure 2. More formally, a dependency graph can be defined as follows:

Definition 1. Let $w = w_1 \dots w_n$ be an input sentence. Given a set L of labels, a *dependency graph* for $w_1 \dots w_n$ is a labeled directed graph $G = (V_w, A)$, where $V_w = \{0, \dots, n\}$ is the set of nodes, and $A \subseteq V_w \times L \times V_w$ is the set of labeled directed arcs.

Each of the nodes in V_w encodes the position of a token in w, except for node 0 that is a dummy node used to mark the root of the sentence and cannot have incoming arcs. Each of the arcs in A encodes a dependency relation between two tokens. We will write the arc $(i, l, j) \in A$ as $i \overset{l}{\rightarrow} j$, which will also be called a *dependency link* labeled l from i to j. We then say that i is the *head* of j and, conversely, that j is a syntactic *dependent* of i. The labels on dependency links are typically used to represent their associated syntactic functions, such as SBJ for subject in Figure 2.

Given a dependency graph $G = (V_w, A)$, we will write $i \rightarrow j \in A$ if there is a dependency link from i to j, regardless of its label. We will write $i \rightarrow^\star j \in A$ if there is a (possibly empty) path from i to j and $i \leftrightarrow^\star j \in A$ if there is a (possibly empty) path connecting i and j in the undirected graph underlying G. When using these notational conventions, we will omit the references to A when the relevant set of edges is clear from the context.

Most dependency-based syntactic formalisms do not allow arbitrary dependency graphs as syntactic representations. Instead, they are typically restricted to acyclic graphs where each node has at most one head. Such dependency graphs are called *dependency forests*.

Definition 2. A dependency graph G is said to be a *dependency forest* if it satisfies the following:

(1) *single-head constraint*: if $i \rightarrow j$, then there is no $k \neq i$ such that $k \rightarrow j$.
(2) *acyclicity constraint*: if $i \rightarrow^\star j$, then there is no arc $j \rightarrow i$.

Nodes that have no head in a dependency forest are called *roots*. Apart from the previous two constraints, some dependency formalisms add the additional constraint that a dependency forest can have only one root (or, equivalently, that it must be connected). A forest of this form is called a *dependency tree*.

In this article, we will work with dependency parsers that output dependency forests, i.e., that enforce the single-head and acyclicity constraints. While we do not require the

parsers to explicitly enforce connectedness, we note that any dependency forest can be easily converted to a dependency tree (and thus made connected) by linking the dummy root node 0 as the parent of all its other root nodes. Therefore, from now on, we will refer to the outputs produced and the training data used by our parsers as dependency trees, keeping in mind that some of these trees may be a representation of dependency analyses that are forests.

2.2. Transition Systems

In a transition-based dependency parser, the dependency analysis for an input sentence is built by a nondeterministic state machine that reads the input sentence and builds dependency arcs. Following the framework of Nivre (2008), this nondeterministic automaton is called a *transition system* and is defined as follows:

Definition 3. A *transition system* for dependency parsing is a tuple $S = (C, T, c_s, C_t)$, where

(1) C is a set of possible parser *configurations*;
(2) T is a finite set of *transitions*, which are partial functions $t : C \rightarrow C$;
(3) c_s is a total initialization function that maps each input string w to a unique *initial configuration* $c_s(w)$; and
(4) $C_t \subseteq C$ is a set of *terminal configurations*.

Although the specific nature of configurations varies among parsers, they are required to contain at least a set A of dependency arcs and a buffer B of unread words, which initially holds all the words in the input sentence. A transition-based parser will be able to read input words by popping them from the buffer and to create dependency arcs by adding them to the set A.

Given an input string $w = w_1 \ldots w_n$, each of the sequences of configurations that the transition system S can traverse by sequentially applying transitions starting from the initial configuration $c_s(w)$ and ending at some terminal configuration $c_t \in C_t$ is called a *transition sequence* for w in S. The *parse* assigned to w by a transition sequence ending at the terminal configuration c_t is the dependency graph $G = (\{0, \ldots, n\}, A_{c_t})$, where A_{c_t} is the set of arcs stored in the configuration c_t.

Note that, from a theoretical standpoint, it is possible to define a transition system such that no transition sequences exist for a given sentence w. However, this is usually avoided in practice, because robustness (the ability to terminate producing a parse for every possible input) is seen as a desirable quality in data-driven natural language parsers. Therefore, all the transition systems that we will use and define in this article have the property that there is at least one (and typically more than one) transition sequence for every sentence w.

To use a transition system to obtain the best dependency analysis for a given sentence, we need to have a mechanism that will select the most suitable among all the transition sequences that the system allows for that sentence. A standard method to achieve this is by using a classifier to select the best transition to execute at each configuration.

To do so, we define an *oracle* for the transition system $S = (C, T, c_s, C_t)$ as a function $o : C \rightarrow T$; i.e., an oracle is a function that selects a single transition to take at each configuration and thus can be used to determinize the parsing process. Given a training treebank containing manually annotated dependency trees for sentences, we train a classifier to approximate an oracle by building a canonical transition sequence for each tree in the treebank and using each of the configurations in the sequence and the corresponding chosen transition as a training instance.

Then, to parse a sentence w, we only need to initialize the parser to the initial configuration $c_s(w)$ and iteratively apply the transitions suggested by the classifier until a terminal configuration is reached. This results in a parser that performs a greedy deterministic search for the best transition sequence, one of the most widely used approaches in transition-based parsing (Yamada and Matsumoto 2003; Nivre, Hall, and Nilsson 2004b; Attardi 2006; Nivre 2008; Goldberg and Elhadad 2010; Tratz and Hovy 2011; Gómez-Rodríguez and Nivre 2013), although other optimization and search strategies, such as beam search, can also be used (Johansson and Nugues 2006; Titov and Herderson 2007; Zhang and Clark 2008; Huang, Jiang, and Liu 2009; Huang and Sagae 2010; Zhang and Nivre 2011; Hayashi et al. 2012).

2.3. An Example Transition System: The Planar Transition System

A simple example of a practical transition system for dependency parsing is the *Planar* transition system (Gómez-Rodríguez and Nivre 2010, 2013). The Planar parser is an extension of the well-known arc-eager projective parser by Nivre (2003), which can handle all dependency trees that are *planar*, i.e., those whose arcs can be drawn above the words (as in Figure 2) in such a way that no two arcs cross. In contrast, the arc-eager parser by Nivre (2003) can only build so-called *projective* trees, which is a slightly more restricted set of syntactic structures (Gómez-Rodríguez and Nivre 2013).

The Planar transition system is a transition system $S = (C, T, c_s, C_t)$ such that

- C is the set of all configurations of the form $c = \langle \sigma, B, A \rangle$, where σ and B are disjoint lists of nodes from V_w (for some input w), and A is a set of dependency arcs over V_w. List B, called the *buffer*, is used to hold nodes corresponding to input words that have not yet been read. List σ, called the *stack*, contains nodes for words that have already been read but still have dependency links pending to be created. For perspicuity, we will use the notation $\sigma|i$ to denote a stack with top i and tail σ and the notation $j|B$ to denote a buffer with top j and tail B. The set A of dependency arcs contains the part of the output parse that the system has constructed at each given point.
- The initial configuration is $c_s(w_1 \dots w_n) = \langle [], [1 \dots n], \emptyset \rangle$; i.e., the buffer initially holds the whole input string while the stack is empty.
- The set of terminal configurations is $C_t = \{\langle \sigma, [], A \rangle \in C\}$; i.e., final configurations are those where the buffer is empty, regardless of the contents of the stack.
- The set T has the following transitions:

SHIFT $\quad \langle \sigma, i|B, A \rangle \Rightarrow \langle \sigma|i, B, A \rangle$

REDUCE $\quad \langle \sigma|i, B, A \rangle \Rightarrow \langle \sigma, B, A \rangle$

LEFT-ARC$_l$ $\quad \langle \sigma|i, j|B, A \rangle \Rightarrow \left\langle \sigma|i, j|B, A \cup \left\{ j \xrightarrow{l} i \right\} \right\rangle$
only if $\nexists k \mid k \rightarrow i \in A$ (single-head) and $i \leftrightarrow^* j \notin A$ (acyclicity).

RIGHT-ARC$_l$ $\quad \langle \sigma|i, j|B, A \rangle \Rightarrow \left\langle \sigma|i, j|B, A \cup \left\{ i \xrightarrow{l} j \right\} \right\rangle$
only if $\nexists k \mid k \rightarrow j \in A$ (single-head) and $i \leftrightarrow^* j \notin A$ (acyclicity).

The SHIFT transition is used to read words from the input string, by moving the next node in the buffer to the top of the stack. The LEFT-ARC and RIGHT-ARC transitions build leftward and rightward dependency arcs, respectively, connecting the first node on the buffer and the topmost node on the stack. Finally, the REDUCE transition is used to pop the topmost node from the stack when we have finished building arcs to or from it.

Transition	Stack (σ)	Buffer (B)	Added Arc
	[ROOT$_0$]	[This$_1$,...,tree$_5$]	
SHIFT	[ROOT$_0$, This$_1$]	[is$_2$,..., tree$_5$]	
LA$_{SBJ}$	[ROOT$_0$, This$_1$]	[is$_2$,..., tree$_5$]	(2, SBJ, 1)
REDUCE	[ROOT$_0$]	[is$_2$,..., tree$_5$]	
RA$_{ROOT}$	[ROOT$_0$]	[is$_2$,..., tree$_5$]	(0, ROOT, 2)
SHIFT	[ROOT$_0$, is$_2$]	[a$_3$,..., tree$_5$]	
SHIFT	[ROOT$_0$, is$_2$, a$_3$]	[dependency$_4$, tree$_5$]	
SHIFT	[ROOT$_0$, is$_2$, a$_3$, dependency$_4$]	[tree$_5$]	
LA$_{NMOD}$	[ROOT$_0$, is$_2$, a$_3$, dependency$_4$]	[tree$_5$]	(5, NMOD, 4)
REDUCE	[ROOT$_0$, is$_2$, a$_3$]	[tree$_5$]	
LA$_{DET}$	[ROOT$_0$, is$_2$, a$_3$]	[tree$_5$]	(5, DET, 3)
REDUCE	[ROOT$_0$, is$_2$]	[tree$_5$]	
RA$_{PRED}$	[ROOT$_0$, is$_2$]	[tree$_5$]	(2, PRED, 5)
SHIFT	[ROOT$_0$, is$_2$, tree$_5$]	[ ]	
REDUCE	[ROOT$_0$, is$_2$]	[ ]	
REDUCE	[ROOT$_0$]	[ ]	

FIGURE 4. Transition sequence for parsing the sentence in Figure 2 using the Planar parser (LA = LEFT-ARC and RA = RIGHT-ARC).

Figure 4 shows a transition sequence in the Planar transition system that derives the labeled dependency graph in Figure 2.

Note that the Planar parser is a linear-time parser, because each word in the input can be shifted and reduced at most once, and the number of arcs that can be built by the LEFT-ARC and RIGHT-ARC transitions is strictly bounded by the number of words by the single-head constraint.

2.4. The 2-Planar and Covington Transition Systems

The undirected dependency parsers defined and tested in this article are based on the Planar transition system described earlier: the 2-Planar transition system (Gómez-Rodríguez and Nivre 2010, 2013) and the version of the Covington (2001) nonprojective parser defined by Nivre (2008). We now outline the two latter parsers briefly, and a more comprehensive description can be found in the aforementioned references.

The 2-Planar transition system is an extension of the Planar system that can recognize a larger set of dependency trees, called 2-planar dependency trees. A dependency tree is said to be 2-planar if it is possible to draw it assigning one out of two colors to each of its dependency arcs, in such a way that arcs sharing the same color do not cross. Gómez-Rodríguez and Nivre (2013) have shown that well over 99% of the dependency trees in natural language treebanks fall into this set, making this parser practical for languages that contain a significant proportion of crossing links, so that planar and projective parsers fall short in coverage.

To handle 2-planar structures, the 2-Planar transition system uses two stacks instead of one, with each stack corresponding to one of the colors that can be assigned to arcs. At each given configuration, one of the stacks is said to be *active* (meaning that we are building arcs of that color), while the other is *inactive*. Configurations are thus of the form $c = \langle \sigma_0, \sigma_1, B, A \rangle$, where σ_0 is the active stack and σ_1 the inactive stack. The initial configuration is $c_s\,(w_1 \ldots w_n) = \langle [], [], [1 \ldots n], \emptyset \rangle$, and the set of terminal configurations is $C_t = \{\langle \sigma_0, \sigma_1, [], A \rangle \in C\}$, analogously to the Planar transition system. The system has the following transitions:

SHIFT $\langle \sigma_0, \sigma_1, i\,|B, A\rangle \Rightarrow \langle \sigma_0|i, \sigma_1|i, B, A\rangle$

REDUCE $\langle \sigma_0|i, \sigma_1, B, A\rangle \Rightarrow \langle \sigma_0, \sigma_1, B, A\rangle$

LEFT-ARC$_l$ $\langle \sigma_0|i, \sigma_1, j\,|B, A\rangle \Rightarrow \left\langle \sigma_0|i, \sigma_1, j\,|B, A \cup \left\{ j \overset{l}{\rightarrow} i \right\} \right\rangle$
only if $\nexists k \mid k \rightarrow i \in A$(single-head) and $i \leftrightarrow^* j \notin A$ (acyclicity).

RIGHT-ARC$_l$ $\langle \sigma_0|i, \sigma_1, j\,|B, A\rangle \Rightarrow \left\langle \sigma_0|i, \sigma_1, j\,|B, A \cup \left\{ i \overset{l}{\rightarrow} j \right\} \right\rangle$
only if $\nexists k \mid k \rightarrow j \in A$ (single-head) and $i \leftrightarrow^* j \notin A$ (acyclicity).

SWITCH $\langle \sigma_0, \sigma_1, B, A\rangle \Rightarrow \langle \sigma_1, \sigma_0, B, A\rangle$.

The SHIFT transition reads words from the input string exactly as in the Planar transition system, but in this case, their corresponding nodes are placed into both stacks. REDUCE, LEFT-ARC, and RIGHT-ARC work similarly as in the Planar parser, but they only take into account the active stack, ignoring the inactive one. Finally, a SWITCH transition is added that makes the active stack inactive and vice versa, allowing us to alternate between the two possible arc colors. Despite this added functionality, the 2-Planar parser still runs in linear time.

On the other hand, the Covington algorithm is a transition system that runs in quadratic time, but it has the advantage of being able to parse every possible dependency tree, without restrictions such as planarity or 2-planarity. The basic algorithm was first described by Covington (1990, 2001). Nivre (2008) implements a variant of this strategy as a transition system, which is the version we use here.

Configurations in this system are of the form $c = \langle \lambda_1, \lambda_2, B, A\rangle$, where λ_1 and λ_2 are *lists* containing nodes associated with partially processed words, and B is the *buffer* of unprocessed words. The idea of the algorithm is that, after reading each given word, we can do a right-to-left traversal of *all* the nodes for already-read words in the input and create links between them and the first node in the buffer. This traversal is implemented by moving nodes from λ_1 (untraversed nodes) to λ_2 (already-traversed nodes). After reading each new input word, all the nodes in both lists are moved back to λ_1 for a new right-to-left traversal to start, hence the quadratic complexity.

The algorithm starts with an initial configuration $c_s\,(w_1 \ldots w_n) = \langle [], [], [1 \ldots n], \emptyset\rangle$ and will terminate in final configurations of the set $C_f = \{\langle \lambda_1, \lambda_2, [], A\rangle \in C\}$. The system has the following transitions:

SHIFT $\langle \lambda_1, \lambda_2, i\,|B, A\rangle \Rightarrow \langle \lambda_1 \cdot \lambda_2|i\,, [], B, A\rangle$

NO-ARC $\langle \lambda_1|i\,, \lambda_2, B, A\rangle \Rightarrow \langle \lambda_1, i\,|\lambda_2, B, A\rangle$

LEFT-ARC$_l$ $\langle \lambda_1|i\,, \lambda_2, j\,|B, A\rangle \Rightarrow \left\langle \lambda_1, i\,|\lambda_2, j\,|B, A \cup \left\{ j \overset{l}{\rightarrow} i \right\} \right\rangle$
only if $\nexists k \mid k \rightarrow i \in A$ (single-head) and $i \leftrightarrow^* j \notin A$ (acyclicity).

RIGHT-ARC$_l$ $\langle \lambda_1|i\,, \lambda_2, j\,|B, A\rangle \Rightarrow \left\langle \lambda_1, i\,|\lambda_2, j\,|B, A \cup \left\{ i \overset{l}{\rightarrow} j \right\} \right\rangle$
only if $\nexists k \mid k \rightarrow j \in A$ (single-head) and $i \leftrightarrow^* j \notin A$ (acyclicity).

The SHIFT transition advances the parsing process by reading the first node in the buffer B and inserting it at the head of a list obtained by concatenating λ_1 and λ_2, thus starting a new right-to-left traversal to find candidate nodes to be linked to the one now heading the buffer. This traversal is implemented by the other three transitions: NO-ARC is used when

there is no dependency relation between the first node in the buffer and the head of the list α_1, and it moves the head of the list α_1 to α_2 without creating any arcs; while LEFT-ARC and RIGHT-ARC create a leftward (rightward) arc connecting the first node in the buffer and the head of the list α_1 and then move the head of α_1 to α_2. The traversal will end when a new SHIFT transition is executed, signifying that no more arcs will be created involving the first node in the buffer and the nodes in α_1.

3. TRANSFORMING DIRECTED PARSERS INTO UNDIRECTED PARSERS

As mentioned in Section 2.2, practical implementations of transition systems use greedy search or beam search to find the best transition sequence (and thus obtain a dependency tree) for each given input. Because these strategies build transition sequences and dependency arcs in a sequential way from the beginning of the sentence to the end, early parsing decisions may condition and restrict later decisions, causing error propagation. McDonald and Nivre (2007) present an empirical study whose results highlight this phenomenon, showing that a transition-based parser tends to be more accurate than a graph-based parser on arcs that are built early in the transition sequence, but less accurate on arcs built later on.

In particular, one possible source of error propagation is the single-head constraint described in Definition 2. To return a valid dependency tree, a transition system must obey this constraint during the whole parsing process. This means that a transition that creates a dependency arc is permissible only if its application does not violate the single-head constraint, i.e., if it does not result in assigning more than one head to the same node. For instance, Figure 4 shows a transition sequence for the Planar parser that correctly parses a sample sentence, assigning it the dependency tree in Figure 2. However, in an alternative scenario where the classifier made a mistake in the eighth transition choosing to apply RA_{NMOD} instead of the correct choice LA_{NMOD}, this would result into building a dependency link from $dependency_4$ to $tree_5$ instead of the correct link from $tree_5$ to $dependency_4$. In turn, this would lead to a situation where creating the (correct) link from is_2 to $tree_5$ would be forbidden by the single-head constraint, as node $tree_5$ would already have an incoming arc. Therefore, in this example, a single erroneous choice of transition initially affecting a single dependency arc propagates to other arcs, because of the single-head constraint, causing at least two attachment errors in the output tree.

To remove this source of error propagation, we transform the Planar, 2-Planar, and Covington transition systems into variants that build *undirected* graphs instead of directed dependency trees. The goal of this transformation is to allow transition-based parsers to work without needing to obey the single-head constraint. This will make these parsers less sensitive to error propagation, because they will be able to create arcs freely at any point in the parsing sequence, regardless of the existing arcs that have been created before.

The mentioned transformation consists in redesigning the transition systems so that they create dependency links without a direction. In this way, it is not necessary to observe the single-head constraint during the parsing process, because the directed concepts of head and dependent do not apply to undirected links. As a result, the output of these new variants is an undirected graph instead of a tree. This transformation has previously been described in Gómez-Rodríguez and Fernández-González (2012).

3.1. The Undirected Planar, 2-Planar, and Covington Transition Systems

With the goal of obtaining undirected transition systems from the directed ones described in Sections 2.3 and 2.4, we replace the LEFT-ARC and RIGHT-ARC transitions in

those systems (which create directed arcs in each direction) with a new transition (ARC) that builds an undirected link. This can be performed because, in these three transition systems, the effect of the two directed transitions is the same except for the direction of the created link, so that their behavior can be collapsed into one common transition: the undirected ARC transition.

In addition to this, the configurations of the undirected transition systems must be changed so that the arc set A is a set of undirected edges, instead of directed arcs. Analogously to our notation for directed arcs, we will use the notation $i \overset{l}{-} j$ as shorthand for an undirected edge labeled l connecting the nodes i and j.

Furthermore, because the direction of the arcs is lost in undirected graphs, the preconditions of transitions that guarantee the single-head constraint are simply removed from the systems.

If we apply these transformations and leave the Planar, 2-Planar, and Covington transition systems otherwise unchanged, we will obtain the respective undirected variants: the *undirected Planar*, the *undirected 2-Planar*, and the *undirected Covington* transition systems. The transition set of each undirected transition system is as follows:

Undirected Planar

SHIFT	$\langle \sigma, i \,\vert\, B, A\rangle \Rightarrow \langle \sigma \vert i, B, A\rangle$
REDUCE	$\langle \sigma \vert i, B, A\rangle \Rightarrow \langle \sigma, B, A\rangle$
ARC_l	$\langle \sigma \vert i, j \,\vert\, B, A\rangle \Rightarrow \left\langle \sigma \vert i, j \,\vert\, B, A \cup \left\{ j \overset{l}{-} i \right\} \right\rangle$ only if $i \leftrightarrow^* j \notin A$ (acyclicity).

Undirected 2-Planar

SHIFT	$\langle \sigma_0, \sigma_1, i \,\vert\, B, A\rangle \Rightarrow \langle \sigma_0 \vert i, \sigma_1 \vert i, B, A\rangle$
REDUCE	$\langle \sigma_0 \vert i, \sigma_1, B, A\rangle \Rightarrow \langle \sigma_0, \sigma_1, B, A\rangle$
ARC_l	$\langle \sigma_0 \vert i, \sigma_1, j \,\vert\, B, A\rangle \Rightarrow \left\langle \sigma_0 \vert i, \sigma_1, j \,\vert\, B, A \cup \left\{ j \overset{l}{-} i \right\} \right\rangle$ only if $i \leftrightarrow^* j \notin A$ (acyclicity).
SWITCH	$\langle \sigma_0, \sigma_1, B, A\rangle \Rightarrow \langle \sigma_1, \sigma_0, B, A\rangle$.

Undirected Covington

SHIFT	$\langle \lambda_1, \lambda_2, i \,\vert\, B, A\rangle \Rightarrow \langle \lambda_1 \cdot \lambda_2 \vert i, [], B, A\rangle$
NO-ARC	$\langle \lambda_1 \vert i, \lambda_2, B, A\rangle \Rightarrow \langle \lambda_1, i \,\vert\, \lambda_2, B, A\rangle$
ARC_l	$\langle \lambda_1 \vert i, \lambda_2, j \,\vert\, B, A\rangle \Rightarrow \left\langle \lambda_1, i \,\vert\, \lambda_2, j \,\vert\, B, A \cup \left\{ j \overset{l}{-} i \right\} \right\rangle$ only if $i \leftrightarrow^* j \notin A$ (acyclicity).

We show in Figure 5 how the undirected Planar parser analyzes a sentence using its own set of transitions. Note that the output of this parsing process is the undirected graph presented in Figure 3 instead of the expected dependency tree in Figure 2. To obtain a dependency tree as the final output of the analysis, we will need to apply a postprocessing step, which will be described in Section 4.

Transition	Stack (σ)	Buffer (B)	Added Arc
	[ROOT$_0$]	[This$_1$,...,tree$_5$]	
SHIFT	[ROOT$_0$, This$_1$]	[is$_2$,..., tree$_5$]	
ARC$_{SBJ}$	[ROOT$_0$, This$_1$]	[is$_2$,..., tree$_5$]	(1, SBJ, 2)
REDUCE	[ROOT$_0$]	[is$_2$,..., tree$_5$]	
ARC$_{ROOT}$	[ROOT$_0$]	[is$_2$,..., tree$_5$]	(0, ROOT, 2)
SHIFT	[ROOT$_0$, is$_2$]	[a$_3$,..., tree$_5$]	
SHIFT	[ROOT$_0$, is$_2$, a$_3$]	[dependency$_4$, tree$_5$]	
SHIFT	[ROOT$_0$, is$_2$, a$_3$, dependency$_4$]	[tree$_5$]	
ARC$_{NMOD}$	[ROOT$_0$, is$_2$, a$_3$, dependency$_4$]	[tree$_5$]	(4, NMOD, 5)
REDUCE	[ROOT$_0$, is$_2$, a$_3$]	[tree$_5$]	
ARC$_{DET}$	[ROOT$_0$, is$_2$, a$_3$]	[tree$_5$]	(3, DET, 5)
REDUCE	[ROOT$_0$, is$_2$]	[tree$_5$]	
ARC$_{PRED}$	[ROOT$_0$, is$_2$]	[tree$_5$]	(2, PRED, 5)
SHIFT	[ROOT$_0$, is$_2$, tree$_5$]	[ ]	
REDUCE	[ROOT$_0$, is$_2$]	[ ]	
REDUCE	[ROOT$_0$]	[ ]	

FIGURE 5. Transition sequence for parsing the sentence in Figure 3 using the undirected Planar parser.

It is worth remarking that, in order to apply this transformation to obtain an undirected parser from a directed one, the original transition system must satisfy the condition that their arc-building transitions (conventionally called LEFT-ARC and RIGHT-ARC) be identical except for the directions of the links that they create. This condition holds in some transition systems in the literature—-such as the three systems described earlier or the arc-eager Directed Acyclic Graph parser for enriched dependency representations described by Sagae and Tsujii (2008)—but not in others. For example, in the well-known arc-eager parser by Nivre (2003), LEFT-ARC transitions pop a node from the stack in addition to creating an arc, while RIGHT-ARC transitions instead remove the topmost buffer node and then push the top stack node back to the buffer. One could still try to transform the arc-eager parser into an undirected variant by converting each of its arc transitions into an undirected transition, without necessarily collapsing them into one. However, this would result into a parser that violates the acyclicity constraint, because the original system is designed in such a way that both constraints are enforced jointly and acyclicity is only guaranteed if the single-head constraint is also kept. It is easy to check that this problem cannot happen in parsers where the LEFT-ARC and RIGHT-ARC transitions are symmetrical in the manner described earlier: In these systems, if a given graph is not parsable in the original system, then its underlying undirected graph will not be parsable in the transformed system.

3.2. Undirected Feature Models

To implement a practical parser on top of a transition system, we need a feature model to extract relevant information from configurations that will serve to train a classifier. Therefore, apart from modifying the transition system, creating a practical undirected parser necessarily implies to adapt its feature model to work with undirected graphs.

Some features usually employed in transition-based parsers depend on the direction of the arcs that have already been created. Examples of such features are the part-of-speech tag associated with the head of the topmost stack node or the label of the arc going from the first node in the buffer to its leftmost dependent.[1] However, because we cannot tell

[1] These example features are taken from the default model for the Planar parser in version 1.5 of MaltParser (Nivre, Hall, and Nilsson 2006).

heads from dependents in an undirected graph, these features cannot be used to train an undirected parser.

Therefore, we convert these features into their closest undirected versions: In the previous examples, those would be the part-of-speech tag associated with the closer node linked to the topmost stack node and the label of the arc that connects the first node in the buffer to the leftmost node linked to it. Notice that now a node (topmost stack or first node in the buffer) has neither head nor dependents, it only has some other nodes linked to it.

More formally, these are the undirected features obtained from the directed ones:

- information (e.g., part of speech, label, and lemma) about the ith node linked to a given node (topmost stack node, topmost buffer node, etc.) on the left or on the right, and about the associated undirected arc, typically for $i = 1, 2, 3$;
- information (e.g., part of speech, label, and lemma) about the closest left and right "undirected siblings" of a given node, i.e., the closest node q located to the left of the given node p such that p and q are linked to some common node r located to the right of both, and vice versa. Note that this notion of undirected siblings does not necessarily correspond to siblings in the directed graph: It can also capture other second-order interactions, such as grandparents.

In addition, we create new features based on undirected relations between nodes that provide further context information for the parser. In particular, we found that the following features worked well in practice:

- a Boolean feature representing whether two given nodes are linked or not in the undirected graph and a feature representing the label of the arc between them.

4. RECOVERING ARC DIRECTIONS

The transformed transition systems described in Section 3 have the drawback that the output they produce is an undirected graph, such as the one in Figure 3, rather than a proper dependency tree. To use these systems and still obtain a directed dependency tree as the final output of the parsing process, we will apply a postprocessing step to assign an orientation to the undirected graph (i.e., choose a direction for each of its edges), in such a way that the single-head constraint is obeyed and the result is a valid dependency tree.

For this purpose, we have developed two different reconstruction techniques to recover arc directions from the undirected graph, previously described in less detail in Gómez-Rodríguez and Fernández-González (2012). The first one, called *naive reconstruction*, is based on using the dummy root node to decide the direction that should be assigned to edges, by choosing the unique orientation of the undirected graph obtained by traversing it from the dummy root. The second technique, *label-based reconstruction*, consists of using the edge labels generated by the transition system to assign a preferred direction to each undirected edge and then choosing the orientation that conforms to as many preferred directions as possible (note that it will not always be possible to conform to the preferred directions of *all* the arcs, as that may generate a graph violating the single-head constraint).

To describe these reconstruction techniques more formally and view them under a common framework, we can formulate the problem of recovering arc directions as an optimum branching (i.e., directed minimum spanning tree) problem on a weighted graph. Given the undirected graph U produced by an undirected parser, we consider its isomorphic symmetric directed graph, i.e., the directed graph which has an arc for each of both possible directions of an undirected edge in U. Each directed spanning tree of that graph corresponds

to an orientation of U. Then, reconstruction techniques can be implemented by assigning weights to each of the arcs in the symmetric graph, so that they encode a criterion to prefer certain orientations of arcs over others, and then using an optimum branching algorithm to find the minimum spanning tree. Different criteria for assigning weights to arcs will produce different reconstruction techniques.

More formally, let $U = (V_w, E)$ be the undirected graph produced by some undirected parser[2] for an input string w (we omit labels for simplicity and readability).

We define the following sets of arcs:

$$A_1(U) = \{(i, j) \mid j \neq 0 \wedge \{i, j\} \in E\},$$
$$A_2(U) = \{(0, i) \mid i \in V_w\}.$$

The set $A_1(U)$ contains the two possible orientations of each edge in U (i.e., the arcs in the symmetric directed graph isomorphic to U) except for those arcs that would have node 0 as a dependent, which we disallow because we are using that node as a dummy root, and therefore, it cannot be assigned a head. On the other hand, the set $A_2(U)$ contains all the possible arcs that link the nodes in V_w as dependents of the dummy root node, regardless of whether their underlying undirected edges were present in U or not. This is so that the reconstruction techniques defined under this framework are allowed to link unattached tokens to the dummy root.

With these arc sets, we define a graph $D(U)$ containing all the candidate arcs that we will consider when reconstructing a dependency structure from U:

$$D(U) = \{V_w, A(U) = A_1(U) \cup A_2(U)\}.$$

The reconstruction process for an undirected graph U consists of finding an optimum branching (i.e., a directed minimum spanning tree) for a weighted directed graph obtained from assigning a cost $c(i, j)$ to each arc (i, j) of the graph $D(U)$; i.e., we are looking for a dependency tree $T = (V_w, A_T \subseteq A(U))$ that minimizes $\sum_{(i,j)\in A_t} c(i, j)$.

Such a tree can be calculated using well-known algorithms for the optimum branching problem, such as the Chu–Liu–Edmonds algorithm (Chu and Liu 1965; Edmonds 1967). In this particular case, we can take advantage of the fact that the graph $D(U)$ has $O(n)$ nodes and $O(n)$ arcs for a string of length n, and use the implementation by Tarjan (1977) to achieve a time complexity of $O(n \log sn)$.

The different reconstruction techniques can be defined by establishing different criteria to assign the costs $c(i, j)$ to the arcs in $A(U)$.

Note that, in practice, the impact of the reconstruction process in overall parsing time is negligible. This is because the optimum branching algorithm does not perform any classification and hence does not need to extract features from configurations, which is the bottleneck of transition-based parsers (Bohnet 2010; Volokh 2013). As a result, when applying undirected parsing to a linear-time parsing algorithm such as the Planar and 2-Planar parsers, the total observed runtimes grow linearly with sentence length, in spite of the theoretical $O(n \log n)$ complexity of the reconstruction algorithm.

[2] Note that, while the approach taken in this article is to obtain undirected parsers by transforming directed parsers, it would also be possible in theory to design an undirected parser from scratch and apply the same reconstruction techniques to it.

4.1. Naive Reconstruction

A first, a very simple reconstruction technique is based on the fact that a rooted tree is implicitly directed; thus, if we train the undirected parser to obtain trees and consider the dummy node 0 as their root, we will directly obtain a directed dependency structure. In terms of the generic framework described earlier, this reconstruction can be defined by assigning costs to the arcs of $D(U)$ as follows:

$$c(i,j)\begin{cases}1 & \text{if } (i,j) \in A_1(U),\\ 2 & \text{if } (i,j) \in A_2(U) \wedge (i,j) \notin A_1(U).\end{cases}$$

This criterion assigns the same cost to both the orientations of each undirected edge in U and a higher cost to attaching any node to the dummy root that was not directly linked to it in U. Note that the particular values that we have assigned to the costs (1 and 2) are arbitrary: We can choose any other pair of values for the cost of the arcs as long as the second value is larger than the first one. The resulting optimum branching does not change: It comes from maximizing the number of arcs that come from the *cheaper* set ($A_1(U)$).

To obtain satisfactory results with this approach, we must train the undirected parser to explicitly build undirected arcs from the dummy root node to the root word(s) of each sentence using arc transitions. This means that, if our training treebank contains forests, we need to transform them into trees by explicitly linking each of their roots as dependents of the node 0, as explained at the end of Section 2.1.

Under this assumption, if no classification errors are made, the undirected graph U output by the undirected parser will always be an undirected tree, and the minimum spanning tree will correspond to the unique orientation of U making its edges point away from the dummy root.[3] It is easy to see that this orientation must be the correct parse, because any other orientation violates the assumption that node 0 is a root.

This naive reconstruction technique has the advantage of being very simple, while guaranteeing that the correct parse will be recovered if the undirected parser is able to correctly generate its underlying undirected tree. However, this approach lacks robustness, because it decides the direction of all the arcs in the final output based on which node(s) are chosen as sentence heads and linked to the dummy root. This means that a parsing error affecting the undirected edges that involve the root may propagate and result in many dependency links being erroneous. For this reason, this approach for recovering arc directions will not produce good empirical results, as will be seen in Section 5. Fortunately, we can define a more robust criterion where the orientation of arcs is defined in a more distributed manner, without being so sensible to the edges involving the root.

4.2. Label-Based Reconstruction

To obtain a more robust and effective reconstruction technique, we first apply a simple transformation to the training corpus so that arcs will have their direction encoded as a part of their label. To do so, if a leftward arc in the training set is labeled X, we relabel it X_l, meaning "a leftward arc labeled X." If a rightward arc in the training set is labeled X, we relabel it X_r, meaning "a rightward arc labeled X."

[3] Note that, while we previously suggested using optimum branching algorithms to find the spanning tree for the sake of generality, in this particular case, it is not necessary to use such a generic algorithm: The spanning tree can simply be built in $O(n)$ by starting a traversal from the root and orienting each arc in the sense of the traversal. However, this is only valid for this particular reconstruction technique.

After training the undirected parser with this modified treebank, its output for a new sentence will be an undirected graph where each edge's label includes an annotation indicating whether the reconstruction process should prefer to link the corresponding pair of nodes with a leftward or with a rightward arc. Note that those annotations represent *preferred directions*—not hard constraints—because, although in the absence of errors it would be possible to simply use the annotations to decode the correct parse for the sentence, in practice parsing errors can create situations where it is not possible to conform to all the annotations without violating the single-head constraint in the directed graph resulting from the reconstruction. In these cases, the reconstruction technique will have to decide which annotations to follow and which have to be ignored. For this purpose, we will assign the costs for our minimum branching algorithm so that it will return a tree agreeing with as many annotations as possible.

To achieve this, we denote by $A_{1+}(U) \subseteq A_1(U)$ the set of arcs in $A_1(U)$ that agree with the annotations, i.e., arcs $(i, j) \in A_1(U)$ where either $i < j$ and $\{i, j\}$ is labeled X_r in U, or $i > j$ and $\{i, j\}$ is labeled X_l in U. Conversely, we call $A_{1-}(U)$ the set of arcs in $A_1(U)$ that disagree with the annotations, i.e., $A_{1-}(U) = A_1(U) \setminus A_{1+}(U)$. Then, we assign costs to the directed arcs in $A(U)$ as follows:

$$c(i, j) \begin{cases} 1 & \text{if } (i, j) \in A_{1+}(U), \\ 2 & \text{if } (i, j) \in A_{1-}(U), \\ 2n & \text{if } (i, j) \in A_2(U) \wedge (i, j) \notin A_1(U), \end{cases}$$

where n is the length of the string.

With these costs, the optimum branching algorithm will find a spanning tree that agrees with as many annotations as possible, because assigning the direction that agrees with an edge's annotation has a lower cost than assigning the opposite direction. Additional arcs from the root not appearing in the parsing output (i.e., arcs in $A_2(U) \setminus A_1(U)$) can be added, but only if this is strictly necessary to guarantee connectedness (i.e., if the graph U was disconnected), because the cost of such an arc $(2n)$ is greater than the sum of costs of any combination of arcs originating from edges in U.

Analogously to the case of the naive reconstruction, the three cost values need not be exactly 1, 2, and $2n$ as chosen earlier: It is sufficient that the second value is larger than the first, and the third value is larger than n times the second value. The latter property trivially ensures that a spanning tree with k arcs from the third set always costs less than one with $k + 1$ arcs from the third set, which guarantees that this kind of arcs is only added when this is strictly necessary to guarantee connectedness (i.e., only one such arc will be used for each connected component that does not include the artificial root). The former property ensures that, among the trees with a minimal number of arcs from the third set, we choose those that minimize the number of arcs from the second set.

While this may be the simplest cost assignment to implement label-based reconstruction, we have found experimentally that better practical results are obtained if we give the algorithm more freedom to create new arcs from the root as follows:

$$c(i, j) \begin{cases} 1 & \text{if } (i, j) \in A_{1+}(U) \wedge (i, j) \notin A_2(U), \\ 2 & \text{if } (i, j) \in A_{1-}(U) \wedge (i, j) \notin A_2(U), \\ 2n & \text{if } (i, j) \in A_2(U). \end{cases}$$

The difference with the previous variant is that arcs originating from the root now have a cost of $2n$ *even if* their underlying undirected arcs were present in the output of the undirected parser. Informally, this means that the postprocessor will not *trust* the links from the dummy root created by the parser and may choose to change them (at no extra cost) if this

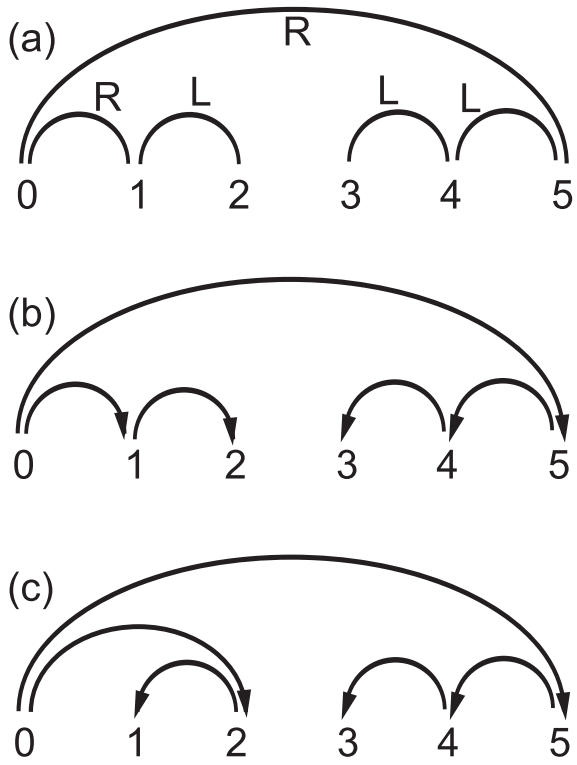


FIGURE 6. (a) An undirected graph obtained by the undirected parser trained with a transformed corpus where arcs have been relabeled to specify their direction. (b) and (c) The dependency graph obtained by each of the variants of the label-based reconstruction (note how the second variant moves an arc from the root).

is convenient to obtain a better agreement with the label annotations of the remaining arcs (see Figure 6 for an example of the difference between both cost assignments). We believe that the higher empirical accuracy obtained with this criterion is probably due to the fact that it is biased toward changing links from the root, which tend to be more problematic for transition-based parsers, while respecting the parser output as much as possible for links located deeper in the dependency structure, for which transition-based parsers have been shown to be more accurate (McDonald and Nivre 2007).

Note that both variants of label-based reconstruction share the property that if the undirected parser produces the correct labeled undirected graph for a given sentence, then the postprocessing will transform it into the correct parse, which is simply the one obtained by following all the annotations in the undirected arcs.

4.3. Example

We can see how the reconstruction techniques work by going back to our running example sentence (Figure 2). In Section 2.3, we saw that the directed Planar parser could parse this sentence with the transition sequence shown in Figure 4. Then, at the beginning of Section 3, we illustrated how a wrong choice by the parser could cause error propagation: If an erroneous link from $dependency_4$ to $tree_5$ was created instead of the correct link from $tree_5$ to $dependency_4$, the single-head constraint would then disallow creating the correct link from is_2 to $tree_5$, causing another attachment error.

If we parsed this sentence with the undirected Planar parser and the naive reconstruction, this error would never happen, because the undirected parser would simply not need to make the choice between linking $dependency_4 \rightarrow tree_5$ or $dependency_4 \leftarrow tree_5$. It would simply execute an ARC transition, as in Figure 5, and produce the undirected arc between $dependency_4$ and $tree_5$ that can be seen in Figure 3. The naive reconstruction technique would then extract from this graph the correct orientation (Figure 2), which is the one where every arc points away from the root.

Note that the naive reconstruction always guarantees that the correct directed parse will be obtained as long as the undirected parse generates the underlying undirected arcs correctly *including those originating from the dummy root*, which point to the root word(s) of the input sentence. This is because rooted trees are implicitly directed. However, if the parser incorrectly identifies the root word of the sentence by linking the dummy root to the wrong node, the naive reconstruction will assign the wrong direction to some of the arcs. An example of this situation can be seen in Figure 7.

On the other hand, if instead of the naive reconstruction we used the label-based reconstruction, the direction error in the example would translate into a labeling error in the undirected parser: Instead of creating an undirected edge labeled $NMOD_L$ between the nodes $dependency_4$ and $tree_5$, the edge would be labeled $NMOD_R$, indicating a preference for right attachment. However, this preference would not be followed by the reconstruction technique, because there would be no possible way to conform to all the preferences at the same time without the node $tree_5$ getting two heads, and the only way of disobeying only one annotation (corresponding to the minimum spanning tree, with cost 15 in the second variant of label-based reconstruction) would be to disregard precisely that annotation and output, again, the parse of Figure 2.

Therefore, in this particular toy example, the combination of undirected parsing and any of the reconstruction techniques not only avoids the error propagation due to erroneously linking from $dependency_4$ to $\rightarrow tree_5$ but even eliminates the original error itself during postprocessing. Of course, not all the cases will be so favorable when applying these techniques in practice (Figure 7), hence the need to evaluate them empirically to see whether undirected parsing can improve accuracy in real-life settings.

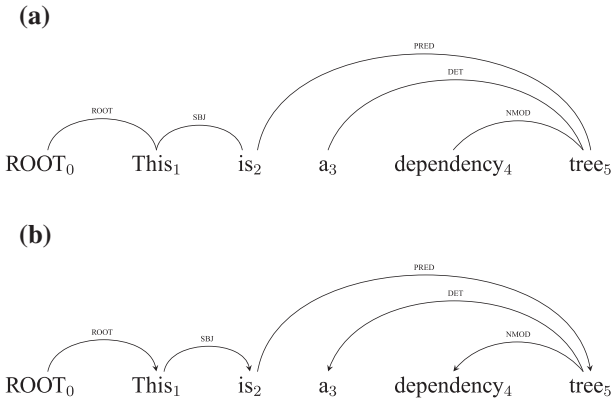


FIGURE 7. Example of an incorrect directed graph obtained by applying the naive reconstruction when the dummy root node was mistakenly attached. (a) The undirected graph obtained after the parsing process where the ROOT was attached to the node $This_1$ instead of the node is_2 as appears in Figure 3 and (b) the resulting dependency tree after applying the naive reconstruction technique, obtaining the arc from $This_1$ to is_2 rather than the correct arc from is_2 to $This_1$ as is described in Figure 2.

5. EXPERIMENTS

In this section, we evaluate the performance of the undirected Planar, 2-Planar, and Covington parsers. For each transition system, we compare the accuracy of the undirected versions with naive and label-based reconstructions to that of the original directed version. In addition, we provide a comparison to well-known state-of-the-art projective and nonprojective parsers.

To evaluate the performance of the parsers in different languages, we use the following eight data sets from the CoNLL-X shared task: Arabic (Hajič et al. 2004), Chinese (Chen et al. 2003), Czech (Hajič et al. 2006), Danish (Kromann 2003), German (Brants et al. 2002), Portuguese (Afonso et al. 2002), Swedish (Nilsson, Hall, and Nivre 2005), and Turkish (Oflazer et al. 2003; Atalay, Oflazer, and Say 2003). These data sets have been chosen for their representativity, as they cover a wide range of language families (Germanic, Romance, Semitic, Sino-Tibetan, Slavic, and Turkic), annotation types (e.g., pure dependency annotation in the case of Danish, dependencies extracted from constituents in the case of Chinese, or from discontinuous constituents in German), and degrees of nonprojectivity (ranging from the fully projective Chinese treebank to the highly nonprojective Czech and German data sets). Buchholz and Marsi (2006) and Havelka (2007) provide detailed information about these and other characteristics of the CoNLL-X treebanks.

In addition to the CoNLL-X data sets, we also perform experiments on English using the *Wall Street Journal* (WSJ) corpus from the well-known Penn Treebank (Marcus, Santorini, and Marcinkiewicz 1993). We obtain dependency structures from the constituency trees in the treebank by using the Penn2Malt converter,[4] with the head-finding rules of Yamada and Matsumoto (2003), and we follow the approach of using sections 2–21 as the training set, 23 as the development set, and 24 as the test set.

For our tests, all the algorithms were implemented in MaltParser (Nivre et al. 2006) and trained with classifiers from the LIBSVM (Chang and Lin 2001) and LIBLINEAR (Fan et al. 2008) packages. In particular, to reduce the training time for larger data sets, we employed the LIBLINEAR package for Chinese, Czech, English, and German, and we used Support Vector Machine classifiers from the LIBSVM package for the remaining languages.

The arc-eager projective and pseudo-projective parsers were trained with the LIBSVM feature models presented in the CoNLL 2006 shared task, where the pseudo-projective version of MaltParser was one of the two top performing systems (Buchholz and Marsi 2006). The feature models for the 2-Planar parser were taken from Gómez-Rodríguez and Nivre (2010) for the languages included in that paper. In the cases where no data set was available from these sources (e.g., for English), we tuned our own feature models.

Regarding the new undirected parsers, their feature models for each algorithm and data set were created from those of the directed parsers as described in Section 3.2.

5.1. Parsing Accuracy

To evaluate the accuracy of the undirected parsers with respect to their directed counterparts, we score each parser on the following standard evaluation metrics:

- *Labeled attachment score (LAS)*: the proportion of tokens (nodes) that are assigned both the correct head and the correct dependency relation label.
- *Unlabeled attachment score (UAS)*: the proportion of tokens (nodes) that are assigned the correct head (regardless of the dependency relation label).

[4] http://w3.msi.vxu.se/~nivre/research/Penn2Malt.html

TABLE 1. Parsing Accuracy of the Undirected Planar Parser with Naive (UPlanarN) and Label-Based (UPlanarL) Postprocessing in Comparison with the Directed Planar Parser (Planar).

	Planar		UPlanarN		UPlanarL	
Language	LAS	UAS	LAS	UAS	LAS	UAS
Arabic	67.34	77.22	66.33	76.75	**67.50**	**77.57**
Chinese	84.20	88.33	83.10	86.95	**84.50**	**88.35**
Czech	77.70	83.24	75.60	81.14	**77.93**	**83.41**
Danish	82.60	86.64	81.94	86.33	**83.83***	**88.17**
German	83.60	85.67	82.77	84.93	**85.67***	**87.69**
Portuguese	83.82	86.88	83.21	86.48	**84.83***	**88.03**
Swedish	82.44	87.36	81.10	85.86	**82.66**	**87.45**
Turkish	71.27	78.57	68.31	75.17	**71.68**	**78.99**
English (WSJ)	85.47	86.55	85.47	86.58	**86.02***	**87.12**
Average	79.83	84.50	78.65	83.35	**80.51***	**85.20**

Results where there is a statistically significant difference at the 0.05 level between the labeled attachment score (LAS) of Planar and UPlanarL are marked with an asterisk and the highest labeled attachment score (LAS) and unlabeled attachment score (UAS) for each language are rendered in bold. The last line shows macro averages across all data sets.

In our results, we show both LAS and UAS considering every token in the input sentences, including punctuation, as a scoring token.

In particular, Table 1 shows the results obtained by the undirected Planar parser with respect to the original Planar parser by Gómez-Rodríguez and Nivre (2013). Table 2 compares the results of the undirected 2-Planar parser with the 2-Planar parser by Gómez-Rodríguez and Nivre (2013). Finally, Table 3 shows the results obtained by the undirected Covington nonprojective parser in comparison with the directed implementation by Nivre (2008).

The results show that the use of undirected parsing with label-based reconstruction (UPlanarL) improves the scores of the Planar parser on all of the nine data sets tested. In most cases, it even attains higher scores than the 2-Planar baseline parser considered, which is remarkable if we take into account that the 2-Planar parser has more theoretical coverage due to its support of crossing links. In the case of 2-planar parsing, applying this technique (U2PlanarL) outperforms the LAS of the directed 2-Planar parser on all the data sets except for Arabic and Portuguese. Finally, the undirected Covington nonprojective parser with label-based reconstruction (UCovingtonL) outperforms the results obtained by the baseline parser (Covington) on all the treebanks except for Portuguese and Swedish.

The improvements achieved in LAS by undirected parsers with label-based reconstruction over the directed versions are statistically significant at the 0.05 level[5] for Danish, English, German, and Portuguese for the Planar parser; English in the case of the 2-Planar parser; and Czech, Danish, English, and Turkish in the case of the Covington parser. Furthermore, no statistically significant *decrease* in accuracy was observed in any of the algorithm/data set combinations.

[5] Statistical significance was assessed using Dan Bikel's randomized comparator: http://www.cis.upenn.edu/~dbikel/software.html

TABLE 2. Parsing Accuracy of the Undirected 2-Planar Parser with Naive (U2PlanarN) and Label-Based (U2PlanarL) Postprocessing in Comparison with the Directed 2-Planar Parser (2Planar).

	2Planar		U2PlanarN		U2PlanarL	
Language	LAS	UAS	LAS	UAS	LAS	UAS
Arabic	**67.19**	**77.11**	66.93	77.09	66.52	76.70
Chinese	84.32	**88.27**	82.98	86.81	**84.42**	88.25
Czech	77.91	83.32	75.19	80.80	**78.59**	**84.21**
Danish	83.61	87.63	81.63	85.80	**83.65**	**87.82**
German	85.76	87.86	82.53	84.81	**85.99**	**87.92**
Portuguese	**84.92**	**88.14**	83.19	86.33	83.74	87.11
Swedish	82.18	**87.43**	80.71	85.68	**82.25**	87.29
Turkish	70.09	77.39	67.44	74.06	**70.64**	**77.46**
English (WSJ)	85.56	86.66	85.36	86.53	**85.96***	**87.04**
Average	80.17	**84.87**	78.44	83.10	**80.20**	**84.87**

Results where there is a statistically significant difference at the 0.05 level between the labeled attachment score (LAS) of 2Planar and U2PlanarL are marked with an asterisk and the highest labeled attachment score (LAS) and unlabeled attachment score (UAS) for each language are rendered in bold. The last line shows macro averages across all data sets.

TABLE 3. Parsing Accuracy of the Undirected Covington Nonprojective Parser with Naive (UCovingtonN) and Label-Based (UCovingtonL) Postprocessing in Comparison with the Directed Algorithm (Covington).

	Covington		UCovingtonN		UCovingtonL	
Language	LAS	UAS	LAS	UAS	LAS	UAS
Arabic	65.49	**75.69**	62.89	72.83	**65.81**	75.66
Chinese	85.61	89.62	83.48	87.13	**86.17**	**90.04**
Czech	77.43	83.15	71.50	77.96	**78.69***	**84.16**
Danish	82.89	87.06	78.90	83.32	**83.85***	**87.75**
German	85.69	87.78	80.01	82.28	**85.90**	**87.95**
Portuguese	**85.21**	**88.21**	81.71	85.17	84.20	87.11
Swedish	**82.76**	**87.61**	81.47	85.96	82.73	87.23
Turkish	72.70	79.75	72.07	79.09	**73.38***	**80.40**
English (WSJ)	84.33	85.44	84.24	85.33	**85.73***	**86.77**
Average	80.23	84.92	77.36	82.12	**80.72***	**85.23**

Results where there is a statistically significant difference at the 0.05 level between the labeled attachment score (LAS) of Covington and UCovingtonL are marked with an asterisk and the highest labeled attachment score (LAS) and unlabeled attachment score (UAS) for each language are rendered in bold. The last line shows macro averages across all data sets.

As expected, the undirected parsers with naive reconstruction (UPlanarN, U2PlanarN, and UCovingtonN) performed worse than those with label-based reconstruction in all the experiments.

To further put these results into context, we provide a comparison of the novel undirected parsers, configured with label-based reconstruction, with well-known projective and nonprojective parsers. Table 4 compares the undirected Planar parser with the arc-eager

TABLE 4. Parsing Accuracy of the Undirected Planar (UPlanarL) with Label-Based Postprocessing in Comparison with the MaltParser Arc-Eager Projective (MaltP) Algorithm.

	UPlanarL		MaltP	
Language	LAS	UAS	LAS	UAS
Arabic	**67.50**	**77.57**	66.74	76.83
Chinese	84.50	88.35	**86.39**	**90.02**
Czech	**77.93**	**83.41**	77.57	83.19
Danish	**83.83**	**88.17**	82.64	86.91
German	**85.67**	**87.69**	85.48	87.58
Portuguese	**84.83**	**88.03**	84.66	87.73
Swedish	**82.66**	87.45	82.44	**87.55**
Turkish	**71.68**	**78.99**	70.96	77.95
English (WSJ)	86.02	87.12	**86.77**	**87.82**
Average	**80.51**	**85.20**	80.41	85.06

The last line shows macro averages across all data sets and the highest labeled attachment score (LAS) and unlabeled attachment score (UAS) for each language are rendered in bold.

TABLE 5. Parsing Accuracy of the Undirected 2-Planar Parser (U2PlanarL) and the Undirected Covington Nonprojective Parser (UCovingtonL) with Label-Based Postprocessing in Comparison with the MaltParser Arc-Eager Pseudo-Projective (MaltPP) Algorithm.

	U2PlanarL		UCovingtonL		MaltPP	
Language	LAS	UAS	LAS	UAS	LAS	UAS
Arabic	**66.52**	**76.70**	65.81	75.66	66.02	76.14
Chinese	84.42	88.25	86.17	**90.04**	**86.39**	90.02
Czech	78.59	**84.21**	**78.69**	84.16	78.47	83.89
Danish	83.65	**87.82**	**83.85**	87.75	83.54	87.70
German	85.99	87.92	85.90	87.95	**86.62**	**88.69**
Portuguese	83.74	87.11	84.20	87.11	**84.90**	**87.95**
Swedish	82.25	87.29	**82.73**	87.23	82.67	**87.38**
Turkish	70.64	77.46	**73.38**	**80.40**	71.33	78.44
English (WSJ)	85.96	87.04	85.73	86.77	**86.77**	**87.82**
Average	80.20	84.87	80.72	85.23	**80.75**	**85.34**

The last line shows macro averages across all data sets and the highest labeled attachment score (LAS) and unlabeled attachment score (UAS) for each language are rendered in bold.

projective parser by Nivre (2003), a well-known algorithm that is also restricted to planar dependency structures.[6] The arc-eager parser is the default parsing algorithm in MaltParser (Nivre et al. 2006) and is also the dependency parser used in other current systems such as

[6] The arc-eager parser covers the set of *projective* dependency structures. Planar structures are a very mild relaxation of projective structures, and in fact, both sets become equivalent when sentences are required to have a single root node at position 0.

ZPar (Zhang and Clark 2011). In addition, Table 5 shows the results of the undirected 2-Planar and the undirected Covington algorithms compared with those of the arc-eager parser with the pseudo-projective transformation of Nivre and Nilsson (2005), which is able to handle nonplanar dependencies and is probably the most widely used method for nonprojective transition-based parsing.

As we can see in these experiments, the undirected Planar parser obtains a better score than the arc-eager algorithm (MaltP) on seven out of nine tests; while the accuracy of the pseudo-projective arc-eager parser (MaltPP) is outperformed on four out of nine languages by the undirected Covington parser, and on the Arabic, Czech and Danish data sets by the undirected 2-Planar algorithm. Therefore, undirected parsing with label-based reconstruction can be used to improve the accuracy of parsing algorithms and produces results that are competitive with state-of-the-art transition-based parsers.

5.2. Undirected Accuracy

The results obtained in Section 5.1 show that undirected parsing with the label-based reconstruction is clearly beneficial for parsing accuracy, while the naive reconstruction is not. To obtain a clearer picture of how and why this happens, we can perform an evaluation of the parsers on *undirected evaluation metrics*, i.e., metrics that only take into account the undirected graph underlying the output of each parser, ignoring arc directions. While the standard metrics used in Section 5.1 are the most relevant for practical applications, undirected metrics can provide insights into the influence of each step of undirected parsing on the final accuracy result (i.e., how the accuracy of structural parsing is affected by modifications of feature models described in Section 3.2 and what proportion of errors are introduced by the naive or label-based transformation that is applied afterward).

Therefore, we will now analyze the results of each parser on the following evaluation metrics:

- *Undirected LAS (ULAS)*: the proportion of labeled arcs that are built correctly by the parser if we disregard direction, i.e., the proportion of arcs $i \overset{l}{\rightarrow} j$ in the gold standard annotation such that the parser produces either $i \overset{l}{\rightarrow} j$ or $j \overset{l}{\rightarrow} i$.
- *Undirected UAS (UUAS)*: the proportion of arcs that are built correctly by the parser if we disregard both label and direction, i.e., the proportion of arcs $i \overset{l}{\rightarrow} j$ in the gold standard annotation such that the parser produces either $i \overset{l'}{\rightarrow} j$ or $j \overset{l'}{\rightarrow} i$ for some label l' not necessarily equal to l.
- *Labeled direction errors (LDE)*: the proportion of labeled arcs that are built correctly by the parser except for their direction, i.e., the proportion of arcs $i \overset{l}{\rightarrow} j$ in the gold standard annotation such that the parser outputs the arc $j \overset{l}{\rightarrow} i$. This is trivially equivalent to the difference between ULAS and LAS.
- *Unlabeled direction errors (UDE)*: the proportion of arcs that are built correctly by the parser except for their direction, disregarding labels. This is the proportion of arcs $i \overset{l}{\rightarrow} j$ in the gold standard annotation such that the parser outputs an arc of the form $j \overset{l'}{\rightarrow} i$ for some label l' and is trivially equivalent to the difference between UUAS and UAS.

Tables 6, 7, and 8 show the undirected scores obtained by the directed and undirected variants of the Planar, 2-Planar, and Covington parsers, respectively.

TABLE 6. Unlabeled Parsing Accuracy Metrics for the Undirected Planar Parser with Naive (UPlanarN) and Label-Based (UPlanarL) Postprocessing in Comparison with the Directed Planar Parser (Planar).

	Planar				UPlanarN				UPlanarL			
Language	ULAS	UUAS	LDE	UDE	ULAS	UUAS	LDE	UDE	ULAS	UUAS	LDE	UDE
Arabic	67.56	78.89	0.22	1.67	67.04	78.67	0.71	1.92	**67.71**	**79.21**	0.21	1.64
Chinese	84.22	**90.34**	0.02	2.01	84.02	89.68	0.92	2.73	**84.58**	90.14	0.08	1.79
Czech	78.06	85.03	0.36	1.79	76.97	84.04	1.37	2.90	**78.35**	**85.34**	0.42	1.93
Danish	82.66	87.66	0.06	1.02	82.79	88.17	0.85	1.84	**83.90**	**88.91**	0.07	0.74
German	83.81	86.34	0.21	0.67	84.26	86.88	1.49	1.95	**85.88**	**88.39**	0.21	0.70
Portuguese	83.84	87.35	0.02	0.47	83.62	87.47	0.41	0.99	**84.85**	**88.58**	0.02	0.55
Swedish	82.50	88.33	0.06	0.97	82.02	87.98	0.92	2.12	**82.74**	**88.35**	0.08	0.90
Turkish	71.35	**79.01**	0.08	0.44	70.17	77.57	1.86	2.40	**71.68**	78.99	0.00	0.00
English (WSJ)	85.91	87.75	0.44	1.20	86.55	88.36	1.08	1.78	**86.66**	**88.55**	0.64	1.43
Average	79.99	85.63	0.16	1.14	79.72	85.42	1.07	2.07	**80.71**	**86.27**	0.19	1.08

The last line shows macro averages across all data sets and the highest undirected labeled attachment score (ULAS) and undirected unlabeled attachment score (UUAS) for each language are rendered in bold. LDE, labeled direction errors; UDE, unlabeled direction errors.

TABLE 7. Unlabeled Parsing Accuracy Metrics for the Undirected 2-Planar Parser with Naive (U2PlanarN) and Label-Based (U2PlanarL) Postprocessing in Comparison with the Directed 2-Planar Parser (2Planar).

	2Planar				U2PlanarN				U2PlanarL			
Language	ULAS	UUAS	LDE	UDE	ULAS	UUAS	LDE	UDE	ULAS	UUAS	LDE	UDE
Arabic	**67.41**	**78.84**	0.22	1.73	67.41	78.89	0.48	1.80	66.74	78.50	0.22	1.80
Chinese	84.34	**90.24**	0.02	1.97	83.84	89.53	0.86	2.72	**84.48**	90.04	0.06	1.79
Czech	78.32	85.24	0.41	1.92	76.70	84.03	1.51	3.23	**79.07**	**86.06**	0.48	1.85
Danish	83.68	88.70	0.07	1.07	82.45	87.68	0.82	1.88	**83.71**	**88.74**	0.06	0.92
German	85.95	88.39	0.19	0.53	83.79	86.53	1.26	1.72	**86.27**	**88.67**	0.28	0.75
Portuguese	**84.92**	**88.68**	0.00	0.54	83.65	87.35	0.46	1.02	83.77	87.73	0.03	0.62
Swedish	82.21	**88.42**	0.03	0.99	81.51	87.43	0.80	1.75	**82.32**	88.19	0.07	0.90
Turkish	70.13	77.83	0.04	0.44	69.74	77.59	2.30	3.53	**70.80**	**77.91**	0.16	0.45
English (WSJ)	86.03	87.91	0.47	1.25	86.52	88.32	1.16	1.79	**86.62**	**88.50**	0.66	1.46
Average	80.33	86.03	0.16	1.16	79.51	85.26	1.07	2.16	**80.42**	**86.04**	0.22	1.17

The last line shows macro averages across all data sets and the highest undirected labeled attachment score (ULAS) and undirected unlabeled attachment score (UUAS) for each language are rendered in bold. LDE, labeled direction errors; UDE, unlabeled direction errors.

TABLE 8. Unlabeled Parsing Accuracy Metrics for the Undirected Covington Parser with Naive (UCovN) and Label-Based (UCovL) Postprocessing in Comparison with the Directed Covington Parser (Cov).

	Cov				UCovN				UCovL			
Language	ULAS	UUAS	LDE	UDE	ULAS	UUAS	LDE	UDE	ULAS	UUAS	LDE	UDE
Arabic	65.77	77.98	0.28	2.29	65.64	77.29	2.75	4.46	**66.72**	**78.41**	0.91	2.75
Chinese	85.65	91.06	0.04	1.44	84.94	90.22	1.46	3.09	**86.27**	**91.50**	0.10	1.46
Czech	77.79	85.08	0.36	1.93	75.11	83.17	3.61	5.21	**79.09**	**86.04**	0.40	1.88
Danish	82.98	88.19	0.09	1.13	79.99	85.59	1.09	2.27	**84.11**	**88.72**	0.26	0.97
German	85.88	88.43	0.19	0.65	82.31	85.16	2.30	2.88	**86.21**	**88.67**	0.31	0.72
Portuguese	**85.21**	**88.75**	0.00	0.54	82.75	86.76	1.04	1.59	84.32	87.85	0.12	0.74
Swedish	82.83	**88.42**	0.07	0.81	82.62	87.78	1.15	1.82	**82.89**	88.15	0.16	0.92
Turkish	72.76	80.06	0.06	0.31	73.05	80.38	0.98	1.29	**73.50**	**80.83**	0.12	0.43
English (WSJ)	84.83	86.71	0.50	1.27	85.99	87.69	1.75	2.36	**86.32**	**88.07**	0.59	1.30
Average	80.41	86.08	0.18	1.15	79.16	84.89	1.79	2.77	**81.05**	**86.47**	0.33	1.24

The last line shows macro averages across all data sets and the highest undirected labeled attachment score (ULAS) and undirected unlabeled attachment score (UUAS) for each language are rendered in bold. LDE, labeled direction errors; UDE, unlabeled direction errors.

A first interesting insight from the tables is that they tell us the prevalence of direction errors, such as the one in the example of Sections 3 and 4.3, in the output of directed parsers. As we can see in the UDE column, more than 1% of the arcs in each parser's output is a direction error (i.e., an arc of the form $a \rightarrow b$ that should have been $b \rightarrow a$); this represents around 7.5% of the errors made by the parsers. In the majority of the cases where the parsers make a direction error, they miss the dependency label as well, probably because there is a strong correspondence between syntactic functions and dependency directions in most treebanks.

Although undirected parsing can be useful in sentences where the directed parser makes a direction error, because the undirected parser does not need to commit to a direction until the preprocessing step, it is worth reminding that the ultimate goal of undirected parsing is *not* to prevent or correct direction errors but to prevent error propagation caused by the single-head constraint. For instance, in the example of Section 4.3, the directed parser made a direction error, but this then caused a nondirection error that was avoided by the undirected parsing technique. Whether the direction error is also finally avoided or not does not depend on the undirected parsing phase but on the reconstruction technique: Thus, in undirected parsers, we can see the ULAS and UUAS metrics as an evaluation of the parsing phase (as its output is an undirected graph) and the LDE and UDE metrics as an evaluation of the reconstruction phase.

In particular, if we compare the direction error metrics for the naive and label-based reconstructions on any of the three algorithms, we can see that the naive reconstruction gets many arc directions wrong. As explained in Section 4.1, this is because the naive approach lacks robustness, as a wrong choice of root can propagate and cause many arcs to be assigned the wrong direction. In contrast, the label-based reconstruction gives very good results, to the point that it does not produce more direction errors than the original directed parsers.

While the differences in direction errors introduced by the transformations explain part of the differences in the LAS and UAS scores between the naive and label-based reconstructions reported in Section 5.1, the tables show that there is also a difference in ULAS and UUAS between both approaches; i.e., the label-based reconstruction already produces a better undirected graph than the naive one even before the reconstruction process. Because the parsing algorithm is the same in both cases, this can only be due to the feature models: In the label-based reconstruction, labels are augmented with preferred directions, and this information is included (in the same way as standard labels) in the feature models, while in the naive reconstruction, no directionality information is encoded in labels or used as features.

Therefore, we can see that having no directionality information during parsing has a negative effect on the accuracy of structural prediction. However, adding the information about preferred direction that is available when using the label-based approach compensates for this, making the feature models in Section 3.2 obtain good results that improve over the directed approach.

6. ERROR ANALYSIS

The results in the previous section show that undirected parsing with label-based reconstruction can improve the accuracy of several parsing algorithms and obtain results that are competitive with the state of the art in transition-based parsing. These results seem to support our hypothesis that the undirected parsing approach can successfully alleviate error propagation, formulated in Section 3.

To further test whether this is in fact the reason for the improvements in accuracy, we conduct a more in-depth analysis of the outputs produced by the parsers in Section 5, going beyond the global directed and undirected accuracy metrics to see the circumstances under which parsing errors are being made.

In particular, as observed by McDonald and Nivre (2007), a good indicator of error propagation in transition-based parsers is the loss of dependency precision for longer dependency arcs (with the length of an arc $i \xrightarrow{l} j$ being defined as $|j-i|$). In most transition-based parsers, shorter arcs tend to be created earlier in transition sequences than longer ones: e.g., all the algorithms considered here (like the one used in McDonald and Nivre (2007)) have the property that given nodes $i < j < k < l$, an arc connecting j and k will always be created before an arc connecting i and l. This means that longer arcs, being created later, will be more likely to be affected by the propagation of errors made in previous parsing decisions.

Figure 8 shows the labeled dependency arc precision obtained by the directed and undirected planar parsers for different predicted dependency lengths; i.e., for each length l, it shows the percentage of correct arcs among the arcs of length l in the parser output. Figure 9 shows the labeled dependency arc recall for those same parsers as a function of gold dependency length, i.e., the percentage of gold standard arcs of each length l in the test set that were correctly predicted by the parser.

Instead of doing a language-by-language analysis, we measure these values across all data sets by aggregating the parser outputs for every language into a single file, and evaluating it with respect to a corresponding joined gold standard file, thus following the same method as McDonald and Nivre (2007). This ensures that we gain a better insight into the distribution of parsing errors, because the data sets for each individual language are too small to have a meaningful sample of arcs for each given distance. Note that, although this form of data aggregation means that the obtained precision and recall values are micro averages, the CoNLL-X test sets purposely have roughly the same size (5,000 tokens). In the case of the Penn Treebank, we randomly chose a sample of that size as well, so that no language is overrepresented and the result would be roughly the same if we computed a macro average instead.

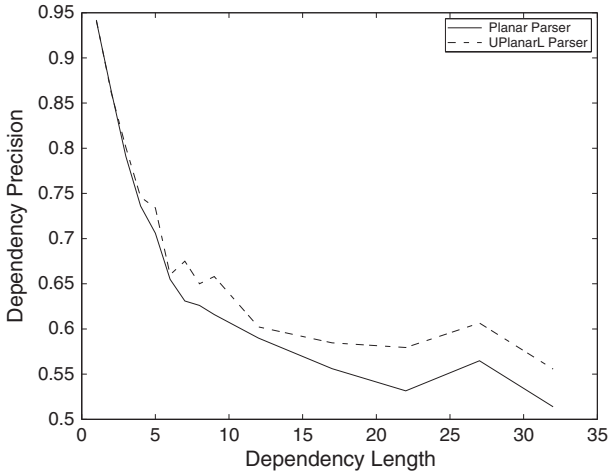


FIGURE 8. Dependency arc precision relative to predicted dependency length for the Planar parser (Planar) and the undirected Planar parser with label-based reconstruction (UPlanarL) on the nine data sets.

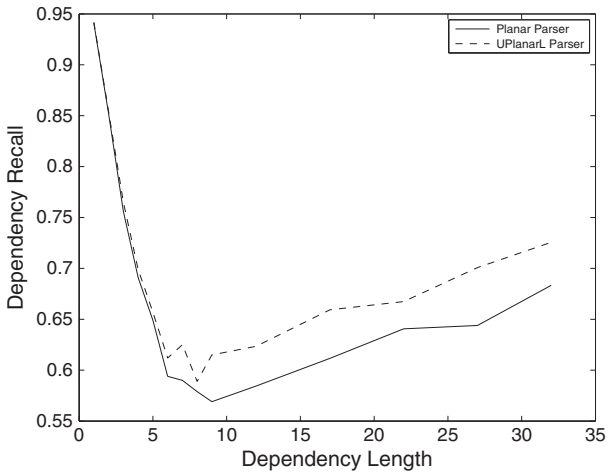


FIGURE 9. Dependency arc recall relative to gold dependency length for the Planar parser (Planar) and the undirected Planar parser with label-based reconstruction (UPlanarL) on the nine data sets.

Starting from length 10, we group the data into bins of size 5—[10, 15] and others—for a more meaningful visualization because, even with the mentioned aggregation, data become sparser from that point and there may be no arcs at all for some particular lengths.

As we can see in Figure 8, both the directed and the undirected planar parsers experiment a drop in precision for longer-distance dependencies, which is expected, among other reasons, because both are transition-based parsers and will exhibit error propagation.[7] However, this phenomenon is significantly more pronounced in the directed parser, and while the accuracies of both algorithms are practically indistinguishable for arcs of lengths smaller than 5, the undirected Planar parser obtains a clearly better precision on longer arcs, with the difference in precision typically being between 3 and 5 points. Note that the difference in global LAS was not so huge (see Section 5) because shorter dependencies are more frequent in treebanks than longer ones, and thus, they have a higher weight in the overall LAS metric.

The recall measurements, shown in Figure 9, exhibit a very similar trend. Note that, in principle, precision is a more useful metric than recall for the purpose of estimating the impact of error propagation, because the order in which dependency arcs are built is directly related to their relative length and position in the output trees, with the relation to the gold standard tree being more indirect. However, we include recall for completeness.

These results for the directed and undirected planar parsers suggest that, as we hypothesized, the undirected variant of the parser is less affected by error propagation than the original directed version. This is the cause of the higher precision and recall for

[7] While error propagation is probably the most important reason why a transition-based parser's performance drops for longer dependencies, it is not the only cause for this phenomenon: McDonald and Nivre (2007) observe that even in the graph-based parser by McDonald, Lerman, and Pereira (2006), which does not analyze sentences sequentially and thus cannot exhibit error propagation, there is a slight drop in accuracy for longer dependencies. A reason for this is that there is less training data for longer dependencies than for shorter ones, because most dependencies in natural language treebanks connect words that are close to each other. Additionally, longer dependencies tend to occur more frequently in ambiguous constructions and in complex, relatively infrequent linguistic phenomena that are difficult to parse (Nivre et al. 2010), while short dependencies include many trivial instances like the attachment of determiners to nouns, where there is little ambiguity and many examples in training sets allowing for a high parsing accuracy.

longer-distance dependencies reflected in Figures 8 and 9 and of the improvement in overall accuracy that we observed in Section 5.

Figures 10 and 11 provide precision and recall measurements analogous to those in Figures 8 and 9, but this time for the directed and undirected 2-Planar parsers. In this case, the results are less conclusive, with the undirected parser slightly outperforming the directed one in precision for arcs with lengths between 5 and 10, but behaving worse for those with lengths greater than 10. It should be noted that the overall precision of the undirected parser for arcs of lengths at least 5 is better than that of the directed version, even though the area under the curve is slightly larger for the latter, because arcs of lengths 5 to 10 are more frequent in treebanks than those of lengths larger than 10. However, the results do not

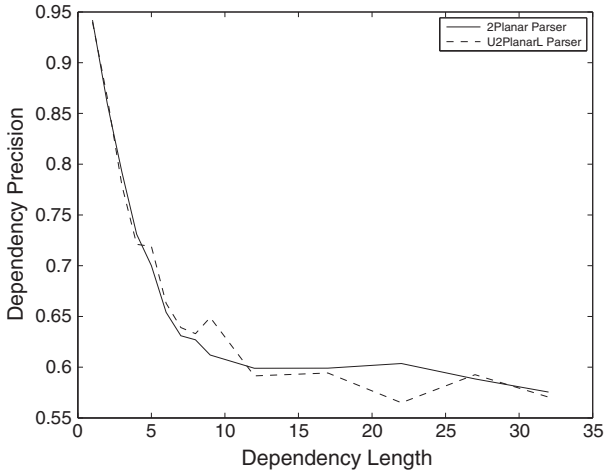


FIGURE 10. Dependency arc precision relative to predicted dependency length for the 2-Planar parser (2Planar) and the undirected 2-Planar parser with label-based reconstruction (U2PlanarL) on the nine data sets.

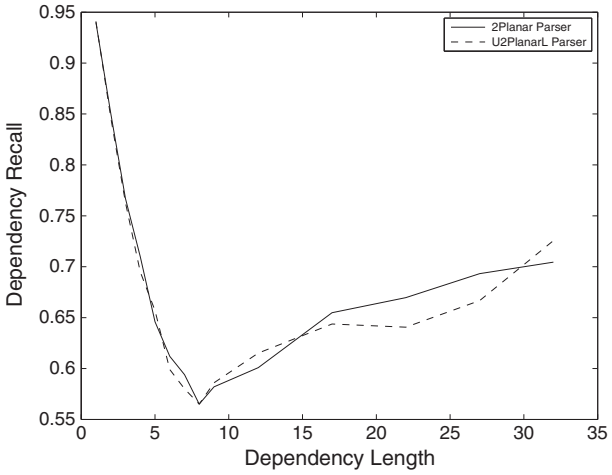


FIGURE 11. Dependency arc recall relative to gold dependency length for the 2-Planar parser (2Planar) and the undirected 2-Planar parser with label-based reconstruction (U2PlanarL) on the nine data sets.

seem marked enough to suggest that there is a consistent mitigation of error propagation across languages in this case. This is coherent with the results of Section 5, where the 2-Planar algorithm was the one that benefitted the least from undirected parsing in terms of LAS, obtaining improvements in several data sets but only a very slight improvement on the cross-language average.

Finally, Figures 12 and 13 provide the same comparison for the two variants of the Covington algorithm. In this case, it is again clear that the undirected variant exhibits less error propagation than the directed one: It obtains better precision for dependencies of lengths between 5 and 20 (which are the vast majority of longer-distance dependencies), and much better recall for those with lengths between 5 and 30, with differences up to 5

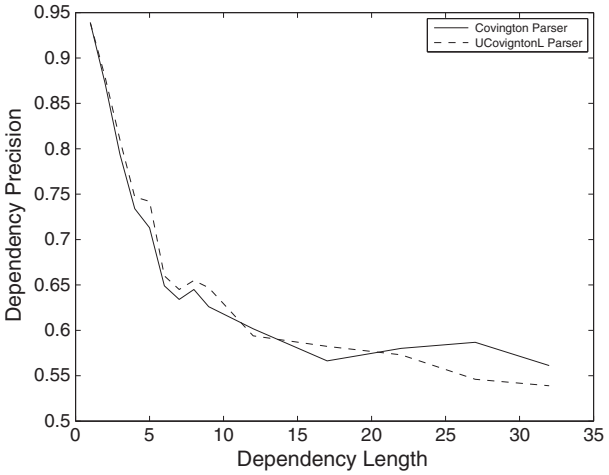


FIGURE 12. Dependency arc precision relative to predicted dependency length for the Covington parser (Covington) and the undirected Covington parser with label-based reconstruction (UCovingtonL) on the nine data sets.

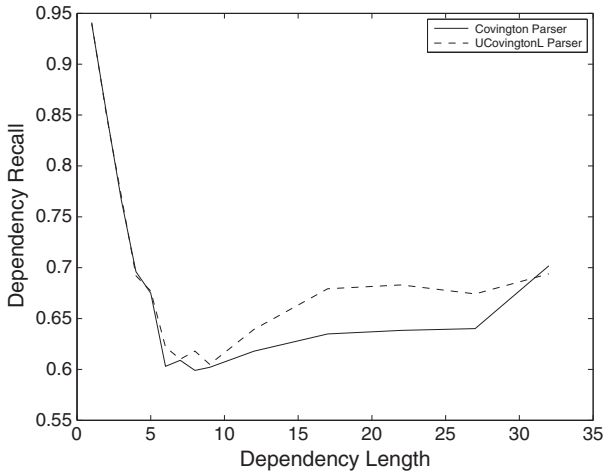


FIGURE 13. Dependency arc recall relative to gold dependency length for the Covington parser (Covington) and the undirected Covington parser with label-based reconstruction (UCovingtonL) on the nine data sets.

percentage points for some of these lengths. The fact that the improvement is larger in recall than in precision indicates that the undirected Covington parser tends to generate more long arcs than the directed one.

Summing up the results of the error analysis, we can see that undirected parsing clearly increases the precision for longer-distance dependency arcs, at least in the Planar and Covington cases, providing evidence that this technique successfully alleviates error propagation. In the case of the 2-Planar algorithm, the results are less conclusive, because no clear increase (or decrease) in precision has been observed for longer dependencies. This is in line with the LAS results in Section 5, where applying the undirected technique achieved much larger LAS improvements on the Planar and Covington algorithms than on the 2-Planar parser.

One possible reason why the undirected technique works better for Planar and Covington than for 2-Planar is that the latter parser suffers from less error propagation than the others in the first place. This can be seen by comparing the results for the directed parsers in Figures 8, 10, and 12: In the Planar and Covington parsers, the precision soon drops below 0.6 when we increase the dependency length, while the 2-Planar parser keeps precisions around 0.6 for longer, suggesting that it is less affected by error propagation.

7. RELATED WORK

In this section, we review existing work in the literature related to using undirected relations between words for parsing, as well as other proposals to address the problem of error propagation in transition-based parsers.

7.1. Undirected Parsing

The idea of parsing with undirected relations between words has been applied before in the work on Link Grammar (Sleator and Temperley 1991). However, in that case, undirected arcs are the desired final result, and not an intermediate result, because the Link Grammar formalism itself represents the syntactic structure of sentences by means of undirected links.

While most languages exhibit both left and right dependency relations, some languages that are strongly head-initial or head-final can be annotated assuming that all the dependency arcs have the same orientation. In such languages, dependency parsers need not worry about arc directions, and therefore, directed and undirected parsing are equivalent. For example, the Japanese parser by Kudo and Matsumoto (2002) simplifies its processing by assuming that all arcs must point to the left, because Japanese is strongly head-final. This is equivalent to producing an undirected dependency structure as the final output. However, such an approach would not be useful for most other languages, as they contain both left and right dependencies.

To the best of our knowledge, the idea of obtaining an undirected graph as an intermediate step for parsing directed dependency structures has not been explored before in the literature.

7.2. Error Propagation Mitigation

On the other hand, the problem of error propagation in transition-based dependency parsers has received considerable attention in the last few years.

One way to reduce error propagation is to expand the parser's search space by using beam search instead of greedy deterministic search, in such a way that the parser is not limited to exploring a single transition sequence (Zhang and Clark 2008; Zhang and Nivre 2011; Bohnet and Kuhn 2012). This approach has been shown to produce considerable

improvements in accuracy when it is combined with global structural learning, instead of local learning in each parser configuration (Zhang and Nivre 2012). While this technique produces larger accuracy improvements than undirected parsing, it also has a much larger computational cost. In theory, using a beam increases parsing time by at least a factor proportional to the beam size. In practice, Goldberg, Zhao and Huang (2013) show that most actual implementations additionally add an extra $O(n)$ factor to each transition, making linear transition systems run in quadratic time. In contrast, our undirected parsing approach does not incur any performance penalty during parsing: The only extra processing needed is the postprocessing step that assigns directions to arcs, but this has a negligible impact in the total runtime because it is a simple graph algorithm that does not need to perform classification or feature extraction, which are the bottlenecks in transition-based parsing (Bohnet 2010; Volokh 2013). It is also worth noting that undirected parsing and beam search are not mutually exclusive, because the former is a transformation on transition systems and the latter is a search strategy that can be applied on top of any transition system. Thus, both approaches can be applied simultaneously.

An alternative approach, introduced very recently, tackles the problem in a more efficient way by employing dynamic oracles (Nivre and Goldberg 2012). In this approach, the parser uses greedy search, but it is trained by an online procedure that considers more than one transition sequence for each given tree (spurious ambiguity) and simulates erroneous transitions to help the parser learn to recover from errors. However, this cannot be applied to any parser because defining a dynamic oracle requires computing a cost function on transitions, and it is only known how to do this efficiently for a very small set of transition-based parsers. In particular, no dynamic oracles have been defined for nonprojective parsers, while our undirected approach can be applied to nonprojective parsers such as 2-Planar and Covington. Note that if dynamic oracles were defined in the future for algorithms supporting undirected parsing, then both techniques would also be applicable simultaneously.

Finally, the easy-first parser by Goldberg and Elhadad (2010) is a transition-based parser that builds the easier and more reliable dependency arcs first, regardless of their position in the sentence. This limits error propagation because it leaves the decisions more likely to cause errors for the end of the process, and allows the parser to have more contextual information available for those hard decisions, which produces significant improvements in accuracy. However, note that this is an entirely new algorithm running in $O(nlog(n))$, while our approach is a transformation applicable to several different algorithms.

8. CONCLUSION

In this article, we have presented a technique to transform transition-based dependency parsers satisfying certain conditions into undirected dependency parsers, which can then be implemented and trained with feature models that do not depend on the directions of dependency links. The resulting parsers have the drawback that they generate undirected graphs instead of dependency trees, but we have shown how arc directions can be recovered from the undirected output by means of one of two different postprocessing techniques, so that the final result is a fully functional dependency parser.

The advantage of the parsers obtained in this way is that they do not need to obey the single-head constraint until the postprocessing step. This gives the parser more freedom when choosing transitions to apply, and alleviates error propagation, thus producing improvements in accuracy with respect to directed parsers. We have backed this claim with experiments in which we evaluated the directed and undirected version of the Planar, 2-Planar (Gómez-Rodríguez and Nivre 2010, 2013), and Covington (Covington 2001; Nivre 2008) parsing algorithms, obtaining improvements in LAS in 23 out of 27 algorithm-data set

combinations, with statistically significant differences for several of them, outperforming well-known state-of-the-art transition-based parsers. A more in-depth analysis has shown that the undirected parsers tend to perform especially well for longer-distance dependencies, which supports the hypothesis that the increase in accuracy is due to alleviation of error propagation.

Note that in this article, we have obtained undirected parsers by transforming existing directed parsers, and this provided a good baseline to assess the usefulness of the undirected parsing technique. However, it would also be possible to define undirected parsers from scratch, without necessarily being based on any directed parsers, and apply the same reconstruction techniques to them so as to obtain directed dependency structures as output. This is an interesting avenue for future work, along with the implementation of undirected dependency parsing with beam-search decoding.

ACKNOWLEDGMENTS

This work was partially funded by the Spanish Ministry of Economy and Competitiveness and FEDER (projects TIN2010-18552-C03-01 and TIN2010-18552-C03-02), the Ministry of Education (FPU grant program), and the Xunta de Galicia (CN2012/008, CN2012/317, and CN2012/319). The experiments were conducted with the help of computing resources provided by the Supercomputing Center of Galicia (CESGA). We thank Joakim Nivre for helpful input in the early stages of this work.

REFERENCES

AFONSO, S., E. BICK, R. HABER, and D. SANTOS. 2002. Floresta sintá(c)tica: a treebank for Portuguese. *In* Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC 2002), ELRA, Paris, France, pp. 1968–1703.

ATALAY, N. B., K. OFLAZER, and B. SAY. 2003. The annotation process in the Turkish treebank. *In* Proceedings of EACL Workshop on Linguistically Interpreted Corpora (LINC-03). Association for Computational Linguistics: Morristown, NJ, pp. 243–246.

ATTARDI, G. 2006. Experiments with a multilanguage non-projective dependency parser. *In* Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL), New York, NY, pp. 166–170.

BERANT, J., I. DAGAN, and J. GOLDBERGER. 2010. Global learning of focused entailment graphs. *In* Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, ACL '10. Association for Computational Linguistics: Stroudsburg, PA, pp. 1220–1229. http://dl.acm.org/citation.cfm?id=1858681.1858805.

BOHNET, B. 2010. Very high accuracy and fast dependency parsing is not a contradiction. *In* Proceedings of the 23rd International Conference on Computational Linguistics, COLING '10. Association for Computational Linguistics: Stroudsburg, PA, pp. 89–97. http://dl.acm.org/citation.cfm?id=1873781.1873792.

BOHNET, B., and J. KUHN. 2012. The best of both worlds: a graph-based completion model for transition-based parsers. *In* Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics, EACL '12. Association for Computational Linguistics: Stroudsburg, PA, pp. 77–87. http://dl.acm.org/citation.cfm?id=2380816.2380828.

BRANAVAN, S. R. K., D. SILVER, and R. BARZILAY. 2012. Learning to win by reading manuals in a Monte-Carlo framework. Journal of Artificial Intelligence Research, **43**(1): 661–704. http://dl.acm.org/citation.cfm?id=2387915.2387932.

BRANTS, S., S. DIPPER, S. HANSEN, W. LEZIUS, and G. SMITH. 2002. The tiger treebank. *In* Proceedings of the Workshop on Treebanks and Linguistic Theories, September 20–21, Sozopol, Bulgaria, pp. 24–41. http://www.coli.uni-sb.de/~sabine/tigertreebank.pdf.

BUCHHOLZ, S., and E. MARSI. 2006. CoNLL-X shared task on multilingual dependency parsing. *In* Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL), New York, NY, pp. 149–164.

CHANG, C.-C., and C.-J. LIN. 2001. LIBSVM: a library for support vector machines. Software available at: http://www.csie.ntu.edu.tw/∼cjlin/libsvm.

CHEN, K., C. LUO, M. CHANG, F. CHEN, C. CHEN, C. HUANG, and Z. GAO. 2003. Sinica treebank: design criteria, representational issues and implementation. *In* Treebanks: Building and Using Parsed Corpora. *Edited by* A. ABEILLÉ. Kluwer: Dordrecht, the Netherlands, pp. 231–248.

CHU, Y. J., and T. H. LIU. 1965. On the shortest arborescence of a directed graph. Science Sinica, **14**: 1396–1400.

COMAS, P. R., J. TURMO, and L. MÁRQUEZ. 2010. Using dependency parsing and machine learning for factoid question answering on spoken documents. *In* Proceedings of the 13th International Conference on Spoken Language Processing (INTERSPEECH 2010), Makuhari, Japan, pp. 1–4.

COVINGTON, M. A. 1990. A dependency parser for variable-word-order languages. Technical Report AI-1990-01, University of Georgia, Athens, GA.

COVINGTON, M. A. 2001. A fundamental algorithm for dependency parsing. *In* Proceedings of the 39th Annual ACM Southeast Conference, Athens, GA, pp. 95–102.

CUI, H., R. SUN, K. LI, M.-Y. KAN, and T.-S. CHUA. 2005. Question answering passage retrieval using dependency relations. *In* SIGIR '05: Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM: New York, NY, pp. 400–407.

CULOTTA, A., and J. SORENSEN. 2004. Dependency tree kernels for relation extraction. *In* ACL '04: Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics: Morristown, NJ, pp. 423–429.

DING, Y., and M. PALMER. 2005. Machine translation using probabilistic synchronous dependency insertion grammars. *In* ACL '05: Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics: Morristown, NJ, pp. 541–548.

EDMONDS, J. 1967. Optimum branchings. Journal of Research of the National Bureau of Standards, **71B**: 233–240.

EISNER, J. M. 1996. Three new probabilistic models for dependency parsing: an exploration. *In* Proceedings of the 16th International Conference on Computational Linguistics (COLING), Copenhagen, Denmark, pp. 340–345.

FAN, R.-E., K.-W. CHANG, C.-J. HSIEH, X.-R. WANG, and C.-J. LIN. 2008. LIBLINEAR: a library for large linear classification. Journal of Machine Learning Research, **9**: 1871–1874.

FUNDEL, K., R. KÜFFNER, and R. ZIMMER. 2006. RelEx—Relation extraction using dependency parse trees. Bioinformatics, **23**(3): 365–371.

GOLDBERG, Y., and M. ELHADAD. 2010. An efficient algorithm for easy-first non-directional dependency parsing. *In* Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL HLT), Los Angeles, CA, pp. 742–750.

GOLDBERG, Y., K. ZHAO, and L. HUANG. 2013. Efficient implementation of beam-search incremental parsers. *In* Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL 2013), Short Papers. Sofia, Bulgaria, pp. 628–633.

GÓMEZ-RODRÍGUEZ, C., and D. FERNÁNDEZ-GONZÁLEZ. 2012. Dependency parsing with undirected graphs. *In* Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics. Association for Computational Linguistics: Stroudsburg, PA, pp. 66–76.

GÓMEZ-RODRÍGUEZ, C., and J. 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 '10. Association for Computational Linguistics: Stroudsburg, PA, pp. 1492–1501. http://portal.acm.org/citation.cfm?id=1858681.1858832.

GÓMEZ-RODRÍGUEZ, C., and J. NIVRE. 2013. Divisible transition systems and multiplanar dependency parsing. Computational Linguistics, **39**(4): 799–845.

HAJIČ, J., J. PANEVOVÁ, E. HAJIČOVÁ, J. PANEVOVÁ, P. SGALL, P. PAJAS, J. ŠTĚPÁNEK, J. HAVELKA, and M. MIKULOVÁ. 2006. Prague Dependency Treebank 2.0. CDROM CAT: LDC2006T01, ISBN 1-58563-370-4. Linguistic Data Consortium.

HAJIČ, J., O. SMRŽ, P. ZEMÁNEK, J. ŠNAIDAUF, and E. BEŠKA. 2004. Prague Arabic Dependency Treebank: development in data and tools. *In* Proceedings of the NEMLAR International Conference on Arabic Language Resources and Tools, Cairo, Egypt, pp. 110–117.

HAVELKA, J. 2007. Beyond projectivity: multilingual evaluation of constraints and measures on non-projective structures. *In* ACL 2007: Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics: Morristown, NJ, pp. 608–615.

HAYASHI, K., T. WATANABE, M. ASAHARA, and Y. MATSUMOTO. 2012. Head-driven transition-based parsing with top-down prediction. *In* Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Jeju Island, Korea, pp. 657–665. http://www.aclweb.org/anthology/P12-1069.

HERRERA, J., A. PEÑAS, and F. VERDEJO. 2005. Textual entailment recognition based on dependency analysis and WordNet. *In* Machine Learning Challenges, volume 3944 of Lecture Notes in Computer Science. Springer-Verlag: Berlin-Heidelberg-New York, pp. 231–239.

HUANG, L., W. JIANG, and Q. LIU. 2009. Bilingually-constrained (monolingual) shift-reduce parsing. *In* Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), Singapore, pp. 1222–1231.

HUANG, L., and K. SAGAE. 2010. Dynamic programming for linear-time incremental parsing. *In* Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL), Uppsala, Sweden, pp. 1077–1086.

JOHANSSON, R., and P. NUGUES. 2006. Investigating multilingual dependency parsing. *In* Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL), New York, NY, pp. 206–210.

JOSHI, M., and C. PENSTEIN-ROSÉ. 2009. Generalizing dependency features for opinion mining. *In* Proceedings of the ACL-IJCNLP 2009 Conference Short Papers, ACLShort '09. Association for Computational Linguistics: Stroudsburg, PA, pp. 313–316. http://dl.acm.org/citation.cfm?id=1667583.1667680.

KATRENKO, S., P. ADRIAANS, and M. VAN SOMEREN. 2010. Using local alignments for relation recognition. Journal of Artificial Intelligence Research, **38**(1): 1–48. http://dl.acm.org/citation.cfm?id=1892211.1892212.

KATZ-BROWN, J., S. PETROV, R. MCDONALD, F. OCH, D. TALBOT, H. ICHIKAWA, and M. SENO. 2011. Training a parser for machine translation reordering. *In* Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing (EMNLP '11), Edinburgh, UK, pp. 183–192. http://petrovi.de/data/emnlp11b.pdf.

KROMANN, M. T. 2003. The Danish dependency treebank and the underlying linguistic theory. *In* Proceedings of the 2nd Workshop on Treebanks and Linguistic Theories (TLT). Växjö University Press: Växjö, Sweden, pp. 217–220.

KUDO, T., and Y. MATSUMOTO. 2002. Japanese dependency analysis using cascaded chunking. *In* Proceedings of the 6th Workshop on Computational Language Learning (CoNLL), Taipei, Taiwan, pp. 63–69.

LOMBARDO, V., and L. LESMO. 1996. An Earley-type recognizer for dependency grammar. *In* Proceedings of the 16th International Conference on Computational Linguistics (COLING 96). ACL/Morgan Kaufmann: San Francisco, CA, pp. 723–728.

MARCUS, M. P., B. SANTORINI, and M. A. MARCINKIEWICZ. 1993. Building a large annotated corpus of English: the Penn Treebank. Computational Linguistics, **19**: 313–330.

MCDONALD, R., K. LERMAN, and F. PEREIRA. 2006. Multilingual dependency analysis with a two-stage discriminative parser. *In* Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL), New York, NY, pp. 216–220.

MCDONALD, R., and J. NIVRE. 2007. Characterizing the errors of data-driven dependency parsing models. *In* Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), Prague, Czech Republic, pp. 122–131.

MCDONALD, R, and J. NIVRE. 2011. Analyzing and integrating dependency parsers. Computational Linguistics, **37**: 197–230.

MCDONALD, R., F. PEREIRA, K. RIBAROV, and J. HAJIČ. 2005. Non-projective dependency parsing using spanning tree algorithms. *In* Proceedings of the Human Language Technology Conference and the Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP), Vancouver, Canada, pp. 523–530.

MIYAO, Y., K. SAGAE, R. SÆTRE, T. MATSUZAKI, and J. TSUJII. 2009. Evaluating contributions of natural language parsers to protein-protein interaction extraction. Bioinformatics, **25**(3): 394–400. http://bioinformatics.oxfordjournals.org/cgi/content/abstract/25/3/394.

NILSSON, J., J. HALL, and J. NIVRE. 2005. MAMBA meets TIGER: reconstructing a Swedish treebank from antiquity. *In* Proceedings of the Nodalida Special Session on Treebanks. *Edited by* P. J. HENRICHSEN. Samfundslitteratur Press: Copenhagen, Denmark, pp. 119–132.

NIVRE, J. 2003. An efficient algorithm for projective dependency parsing. *In* Proceedings of the 8th International Workshop on Parsing Technologies (IWPT), Nancy, France, pp. 149–160.

NIVRE, J. 2008. Algorithms for deterministic incremental dependency parsing. Computational Linguistics, **34**(4): 513–553. ISSN 0891-2017. http://www.mitpressjournals.org/doi/abs/10.1162/coli.07-056-R1-07-027.

NIVRE, J., and Y. GOLDBERG. 2012. A dynamic oracle for arc-eager dependency parsing. *In* Proceedings of the 24th International Conference on Computational Linguistics (COLING), Mumbai, India, pp. 959–976.

NIVRE, J., J. HALL, and J. NILSSON. 2004a. Memory-based dependency parsing. *In* Proceedings of the 8th Conference on Computational Natural Language Learning (CONLL-2004). Association for Computational Linguistics: Morristown, NJ, pp. 49–56.

NIVRE, J., J. HALL, and J. NILSSON. 2004b. Memory-based dependency parsing. *In* Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL). Association for Computational Linguistics: Morristown, NJ, pp. 49–56.

NIVRE, J., J. HALL, and J. NILSSON. 2006. Maltparser: a data-driven parser-generator for dependency parsing. *In* Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC), Genoa, Italy, pp. 2216–2219.

NIVRE, J., and J. NILSSON. 2005. Pseudo-projective dependency parsing. *In* Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL), Ann Arbor, MI, pp. 99–106.

NIVRE, J., L. RIMELL, R. MCDONALD, and C. GÓMEZ RODRÍGUEZ. 2010. Evaluation of dependency parsers on unbounded dependencies. *In* Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010), Beijing, China, pp. 833–841.

OFLAZER, K., B. SAY, D. Z. HAKKANI-TÜR, and G. TÜR. 2003. Building a Turkish treebank. *In* Treebanks: Building and Using Parsed Corpora. *Edited by* A. ABEILLÉ. Kluwer: Dordrecht, the Netherlands, pp. 261–277.

SAGAE, K., and J. TSUJII. 2008. Shift-reduce dependency DAG parsing. *In* COLING '08: Proceedings of the 22nd International Conference on Computational Linguistics. Association for Computational Linguistics: Morristown, NJ, pp. 753–760.

SHEN, L., J. XU, and R. WEISCHEDEL. 2008. A new string-to-dependency machine translation algorithm with a target dependency language model. *In* Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-08: HLT). Association for Computational Linguistics: Morristown, NJ, pp. 577–585.

SLEATOR, D., and D. TEMPERLEY. 1991. Parsing English with a link grammar. Technical Report CMU-CS-91-196, Computer Science, Carnegie Mellon University, Pittsburgh, PA.

TAPANAINEN, P., and T. JÄRVINEN. 1997. A non-projective dependency parser. *In* Proceedings of the Fifth Conference on Applied Natural Language Processing, ANLC '97. Association for Computational Linguistics: Stroudsburg, PA, pp. 64–71. http://dx.doi.org/10.3115/974557.974568.

TARJAN, R. E. 1977. Finding optimum branchings. Networks, **7**: 25–35.

TITOV, I., and J. HENDERSON. 2007. A latent variable model for generative dependency parsing. *In* Proceedings of the 10th International Conference on Parsing Technologies (IWPT), Prague, Czech Republic, pp. 144–155.

TRATZ, S., and E. HOVY. 2011. A fast, accurate, non-projective, semantically-enriched parser. *In* Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Edinburgh, UK, pp. 1257–1268. http://www.aclweb.org/anthology/D11-1116.

VOLOKH, A. 2013. Performance-Oriented dependency parsing, PhD dissertation, Saarbrücken, Germany.

XU, P., J. KANG, M. RINGGAARD, and F. OCH. 2009. Using a dependency parser to improve SMT for subject-object-verb languages. *In* Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, NAACL '09. Association for Computational Linguistics: Stroudsburg, PA, pp. 245–253. http://dl.acm.org/citation.cfm?id=1620754.1620790.

YAMADA, H., and Y. MATSUMOTO. 2003. Statistical dependency analysis with support vector machines. *In* Proceedings of the 8th International Workshop on Parsing Technologies (IWPT), Nancy, France, pp. 195–206.

ZHANG, Y., and S. CLARK. 2008. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing. *In* Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), Honolulu, HI, pp. 562–571.

ZHANG, Y., and S. CLARK. 2011. Syntactic processing using the generalized perceptron and beam search. Computational Linguistics, **37**(1): 105–151.

ZHANG, Y., and J. NIVRE. 2011. Transition-based dependency parsing with rich non-local features. *In* Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: Short Papers - Volume 2, HLT '11. Association for Computational Linguistics: Stroudsburg, PA, pp. 188–193.

ZHANG, Y., and J. NIVRE. 2012. Analyzing the effect of global learning and beam-search on transition-based dependency parsing. *In* COLING (Posters). *Edited by* M. KAY, and C. BOITET. Indian Institute of Technology: Bombay, Mumbai, India, pp. 1391–1400.

Improving Transition-Based Dependency Parsing with Buffer Transitions

Daniel Fernández-González
Departamento de Informática
Universidade de Vigo
Campus As Lagoas, 32004
Ourense, Spain
danifg@uvigo.es

Carlos Gómez-Rodríguez
Departamento de Computación
Universidade da Coruña
Campus de Elviña, 15071
A Coruña, Spain
carlos.gomez@udc.es

Abstract

In this paper, we show that significant improvements in the accuracy of well-known transition-based parsers can be obtained, without sacrificing efficiency, by enriching the parsers with simple transitions that act on buffer nodes.

First, we show how adding a specific transition to create either a left or right arc of length one between the first two buffer nodes produces improvements in the accuracy of Nivre's arc-eager projective parser on a number of datasets from the CoNLL-X shared task. Then, we show that accuracy can also be improved by adding transitions involving the topmost stack node and the second buffer node (allowing a limited form of non-projectivity).

None of these transitions has a negative impact on the computational complexity of the algorithm. Although the experiments in this paper use the arc-eager parser, the approach is generic enough to be applicable to any stack-based dependency parser.

1 Introduction

Dependency parsing has become a very active research area in natural language processing in recent years. The dependency representation of syntax simplifies the syntactic parsing task, since no non-lexical nodes need to be postulated by the parsers; while being convenient in practice, since dependency representations directly show the head-modifier and head-complement relationships which form the basis of predicate-argument structure. This has led to the development of various data-driven dependency parsers, such as those by Yamada and Matsumoto (2003), Nivre et al. (2004), McDonald et al. (2005), Martins et al. (2009), Huang and Sagae (2010) or Tratz and Hovy (2011), which can be trained directly from annotated data and produce accurate analyses very efficiently.

Most current data-driven dependency parsers can be classified into two families, commonly called **graph-based** and **transition-based** parsers (McDonald and Nivre, 2011). Graph-based parsers (Eisner, 1996; McDonald et al., 2005) are based on global optimization of models that work by scoring subtrees. On the other hand, transition-based parsers (Yamada and Matsumoto, 2003; Nivre et al., 2004), which are the focus of this work, use local training to make greedy decisions that deterministically select the next parser state. Among the advantages of transition-based parsers are the linear time complexity of many of them and the possibility of using rich feature models (Zhang and Nivre, 2011).

In particular, many transition-based parsers (Nivre et al., 2004; Attardi, 2006; Sagae and Tsujii, 2008; Nivre, 2009; Huang and Sagae, 2010; Gómez-Rodríguez and Nivre, 2010) are **stack-based** (Nivre, 2008), meaning that they keep a stack of partially processed tokens and an input buffer of unread tokens. In this paper, we show how the accuracy of this kind of parsers can be improved, without compromising efficiency, by extending their set of available transitions with **buffer transitions**. These are transitions that create a dependency arc involving some node in the buffer, which would typically be considered unavailable for linking by these algo-

Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pages 308–319, Jeju Island, Korea, 12–14 July 2012.

rithms. The rationale is that buffer transitions construct some "easy" dependency arcs in advance, before the involved nodes reach the stack, so that the classifier's job when choosing among standard transitions is simplified.

To test the approach, we use the well-known arc-eager parser by (Nivre, 2003; Nivre et al., 2004) as a baseline, showing improvements in accuracy on most datasets of the CoNLL-X shared task (Buchholz and Marsi, 2006). However, the techniques discussed in this paper are generic and can also be applied to other stack-based dependency parsers.

The rest of this paper is structured as follows: Section 2 is an introduction to transition-based parsers and the arc-eager parsing algorithm. Section 3 presents the first novel contribution of this paper, **projective buffer transitions**, and discusses their empirical results on CoNLL-X datasets. Section 4 does the same for a more complex set of transitions, **non-projective buffer transitions**. Finally, Section 5 discusses related work and Section 6 sums up the conclusions and points out avenues for future work.

2 Preliminaries

We now briefly present some basic definitions for transition-based dependency parsing; a more thorough explanation can be found in (Nivre, 2008).

2.1 Dependency graphs

Let $w = w_1 \dots w_n$ be an input string. A **dependency graph** for w is a directed graph $G = (V_w, A)$; where $V_w = \{0, 1, \dots, n\}$ is a set of nodes, and $A \subseteq V_w \times L \times V_w$ is a set of labelled arcs. Each node in V_w encodes the position of a token in w, where 0 is a dummy node used as artificial root. An arc (i, l, j) will also be called a **dependency link** labelled l from i to j. We say that i is the syntactic **head** of j and, conversely, that j is a **dependent** of i. The **length** of the arc (i, l, j) is the value $|j - i|$.

Most dependency representations of syntax do not allow arbitrary dependency graphs. Instead, they require dependency graphs to be **forests**, i.e., acyclic graphs where each node has at most one head. In this paper, we will work with parsers that assume dependency graphs $G = (V_w, A)$ to satisfy the following properties:

- Single-head: every node has at most one incoming arc (if $(i, l, j) \in A$, then for every $k \neq i$, $(k, l', j) \notin A$).
- Acyclicity: there are no directed cycles in G.
- Node 0 is a root, i.e., there are no arcs of the form $(i, l, 0)$ in A.

A dependency forest with a single root (i.e., where all the nodes but one have at least one incoming arc) is called a **tree**. Every dependency forest can trivially be represented as a tree by adding arcs from the dummy root node 0 to every other root node.

For reasons of computational efficiency, many dependency parsers are restricted to work with forests satisfying an additional restriction called projectivity. A dependency forest is said to be **projective** if the set of nodes reachable by traversing zero or more arcs from any given node k corresponds to a continuous substring of the input (i.e., is an interval $\{x \in V_w \mid i \leq x \leq j\}$). For trees with a dummy root node at position 0, this is equivalent to not allowing dependency links to cross when drawn above the nodes (planarity).

2.2 Transition systems

A **transition system** is a nondeterministic state machine that maps input strings to dependency graphs. In this paper, we will focus on **stack-based transition systems**. A stack-based transition system is a quadruple $S = (C, T, c_s, C_t)$ where

- C is a set of parser **configurations**. Each configuration is of the form $c = (\sigma, \beta, A)$ where σ is a list of nodes of V_w called the **stack**, β is a list of nodes of V_w called the **buffer**, and A is a set of dependency arcs,
- T is a finite set of **transitions**, each of which is a partial function $t : C \rightarrow C$,
- c_s is an initialization function, mapping a sentence $w_1 \dots w_n$ to an **initial configuration** $c_s = ([0], [1, \dots, n], \emptyset)$,
- C_t is the set of **terminal configurations** $C_t = (\sigma, [], A) \in C$.

Transition systems are nondeterministic devices, since several transitions may be applicable to the same configuration. To obtain a deterministic parser

from a transition system, a classifier is trained to greedily select the best transition at each state. This training is typically done by using an **oracle**, which is a function $o : C \rightarrow T$ that selects a single transition at each configuration, given a tree in the training set. The classifier is then trained to approximate this oracle when the target tree is unknown.

2.3 The arc-eager parser

Nivre's arc-eager dependency parser (Nivre, 2003; Nivre et al., 2004) is one of the most widely known and used transition-based parsers (see for example (Zhang and Clark, 2008; Zhang and Nivre, 2011)). This parser works by reading the input sentence from left to right and creating dependency links as soon as possible. This means that links are created in a strict left-to-right order, and implies that while leftward links are built in a bottom-up fashion, a rightward link $a \rightarrow b$ will be created before the node b has collected its right dependents.

The arc-eager transition system has the following four transitions (note that, for convenience, we write a stack with node i on top as $\sigma|i$, and a buffer whose first node is i as $i|\beta$):

- SHIFT : $(\sigma, i|\beta, A) \Rightarrow (\sigma|i, \beta, A)$.
- REDUCE : $(\sigma|i, \beta, A) \Rightarrow (\sigma, \beta, A)$. Precondition: $\exists k, l' \mid (k, l', i) \in A$.
- LEFT-ARC$_l$: $(\sigma|i, j|\beta, A) \Rightarrow (\sigma, j|\beta, A \cup \{(j, l, i)\})$. Preconditions: $i \neq 0$ and $\nexists k, l' \mid (k, l', i) \in A$ (single-head)
- RIGHT-ARC$_l$: $(\sigma|i, j|\beta, A) \Rightarrow (\sigma|i|j, \beta, A \cup \{(i, l, j)\})$.

The SHIFT transition reads an input word by removing the first node from the buffer and placing it on top of the stack. The REDUCE transition pops the stack, and it can only be executed if the topmost stack node has already been assigned a head. The LEFT-ARC transition creates an arc from the first node in the buffer to the node on top of the stack, and then pops the stack. It can only be executed if the node on top of the stack does not already have a head. Finally, the RIGHT-ARC transition creates an arc from the top of the stack to the first buffer node, and then removes the latter from the buffer and moves it to the stack.

The arc-eager parser has linear time complexity. In principle, it is restricted to projective dependency forests, but it can be used in conjunction with the pseudo-projective transformation (Nivre et al., 2006) in order to capture a restricted subset of non-projective forests. Using this setup, it scored as one of the top two systems in the CoNLL-X shared task.

3 Projective buffer transitions

In this section, we show that the accuracy of stack-based transition systems can benefit from adding one of a pair of new transitions, which we call **projective buffer transitions**, to their transition sets.

3.1 The transitions

The two projective buffer transitions are defined as follows:

- LEFT-BUFFER-ARC$_l$: $(\sigma, i|j|\beta, A) \Rightarrow (\sigma, j|\beta, A \cup \{(j, l, i)\})$.
- RIGHT-BUFFER-ARC$_l$: $(\sigma, i|j|\beta, A) \Rightarrow (\sigma, i|\beta, A \cup \{(i, l, j)\})$.

The LEFT-BUFFER-ARC transition creates a leftward dependency link from the second node to the first node in the buffer, and then removes the first node from the buffer. Conversely, the RIGHT-BUFFER-ARC transition creates a rightward dependency link from the first node to the second node in the buffer, and then removes the second node. We call these transitions projective buffer transitions because, since they act on contiguous buffer nodes, they can only create projective arcs.

Adding one (or both) of these transitions to a projective or non-projective stack-based transition system does not affect its correctness, as long as this starting system cannot generate configurations (σ, β, A) where a buffer node has a head in A[1]: it cannot affect completeness because we are not removing existing transitions, and therefore any dependency graph that the original system could build

[1]Most stack-based transition systems in the literature disallow such configurations. However, in parsers that allow them (such as those defined by Gómez-Rodríguez and Nivre (2010)), projective buffer transitions can still be added without affecting correctness if we impose explicit single-head and acyclicity preconditions on them. We have not included these preconditions by default for simplicity of presentation.

will still be obtainable by the augmented one; and it cannot affect soundness (be it for projective dependency forests or for any superset of them) because the new transitions can only create projective arcs and cannot violate the single-head or acyclicity constraints, given that a buffer node cannot have a head.

The idea behind projective buffer transitions is to create dependency arcs of length one (i.e., arcs involving contiguous nodes) *in advance* of the standard arc-building transitions that need at least one of the nodes to get to the stack (LEFT-ARC and RIGHT-ARC in the case of the arc-eager transition system).

Our hypothesis is that, as it is known that short-distance dependencies are easier to learn for transition-based parsers than long-distance ones (McDonald and Nivre, 2007), handling these short arcs in advance and removing their dependent nodes will make it easier for the classifier to learn how to make decisions involving the standard arc transitions.

Note that the fact that projective buffer transitions create arcs of length 1 is not explicit in the definition of the transitions. For instance, if we add the LEFT-BUFFER-ARC$_l$ transition only to the arc-eager transition system, LEFT-BUFFER-ARC$_l$ will only be able to create arcs of length 1, since it is easy to see that the first two buffer nodes are contiguous in all the accessible configurations. However, if we add RIGHT-BUFFER-ARC$_l$, this transition will have the potential to create arcs of length greater than 1: for example, if two consecutive RIGHT-BUFFER-ARC$_l$ transitions are applied starting from a configuration $(\sigma, i|i+1|i+2|\beta, A)$, the second application will create an arc $i \rightarrow i+2$ of length 2.

Although we could have added the length-1 restriction to the transition definitions, we have chosen the more generic approach of leaving it to the oracle instead. While the oracle typically used for the arc-eager system follows the simple principle of executing transitions that create an arc as soon as it has the chance to, adding projective buffer transitions opens up new possibilities: we may now have several ways of creating an arc, and we have to decide in which cases we train the parser to use one of the buffer transitions and in which cases we prefer to train it to ignore the buffer transitions and delegate to the standard ones. Following the hypothesis explained above, our policy has been to train the parser to use buffer transitions whenever possible for arcs of length one, and to not use them for arcs of length larger than one. To test this idea, we also conducted experiments with the alternative policy "use buffer transitions whenever possible, regardless of arc length": as expected, the obtained accuracies were (slightly) worse.

The chosen oracle policy is generic and can be plugged into any stack-based parser: for a given transition, first check whether it is possible to build a gold-standard arc of length 1 with a projective buffer transition.[2] If so, choose that transition, and if not, just delegate to the original parser's oracle.

3.2 Experiments

To empirically evaluate the effect of projective buffer transitions on parsing accuracy, we have conducted experiments on eight datasets of the CoNLL-X shared task (Buchholz and Marsi, 2006): Arabic (Hajič et al., 2004), Chinese (Chen et al., 2003), Czech (Hajič et al., 2006), Danish (Kromann, 2003), German (Brants et al., 2002), Portuguese (Afonso et al., 2002), Swedish (Nilsson et al., 2005) and Turkish (Oflazer et al., 2003; Atalay et al., 2003).

As our baseline parser, we use the arc-eager projective transition system by Nivre (2003). Table 1 compares the accuracy obtained by this system alone with that obtained when the LEFT-BUFFER-ARC and RIGHT-BUFFER-ARC transitions are added to it as explained in Section 3.1.

Accuracy is reported in terms of labelled (LAS) and unlabelled (UAS) attachment score. We used SVM classifiers from the LIBSVM package (Chang and Lin, 2001) for all languages except for Chinese, Czech and German. In these, we used the LIBLINEAR package (Fan et al., 2008) for classification, since it reduces training time in these larger datasets. Feature models for all parsers were specifically tuned for each language.[3]

[2] In this context, "possible" means that we can create the arc without losing the possibility of creating other gold-standard arcs. In the case of RIGHT-BUFFER-ARC, this involves checking that the candidate dependent node has no dependents in the gold-standard tree (if it has any, we cannot remove it from the stack or it would not be able to collect its dependents, so we do not use the buffer transition).

[3] All the experimental settings and feature models used are included in the supplementary material and also available at `http://www.grupolys.org/~cgomezr/exp/`.

Language	NE		NE+LBA		NE+RBA	
	LAS	UAS	LAS	UAS	LAS	UAS
Arabic	66.43	77.19	**67.78**	**78.26**	63.87	74.63
Chinese	86.46	90.18	82.47	86.14	**86.62**	**90.64**
Czech	77.24	83.40	**78.70**	**84.24**	78.28	83.94
Danish	84.91	89.80	**85.21**	**90.20**	82.53	87.35
German	86.18	88.60	84.31	86.50	**86.48**	**88.90**
Portug.	86.60	90.20	**86.92**	**90.58**	85.55	89.28
Swedish	**83.33**	**88.83**	82.81	88.03	81.66	88.03
Turkish	63.77	74.35	57.42	66.24	**64.33**	**74.73**

Table 1: Parsing accuracy (LAS and UAS, excluding punctuation) of Nivre's arc-eager parser without modification (NE), with the LEFT-BUFFER-ARC transition added (NE+LBA) and with the RIGHT-BUFFER-ARC transition added (NE+RBA). Best results for each language are shown in boldface.

As can be seen in Table 1, adding a projective buffer transition improves the performance of the parser in seven out of the eight tested languages. The improvements in LAS are statistically significant at the .01 level[4] in the Arabic and Czech treebanks.

Note that the decision of *which* buffer transition to add strongly depends on the dataset. In the majority of the treebanks, we can see that when the LEFT-BUFFER-ARC transition improves performance the RIGHT-BUFFER-ARC transition harms it, and vice versa. The exceptions are Czech, where both transitions are beneficial, and Swedish, where both are harmful. Therefore, when using projective buffer transitions in practice, the language and annotation scheme should be taken into account (or tests should be made) to decide which one to use.

Table 2 hints at the reason for this treebank-sensitiveness. By analyzing the relative frequency of leftward and rightward dependency links (and, in particular, of leftward and rightward links of length 1) in the different treebanks, we see a reasonably clear tendency: the LEFT-BUFFER-ARC transition works better in treebanks that contain a large proportion of *rightward* arcs of length 1, and the RIGHT-BUFFER-ARC transition works better in treebanks with a large proportion of *leftward* arcs of length 1. Note that, while this might seem counterintuitive at a first glance, it is coherent with the hypothesis that we formulated in Section 3.1: the advantage of projective buffer transitions is not that they build arcs more accurately than standard arc-building transitions (in fact the opposite might be expected, since they work on nodes while they are still on the buffer and we have less information about their surrounding nodes in our feature models), but that they make it easier for the classifier to decide among standard transitions. The analysis on Table 2 agrees with that explanation: LEFT-BUFFER-ARC improves performance in treebanks where it is not used too often but it can filter out leftward arcs of length 1, making it easier for the parser to be accurate on rightward arcs of length 1; and the converse happens for RIGHT-BUFFER-ARC.

Language	L%	R%	L1%	R1%	Best PBT
Arabic	12.3	87.7	6.5	55.1	LBA
Chinese	58.4	41.6	35.8	15.1	RBA
Czech	41.4	58.6	22.1	24.9	LBA*
Danish	17.1	82.9	10.9	43.0	LBA
German	39.8	60.2	20.3	19.9	RBA
Portug.	32.6	67.4	22.5	26.9	LBA
Swedish	38.2	61.8	24.1	21.8	LBA*
Turkish	77.8	22.2	47.2	10.4	RBA

Table 2: Analysis of the datasets used in the experiments in terms of: percentage of leftward and rightward links (L%, R%), percentage of leftward and rightward links of length 1 (L1%, R1%), and which projective buffer transition works better for each dataset according to the results in Table 1 (LBA = LEFT-BUFFER-ARC, RBA = RIGHT-BUFFER-ARC). Languages where both transitions are beneficial (Czech) or harmful (Swedish) are marked with an asterisk.

[4] Statistical significance was assessed using Dan Bikel's randomized parsing evaluation comparator: `http://www.cis.upenn.edu/~dbikel/software.html#comparator`

	NE		NE+LBA			NE+RBA			NE+LBA+RBA			
Language	LA	RA	LA*	RA	LBA	LA	RA*	RBA	LA*	RA*	LBA	RBA
Arabic	**58.28**	67.77	42.61	**68.65**	**77.46**	55.88	60.63	**79.70**	37.40	62.28	66.78	75.94
Chinese	85.69	**85.79**	80.92	84.19	**89.00**	**85.96**	84.77	**88.01**	81.08	79.46	87.72	86.33
Czech	85.73	76.44	80.79	**78.34**	**91.07**	**86.25**	76.62	**82.58**	79.49	75.98	90.26	81.97
Danish	89.47	83.92	88.65	**84.16**	**91.72**	86.27	78.04	**92.30**	**90.23**	77.52	88.79	92.10
German	89.15	87.11	83.75	**87.23**	**94.30**	**89.55**	84.38	**95.98**	79.26	81.60	91.66	90.73
Portuguese	**94.77**	84.91	90.83	**85.11**	**97.07**	93.84	81.86	**92.29**	88.72	79.86	96.02	89.26
Swedish	**87.75**	80.74	84.62	**81.30**	**92.83**	87.12	74.77	**90.73**	78.10	72.50	90.86	89.89
Turkish	59.68	**74.21**	53.02	74.01	**72.78**	**60.23**	69.23	**73.91**	49.34	48.48	65.57	41.94

Table 3: Labelled precision of the arcs built by each transition of Nivre's arc-eager parser without modification (NE), with a projective buffer transition added (NE+LBA, NE+RBA) and with both projective buffer transitions added (NE+LBA+RBA). We mark a standard LEFT-ARC (LA) or RIGHT-ARC (LA) transition with an asterisk (LA*, RA*) when it is acting only on a "hard" subset of leftward (rightward) arcs, and thus its precision is not directly comparable to that of (LA, RA). Best results for each language and transition are shown in boldface.

To further test this idea, we computed the labelled precision of each individual transition of the parsers with and without projective buffer transitions, as shown in Table 3. As we can see, projective buffer transitions achieve better precision than standard transitions, but this is not surprising since they act only on "easy" arcs of length 1. Therefore, this high precision does not mean that they actually build arcs more accurately than the standard transitions, since it is not measured on the same set of arcs. Similarly, adding a projective buffer transition decreases the precision of its corresponding standard transition, but this is because the standard transition is then dealing only with "harder" arcs of length greather than 1, not because it is making more errors. A more interesting insight comes from comparing transitions that are acting on the same target set of arcs: we see that, in the languages where LEFT-BUFFER-ARC is beneficial, the addition of this transition always improves the precision of the standard RIGHT-ARC transition; and the converse happens with RIGHT-BUFFER-ARC with respect to LEFT-ARC. This further backs the hypothesis that the filtering of "easy" links achieved by projective buffer transitions makes it easier for the classifier to decide among standard transitions.

We also conducted experiments adding both transitions at the same time (NE+LBA+RBA), but the results were worse than adding the suitable transition for each dataset. Table 3 hints at the reason: the precision of buffer transitions noticeably decreases when both of them are added at the same time, presumably because it is difficult for the classifier to decide between both with the restricted feature information available for buffer nodes.

	NE+LBA/RBA		NE+PP (CoNLL X)	
Language	LAS	UAS	LAS	UAS
Arabic	**67.78**	**78.26**	66.71	77.52
Chinese	86.62	**90.64**	**86.92**	90.54
Czech	**78.70**	84.24	78.42	**84.80**
Danish	**85.21**	**90.20**	84.77	89.80
German	**86.48**	**88.90**	85.82	88.76
Portug.	86.92	90.58	**87.60**	**91.22**
Swedish	82.81	88.03	**84.58**	**89.50**
Turkish	64.33	74.73	**65.68**	**75.82**

Table 4: Comparison of the parsing accuracy (LAS and UAS, excluding punctuation) of Nivre's arc-eager parser with projective buffer transitions (NE+LBA/RBA) and the parser with the pseudo-projective transformation (Nivre et al., 2006)

To further put the obtained results into context, Table 4 compares the performance of the arc-eager parser with the projective buffer transition most suitable for each dataset with the results obtained by the parser with the pseudo-projective transformation by Nivre et al. (2006) in the CoNLL-X shared task, one of the top two performing systems in that event. The reader should be aware that the purpose of this table is only to provide a broad idea of how our approach performs with respect to a well-known reference point, and not to make a detailed comparison, since the two parsers have not been tuned in homogeneous conditions: on the one hand, we had access to the CoNLL-X test sets which were unavailable

System	Arabic	Danish
Nivre et al. (2006)	66.71	84.77
McDonald et al. (2006)	66.91	84.79
Nivre (2009)	67.3	84.7
Gómez-Rodríguez and Nivre (2010)	N/A	83.81
NE+LBA/RBA	**67.78**	**85.21**

Table 5: Comparison of the Arabic and Danish LAS obtained by the arc-eager parser with projective buffer transitions in comparison to other parsers in the literature that report results on these datasets.

for the participants in the shared task; on the other hand, we did not fine-tune the classifier parameters for each dataset like Nivre et al. (2006), but used default values for all languages.

As can be seen in the table, even though the pseudo-projective parser is able to capture non-projective syntactic phenomena, the algorithm with projective buffer transitions (which is strictly projective) outperforms it in four of the eight treebanks, including non-projective treebanks such as the German one.

Furthermore, to our knowledge, our LAS results for Arabic and Danish are currently the best published results for a single-parser system on these datasets, not only outperforming the systems participating in CoNLL-X but also other parsers tested on these treebanks in more recent years (see Table 5).

Finally, it is worth noting that adding projective buffer transitions has no negative impact on efficiency, either in terms of computational complexity or of empirical runtime. Since each projective buffer transition removes a node from the buffer, no more than n such transitions can be executed for a sentence of length n, so adding these transitions cannot increase the complexity of a transition-based parser. In the particular case of the arc-eager parser, using projective buffer transitions reduces the average number of transitions needed to obtain a given dependency forest, as some nodes can be dispatched by a single transition rather than being shifted and later popped from the stack. In practice, we observed that the training and parsing times of the arc-eager parser with projective buffer transitions were slightly faster than without them on the Arabic, Chinese, Swedish and Turkish treebanks, and slightly slower than without them on the other four treebanks, so adding these transitions does not seem to noticeably degrade (or improve) practical efficiency.

4 Non-projective buffer transitions

We now present a second set of transitions that still follow the idea of early processing of some dependency arcs, as in Section 3; but which are able to create arcs skipping over a buffer node, so that they can create some non-projective arcs. For this reason, we call them **non-projective buffer transitions**.

4.1 The transitions

The two non-projective buffer transitions are defined as follows:

- LEFT-NONPROJ-BUFFER-ARC$_l$:
  $(\sigma|i, j|k|\beta, A) \Rightarrow (\sigma, j|k|\beta, A \cup \{(k, l, i)\})$.
  Preconditions: $i \neq 0$ and $\nexists m, l' \mid (m, l', i) \in A$ (single-head)

- RIGHT-NONPROJ-BUFFER-ARC$_l$:
  $(\sigma|i, j|k|\beta, A) \Rightarrow (\sigma|i, j|\beta, A \cup \{(i, l, k)\})$.

The LEFT-NONPROJ-BUFFER-ARC transition creates a leftward arc from the second buffer node to the node on top of the stack, and then pops the stack. It can only be executed if the node on top of the stack does not already have a head. The RIGHT-NONPROJ-BUFFER-ARC transition creates an arc from the top of the stack to the second node in the buffer, and then removes the latter from the buffer. Note that these transitions are analogous to projective buffer transitions, and they use the second node in the buffer in the same way, but they create arcs involving the node on top of the stack rather than the first buffer node. This change makes the precondition that checks for a head necessary for the transition LEFT-NONPROJ-BUFFER-ARC to respect the single-head constraint, since many stack-based parsers can generate configurations where the node on top of the stack has a head.

We call these transitions non-projective buffer transitions because, as they act on non-contiguous nodes in the stack and buffer, they allow the creation of a limited set of non-projective dependency arcs. This means that, when added to a projective parser, they will increase its coverage.[5] On the other hand,

[5]They may also increase the coverage of parsers allowing restricted forms of non-projectivity, but that depends on the par-

	NE		NE+LNBA		NE+RNBA	
Language	LAS	UAS	LAS	UAS	LAS	UAS
Arabic	66.43	77.19	67.13	77.90	**67.21**	**77.92**
Chinese	86.46	90.18	**87.71**	**91.39**	86.98	90.76
Czech	77.24	83.40	**78.88**	**84.72**	78.12	83.78
Danish	84.91	89.80	**85.17**	**90.10**	84.25	88.92
German	86.18	88.60	**86.96**	**88.98**	85.56	88.30
Portug.	86.60	90.20	**86.78**	**90.34**	86.07	89.92
Swedish	83.33	88.83	**83.55**	**89.30**	83.17	88.59
Turkish	63.77	74.35	63.04	73.99	**65.01**	**75.70**

Table 6: Parsing accuracy (LAS and UAS, excluding punctuation) of Nivre's arc-eager parser without modification (NE), with the LEFT-NONPROJ-BUFFER-ARC transition added (NE+LNBA) and with the RIGHT-NONPROJ-BUFFER-ARC transition added (NE+RNBA). Best results for each language are shown in boldface.

adding these transitions to a stack-based transition system does not affect soundness under the same conditions and for the same reasons explained for projective buffer transitions in Section 3.1.

Note that the fact that non-projective buffer transitions are able to create non-projective dependency arcs does not mean that *all* the arcs that they build are non-projective, since an arc on non-contiguous nodes in the stack and buffer may or may not cross other arcs. This means that non-projective buffer transitions serve a dual purpose: not only they increase coverage, but they also can create some "easy" dependency links in advance of standard transitions, just like projective buffer transitions.

Contrary to projective buffer transitions, we do not impose any arc length restrictions on non-projective buffer transitions (either as a hard constraint in the transitions themselves or as a policy in the training oracle), since we would like the increase in coverage to be as large as possible. We wish to allow the parsers to create non-projective arcs in a straightforward way and without compromising efficiency. Therefore, to train the parser with these transitions, we use an oracle that employs them whenever possible, and delegates to the original parser's oracle otherwise.

4.2 Experiments

We evaluate the impact of non-projective buffer transitions on parsing accuracy by using the same baseline parser, datasets and experimental settings as for projective buffer transitions in Section 3.2. As can be seen in Table 6, adding a non-projective buffer transition to the arc-eager parser improves its performance on all eight datasets. The improvements in LAS are statistically significant at the .01 level (Dan Bikel's comparator) for Chinese, Czech and Turkish. Note that the Chinese treebank is fully projective, this means that non-projective buffer transitions are also beneficial when creating projective arcs.

While with projective buffer transitions we observed that each of them was beneficial for about half of the treebanks, and we related this to the amount of leftward and rightward links of length 1 in each; in the case of non-projective buffer transitions we do not observe this tendency. In this case, LEFT-NONPROJ-BUFFER-ARC works better than RIGHT-NONPROJ-BUFFER-ARC in all datasets except for Turkish and Arabic.

As with the projective transitions, we gathered data about the individual precision of each of the transitions. The results were similar to those for the projective transitions, and show that adding a non-projective buffer transition improves the precision of the standard transitions. We also experimentally checked that adding both non-projective buffer transitions at the same time (NE+LNBA+RNBA) achieved worse performance than adding only the most suitable transition for each dataset.

Table 7 compares the performance of the arc-eager parser with the best non-projective buffer transition for each dataset with the results obtained by

ticular subset of non-projective structures captured by each such parser.

	NE+LNBA/RNBA		NE+PP (CoNLL X)	
Language	LAS	UAS	LAS	UAS
Arabic	**67.21**	**77.92**	66.71	77.52
Chinese	**87.71**	**91.39**	86.92	90.54
Czech	**78.88**	84.72	78.42	**84.80**
Danish	**85.09**	**89.98**	84.77	89.80
German	**86.96**	**88.98**	85.82	88.76
Portug.	86.78	90.34	**87.60**	**91.22**
Swedish	83.55	89.30	**84.58**	**89.50**
Turkish	65.01	75.70	**65.68**	**75.82**

Table 7: Comparison of the parsing accuracy (LAS and UAS, excluding punctuation) of Nivre's arc-eager parser with non-projective buffer transitions (NE+LNBA/RNBA) and the parser with the pseudo-projective transformation (Nivre et al., 2006).

System	PP	PR	NP	NR
NE	80.40	80.76	-	-
NE+LNBA/RNBA	**80.96**	**81.33**	**58.87**	15.66
NE+PP (CoNLL-X)	80.71	81.00	50.72	**29.57**

Table 8: Comparison of the precision and recall for projective (PP, PR) and non-projective (NP, NR) arcs, averaged over all datasets, obtained by Nivre's arc-eager parser with and without non-projective buffer transitions (NE+LNBA/RNBA, NE) and the parser with the pseudo-projective transformation (Nivre et al., 2006).

the parser with the pseudo-projective transformation by Nivre et al. (2006) in the CoNLL-X shared task. Note that, like the one in Table 4, this should not be interpreted as a homogeneous comparison. We can see that the algorithm with non-projective buffer transitions obtains better LAS in five out of the eight treebanks. Precision and recall data on projective and non-projective arcs (Table 8) show that, while our parser does not capture as many non-projective arcs as the pseudo-projective transformation (unsurprisingly, as it can only build non-projective arcs in one direction: that of the particular non-projective buffer transition used for each dataset); it does so with greater precision and is more accurate than that algorithm in projective arcs.

Like projective buffer transitions, non-projective transitions do not increase the computational complexity of stack-based parsers. The observed training and parsing times for the arc-eager parser with non-projective buffer transitions showed a small overhead with respect to the original arc-eager (7.1% average increase in training time, 17.0% in parsing time). For comparison, running the arc-eager parser with the pseudo-projective transformation (Nivre et al., 2006) on the same machine produced a 23.5% increase in training time and a 87.5% increase in parsing time.

5 Related work

The approach of adding an extra transition to a parser to improve its accuracy has been applied in the past by Choi and Palmer (2011). In that paper, the LEFT-ARC transition from Nivre's arc-eager transition system is added to a list-based parser. However, the goal of that transition is different from ours (selecting between projective and non-projective parsing, rather than building some arcs in advance) and the approach is specific to one algorithm while ours is generic – for example, the LEFT-ARC transition cannot be added to the arc-standard and arc-eager parsers, or to extensions of those like the ones by Attardi (2006) or Nivre (2009), because these already have it.

The idea of creating dependency arcs of length 1 in advance to help the classifier has been used by Cheng et al. (2006). However, their system creates such arcs in a separate preprocessing step rather than dynamically by adding a transition to the parser, and our approach obtains better LAS and UAS results on all the tested datasets.

The projective buffer transitions presented here bear some resemblance to the easy-first parser by Goldberg and Elhadad (2010), which allows creation of dependency arcs between any pair of contiguous nodes and is based on the idea of "easy" dependency links being created first. However, while the easy-first parser is an entirely new $O(n \log(n))$ algorithm, our approach is a generic extension for stack-based parsers that does not increase their complexity (so, for example, applying it to the arc-eager system as in the experiments in this paper yields $O(n)$ complexity).

Non-projective transitions that create dependency arcs between non-contiguous nodes have been used in the transition-based parser by Attardi (2006). However, the transitions in that parser do not use the second buffer node, since they are not intended

to create some arcs in advance. The non-projective buffer transitions presented in this paper can also be added to Attardi's parser.

6 Discussion

We have presented a set of two transitions, called *projective buffer transitions*, and showed that adding one of them to Nivre's arc-eager parser improves its accuracy in seven out of eight tested datasets from the CoNLL-X shared task. Furthermore, adding one of a set of *non-projective buffer transitions* achieves accuracy improvements in all of the eight datasets. The obtained improvements are statistically significant for several of the treebanks, and the parser with projective buffer transitions surpassed the best published single-parser LAS results on two of them. This comes at no cost either on computational complexity or (in the case of projective transitions) on empirical training and parsing times with respect to the original parser.

While we have chosen Nivre's well-known arc-eager parser as our baseline, we have shown that these transitions can be added to any stack-based dependency parser, and we are not aware of any specific property of arc-eager that would make them work better in practice on this parser than on others. Therefore, future work will include an evaluation of the impact of buffer transitions on more transition-based parsers. Other research directions involve investigating the set of non-projective arcs allowed by non-projective buffer transitions, defining different variants of buffer transitions (such as non-projective buffer transitions that work with nodes located deeper in the buffer) or using projective and non-projective buffer transitions at the same time.

Acknowledgments

This research has been partially funded by the Spanish Ministry of Economy and Competitiveness and FEDER (projects TIN2010-18552-C03-01 and TIN2010-18552-C03-02), Ministry of Education (FPU Grant Program) and Xunta de Galicia (Rede Galega de Recursos Lingüísticos para unha Sociedade do Coñecemento).

References

Susana Afonso, Eckhard Bick, Renato Haber, and Diana Santos. 2002. "Floresta sintá(c)tica": a treebank for Portuguese. In *Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC 2002)*, pages 1968–1703, Paris, France. ELRA.

Nart B. Atalay, Kemal Oflazer, and Bilge Say. 2003. The annotation process in the Turkish treebank. In *Proceedings of EACL Workshop on Linguistically Interpreted Corpora (LINC-03)*, pages 243–246, Morristown, NJ, USA. Association for Computational Linguistics.

Giuseppe Attardi. 2006. Experiments with a multilanguage non-projective dependency parser. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pages 166–170.

Sabine Brants, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius, and George Smith. 2002. The tiger treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories, September 20-21*, Sozopol, Bulgaria.

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pages 149–164.

Chih-Chung Chang and Chih-Jen Lin, 2001. *LIBSVM: A Library for Support Vector Machines*. Software available at http://www.csie.ntu.edu.tw/∼cjlin/libsvm.

K. Chen, C. Luo, M. Chang, F. Chen, C. Chen, C. Huang, and Z. Gao. 2003. Sinica treebank: Design criteria, representational issues and implementation. In Anne Abeillé, editor, *Treebanks: Building and Using Parsed Corpora*, chapter 13, pages 231–248. Kluwer.

Yuchang Cheng, Masayuki Asahara, and Yuji Matsumoto. 2006. Multi-lingual dependency parsing at NAIST. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, CoNLL-X '06, pages 191–195, Stroudsburg, PA, USA. Association for Computational Linguistics.

Jinho D. Choi and Martha Palmer. 2011. Getting the most out of transition-based dependency parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers - Volume 2*, HLT '11, pages 687–692, Stroudsburg, PA, USA. Association for Computational Linguistics.

Jason M. Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, pages 340–345.

R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. 2008. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874.

Yoav Goldberg and Michael Elhadad. 2010. An efficient algorithm for easy-first non-directional dependency parsing. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL HLT)*, pages 742–750.

Carlos Gómez-Rodríguez and Joakim Nivre. 2010. A transition-based parser for 2-planar dependency structures. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 1492–1501, Stroudsburg, PA, USA. Association for Computational Linguistics.

Jan Hajič, Otakar Smrž, Petr Zemánek, Jan Šnaidauf, and Emanuel Beška. 2004. Prague Arabic Dependency Treebank: Development in data and tools. In *Proceedings of the NEMLAR International Conference on Arabic Language Resources and Tools*.

Jan Hajič, Jarmila Panevová, Eva Hajičová, Jarmila Panevová, Petr Sgall, Petr Pajas, Jan Štěpánek, Jiří Havelka, and Marie Mikulová. 2006. Prague Dependency Treebank 2.0. CDROM CAT: LDC2006T01, ISBN 1-58563-370-4. Linguistic Data Consortium.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 1077–1086, Stroudsburg, PA, USA. Association for Computational Linguistics.

Matthias T. Kromann. 2003. The Danish dependency treebank and the underlying linguistic theory. In *Proceedings of the 2nd Workshop on Treebanks and Linguistic Theories (TLT)*, pages 217–220, Växjö, Sweden. Växjö University Press.

Andre Martins, Noah Smith, and Eric Xing. 2009. Concise integer linear programming formulations for dependency parsing. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP (ACL-IJCNLP)*, pages 342–350.

Ryan McDonald and Joakim Nivre. 2007. Characterizing the errors of data-driven dependency parsing models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 122–131.

Ryan McDonald and Joakim Nivre. 2011. Analyzing and integrating dependency parsers. *Comput. Linguist.*, 37:197–230.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the Human Language Technology Conference and the Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP)*, pages 523–530.

Ryan McDonald, Kevin Lerman, and Fernando Pereira. 2006. Multilingual dependency analysis with a two-stage discriminative parser. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pages 216–220.

Jens Nilsson, Johan Hall, and Joakim Nivre. 2005. MAMBA meets TIGER: Reconstructing a Swedish treebank from Antiquity. In Peter Juel Henrichsen, editor, *Proceedings of the NODALIDA Special Session on Treebanks*.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2004. Memory-based dependency parsing. In *Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL-2004)*, pages 49–56, Morristown, NJ, USA. Association for Computational Linguistics.

Joakim Nivre, Johan Hall, Jens Nilsson, Gülsen Eryiğit, and Svetoslav Marinov. 2006. Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pages 221–225.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, pages 149–160. ACL/SIGPARSE.

Joakim Nivre. 2008. Algorithms for Deterministic Incremental Dependency Parsing. *Computational Linguistics*, 34(4):513–553.

Joakim Nivre. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP (ACL-IJCNLP)*, pages 351–359.

Kemal Oflazer, Bilge Say, Dilek Zeynep Hakkani-Tür, and Gökhan Tür. 2003. Building a Turkish treebank. In Anne Abeillé, editor, *Treebanks: Building and Using Parsed Corpora*, pages 261–277. Kluwer.

Kenji Sagae and Jun'ichi Tsujii. 2008. Shift-reduce dependency DAG parsing. In *Proceedings of the 22nd International Conference on Computational Linguistics (COLING)*, pages 753–760.

Stephen Tratz and Eduard Hovy. 2011. A fast, accurate, non-projective, semantically-enriched parser. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1257–1268, Edinburgh, Scotland, UK., July. Association for Computational Linguistics.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines.

In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 195–206.

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 562–571.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers - Volume 2*, HLT '11, pages 188–193, Stroudsburg, PA, USA. Association for Computational Linguistics.

Computing and Informatics, Vol. 32, 2013, 1001–1031, V 2013-Dec-27

IMPROVING THE ARC-EAGER MODEL WITH REVERSE PARSING

Daniel Fernández-González

Departamento de Informática
Universidade de Vigo
Campus As Lagoas
32004 Ourense, Spain
e-mail: danifg@uvigo.es

Carlos Gómez-Rodríguez

Departamento de Computación
Facultad de Informática
Universidade da Coruña
Campus Elviña
s/n, 15071 A Coruña, Spain
e-mail: cgomezr@udc.es

David Vilares

Departamento de Computación
Facultad de Informática
Universidade da Coruña
Campus Elviña
s/n, 15071 A Coruña, Spain
e-mail: david.vilares@udc.es

Abstract. A known way to improve the accuracy of dependency parsers is to combine several different parsing algorithms, in such a way that the weaknesses of each of the models can be compensated by the strengths of others. For example, voting-based combination schemes are based on variants of the idea of analyzing

each sentence with various parsers, and constructing a combined output where the head of each node is determined by "majority vote" among the different parsers. Typically, such approaches combine very different parsing models to take advantage of the variability in the parsing errors they make.

In this paper, we show that consistent improvements in accuracy can be obtained in a much simpler way by combining a single parser with itself. In particular, we start with a greedy implementation of the Nivre pseudo-projective arc-eager algorithm, a well-known left-to-right transition-based parser, and we combine it with a "mirrored" version of the algorithm that analyzes sentences from right to left. To determine which of the two obtained outputs we trust for the head of each node, we use simple criteria based on the length and position of dependency arcs.

Experiments on several datasets from the CoNLL-X shared task and the WSJ section of the English Penn Treebank show that the novel combination system obtains better performance than the baseline arc-eager parser in all cases. To test the generality of the approach, we also perform experiments with a different transition system (arc-standard) and a different search strategy (beam search), obtaining similar improvements in all these settings.

Keywords: automata, computational linguistics, dependency parsing, natural language processing, parsing.

Mathematics Subject Classification 2010: 68T50

1 INTRODUCTION

Nowadays the huge amount of raw textual information that computers must process gives a vital role to tasks such as information extraction, machine translation or question answering in many different fields. All these tasks involve a transformation of unrestricted natural language text into representations that a machine can handle easily. This is, in fact, the main goal of natural language processing (NLP).

One of the most ubiquitous and useful NLP processes is syntactic parsing. This consists of mapping a sentence in natural language into its syntactic representation. Two different syntactic formalisms are popular for this purpose: *constituency* representations [5, 12] or *dependency* representations [43]. Parsing a sentence with constituency representations means decomposing it into constituents or phrases, and in that way a phrase structure tree is created with relationships between words and phrases, as in Figure 1. In contrast, the goal of parsing a sentence with dependency representations is to create a dependency graph consisting of lexical nodes linked by binary relations called dependencies. A dependency relation connects two words, with one of them acting as the *head* and the other one as the *dependent*. A dependency graph can also be called a dependency tree, if each node of the graph has only one head and the structure is acyclic. Figure 2 shows a dependency tree for an English sentence, where each edge is labeled with a dependency type.

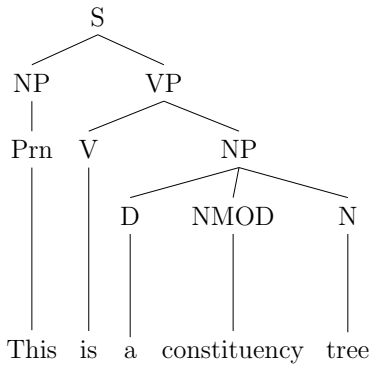


Fig. 1. Constituency tree for an English sentence.

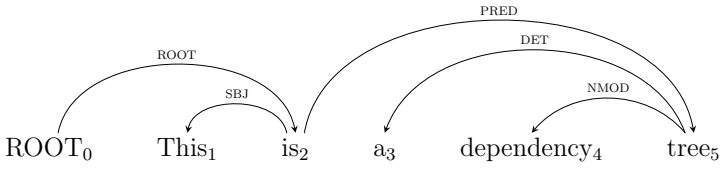


Fig. 2. Dependency tree for an English sentence.

Dependency parsing has recently gained a wide interest in the natural language processing community and has been used for many problems ranging from machine translation [13] to ontology construction [41]. Some of the most accurate and efficient dependency parsers are based on *data-driven* parsing models such as those by Nivre et al [34], McDonald et al [28], Titov and Henderson [44], Martins et al [25], Huang and Sagae [21], Koo and Collins [22], Zhang and Nivre [49], Bohnet and Nivre [7] or Gómez-Rodríguez and Nivre [17]. These dependency parsers can be trained from syntactically annotated text without the need for a formal grammar, and they provide a simple representation of syntax that maps to predicate-argument structure in a straightforward way.

Most data-driven dependency parsers can be classified into two families: *graph-based* and *transition-based* parsers [27]. On the one hand, graph-based parsers [14, 28] learn a model for scoring possible dependency graphs for a given sentence and, then, the parsing process consists of searching for the highest-scoring graph. In transition-based parsing [45, 34], a model is learned for scoring transitions from one parser state to the next, and the parsing process consists of finding a high-scoring sequence of transitions that will traverse a series of states until a complete dependency graph is created. The most commonly used graph-based and transition-based parsers are the *maximum spanning tree* parser by McDonald et al [28] and the *arc-eager* parser with the pseudo-projective algorithm by Nivre and Nilsson [37],

respectively.

It has been proved by McDonald and Nivre [26] that transition-based parsers suffer from *error propagation*: a transition erroneously chosen at an early moment can place the parser in an incorrect state that will in turn lead to more incorrect transitions in the future. Since some transition-based parsers (as the arc-eager parser) analyze the sentence from left to right, the probability of choosing an incorrect transition tends to be higher as we approach the end of a given sentence. As a consequence, the dependency tree obtained by a transition-based parser typically presents more (propagating) errors in the rightmost arcs of the graph than in the leftmost arcs.

With the goal of reducing the effect of error propagation on the rightmost arcs of the graph, Nilsson [29] proposes the application of a reverse parsing strategy on the arc-eager parser by Nivre [31]. This proceeds by transforming the left-to-right arc-eager parser into a right-to-left variant. That way, the reverse parser analyses a sentence from the end to the beginning, likely making more errors in the leftmost arcs of the dependency tree than in the rightmost arcs in relation to the standard parser.

Nilsson [29] proved experimentally that analyzing a sentence in reverse order does not improve the global accuracy of the arc-eager parser. However, the reverse arc-eager parser is able to build correctly some arcs of the dependency tree that the original arc-eager version creates erroneously. Concretely, we found out that, in addition to having a better performance in rightmost arcs, the right-to-left arc-eager version is able to achieve higher accuracy in arcs with certain lengths. To take advantage of that, we present an efficient combination system that obtains a new dependency tree by joining the dependency trees created by each parser. This system uses two different strategies to accomplish the combination: a strategy based on the position of the arcs and another based on the length of the arcs.

Our combination system presents several advantages in comparison to other strategies such as voting or stacking,[1] where a complex combination process must be done involving several parsers with different natures. The simplicity of our approach allows the pseudo-projective arc-eager parser by Nivre and Nilsson [37] to improve its own accuracy without increasing its execution time and by using exclusively one parser that analyses a sentence in parallel in both directions at the same time.

We test the accuracy of the combinative approach on eight datasets from CoNLL-X shared task [9] and on the English WSJ corpus from the Penn Treebank [24]. In these experiments, the combination of the arc-eager parser and its reverse variant outperforms the accuracy of both parsers in the nine languages tested, and even improves over the scores of the maximum spanning tree parser by McDonald et

[1] Strictly speaking, our approach can be seen as a degenerate instance of weighted voting, where there are only two systems and therefore the weighting scheme reduces to a boolean criterion to choose among them at each node. In this article, we call "voting systems" those that use more than two systems, thus requiring more complex schemes involving majority voting or numeric weights.

al [28]. Moreover, the combinative approach is not only beneficial for the pseudo-projective arc-eager parser with greedy search, but also for other transition-based models like the pseudo-projective arc-standard parser [32] and other search strategies like the beam search used by the ZPar system [48], as we show experimentally in Section 6.

The rest of this article is organized as follows: in Section 2, we discuss other research work that deals with parser combination. Section 3 introduces some notation and concepts about transition-based dependency parsing. Section 4 describes the pseudo-projective arc-eager parser [37] and its reverse variant. In Section 5, we discuss different strategies to implement the parser combination system. Section 6 presents an empirical study of the performance of the reverse arc-eager parser and the combinative approaches, as well as the effect of this novel technique on the arc-standard parser and a beam-search-based parser. It also shows an analysis that explains why the combination system improves over the individual scores. Finally, Section 7 contains a final discussion.

2 RELATED WORK

Some dependency parser combination approaches have been applied successfully in the literature. One of the most influential is the approach by Sagae and Lavie [39]. Following on the work by Zeman and Žabokrtský [46], they present a framework for combining the output of different parsers by applying a *voting* system. This approach consists of letting several parsers assign votes to the dependency links that they consider that should belong to the final dependency graph. In that way, a weighted graph is created. Afterwards, a quadratic maximum spanning tree algorithm must be applied to find the final output. This combination system has the drawback of increasing the time complexity of the parsing process significantly. As the maximum spanning tree algorithm must be used, a combination of different linear parsers results into a quadratic system. On the contrary, our approach does not increase the time complexity of the combined parsers.

The combination method described by Sagae and Lavie [39] was used in other works to combine several transition-based parsers. Concretely, Hall et al [20] and Nilsson [29] use this voting system to combine six transition-based parsers, where two of them are the arc-eager parser by Nivre [31] and its reverse version. Only Nilsson [29] presents an individual evaluation of the reverse arc-eager parsing accuracy. In his results, he shows that the reverse arc-eager version performs worse than the standard version on ten datasets of the CoNLL 2007 shared task. However, the author confirms that the combined parsers were not properly optimized. This does not happen in our research, where both models were conveniently tuned and, as a consequence, the reverse arc-eager parser proves more accurate than the conventional left-to-right version on the Czech dataset.

Another example of using the voting combination by Sagae and Lavie [39] is the work by Samuelsson et al [40]. In this research, two more transition-based parsers

are added to those combined in Hall et al [20] and Nilsson [29]. In addition to this, the authors join the eight combined parsers with a semantic parser in order to achieve a better accuracy. As the other approaches based in voting combination, this one tries to combine different parsers with different time complexities. The resulting system's complexity is the maximum among those of the combined systems, which is quadratic. In our combination system only one single algorithm is used (the arc-eager parser by Nivre and Nilsson [37]) and its time complexity remains linear.

A different combinative approach is the one undertaken by Nivre and McDonald [36]. They implement a feature-based integration which tries to combine a graph-based parser with a transition-based model only during learning time: one parser helps the other to create the trained model. This method receives the name of *stacking* combination. The main drawback is that the quadratic time complexity of the graph-based parser increases the overall time complexity. To prevent that, Attardi and Dell'Orletta [3] propose a stacking combination of one linear transition-based parser with its own reverse version. In that way, the linear time complexity is maintained. However, this approach has the drawback that the right-to-left parser cannot be applied until the left-to-right parser ends, whilst our combination system allows both parsers to run in parallel, reducing the execution time. This means that, in multicore machines, our approach takes practically the same time to parse a sentence as the single arc-eager parser does, and additionally, it achieves an improvement in accuracy.

An evaluation and comparison between the voting and stacking combination approaches, as well as further information about these approaches, can be found in Fishel and Nivre [16] and Surdeanu and Manning [42].

Finally, Zhang and Clark [47] propose a beam-search parser that combines both graph-based and transition-based parsing into a single system that uses a transition-based decoder with a scoring model using graph-based information. This approach, which has also been used in other recent works [6, 7], is different from stacking: instead of using two separately trained models, it combines the graph-based and transition-based approaches into a single model. In spite of the fact that the resulting system is also linear, the approach developed by Zhang and Clark is not as fast as the greedy arc-eager algorithm that we use in this paper.

3 TRANSITION-BASED DEPENDENCY PARSING

In this section, we introduce some definitions and notation concerning transition-based dependency parsing that will be used throughout the article.

3.1 Dependency parsing

A *dependency graph* is a labeled directed graph that represents the syntactic structure of a given sentence. More formally, it can be defined as follows:

Definition 1. Let $w = w_1 \dots w_n$ be an input string. A *dependency graph* for

$w_1 \dots w_n$ is a labelled directed graph $G = (V_w, E)$, where $V_w = \{0, \dots, n\}$ is the set of nodes, and $E \subseteq V_w \times L \times V_w$ is the set of labelled directed arcs.

The set V_w is the set of *nodes*. This means that every token index i of the sentence is a node ($1 \leq i \leq n$) and that there is a special node 0, which does not correspond to any token of the sentence and which will always be a root of the dependency graph (normally the only root).

Each arc in E encodes a dependency relation between two tokens. We call an edge (w_i, l, w_j) in a dependency graph G a *dependency link* from w_i to w_j with label l, represented as $w_i \xrightarrow{l} w_j$. We say that w_i is the *head* of w_j and, conversely, that w_j is a *dependent* of w_i. The labels on dependency links are typically used to represent their associated syntactic functions, such as SBJ for subject in the dependency link $is_2 \rightarrow This_1$ in Figure 2.

For convenience, we write $w_i \rightarrow w_j \in E$ if the link (w_i,w_j) exists (regardless of its label) and $w_i \rightarrow^* w_j \in E$ if there is a (possibly empty) directed path from w_i to w_j.

Most dependency-based syntactic formalisms do not allow arbitrary dependency graphs as syntactic representations. Instead, they are typically restricted to acyclic graphs where each node has at most one head. Such dependency graphs are called *dependency forests*.

Definition 2. A dependency graph G is said to be a *dependency forest* if it satisfies the following:

1. *Single-head constraint*: if $w_i \rightarrow w_j$, then there is no $w_k \neq w_i$ such that $w_k \rightarrow w_j$.
2. *Acyclicity constraint*: if $w_i \rightarrow^\star w_j$, then there is no arc $w_j \rightarrow w_i$.

Nodes that have no head in a dependency forest are called *roots*. Apart from the previous two constraints, some dependency formalisms add the additional constraint that a dependency forest can have only one root (or, equivalently, that it must be connected). A forest of this form is called a *dependency tree*.

The system in charge of parsing a given sentence producing a dependency graph is called a *dependency parser*. In this article, we will work with dependency parsers that output dependency trees. These parsers enforce the single-head and acyclicity constraints, and they link all of their root nodes as dependents of a dummy root node 0.

For reasons of computational efficiency, many dependency parsers are restricted to work with *projective* dependency structures, that is, dependency trees in which the projection of each node corresponds to a contiguous substring of the input:

Definition 3. An arc $w_i \rightarrow w_k$ is *projective* iff, for every word w_j occurring between w_i and w_k in the sentence (w_i<w_j<w_k or w_i>w_j>w_k), $w_i \rightarrow^\star w_j$.

Definition 4. A dependency graph $G = (V_w, E)$ is projective iff every arc in E is projective.

Projective dependency trees are not sufficient to represent all the linguistic phenomena observed in natural languages, but they have the advantage of being efficiently parsable. Even so, non-projective dependency structures present in natural languages represent, in many languages, a rather reduced portion of the total.

3.2 Transition systems

In this article, we work with transition-based dependency parsers that are defined following the framework of Nivre [33]. According to this, a deterministic dependency parser is defined by a non-deterministic *transition system*, specifying a set of elementary operations that can be executed during the parsing process, and an *oracle* that deterministically selects a single transition at each choice point of the parsing process. More formally, they are defined as follows:

Definition 5. A *transition system* for dependency parsing is a tuple $S = (C, T, c_s, C_t)$, where

1. C is a set of possible parser *configurations*,
2. T is a finite set of *transitions*, which are partial functions $t : C \rightarrow C$,
3. c_s is a total initialization function that maps each input string w to a unique *initial configuration* $c_s(w)$, and
4. $C_t \subseteq C$ is a set of *terminal configurations*.

Definition 6. An *oracle* for a transition system is a function $o : C \rightarrow T$.

Although the specific nature of configurations varies among parsers, they are required to contain at least a set A of dependency arcs and a buffer B of unread words, which initially holds all the words in the input sentence. A transition-based parser will be able to read input words by popping them from the buffer, and to create dependency arcs by adding them to the set A.

An input sentence w can be parsed using a transition system $S = (C, T, c_s, C_t)$ and an oracle o by starting in the initial configuration $c_s(w)$, calling the oracle function on the current configuration c, and updating the configuration by applying the transition $o(c)$ returned by the oracle. This process is repeated until a terminal configuration is reached, and the dependency analysis of the sentence is defined by the terminal configuration. Each sequence of configurations that the parser can traverse from an initial configuration to a terminal configuration for some input w is called a *transition sequence*.

Note that, apart from a correct transition system, a practical parser needs a good oracle to achieve the desired results, since a transition system only specifies how to reach all the possible dependency graphs that could be associated to a sentence, but not how to select the correct one. Oracles for practical parsers can be obtained by training classifiers on treebank data [34].

4 REVERSING THE ARC-EAGER PARSER

4.1 Arc-eager Parser

In this article, we use as our main baseline the well-known parser called pseudo-projective arc-eager by Nivre and Nilsson [37]. This is the result of adding a pseudo-projective transformation to the arc-eager parser by Nivre [31]. As a transition-based parser, the basic arc-eager parser is defined by a transition system $S = (C, T, c_s, C_t)$ such that:

- C is the set of all configurations of the form $c = \langle \sigma, B, A \rangle$, where σ and B are disjoint lists of nodes from V_w (for some input w), and A is a set of dependency arcs over V_w. The list B, called the *buffer*, is used to hold nodes corresponding to input words that have not yet been read. The list σ, called the *stack*, contains nodes for words that have already been read, but still have dependency links pending to be created. For convenience, we will use the notation $\sigma|i$ to denote a stack with top i and tail σ, and the notation $j|B$ to denote a buffer with top j and tail B. The set A of dependency arcs contains the part of the output parse that the system has constructed at each given point.
- The initial configuration is $c_s(w_1 \dots w_n) = \langle [], [1 \dots n], \emptyset \rangle$, i.e., the buffer initially holds the whole input string while the stack is empty.
- The set of terminal configurations is $C_t = \{\langle \sigma, [], A \rangle \in C\}$, i.e., final configurations are those where the buffer is empty, regardless of the contents of the stack.
- The set T has the following transitions:

SHIFT	$\langle \sigma, i\|B, A \rangle \Rightarrow \langle \sigma\|i, B, A \rangle$
REDUCE	$\langle \sigma\|i, B, A \rangle \Rightarrow \langle \sigma, B, A \rangle$
LEFT-ARC$_l$	$\langle \sigma\|i, j\|B, A \rangle \Rightarrow \langle \sigma, j\|B, A \cup \{j \xrightarrow{l} i\} \rangle$ only if $\nexists k \mid k \to i \in A$ (single-head)
RIGHT-ARC$_l$	$\langle \sigma\|i, j\|B, A \rangle \Rightarrow \langle \sigma\|i\|j, B, A \cup \{i \xrightarrow{l} j\} \rangle$ only if $\nexists k \mid k \to j \in A$ (single-head)

The SHIFT transition is used to read words from the input string, by moving the next node in the buffer to the top of the stack. The LEFT-ARC transition creates a leftward dependency arc from the first node on the buffer to the topmost node on the stack and pops the stack. The RIGHT-ARC transition builds a rightward dependency arc from the topmost node on the stack to the first node on the buffer and pushes the first node on the buffer onto the stack. Finally, the REDUCE transition is used to pop the topmost node from the stack when we have finished building arcs to or from it.

Figure 3 shows a transition sequence in the arc-eager transition system which derives the labelled dependency graph in Figure 2.

Note that the arc-eager parser is a linear-time parser, since each word in the input can be shifted and reduced at most once, and the number of arcs that can be built by LEFT-ARC and RIGHT-ARC transitions is strictly bounded by the number of words by the single-head constraint. Besides, the arc-eager algorithm by Nivre [31] is not able to parse non-projective syntactic structures. In order to solve that, the arc-eager parser by Nivre and Nilsson [37] implements a pseudo-projective transformation, which projectivizes the non-projective structures so that the arc-eager parser can handle them.

Transition	**Stack (σ)**	**Buffer (B)**	**Added Arc**
	[ROOT$_0$]	[This$_1$,...,tree$_5$]	
SHIFT	[ROOT$_0$,This$_1$]	[is$_2$,..., tree$_5$]	
LA$_{SBJ}$	[ROOT$_0$]	[is$_2$,...,tree$_5$]	(2,SBJ,1)
RA$_{ROOT}$	[ROOT$_0$, is$_2$]	[a$_3$,...,tree$_5$]	(0,ROOT,2)
SHIFT	[ROOT$_0$,is$_2$,a$_3$]	[dependency$_4$,tree$_5$]	
SHIFT	[ROOT$_0$,is$_2$,a$_3$,dependency$_4$]	[tree$_5$]	
LA$_{NMOD}$	[ROOT$_0$,is$_2$,a$_3$]	[tree$_5$]	(5,NMOD,4)
LA$_{DET}$	[ROOT$_0$,is$_2$]	[tree$_5$]	(5,DET,3)
RA$_{PRED}$	[ROOT$_0$,is$_2$,tree$_5$]	[ ]	(2,PRED,5)
REDUCE	[ROOT$_0$,is$_2$]	[ ]	
REDUCE	[ROOT$_0$]	[ ]	

Fig. 3. Transition sequence for parsing the sentence in Figure 2 using the arc-eager parser (LA=LEFT-ARC, RA=RIGHT-ARC).

4.2 Reverse Arc-eager Parser

In order to reduce the amount of errors produced in the rightmost side of the dependency tree, we apply a reverse strategy on the arc-eager parser like that of Nilsson [29]. The *reverse arc-eager parser* is a right-to-left dependency parser that analyses a sentence in reverse order. The main advantage of this approach is that it improves the accuracy of arcs located in the rightmost side of the dependency tree, as well as those with certain lengths.

The reverse arc-eager variant is defined with the same transition system as the original arc-eager parser with the difference that, in the initial configuration, the sentence is put in reverse order. Concretely, the initial configuration $c_s(w_1 \dots w_n) = \langle [], [1 \dots n], \emptyset \rangle$ in the arc-eager transition system is changed into an initial configuration where the sentence is inverted in the buffer: $c_s(w_n \dots w_1) = \langle [], [1 \dots n], \emptyset \rangle$.

Figure 4 describes the transition sequence followed by the reverse arc-eager parser to analyze the sentence in Figure 2. The result is the dependency tree in

Figure 5. Note that the dependency graph obtained is the reverse of the one which appears in Figure 2, except for the dummy root arc, which is not affected by our reversing process. Therefore, the results in this paper are not influenced by the effect of placing the dummy root at the end of the sentence, recently studied by Ballesteros and Nivre [4].

Transition	**Stack (σ)**	**Buffer (B)**	**Added Arc**
	[ROOT$_0$]	[tree$_1$,...,This$_5$]	
SHIFT	[ROOT$_0$,tree$_1$]	[dependency$_2$,...,This$_5$]	
RA$_{NMOD}$	[ROOT$_0$,tree$_1$,dependency$_2$]	[a$_3$,...,This$_5$]	(1,NMOD,2)
REDUCE	[ROOT$_0$,tree$_1$]	[a$_3$,...,This$_5$]	
RA$_{DET}$	[ROOT$_0$,tree$_1$,a$_3$]	[is$_4$,This$_5$]	(1,DET,3)
REDUCE	[ROOT$_0$,tree$_1$]	[is$_4$,This$_5$]	
LA$_{PRED}$	[ROOT$_0$]	[is$_4$,This$_5$]	(4,PRED,1)
RA$_{ROOT}$	[ROOT$_0$, is$_4$]	[This$_5$]	(0,ROOT,4)
RA$_{SBJ}$	[ROOT$_0$,is$_4$,This$_5$]	[ ]	(4,SBJ,5)
REDUCE	[ROOT$_0$,is$_4$]	[ ]	
REDUCE	[ROOT$_0$]	[ ]	

Fig. 4. Transition sequence for parsing the sentence in Figure 2 using the reverse arc-eager parser (LA=LEFT-ARC, RA=RIGHT-ARC).

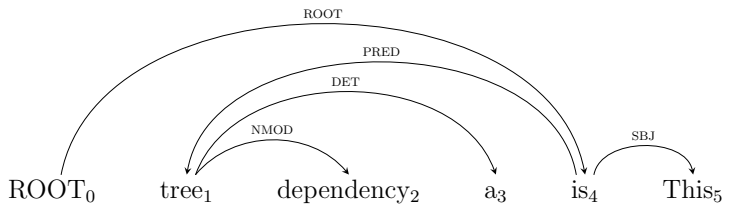


Fig. 5. Dependency tree obtained by applying the reverse arc-eager parser on the English sentence in Figure 2.

5 COMBINING THE ARC-EAGER AND THE REVERSE ARC-EAGER PARSERS

While the arc-eager parser makes more mistakes in the rightmost side of the graph due to error propagation, the reverse version achieves better precision on the arcs created on the rightmost part of the tree. Furthermore, we have observed that some arcs with certain lengths tend to be correctly built more often by the reverse parser than by the arc-eager parser. Therefore, a combination of parsers is a logical next step.

5.1 Parser Combination System

In this section, we introduce a combination system that takes advantage of the strengths of each parser and discards their weaknesses. Concretely, the developed combination system is applied on the dependency trees obtained by the parsers after the parsing process. This system builds a new dependency tree by selecting the arcs from one dependency tree or the other according to a certain strategy. The combination strategies that we use are:

- **Position-based strategy:** This strategy uses the position of dependents in the sentence to distinguish which arcs are selected from the first parser and which from the second parser. This approach is based on the idea that each parser is good at certain positions of the graph. For instance, if the first parser is good at doing the leftmost arcs, whose dependents are located from position 1 to 4, and the second parser obtains a higher accuracy on the rest of the arcs, then the combination system will trust the first parser to build the dependency graph until position 4, and use the dependency tree of the second parser to complete the output. In this case, we will say that the combination is done with a *reliability parameter* p=4;
- **Length-based strategy:** This combination technique selects the arcs built by the first parser or the second parser depending on their length. There are some parsers that create arcs with a certain length more accurately. For example, if the first parser builds long arcs with a higher accuracy than the second parser, then the combination of both parsers will trust the first parser to build long arcs and will use the arcs created by the second parser to complete the rest. In that case, if we assume that long arcs are those with a length higher than 15, we will say that this combination has a *reliability parameter* l=15.[2]

In both strategies, firstly it is necessary to identify the order of the parser combination: which is the first parser and which is the second parser. This is because the result is not the same if we combine the arc-eager parser plus the reverse variant as if we use the configuration with reverse arc-eager plus the original version. Secondly, the reliability parameters must be selected. In the example described above, only one parameter divides the set of arcs by assigning a certain kind of arcs to a parser. In the case of the position-based strategy, the parameter p divides the sentence into two parts: the first part is done by the first parser and the other by the second parser. On the other hand, the parameter l of the length-based technique divides the set of arcs in such a way that those with length lower than l are created by the first parser and those with higher length by the second parser.

[2] Note that, when each parser assigns a different head to the same node, the length of the arcs created by each parser on that node may be different. Therefore, during the combination process, we trust the length of the arcs produced by the first parser to decide whether the arcs are longer than the reliability parameter l or not.

Note that a huge amount of arcs of both parsers can coincide, but it is in a small set of arcs where parsers differ. In the same way that parsers differ in the arcs created, they can build the same arcs but assign a different label to the same dependency link. When that happens, the combination system applies the strategies described above to decide which parser we should trust to choose the correct label. For instance, suppose that both parsers create the same arc but they assign a different label to each arc and suppose that the dependent of these arcs is situated before p; then if we apply a position-based strategy with a parameter p, the label used in the new output arc is the label of the arc created by the first parser and not the one assigned by the second parser.

The implementation of the position-based and the length-based strategies is described in Figure 6.

Note that this combination process can produce dependency graphs with cycles, which we do not consider desirable because we wish to obtain dependency forests, which must satisfy the acyclicity constraint. As we will see in Section 6.3, the presence of cycles using our approach is significantly low. However, in case that a cycle is present in the final dependency tree of a given sentence, the combination process is undone for this sentence and the output obtained by the original arc-eager parser is chosen as the final dependency tree. This is because, in general, the arc-eager parser obtains higher scores than the reverse version.

5.2 Example

Using the combination system defined in Section 5.1, we combine the arc-eager and the reverse arc-eager parsers. Concretely, we detail an artificial example in Figure 7 where the position-based strategy is used to undertake the combination.

Firstly, in Figure 7a we present the dependency tree returned by the reverse arc-eager parser after analyzing a sentence. Since the reverse parser outputs a dependency tree with nodes in reverse order, we have to invert them in order to continue with the combination process. The dependency graph obtained after applying an inverter process is shown in Figure 7b. Note that the analysis made by the reverse parser presents two incorrect arcs: the two leftmost arcs $ROOT_0 \rightarrow This_1$ and $This_1 \rightarrow is_2$. Secondly, Figure 7c presents the dependency tree obtained by the original arc-eager parser. In this tree, there are also two mistakes: the incorrect rightmost arc $dependency_4 \rightarrow a_3$ and the incorrect label DET in arc $tree_5 \rightarrow dependency_4$. Notice that this example tries to remark that the arc-eager parser has less accuracy in rightmost arcs, whilst the reverse variant is worse at creating leftmost arcs. Finally, the Figure 7d shows the resulting dependency graph after combining the dependency trees in Figure 7b and Figure 7c. Concretely, the combination system uses the position-based strategy with a reliability parameter p=2 and the combination order is arc-eager+reverse. This means that we trust the arc-eager dependency tree (Figure 7c) to assign head nodes to words located before and at position 2 ($ROOT_0 \rightarrow is_2$, $is_2 \rightarrow This_1$ and $is_2 \rightarrow tree_5$), and we complete the new graph with arcs $tree_5 \rightarrow dependency_4$ and $tree_5 \rightarrow a_3$ provided by the reverse dependency tree

```
Combination_method(dep_tree_1, dep_tree_2):combined_dep_tree
begin
 for all dependency_tree_nodes
 do
  if head_node_1 =/= head_node_2
  then
     if strategy(head_node_1,dependency_tree_node) <= parameter
     then
      create_arc(head_node_1,dependency_tree_node,label_node_1)
     else
      create_arc(head_node_2,dependency_tree_node label_node_2)
  else
    if label_1 =/= label_2
    then
      if strategy(head_node_1,dependency_tree_node) <= parameter
      then
       create_arc(head_node_2,dependency_tree_node,label_node_1)
      else
       create_arc(head_node_2,dependency_tree_node,label_node_2)
    else
      create_arc(head_node_2,dependency_tree_node,label_node_2)
 done
 find_and_process_cycles(combine_dep_tree)
end
```

Fig. 6. Generic algorithm that combines two dependency trees for a given sentence (`dep_tree_1`, `dep_tree_2`) and builds a new output (`combined_dep_tree`); where the method **strategy** returns either the position of the dependent or the length of the arc defined by the nodes `head_node_1` and `dependency_tree_node` depending on the strategy used (`position-based` or `length-based`, respectively), *parameter* is either p or l depending on the strategy followed, `dependency_tree_nodes` is the set of nodes of the input dependency trees (note that, since the sentence analyzed is the same, the dependency trees 1 and 2 have the same nodes), `head_node_X` and `label_X` determines the head node and the label assigned by a parser `X` (1 or 2) to the current node of the dependency tree (`dependency_tree_node`) to create an arc, and the function **create_arc()** builds an arc in the output dependency tree with a certain head and label. Finally, the method **find_and_process_cycles()** is in charge of detecting the arcs involved in a cycle on the resulting combination output and solving them by trusting only the original arc-eager parser on that sentence.

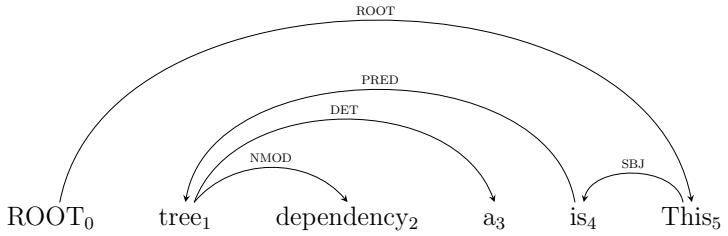
(Figure 7b). Since we rely on the reverse parser to build the $tree_5 \rightarrow dependency_4$, the correct label of this arc is taken from the reverse arc-eager dependency tree. As we can see, the output in Figure 7d solves all the mistakes made by both parsers.

6 EXPERIMENTS

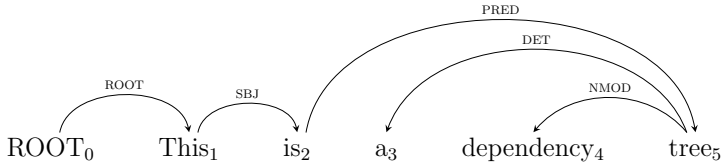
In this section, we evaluate the performance of the reverse arc-eager parser and the parsers obtained by combining the pseudo-projective arc-eager parser by Nivre and Nilsson [37] , implemented in MaltParser [35], and its reverse version; using each of the combination strategies described in Section 5.

In addition we also provide a comparison between our approaches and two

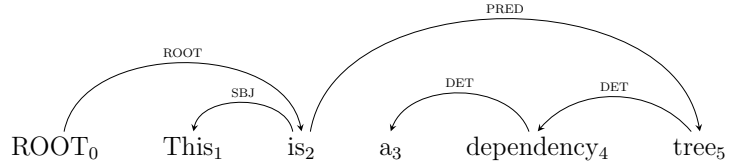
a)



b)



c)



d)

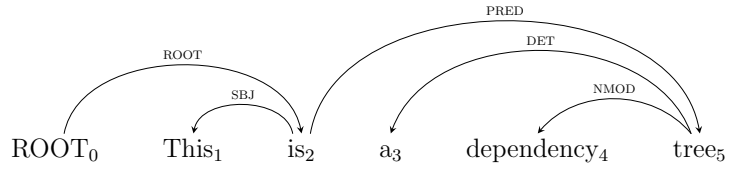


Fig. 7. **a)** Dependency tree of an English sentence analyzed by the reverse arc-eager parser. **b)** Dependency tree obtained by inverting the output of the reverse parser in Figure 7a. **c)** Dependency tree derived by the original arc-eager parser. **d)** Combination of the reverse arc-eager dependency tree in Figure 7b and the arc-eager dependency tree in Figure 7c using the position-based strategy with p=2.

widely-used parsers: the pseudo-projective arc-eager parser by Nivre and Nilsson [37] and the maximum spanning tree parser by McDonald et al [28].

Finally, and in order to test our approach more deeply, we provide further experiments on two different transition-based parsers: the pseudo-projective arc-standard parser [32], also using MaltParser, and the beam-search-based implementation of the arc-eager parser provided in ZPar [48]. In both cases we combine the original parser with its reverse variant following the two available strategies.

To undertake these experiments, we use the following datasets: Arabic [19], Chinese [11], Czech [18], Danish [23], German [8], Portuguese [1], Swedish [30] and Turkish [38, 2] from the CoNLL-X shared task,[3] and the English WSJ corpus from the Penn Treebank [24] with the same dependency conversion and split as described in Zhang and Nivre [49]. To measure accuracy, we employ the following standard evaluation metrics:

- **Labelled Attachment Score (LAS):** The proportion of tokens (nodes) that are assigned both the correct head and the correct dependency relation label.
- **Unlabelled Attachment Score (UAS):** The proportion of tokens (nodes) that are assigned the correct head (regardless of the dependency relation label).

In our results, we show LAS and UAS without considering punctuation as a scoring node.

6.1 Greedy Arc-eager Parser Results

Table 1 shows the results obtained by the reverse arc-eager parser with respect to the original arc-eager parser by Nivre and Nilsson [37].

For our experiments, we used classifiers from the LIBSVM [10] and LIBLINEAR [15] packages. Concretely, in order to reduce the training time on larger datasets, we employ the LIBLINEAR package for Chinese, Czech, English and German; and for the rest of languages, we use SVM classifiers from the LIBSVM package. Feature models were optimized for each language.[4]

Note that, unlike the reverse arc-eager parser by Nilsson [29], our version was specifically tuned for each language independently from the original version, by performing feature optimization using the training set. This allows for a fairer comparison between the original and the reverse parsers, since training the reverse parser with feature models originally optimized for the standard one could introduce

[3] These treebanks have been chosen for their representativity, since they cover a wide range of language families (Germanic, Romance, Semitic, Sino-Tibetan, Slavic, and Turkic); annotation types (e.g. pure dependency annotation in the case of Danish, dependencies extracted from constituents in the case of Chinese, or from discontinuous constituents in German); and degrees of non-projectivity (ranging from the highly non-projective Czech and German treebanks to the fully projective Chinese dataset).

[4] For replicability, all the feature models are available at `http://www.grupolys.org/~cgomezr/exp/`.

a bias against the former. As we can see, the reverse version has worse performance than the standard one on all datasets, except in the Czech language dataset.

Table 2 and Table 3 detail the scores attained by the combination of the arc-eager and reverse arc-eager parsers using the position-based and the length-based strategies, respectively, in comparison to the original arc-eager parser by Nivre and Nilsson [37]. In both combination strategies, the reliability parameter and the combination order were determined using exclusively the training dataset by applying a 10-fold cross-validation process. This means that 10 different training-development set pairs were obtained from the original training dataset to undertake the cross-validation process.

Table 1. Parsing accuracy of the arc-eager parser (Arc-eager) in comparison to the reverse arc-eager parser (Reverse).

	Arc-eager		Reverse	
Language	LAS	UAS	LAS	UAS
Arabic	**67.19**	78.42	66.83	**78.50**
Chinese	**87.04**	**90.78**	85.41	89.05
Czech	79.68	85.00	**80.40**	**85.82**
Danish	**85.51**	**90.34**	84.81	89.62
German	**87.30**	**89.68**	86.08	88.40
Portug.	**88.04**	**91.40**	86.24	90.18
Swedish	**84.58**	**90.20**	81.76	87.79
Turkish	**65.80**	75.74	65.58	**75.94**
English (WSJ)	**89,09**	**90,34**	88,01	89,14

The results show that the use of the parser combination system with any strategy improves over the scores of the arc-eager parser on all of the nine datasets tested. The only cases where the combination of parsers does not outperform the score of the arc-eager parser is in the English and Portuguese datasets using the length-based strategy. But even in those cases, the results of the parser combination system are the same as with the arc-eager baseline.[5] The LAS and UAS averages show that the length-based strategy achieves a higher increment in scores than the position-based strategy. However, in some languages the position-based approach has a better performance.

To further put the obtained results into context, we provide the comparison of the parser combination to the arc-eager parser and another well-known parser. In order to show the best results of the combination of the original and reverse arc-eager, we configure them with the best strategy for each language. Concretely, Table 4 compares the accuracy of the parser combination system to the maximum

[5] In fact, the scores in this case are identical to the baseline because the chosen value for the l parameter is 0, meaning that we trust the first parser on all dependency links and the second parser on none.

Table 2. Parsing accuracy of the position-based combination (PosComb) of the arc-eager parser (ARC) and its reverse variant (REV) in comparison to the original arc-eager parser (Arc-eager). The parameter p was determined from the training dataset. For each language, the table shows the value of p and the combination order (ARC+REV or REV+ARC) that were used, obtained from the cross-validation process.

	Arc-eager		PosComb			
Language	LAS	UAS	LAS	UAS	p	Order
Arabic	67.19	78.42	**67.60**	**78.74**	15	ARC+REV
Chinese	87.04	90.78	**87.06**	**90.80**	18	ARC+REV
Czech	79.68	85.00	**80.98**	**86.32**	7	ARC+REV
Danish	**85.51**	90.34	**85.51**	**90.46**	34	ARC+REV
German	87.30	**89.68**	**87.32**	89.66	40	ARC+REV
Portug.	88.04	**91.40**	**88.18**	**91.40**	1	REV+ARC
Swedish	84.58	90.20	**84.62**	**90.22**	40	ARC+REV
Turkish	65.80	75.74	**66.00**	**76.20**	3	REV+ARC
English (WSJ)	89.09	90.34	**89.12**	**90.37**	1	REV+ARC
Average	81.58	86.88	**81.82**	**87.13**		

spanning tree parser by McDonald et al [28] and the original pseudo-projective arc-eager parser by Nivre and Nilsson [37]. Note that the arc-eager and the maximum spanning tree parsers were the two top performing systems in the CoNLL 2006 shared task [9]. For each dataset, the strategy followed to obtain the best score is shown. As we can see, the combination of the arc-eager parser with its reverse variant outperforms the score of these two widely-used parsers in all datasets.

6.2 Results with the Arc-standard Model and with Beam Search

Before proceeding to a more in-depth analysis of the results of applying our parser combination approach to the arc-eager parser, we test the generality of the approach by performing experiments with a different transition system and with a different search strategy, and seeing whether it also produces gains in accuracy.

Table 5 shows the results obtained by the greedy pseudo-projective arc-standard parser [32] and its reverse variant, using MaltParser in the same way as in the arc-eager model experiments of Section 6.1. Table 6 shows analogous results for the variant of the arc-eager parser implemented in the ZPar system [48], which uses global learning and beam search to provide state-of-the-art accuracy, at the cost of being computationally more expensive than greedy search. These beam search experiments were performed with the default settings and feature models of ZPar, but performing the pseudo-projective transformation on the training data and undoing it on the output parses in order to handle the non-projective treebanks in the same way as in the greedy implementations.

On the one hand, the original arc-standard algorithm is only outperformed by

Table 3. Parsing accuracy of the length-based combination (LenComb) of the arc-eager parser (ARC) and its reverse variant (REV) in comparison to the original arc-eager parser (Arc-eager). The parameter l was determined from the training dataset. For each language, the table shows the value of l and the combination order (ARC+REV or REV+ARC) that were used, obtained from the cross-validation process.

	Arc-eager		LenComb			
Language	LAS	UAS	LAS	UAS	l	Order
Arabic	67.19	78.42	**67.56**	**79.44**	2	REV+ARC
Chinese	87.04	90.78	**87.06**	**90.80**	19	ARC+REV
Czech	79.68	85.00	**80.88**	**86.22**	5	REV+ARC
Danish	85.51	90.34	**85.79**	**90.68**	2	REV+ARC
German	87.30	**89.68**	**87.42**	**89.68**	1	REV+ARC
Portug.	**88.04**	**91.40**	**88.04**	**91.40**	0	REV+ARC
Swedish	84.58	**90.20**	**84.60**	**90.20**	40	ARC+REV
Turkish	65.80	75.74	**66.20**	**76.12**	1	REV+ARC
English (WSJ)	**89,09**	**90,34**	**89,09**	**90,34**	0	REV+ARC
Average	81.58	86.88	**81.85**	**87.21**		

Table 4. Parsing accuracy of the best combinative configuration detailed in Table 2 and Table 3 in comparison to the arc-eager parser (Arc-eager) and the maximum spanning tree parser (MSTParser) on eight datasets form the CoNLL 2006 shared task.

	Arc-eager		MSTParser		BestCombination		
Language	LAS	UAS	LAS	UAS	LAS	UAS	Strategy
Arabic	67.19	78.42	66.91	79.34	**67.56**	**79.44**	Length-based
Chinese	87.04	90.78	85.90	**91.07**	**87.06**	90.80	Length-based
Czech	79.68	85.00	80.18	**87.30**	**80.98**	86.32	Position-based
Danish	85.51	90.34	84.79	90.58	**85.79**	**90.68**	Length-based
German	87.30	89.68	87.34	**90.38**	**87.42**	89.68	Length-based
Portug.	88.04	**91.40**	86.82	91.36	**88.18**	**91.40**	Position-based
Swedish	84.58	90.20	82.55	88.93	**84.62**	**90.22**	Position-based
Turkish	65.80	75.74	63.19	74.67	**66.20**	**76.12**	Length-based

its reverse version in the Arabic and Czech datasets. Taking into account these results and those of the reverse arc-eager model in Table 1, we can clearly see that the reverse strategy by itself is beneficial for the Czech dataset in greedy transition-based parsing. On the other hand, the reverse variant of the beam-search parser improves over the original algorithm in five out of nine datasets: it seems that the beam-search parser takes more advantage of this strategy than the greedy parsers.

Table 7 and Table 8 present the accuracy attained by the combination of the arc-standard and the reverse arc-standard parsers following the position-based and the length-based strategies, respectively, in comparison to the original arc-standard

Table 5. Parsing accuracy of the pseudo-projective arc-standard parser (Arc-standard) in comparison to the reverse arc-standard parser (Reverse).

	Arc-standard		Reverse	
Language	LAS	UAS	LAS	UAS
Arabic	66.69	**78.40**	**67.03**	77.66
Chinese	**86.22**	**90.08**	84.99	89.58
Czech	80.92	86.72	**81.96**	**87.52**
Danish	**84.55**	**89.72**	84.35	89.48
German	**86.92**	**89.36**	86.38	88.90
Portug.	**87.38**	**90.86**	87.12	90.78
Swedish	**83.05**	**88.77**	81.94	88.31
Turkish	**65.52**	**75.82**	65.15	75.34
English (WSJ)	**88.81**	**90.10**	88.79	90,00

Table 6. Parsing accuracy of the beam-search parser (ZPar) in comparison to its reversed version (Reverse).

	ZPar		Reverse	
Language	LAS	UAS	LAS	UAS
Arabic	**66.95**	**77.66**	65.33	77.09
Chinese	88.27	92.39	**88.31**	**92.41**
Czech	**84.16**	**89.66**	82.70	88.70
Danish	**86.51**	**91.30**	86.03	90.84
German	90.24	92.45	**90.26**	**92.47**
Portug.	88.70	92.53	**89.28**	**92.81**
Swedish	**85.44**	**90.86**	85.20	90.84
Turkish	65.52	76.00	**66.26**	**76.78**
English (WSJ)	91.45	92.50	**91.46**	**92.53**

parser [32]. In this case, the length-based strategy also achieves slightly higher scores than the position-based technique according to the LAS and UAS averages. However, it is in the length-based strategy where the combination process seems to be less universally useful, since in three out of nine datasets it does not outperform the original version. In spite of that, it is worth highlighting the good scores obtained in general by the position-based and length-based combinations, especially on the Arabic, Czech and English datasets. In the case of the English language, the length-based strategy allows the arc-standard parser to achieve an accuracy on par with the original arc-eager parser (Table 1), which was better without the combination approach.

Table 9 and Table 10 detail the accuracy obtained by the combination of the original and reversed beam-search parsers following the position-based and the length-based strategies, respectively, in comparison to the original beam-search ZPar parser

Table 7. Parsing accuracy of the position-based combination (PosComb) of the arc-standard parser (ARC) and its reverse variant (REV) in comparison to the original arc-standard parser (Arc-standard). The parameter p was determined from the training dataset. For each language, the table shows the value of p and the combination order (ARC+REV or REV+ARC) that were used, obtained from the cross-validation process.

	Arc-standard		PosComb			
Language	LAS	UAS	LAS	UAS	p	Order
Arabic	66.69	78.40	**67.88**	**78.42**	11	ARC+REV
Chinese	**86.22**	**90.08**	**86.22**	**90.08**	25	ARC+REV
Czech	80.92	86.72	**82.26**	**87.70**	7	ARC+REV
Danish	84.55	89.72	**84.89**	**89.86**	5	REV+ARC
German	86.92	89.36	**86.94**	**89.40**	1	REV+ARC
Portug.	**87.38**	**90.86**	**87.38**	**90.86**	0	REV+ARC
Swedish	83.05	**88.77**	**83.13**	88.69	1	REV+ARC
Turkish	65.52	75.34	**65.66**	**76.00**	9	ARC+REV
English (WSJ)	88.81	90.10	**88.84**	**90.12**	1	REV+ARC
Average	81.12	86.59	**81.47**	**86.79**		

[48]. As we can see, even though the global learning model and beam-search decoding used in this system reduce error propagation with respect to the greedy algorithms [50], our combination approach still provides clear benefits in terms of accuracy. In this case, the position-based combination attains higher scores in LAS and the length-based strategy obtains better scores in UAS according to the LAS and UAS averages.

6.3 Analysis

It is clear that combination of parsers makes sense when one of them can correctly analyze some structures that the other cannot and vice versa.

When we combine the arc-eager parser with the reverse arc-eager parser, we expect the reverse approach to build arcs that the original version is not able to. This is, in fact, what happened in our experiments. For instance, Figure 8 shows the precision relative to dependent position in the sentence for the arc-eager parser (Arceager) and the reverse arc-eager parser (Reverse) on the Czech dataset. We can see that the precision of the reverse parser is higher than the obtained by the arc-eager parser from position 8 to the end of the sentence (the rightmost arcs). Thus, we can use a position-based strategy with p=7 and order ARC+REV to take advantage of this phenomenon. Indeed, this is what appears in Table 2 for the Czech dataset.

Note that there are two languages (Portuguese and Turkish) in Table 2 where the combination order is REV+ARC instead of ARC+REV. This means that the

Table 8. Parsing accuracy of the length-based combination (LenComb) of the arc-standard parser (ARC) and its reverse variant (REV) in comparison to the original arc-standard parser (Arc-standard). The parameter l was determined from the training dataset. For each language, the table shows the value of l and the combination order (ARC+REV or REV+ARC) that were used, obtained from the cross-validation process.

	Arc-standard		LenComb			
Language	LAS	UAS	LAS	UAS	l	Order
Arabic	66.69	78.40	**67.86**	**78.86**	3	REV+ARC
Chinese	**86.22**	**90.08**	**86.22**	**90.08**	20	ARC+REV
Czech	80.92	86.72	**82.06**	**87.50**	6	REV+ARC
Danish	84.55	89.72	**84.95**	**90.02**	4	REV+ARC
German	**86.92**	**89.36**	**86.92**	**89.36**	0	REV+ARC
Portug.	**87.38**	**90.86**	**87.38**	**90.86**	0	REV+ARC
Swedish	**83.05**	**88.77**	**83.05**	**88.77**	0	REV+ARC
Turkish	65.52	75.34	**65.90**	**76.30**	2	REV+ARC
English (WSJ)	88.81	90.10	**89.09**	**90.31**	3	REV+ARC
Average	81.12	86.59	**81.49**	**86.90**		

reverse parser obtains better accuracy on the leftmost arcs than on the rightmost ones, which is more unusual. Concretely, in these languages the reverse parser improves the score obtained by the arc-eager parser in arcs originating from the root node 0 (the leftmost arcs). For instance, the reverse parser achieves a 94.03% of precision in arcs created from the root node in the Turkish dataset, whilst the arc-eager obtains 86.97% precision in the same language and doing the same task. Therefore, if we combine both parsers with a position-based strategy with a low p=3 and order REV+ARC, we will use the strength of the reverse parser on creating root arcs (usually situated between nodes 1 and 3) in the Turkish dataset, as shown in Table 2.

In addition to offering improvements at some positions in the sentence, the reverse arc-eager parser improves over the original version on arcs with a certain length. For instance, the reverse parser obtains better accuracy on short arcs (length lower than 5) in the Czech dataset, whilst the original parser achieves better scores on long arcs. This is shown in Figure 9. Note that, although the reverse parser also performs better on very long arcs (length larger than 25), it is more important to take advantage of it in the short arcs because the proportion of short arcs is higher than that of very long ones. Therefore, a length-based combination with parameter l=5 and order REV+ARC is the proper configuration to obtain the best results, and that was the one selected by cross-validation on the training set and described in Table 3.

Finally, we have to mention that our combinative approach is less sensitive to cycles than other strategies such as voting. This is probably because we are

Table 9. Parsing accuracy of the position-based combination (PosComb) of the beam-search parser (ZP) and its reverse variant (REV) in comparison to the original beam-search parser (ZPar). The parameter p was determined from the training dataset. For each language, the table shows the value of p and the combination order (ZP+REV or REV+ZP) that were used, obtained from the cross-validation process.

	ZPar		PosComb			
Language	LAS	UAS	LAS	UAS	p	Order
Arabic	66.95	77.66	**67.52**	**78.18**	15	ZP+REV
Chinese	88.28	92.39	**88.33**	**92.41**	12	ZP+REV
Czech	84.16	89.66	**84.34**	**89.72**	30	ZP+REV
Danish	86.51	**91.30**	**86.57**	91.18	14	ZP+REV
German	90.24	92.45	**90.48**	**92.63**	1	ZP+REV
Portug.	88.70	92.53	**88.94**	**92.59**	9	REV+ZP
Swedish	85.44	90.86	**85.68**	**91.10**	2	ZP+REV
Turkish	65.52	76.00	**66.22**	**76.74**	25	REV+ZP
English (WSJ)	91.45	92.50	**91.49**	**92.55**	14	ZP+REV
Average	83.03	88.37	**83.29**	**88.57**		

working with a single transition-based algorithm (in our main experiments, the arc-eager parser). Other combinative approaches likely suffer from a high number of cycles due to joining parsers of different kinds. The percentage of sentences of the treebank where a cycle is created during our combination process is shown in Table 11. Although the table focuses on the greedy arc-eager parser, the parsers of Section 6.2 yield similar figures. As we can see, the percentage of sentences with cycles is significantly low and the length-based strategy is more prone to present cycles than the position-based technique. This is because the position-based combination takes one part of the output graph from the first parser and the other part from the second one, in that way, each part of the graph taken does not present inner cycles (although there could be cycles spanning both parts of the graph at once). However, the length-based combination creates the output by choosing arcs individually from each parser regardless of the position, and therefore it has a tendency to cause more cycles.

7 DISCUSSION

We presented an optimized version of the *reverse arc-eager parser* introduced by Nilsson [29]. This is obtained by applying a reverse strategy on the pseudo-projective arc-eager parser by Nivre and Nilsson [37], which makes this parser analyze a given sentence in reverse order: from right to left. We found out that the reverse arc-eager parser can correctly handle some syntactic structures that the original parser cannot. Initially, we expected a better accuracy of the reverse variant on the rightmost

Table 10. Parsing accuracy of the length-based combination (LenComb) of the beam-search parser (ZP) and its reverse variant (REV) in comparison to the original beam-search parser (ZPar). The parameter l was determined from the training dataset. For each language, the table shows the value of l and the combination order (ZP+REV or REV+ZP) that were used, obtained from the cross-validation process.

	ZPar		LenComb			
Language	LAS	UAS	LAS	UAS	l	Order
Arabic	**66.95**	77.66	66.77	**78.26**	1	REV+ZP
Chinese	88.28	92.39	**88.57**	**92.56**	1	ZP+REV
Czech	**84.16**	**89.66**	**84.16**	**89.66**	0	REV+ZP
Danish	86.51	91.30	**86.55**	**91.34**	1	REV+ZP
German	90.24	92.45	**90.52**	**92.65**	1	REV+ZP
Portug.	88.70	92.53	**89.10**	**92.69**	3	REV+ZP
Swedish	85.44	90.86	**85.64**	**91.08**	1	ZP+REV
Turkish	65.52	76.00	**66.70**	**77.06**	2	REV+ZP
English (WSJ)	91.45	92.50	**91.51**	**92.59**	1	REV+ZP
Average	83.03	88.37	**83.28**	**88.65**		

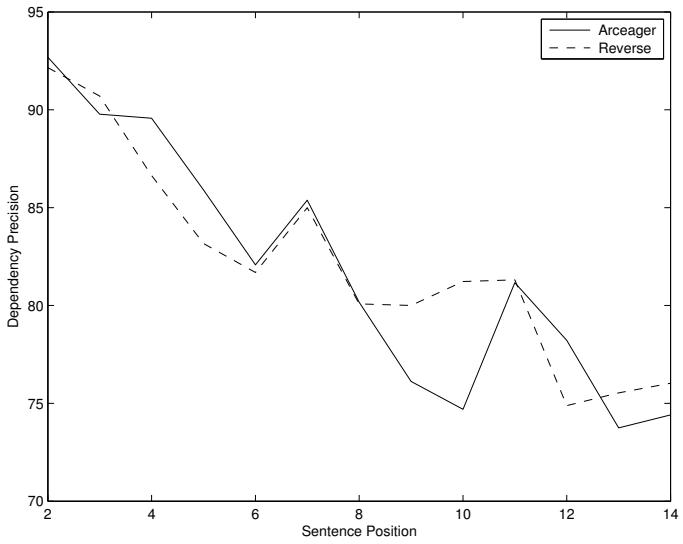


Fig. 8. Dependency arc precision relative to position in the sentence, for the arc-eager parser (Arceager) and the reverse arc-eager parser (Reverse), on the Czech dataset.

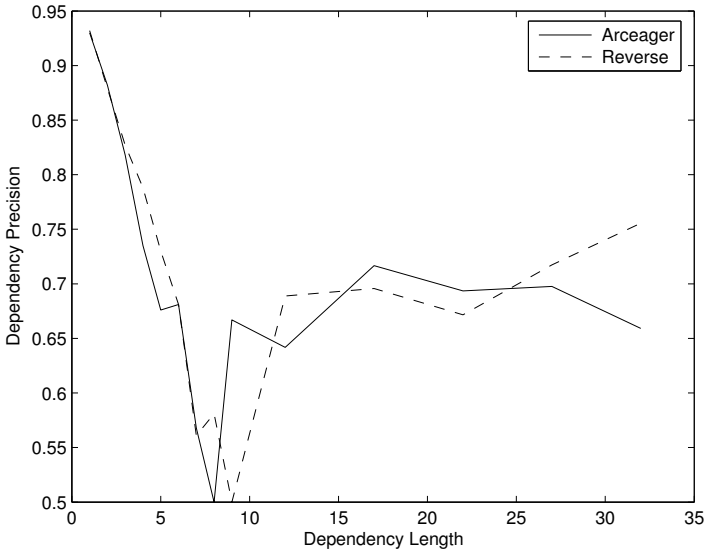


Fig. 9. Dependency arc precision relative to predicted dependency length, for the arc-eager parser (Arceager) and the reverse arc-eager parser (Reverse) on the Czech dataset.

arcs of the dependency graph as predicted by Nilsson [29]. However, we noticed that in some languages, such as Turkish, the reverse arc-eager parser performed better on the leftmost arcs of the graph. In addition to this, we discovered that the reverse variant produced better results than the arc-eager parser on arcs with certain lengths.

To take advantage of these findings, we introduced a *parser combination system*, that is able to integrate the dependency trees output by two different parsers into a new dependency tree that gathers the best of each one. We present two different strategies to undertake the parser combination: the *position-based* strategy, which combines the models regarding the position of the arcs in the sentence, and the *length-based* combination, which integrates two parsers taking into account the length of the arcs. We use this combination system to improve the arc-eager parser by combining it with its reverse variant.

The results obtained show that this approach with any of both strategies produces improvements in the arc-eager parsing accuracy on all of the nine datasets used in the experiments and is even able to outperform widely-used dependency parsers. Moreover, we also showed that this combination process can be successfully applied to different dependency parsers such as the arc-standard parser [32], and different search strategies and learning models such as the global learning and beam search used in the ZPar parser [48].

Table 11. Percentage of sentences of each language that presented a cycle during the combination process using the position-based (% Position-based) or the length-based (% Length-based) strategies with the arc-eager parser.

Language	% Position-based	% Length-based
Arabic	2.74	17.81
Chinese	0.00	0.00
Czech	1.37	7.40
Danish	0.00	2.17
German	0.00	0.56
Portug.	1.39	0.00
Swedish	0.00	0.00
Turkish	0.32	1.28
English	0.82	0.00
Average	0.74	3.25

In addition, our combination system does not add any extra time complexity and allows the parallel execution of a parser and its reverse version. Therefore, by applying this approach on a single parser, more accuracy is achieved in the same amount of time as if we use this parser in a regular way.[6] Thus, our technique is especially useful in settings where parsing speed is important, so that combination approaches that incur significant speed penalties are not desirable.

Furthermore, the combination method presented in this article interferes neither in the learning nor in the parsing process, but is used in a post-parsing step. This means that it can be applied on any dependency parser, regardless of its nature, because it does not depend on each parser's characteristics.

As future work, this system can be extended by adding new combination strategies such as combining two (or more) parsers, where each one is good at doing certain part of the dependency tree; developing a new direction-based strategy, which trusts one parser on building the leftward arcs and uses the other parser to create the rightward arcs; or implementing combination strategies with a range of reliability parameters, in that way, the combination could be more specific.

Acknowledgements This research has been partially funded by Spanish Ministry of Economy and Competitiveness and FEDER (projects TIN2010-18552-C03-01 and TIN2010-18552-C03-02), Ministry of Education (FPU Grant Program) and Xunta de Galicia (CN 2012/008, Rede Galega de Procesamento da Linguaxe e Recu-

[6] Using the position-based combination, it is even possible to execute the reversed and the original arc-eager parser in a sequential way, while still spending roughly the same amount of time as with a single parser. To achieve that, one parser would analyze one portion of the dependency graph until position p and the other parser would create the other part of the graph.

peración da Información, Rede Galega de Recursos Lingüísticos para unha Sociedade do Coñecemento).

REFERENCES

[1] Afonso, S., Bick, E., Haber, R., Santos, D.: "Floresta sint(c)tica": a treebank for Portuguese. In: Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC 2002). pp. 1968–1703. ELRA, Paris, France (2002)

[2] Atalay, N.B., Oflazer, K., Say, B.: The annotation process in the Turkish treebank. In: Proceedings of EACL Workshop on Linguistically Interpreted Corpora (LINC-03). pp. 243–246. Association for Computational Linguistics, Morristown, NJ, USA (2003)

[3] Attardi, G., Dell'Orletta, F.: Reverse revision and linear tree combination for dependency parsing. In: NAACL-Short 09. pp. 261–264 (2009)

[4] Ballesteros, M., Nivre, J.: Going to the roots of dependency parsing. Computational Linguistics 39(1), 5–13 (2013)

[5] Bloomfield, L.: Language. University of Chicago Press (1933)

[6] Bohnet, B., Kuhn, J.: The best of both worlds - a graph-based completion model for transition-based parsers. In: Daelemans, W., Lapata, M., Màrquez, L. (eds.) EACL. pp. 77–87. The Association for Computational Linguistics (2012)

[7] Bohnet, B., Nivre, J.: A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In: Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning. pp. 1455–1465. EMNLP-CoNLL '12, Association for Computational Linguistics, Stroudsburg, PA, USA (2012), http://dl.acm.org/citation.cfm?id=2390948.2391114

[8] Brants, S., Dipper, S., Hansen, S., Lezius, W., Smith, G.: The tiger treebank. In: Proceedings of the Workshop on Treebanks and Linguistic Theories, September 20-21. Sozopol, Bulgaria (2002), http://www.coli.uni-sb.de/~sabine/tigertreebank.pdf

[9] Buchholz, S., Marsi, E.: CoNLL-X shared task on multilingual dependency parsing. In: Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL). pp. 149–164 (2006)

[10] Chang, C.C., Lin, C.J.: LIBSVM: A Library for Support Vector Machines (2001), software available at http://www.csie.ntu.edu.tw/∼cjlin/libsvm

[11] Chen, K., Luo, C., Chang, M., Chen, F., Chen, C., Huang, C., Gao, Z.: Sinica treebank: Design criteria, representational issues and implementation. In: Abeillé, A. (ed.) Treebanks: Building and Using Parsed Corpora, chap. 13, pp. 231–248. Kluwer (2003)

[12] Chomsky, N.: Three models for the description of language. IRE Transactions on Information Theory IT-2, 113–124 (1956)

[13] Ding, Y., Palmer, M.: Synchronous dependency insertion grammars: A grammar formalism for syntax based statistical MT. In: Proceedings of the Workshop on Recent Advances in Dependency Grammar. pp. 90–97 (2004)

[14] Eisner, J.M.: Three new probabilistic models for dependency parsing: An exploration. In: Proceedings of the 16th International Conference on Computational Linguistics (COLING). pp. 340–345 (1996)

[15] Fan, R.E., Chang, K.W., Hsieh, C.J., Wang, X.R., Lin, C.J.: LIBLINEAR: A library for large linear classification. Journal of Machine Learning Research 9, 1871–1874 (2008)

[16] Fishel, M., Nivre, J.: Voting and stacking in data-driven dependency parsing (2009)

[17] Gómez-Rodríguez, C., Nivre, J.: Divisible transition systems and multiplanar dependency parsing. Computational Linguistics 39(4), 799–845 (2013)

[18] Hajič, J., Panevová, J., Hajičová, E., Panevová, J., Sgall, P., Pajas, P., Štěpánek, J., Havelka, J., Mikulová, M.: Prague Dependency Treebank 2.0. CDROM CAT: LDC2006T01, ISBN 1-58563-370-4. Linguistic Data Consortium (2006)

[19] Hajič, J., Smrž, O., Zemánek, P., Šnaidauf, J., Beška, E.: Prague Arabic Dependency Treebank: Development in data and tools. In: Proceedings of the NEMLAR International Conference on Arabic Language Resources and Tools (2004)

[20] Hall, J., Nilsson, J., Nivre, J., Eryiğit, G., Megyesi, B., Nilsson, M., Saers, M.: Single malt or blended? A study in multilingual parser optimization. In: Proceedings of the CoNLL Shared Task of EMNLP-CoNLL 2007. pp. 933–939 (2007)

[21] Huang, L., Sagae, K.: Dynamic programming for linear-time incremental parsing. In: Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics. pp. 1077–1086. ACL '10, Association for Computational Linguistics, Stroudsburg, PA, USA (2010), http://portal.acm.org/citation.cfm?id=1858681.1858791

[22] Koo, T., Collins, M.: Efficient third-order dependency parsers. In: Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL). pp. 1–11 (2010)

[23] Kromann, M.T.: The Danish dependency treebank and the underlying linguistic theory. In: Proceedings of the 2nd Workshop on Treebanks and Linguistic Theories (TLT). pp. 217–220. Växjö University Press, Växjö, Sweden (2003)

[24] Marcus, M.P., Santorini, B., Marcinkiewicz, M.A.: Building a large annotated corpus of English: The Penn Treebank. Computational Linguistics 19, 313–330 (1993)

[25] Martins, A., Smith, N., Xing, E.: Concise integer linear programming formulations for dependency parsing. In: Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP (ACL-IJCNLP). pp. 342–350 (2009)

[26] McDonald, R., Nivre, J.: Characterizing the errors of data-driven dependency parsing models. In: Proceedings of the 2007 Joint Conference on Empirical Meth-

ods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL). pp. 122–131 (2007)

[27] McDonald, R., Nivre, J.: Analyzing and integrating dependency parsers. Comput. Linguist. 37, 197–230 (2011)

[28] McDonald, R., Pereira, F., Ribarov, K., Hajič, J.: Non-projective dependency parsing using spanning tree algorithms. In: Proceedings of the Human Language Technology Conference and the Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP). pp. 523–530 (2005)

[29] Nilsson, J.: Transformation and Combination in Data-Driven Dependency Parsing. Ph.D. thesis, Växjö University (2009)

[30] Nilsson, J., Hall, J., Nivre, J.: MAMBA meets TIGER: Reconstructing a Swedish treebank from Antiquity. In: Henrichsen, P.J. (ed.) Proceedings of the NODALIDA Special Session on Treebanks (2005)

[31] Nivre, J.: An efficient algorithm for projective dependency parsing. In: Proceedings of the 8th International Workshop on Parsing Technologies (IWPT). pp. 149–160 (2003)

[32] Nivre, J.: Algorithms for deterministic incremental dependency parsing. Computational Linguistics 34, 513–553 (2008)

[33] Nivre, J.: Algorithms for Deterministic Incremental Dependency Parsing. Computational Linguistics 34(4), 513–553 (2008), http://www.mitpressjournals.org/doi/abs/10.1162/coli.07-056-R1-07-027

[34] Nivre, J., Hall, J., Nilsson, J.: Memory-based dependency parsing. In: Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL-2004). pp. 49–56. Association for Computational Linguistics, Morristown, NJ, USA (2004)

[35] Nivre, J., Hall, J., Nilsson, J.: Maltparser: A data-driven parser-generator for dependency parsing. In: Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC). pp. 2216–2219 (2006)

[36] Nivre, J., McDonald, R.: Integrating graph-based and transition-based dependency parsers. In: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL). pp. 950–958 (2008)

[37] Nivre, J., Nilsson, J.: Pseudo-projective dependency parsing. In: Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL). pp. 99–106 (2005)

[38] Oflazer, K., Say, B., Hakkani-Tür, D.Z., Tür, G.: Building a Turkish treebank. In: Abeillé, A. (ed.) Treebanks: Building and Using Parsed Corpora, pp. 261–277. Kluwer (2003)

[39] Sagae, K., Lavie, A.: Parser combination by reparsing. In: Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers. pp. 129–132 (2006)

[40] Samuelsson, Y., Eklund, J., Täckström, O., Velupillai, S., Saers, M.: Mixing and blending syntactic and semantic dependencies. In: In Proc. of CoNLL-2008 Shared Task (2008)

[41] Snow, R., Jurafsky, D., Ng, A.Y.: Learning syntactic patterns for automatic hypernym discovery. In: Advances in Neural Information Processing Systems (NIPS) (2005)

[42] Surdeanu, M., Manning, C.D.: Ensemble models for dependency parsing: cheap and good? In: Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics. pp. 649–652. HLT '10, Association for Computational Linguistics, Stroudsburg, PA, USA (2010), http://dl.acm.org/citation.cfm?id=1857999.1858090

[43] Tesnière, L.: Éléments de syntaxe structurale. Editions Klincksieck (1959)

[44] Titov, I., Henderson, J.: A latent variable model for generative dependency parsing. In: Proceedings of the 10th International Conference on Parsing Technologies (IWPT). pp. 144–155 (2007)

[45] Yamada, H., Matsumoto, Y.: Statistical dependency analysis with support vector machines. In: Proceedings of the 8th International Workshop on Parsing Technologies (IWPT). pp. 195–206 (2003)

[46] Zeman, D., Žabokrtský, Z.: Improving parsing accuracy by combining diverse dependency parsers. In: Proceedings of the Ninth International Workshop on Parsing Technology. pp. 171–178. Parsing '05, Association for Computational Linguistics, Stroudsburg, PA, USA (2005), http://dl.acm.org/citation.cfm?id=1654494.1654512

[47] Zhang, Y., Clark, S.: A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 562–571 (2008)

[48] Zhang, Y., Clark, S.: Syntactic processing using the generalized perceptron and beam search. Computational Linguistics 37(1), 105–151 (2011)

[49] Zhang, Y., Nivre, J.: Transition-based parsing with rich non-local features. In: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL) (2011)

[50] Zhang, Y., Nivre, J.: Analyzing the effect of global learning and beam-search on transition-based dependency parsing. In: Kay, M., Boitet, C. (eds.) COLING (Posters). pp. 1391–1400. Indian Institute of Technology Bombay (2012)

Daniel Fernández-González received his M.Sc. degree in Computer Science from University of Vigo (UVIGO) in 2010. Currently, he is a Ph.D. student at the University of A Coruña and he works as a researcher at the Computer Science Department of UVIGO, Spain. His main research interest is data-driven dependency parsing, especially transition-based parsing.

Carlos Gómez-Rodríguez received M.Sc. and Ph.D. degrees in Computer Science from the University of A Coruña in 2005 and 2009, respectively, and is currently an Associate Professor at the same institution. His main research focus is on natural language parsing, and he has authored a monograph and several dozens of papers in this field. His contributions include both theoretical and empirical work on constituency and dependency-based parsing algorithms, as well as on applications of parsing to other natural language processing tasks.

David Vilares received his M. Sc. degree in Computer Science from University of A Coruña in 2012. He is currently a researcher at the Computer Science Department of the University of A Coruña. His research interests include sentiment analysis and natural language processing.

Arc-Eager Parsing with the Tree Constraint

Joakim Nivre*
Uppsala University

Daniel Fernández-González**
University of Vigo

The arc-eager system for transition-based dependency parsing is widely used in natural language processing despite the fact that it does not guarantee that the output is a well-formed dependency tree. We propose a simple modification to the original system that enforces the tree constraint without requiring any modification to the parser training procedure. Experiments on multiple languages show that the method on average achieves 72% of the error reduction possible and consistently outperforms the standard heuristic in current use.

1. Introduction

One of the most widely used transition systems for dependency parsing is the arc-eager system first described in Nivre (2003), which has been used as the backbone for greedy deterministic dependency parsers (Nivre, Hall, and Nilsson 2004; Goldberg and Nivre 2012), beam search parsers with structured prediction (Zhang and Clark 2008; Zhang and Nivre 2011), neural network parsers with latent variables (Titov and Henderson 2007), and delexicalized transfer parsers (McDonald, Petrov, and Hall 2011). However, in contrast to most similar transition systems, the arc-eager system does not guarantee that the output is a well-formed dependency tree, which sometimes leads to fragmented parses and lower parsing accuracy. Although various heuristics have been proposed to deal with this problem, there has so far been no clean theoretical solution that also gives good parsing accuracy. In this article, we present a modified version of the original arc-eager system, which is provably correct for the class of projective dependency trees, which maintains the linear time complexity of greedy (or beam search) parsers, and which does not require any modifications to the parser training procedure. Experimental evaluation on the CoNLL-X data sets show that the new system consistently outperforms the standard heuristic in current use, on average achieving 72% of the error reduction possible (compared with 41% for the old heuristic).

* Uppsala University, Department of Linguistics and Philology, Box 635, SE-75126, Uppsala, Sweden. E-mail: joakim.nivre@lingfil.uu.se.
** Universidad de Vigo, Departamento de Informática, Campus As Lagoas, 32004, Ourense, Spain. E-mail: danifg@uvigo.es.

Submission received: 25 June 2013; accepted for publication: 4 November 2013.

doi:10.1162/COLI_a_00185

2. The Problem

The dependency parsing problem is usually defined as the task of mapping a sentence $x = w_1, \ldots, w_n$ to a dependency tree T, which is a directed tree with one node for each input token w_i, plus optionally an artificial root node corresponding to a dummy word w_0, and with arcs representing dependency relations, optionally labeled with dependency types (Kübler, McDonald, and Nivre 2009). In this article, we will furthermore restrict our attention to dependency trees that are projective, meaning that every subtree has a contiguous yield. Figure 1 shows a labeled projective dependency tree.

Transition-based dependency parsing views parsing as heuristic search through a non-deterministic transition system for deriving dependency trees, guided by a statistical model for scoring transitions from one configuration to the next. Figure 2 shows the arc-eager transition system for dependency parsing (Nivre 2003, 2008). A parser configuration consists of a stack σ, a buffer β, and a set of arcs A. The initial configuration for parsing a sentence $x = w_1, \ldots, w_n$ has an empty stack, a buffer containing the words $w_1, \ldots, w_n$, and an empty arc set. A terminal configuration is any configuration with an empty buffer. Whatever arcs have then been accumulated in the arc set A defines the output dependency tree. There are four possible transitions from a configuration

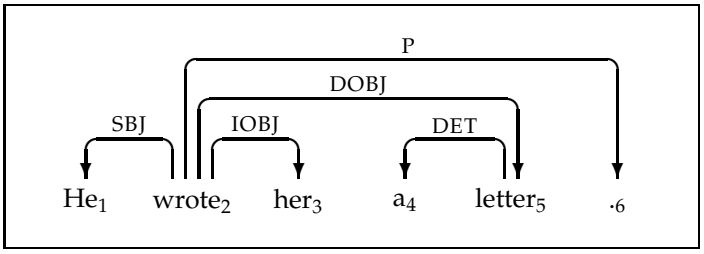


Figure 1
Projective labeled dependency tree for an English sentence.

Initial:	$([\,], [w_1, \ldots, w_n], \{\,\})$			
Terminal:	$(\sigma, [\,], A)$			
Shift:	$(\sigma, w_i \mid \beta, A)$	$\Rightarrow$	$(\sigma \mid w_i, \beta, A)$	
Reduce:	$(\sigma \mid w_i, \beta, A)$	$\Rightarrow$	(σ, β, A)	$\text{HEAD}(w_i)$
Right-Arc:	$(\sigma \mid w_i, w_j \mid \beta, A)$	$\Rightarrow$	$(\sigma \mid w_i \mid w_j, \beta, A \cup \{w_i \rightarrow w_j\})$	
Left-Arc:	$(\sigma \mid w_i, w_j \mid \beta, A)$	$\Rightarrow$	$(\sigma, w_j \mid \beta, A \cup \{w_i \leftarrow w_j\})$	$\neg\text{HEAD}(w_i)$

Figure 2
Arc-eager transition system for dependency parsing. We use | as list constructor, meaning that $\sigma \mid w_i$ is a stack with top w_i and remainder σ and $w_j \mid \beta$ is a buffer with head w_j and tail β. The condition $\text{HEAD}(w_i)$ is true in a configuration (σ, β, A) if A contains an arc $w_k \rightarrow w_i$ (for some k).

where *top* is the word on top of the stack (if any) and *next* is the first word of the buffer:[1]

1. **Shift** moves *next* to the stack.
2. **Reduce** pops the stack; allowed only if *top* has a head.
3. **Right-Arc** adds a dependency arc from *top* to *next* and moves *next* to the stack.
4. **Left-Arc** adds a dependency arc from *next* to *top* and pops the stack; allowed only if *top* has no head.

The arc-eager system defines an incremental left-to-right parsing order, where left dependents are added bottom–up and right dependents top–down, which is advantageous for postponing certain attachment decisions. However, a fundamental problem with this system is that it does not guarantee that the output parse is a projective dependency tree, only a projective dependency forest, that is, a sequence of adjacent, non-overlapping projective trees (Nivre 2008). This is different from the closely related arc-standard system (Nivre 2004), which constructs all dependencies bottom–up and can easily be constrained to only output trees. The failure to implement the tree constraint may lead to fragmented parses and lower parsing accuracy, especially with respect to the global structure of the sentence. Moreover, even if the loss in accuracy is not substantial, this may be problematic when using the parser in applications where downstream components may not function correctly if the parser output is not a well-formed tree.

The standard solution to this problem in practical implementations, such as MaltParser (Nivre, Hall, and Nilsson 2006), is to use an artificial root node and to attach all remaining words on the stack to the root node at the end of parsing. This fixes the formal problem, but normally does not improve accuracy because it is usually unlikely that more than one word should attach to the artificial root node. Thus, in the error analysis presented by McDonald and Nivre (2007), MaltParser tends to have very low precision on attachments to the root node. Other heuristic solutions have been tried, usually by post-processing the nodes remaining on the stack in some way, but these techniques often require modifications to the training procedure and/or undermine the linear time complexity of the parsing system. In any case, a clean theoretical solution to this problem has so far been lacking.

3. The Solution

We propose a modified version of the arc-eager system, which guarantees that the arc set A in a terminal configuration forms a projective dependency tree. The new system, shown in Figure 3, differs in four ways from the old system:

1. Configurations are extended with a boolean variable e, keeping track of whether we have seen the end of the input, that is, whether we have passed through a configuration with an empty buffer.

1 For simplicity, we only consider unlabeled parsing here. In labeled parsing, which is used in all experiments, **Right-Arc** and **Left-Arc** also have to select a label for the new arc.

Initial:	$([\,], [w_1, \ldots, w_n], \{\,\}, \textbf{false})$	
Terminal:	$([w_i], [\,], A, \textbf{true})$	
Shift:	$(\sigma, w_i\|\beta, A, \textbf{false}) \Rightarrow (\sigma\|w_i, \beta, A, [\![\beta = [\,]]\!])$	
Unshift:	$(\sigma\|w_i, [\,], A, \textbf{true}) \Rightarrow (\sigma, [w_i], A, \textbf{true})$	$\neg\text{HEAD}(w_i)$
Reduce:	$(\sigma\|w_i, \beta, A, e) \Rightarrow (\sigma, \beta, A, e)$	$\text{HEAD}(w_i)$
Right-Arc:	$(\sigma\|w_i, w_j\|\beta, A, e) \Rightarrow$ $(\sigma\|w_i\|w_j, \beta, A \cup \{w_i \rightarrow w_j\}, [\![e \vee \beta = [\,]]\!])$	
Left-Arc:	$(\sigma\|w_i, w_j\|\beta, A, e) \Rightarrow (\sigma, w_j\|\beta, A \cup \{w_i \leftarrow w_j\}, e)$	$\neg\text{HEAD}(w_i)$

Figure 3
Arc-eager transition system enforcing the tree constraint. The expression $[\![\phi]\!]$ evaluates to **true** if ϕ is true and **false** otherwise.

2. Terminal configurations have the form $([w_i], [\,], A, \textbf{true})$, that is, they have an empty buffer, exactly one word on the stack, and $e = \textbf{true}$.
3. The **Shift** transition is allowed only if $e = \textbf{false}$.
4. There is a new transition **Unshift**, which moves *top* back to the buffer and which is allowed only if *top* has no head and the buffer is empty.

The new system behaves exactly like the old system until we reach a configuration with an empty buffer, after which there are two alternatives. If the stack contains exactly one word, we terminate and output a tree, which was true also in the old system. However, if the stack contains more than one word, we now go on parsing but are forbidden to make any **Shift** transitions. After this point, there are two cases. If the buffer is empty, we make a deterministic choice between **Reduce** and **Unshift** depending on whether *top* has a head or not. If the buffer is not empty, we non-deterministically choose between **Right-Arc** and either **Left-Arc** or **Reduce** (the latter again depending on whether *top* has a head). Because the new **Unshift** transition is only used in completely deterministic cases, we can use the same statistical model to score transitions both before and after we have reached the end of the input, as long as we make sure to block any **Shift** transition favored by the model.

We first show that the new system is still guaranteed to terminate and that the maximum number of transitions is $O(n)$, where n is the length of the input sentence, which guarantees linear parsing complexity for greedy (and beam search) parsers with constant-time model predictions and transitions. From previous results, we know that the system is guaranteed to reach a configuration of the form $(\sigma, [\,], A)$ in $2n - k$ transitions, where $k = |\sigma|$ (Nivre 2008).[2] In any non-terminal configuration arising from this point on, we can always perform **Reduce** or **Unshift** (in case the buffer is empty) or **Right-Arc** (otherwise), which means that termination is guaranteed if we can show that the number of additional transitions is bounded.

2 This holds because we must move n words from the buffer to the stack (in either a **Shift** or a **Right-Arc** transition) and pop $n - k$ words from the stack (in either a **Reduce** or **Left-Arc** transition).

Note first that we can perform at most $k-1$ **Unshift** transitions moving a word back to the buffer (because a word can only be moved back to the buffer if it has no head, which can only happen once since **Shift** is now forbidden).[3] Therefore, we can perform at most $k-1$ **Right-Arc** transitions, moving a word back to the stack and attaching it to its head. Finally, we can perform at most $k-1$ **Reduce** and **Left-Arc** transitions, removing a word from the stack (regardless of whether it has first been moved back to the buffer). In total, we can thus perform at most $2n-k+3(k-1) < 4n$ transitions, which means that the number of transitions is $O(n)$.

Having shown that the new system terminates after a linear number of transitions, we now show that it also guarantees that the output is a well-formed dependency tree. In order to reach a terminal configuration, we must pop $n-1$ words from the stack, each of which has exactly one incoming arc and is therefore connected to at least one other node in the graph. Because the word remaining in the stack has no incoming arc but must be connected to (at least) the last word that was popped, it follows that the resulting graph is connected with exactly $n-1$ arcs, which entails that it is a tree.

It is worth noting that, although the new system can always construct a tree over the unattached words left on the stack in the first configuration of the form $(\sigma, [\,], A)$, it may not be able to construct every possible tree over these nodes. More precisely, a sequence of words $w_j, \ldots, w_k$ can only attach to a word on the left in the form of a chain (not as siblings) and can only attach to a word on the right as siblings (not as a chain). Nevertheless, the new system is both sound and complete for the class of projective dependency trees, because every terminating transition sequence derives a projective tree (soundness) and every projective tree is derived by some transition sequence (completeness). By contrast, the original arc-eager system is complete but not sound for the class of projective trees.

4. Experiments

In our empirical evaluation we make use of the open-source system MaltParser (Nivre, Hall, and Nilsson 2006), which is a data-driven parser-generator for transition-based dependency parsing supporting the use of different transition systems. Besides the original arc-eager system, which is already implemented in MaltParser, we have added an implementation of the new modified system. The training procedure used in MaltParser derives an oracle transition sequence for each sentence and gold tree in the training corpus and uses every configuration–transition pair in these sequences as a training instance for a multi-class classifier. Because the oracle sequences in the arc-eager system always produce a well-formed tree, there will be no training instances corresponding to the extended transition sequences in the new system (i.e., sequences containing one or more non-terminal configurations of the form $(\sigma, [\,], A)$). However, because the **Unshift** transition is only used in completely deterministic cases, where the classifier is not called upon to rank alternative transitions, we can make use of exactly the same classifier for both the old and the new system.[4]

We compare the original and modified arc-eager systems on all 13 data sets from the CoNLL-X shared task on multilingual dependency parsing (Buchholz and Marsi 2006),

3 The number is $k-1$, rather than k, because **Unshift** requires an empty buffer, which together with only one word on the stack would imply a terminal configuration.

4 Although this greatly simplifies the integration of the new system into existing parsing frameworks, it is conceivable that accuracy could be improved further through specialized training methods, for example, using a dynamic oracle along the lines of Goldberg and Nivre (2012). We leave this for future research.

which all assume the existence of a dummy root word prefixed to the sentence. We tune the feature representations separately for each language and projectivize the training data for languages with non-projective dependencies but otherwise use default settings in MaltParser (including the standard heuristic of attaching any unattached tokens to the artificial root node at the end of parsing for the original system). Because we want to perform a detailed error analysis for fragmented parses, we initially avoid using the dedicated test set for each language and instead report results on a development set created by splitting off 10% of the training data.

Table 1 (columns 2–3) shows the unlabeled attachment score (including punctuation) achieved with the two systems. We see that the new system improves over the old one by 0.19 percentage points on average, with individual improvements ranging from 0.00 (Japanese) to 0.50 (Slovene). These differences may seem quantitatively small, but it must be remembered that the unattached tokens left on the stack in fragmented parses constitute a very small fraction of the total number of tokens on which these scores are calculated. In order to get a more fine-grained picture of the behavior of the two systems, we therefore zoom in specifically on these tokens in the rest of Table 1.

Column 4 shows the number of unattached tokens left on the stack when reaching the end of the input (excluding the artificial root node). Column 5 shows for how many of these tokens the correct head is also on the stack (including the artificial root node). Both statistics are summed over all sentences in the development set. We see from these figures that the amount of fragmentation varies greatly between languages, from only four unattached tokens for Japanese to 230 tokens for Slovene. These tendencies seem to reflect properties of the data sets, with Japanese having the lowest average sentence length of all languages and Slovene having a high percentage of non-projective dependencies and a very small training set. They also partly explain why these languages show the smallest and largest improvement, respectively, in overall attachment score.

Table 1
Experimental results for the old and new arc-eager transition systems (development sets). UAS = unlabeled attachment score; Stack-Token = number of unattached tokens left in the stack when reaching the end of the input (excluding the artificial root node); Stack-Head = number of unattached tokens for which the head is also left in the stack (including the artificial root node); Correct = number of tokens left in the stack that are correctly attached in the final parser output; Recall = Correct/Stack-Head (%).

	UAS		Stack		Correct		Recall	
Language	**Old**	**New**	**Token**	**Head**	**Old**	**New**	**Old**	**New**
Arabic	77.38	77.74	38	24	3	22	12.50	91.67
Bulgarian	90.32	90.42	20	15	7	13	46.67	86.67
Chinese	89.28	89.48	33	31	8	18	25.81	58.06
Czech	83.26	83.46	41	36	8	21	22.22	58.33
Danish	88.10	88.17	29	23	18	22	78.26	95.65
Dutch	86.23	86.49	63	57	13	28	22.80	49.12
German	87.44	87.75	66	39	6	27	15.38	69.23
Japanese	93.53	93.53	4	4	4	4	100.00	100.00
Portuguese	87.68	87.84	44	36	15	26	41.67	72.22
Slovene	76.50	77.00	230	173	50	97	28.90	56.07
Spanish	81.43	81.59	57	42	13	22	30.95	52.38
Swedish	88.18	88.33	43	28	13	24	46.43	85.71
Turkish	81.44	81.45	24	16	9	10	56.25	62.50
Average	85.44	85.63	53.23	40.31	12.85	25.69	40.60	72.12

Table 2
Results on final test sets. LAS = labeled attachment score. UAS = unlabeled attachment score.

		Ara	Bul	Cze	Chi	Dan	Dut	Ger	Jap	Por	Slo	Spa	Swe	Tur	Ave
LAS	Old	65.90	87.39	86.11	77.82	84.11	77.28	85.42	90.88	83.52	70.30	79.72	83.19	73.92	80.43
	New	66.13	87.45	86.27	77.93	84.16	77.37	85.51	90.84	83.76	70.86	79.73	83.19	73.98	80.55
UAS	Old	76.33	90.51	89.79	83.09	88.74	79.95	87.92	92.44	86.28	77.09	82.47	88.70	81.09	84.95
	New	76.49	90.58	89.94	83.09	88.82	80.18	88.00	92.40	86.33	77.34	82.49	88.76	81.20	85.05

Columns 6 and 7 show, for the old and the new system, how many of the unattached tokens on the stack are attached to their correct head in the final parser output, as a result of heuristic root attachment for the old system and extended transition sequences for the new system. Columns 8 and 9 show the same results expressed in terms of recall or error reduction (dividing column 6/7 by column 5). These results clearly demonstrate the superiority of the new system over the old system with heuristic root attachment. Whereas the old system correctly attaches 40.60% of the tokens for which a head can be found on the stack, the new system finds correct attachments in 72.12% of the cases. For some languages, the effect is dramatic, with Arabic improving from just above 10% to over 90% and German from about 15% to almost 70%, but all languages clearly benefit from the new technique for enforcing the tree constraint.

Variation across languages can to a large extent be explained by the proportion of unattached tokens that should be attached to the artificial root node. Because the old root attachment heuristic attaches all tokens to the root, it will have 100% recall on tokens for which this is the correct attachment and 0% recall on all other tokens. This explains why the old system gets 100% recall on Japanese, where all four tokens left on the stack should indeed be attached to the root. It also means that, on average, root attachment is only correct for about 40% of the cases (which is the overall recall achieved by this method). By contrast, the new system only achieves a recall of 82.81% on root attachments, but this is easily compensated by a recall of 63.50% on non-root attachments.

For completeness, we report also the labeled and unlabeled attachment scores (including punctuation) on the dedicated test sets from the CoNLL-X shared task, shown in Table 2. The results are perfectly consistent with those analyzed in depth for the development sets. The average improvement is 0.12 for LAS and 0.10 for UAS. The largest improvement is again found for Slovene (0.58 LAS, 0.25 UAS) and the smallest for Japanese, where there is in fact a marginal drop in accuracy (0.04 LAS/UAS).[5] For all other languages, however, the new system is at least as good as the old system and in addition guarantees a well-formed output without heuristic post-processing. Moreover, although the overall improvement is small, there is a statistically significant improvement in either LAS or UAS for all languages except Bulgarian, Czech, Japanese, Spanish, and Swedish, and in both LAS and UAS on average over all languages according to a randomized permutation test ($\alpha = .05$) (Yeh 2000). Finally, it is worth noting that there is no significant difference in running time between the old and the new system.

5 As we saw in the previous analysis, fragmentation happens very rarely for Japanese and all unattached tokens should normally be attached to the root node, which gives 100% recall for the baseline parser.

5. Conclusion

In conclusion, we have presented a modified version of the arc-eager transition system for dependency parsing, which, unlike the old system, guarantees that the output is a well-formed dependency tree. The system is provably sound and complete for the class of projective dependency trees, and the number of transitions is still linear in the length of the sentence, which is important for efficient parsing. The system can be used without modifying the standard training procedure for greedy transition-based parsers, because the statistical model used to score transitions is the same as for the old system. An empirical evaluation on all 13 languages from the CoNLL-X shared task shows that the new system consistently outperforms the old system with the standard heuristic of attaching all unattached tokens to the artificial root node. Whereas the old method only recovers about 41% of the attachments that are still feasible, the new system achieves an average recall of 72%. Although this gives only a marginal effect on overall attachment score (at most 0.5%), being able to guarantee that output parses are always well formed may be critical for downstream modules that take these as input. Moreover, the proposed method achieves this guarantee as a theoretical property of the transition system without having to rely on ad hoc post-processing and works equally well regardless of whether a dummy root word is used or not.

Acknowledgments

This research has been partially funded by the Spanish Ministry of Economy and Competitiveness and FEDER (project TIN2010-18552-C03-01), Ministry of Education (FPU Grant Program), and Xunta de Galicia (projects CN 2012/319 and CN 2012/317).

References

Buchholz, Sabine and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pages 149–164, New York, NY.

Goldberg, Yoav and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING)*, pages 959–976, Jeju Island.

Kübler, Sandra, Ryan McDonald, and Joakim Nivre. 2009. *Dependency Parsing*. Morgan and Claypool.

McDonald, Ryan and Joakim Nivre. 2007. Characterizing the errors of data-driven dependency parsing models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 122–131, Prague.

McDonald, Ryan, Slav Petrov, and Keith Hall. 2011. Multi-source transfer of delexicalized dependency parsers. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 62–72, Edinburgh.

Nivre, Joakim. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 149–160, Nancy.

Nivre, Joakim. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together (ACL)*, pages 50–57, Stroudsburg, PA.

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

Nivre, Joakim, Johan Hall, and Jens Nilsson. 2004. Memory-based dependency parsing. In *Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL)*, pages 49–56, Boston, MA.

Nivre, Joakim, Johan Hall, and Jens Nilsson. 2006. Maltparser: A data-driven parser-generator for dependency parsing. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC)*, pages 2,216–2,219, Genoa.

Titov, Ivan and James Henderson. 2007. A latent variable model for generative

dependency parsing. In *Proceedings of the 10th International Conference on Parsing Technologies (IWPT)*, pages 144–155, Prague.

Yeh, Alexander. 2000. More accurate tests for the statistical significance of result differences. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING)*, pages 947–953, Saarbrüken.

Zhang, Yue and Stephen Clark. 2008. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 562–571, Honolulu, HI.

Zhang, Yue and Joakim Nivre. 2011. Transition-based parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 188–193, Portland, OR.

An Efficient Dynamic Oracle for Unrestricted Non-Projective Parsing

Carlos Gómez-Rodríguez
Departamento de Computación
Universidade da Coruña
Campus de Elviña, s/n
15071 A Coruña, Spain
carlos.gomez@udc.es

Daniel Fernández-González
Departamento de Informática
Universidade de Vigo
Campus As Lagoas, s/n
32004 Ourense, Spain
danifg@uvigo.es

Abstract

We define a dynamic oracle for the Covington non-projective dependency parser. This is not only the first dynamic oracle that supports arbitrary non-projectivity, but also considerably more efficient ($O(n)$) than the only existing oracle with restricted non-projectivity support. Experiments show that training with the dynamic oracle significantly improves parsing accuracy over the static oracle baseline on a wide range of treebanks.

1 Introduction

Greedy transition-based dependency parsers build analyses for sentences incrementally by following a sequence of transitions defined by an automaton, using a scoring model to choose the best transition to take at each state (Nivre, 2008). While this kind of parsers have become very popular, as they achieve competitive accuracy with especially fast parsing times; their raw accuracy is still behind that of slower alternatives like transition-based parsers that use beam search (Zhang and Nivre, 2011; Choi and McCallum, 2013). For this reason, a current research challenge is to improve the accuracy of greedy transition-based parsers as much as possible without sacrificing efficiency.

A relevant recent advance in this direction is the introduction of dynamic oracles (Goldberg and Nivre, 2012), an improvement in the training procedure of greedy parsers that can boost their accuracy without any impact on parsing speed. An oracle is a training component that selects the best transition(s) to take at a given configuration, using knowledge about the gold tree. Traditionally, transition-based parsers were trained to follow a so-called static oracle, which is only defined on the configurations of a canonical computation that generates the gold tree, returning the next transition in said computation. In contrast, dynamic oracles are non-deterministic (not limited to one sequence, but supporting all the possible computations leading to the gold tree), and complete (also defined for configurations where the gold tree is unreachable, choosing the transition(s) that lead to a tree with minimum error). This extra robustness in training provides higher parsing accuracy.

However, defining a usable dynamic oracle for a given parser is non-trivial in general, due to the need of calculating the loss of each configuration, i.e., the minimum Hamming loss to the gold tree from a tree reachable from that configuration. While it is always easy to do this in exponential time by simulating all possible computations in the algorithm to obtain all reachable trees, it is not always clear how to achieve this calculation in polynomial time. At the moment, this problem has been solved for several projective parsers exploiting either arc-decomposability (Goldberg and Nivre, 2013) or tabularization of computations (Goldberg et al., 2014). However, for parsers that can handle crossing arcs, the only known dynamic oracle (Gómez-Rodríguez et al., 2014) has been defined for a variant of the parser by Attardi (2006) that supports a restricted set of non-projective trees. To our knowledge, no dynamic oracles are known for any transition-based parser that can handle unrestricted non-projectivity.

In this paper, we define such an oracle for the Covington non-projective parser (Covington, 2001; Nivre, 2008), which can handle arbitrary non-projective dependency trees. As this algorithm is not arc-decomposable and its tabularization is NP-hard (Neuhaus and Bröker, 1997), we do not use the existing techniques to define dynamic oracles, but a reasoning specific to this parser. It is worth noting that, apart from being the first dynamic oracle supporting unrestricted non-projectivity, our oracle is very efficient, solving the loss calculation in $O(n)$. In contrast, the restricted non-projective oracle of Gómez-Rodríguez et al.

(2014) has $O(n^8)$ time complexity.

The rest of the paper is organized as follows: after a quick outline of Covington's parser in Sect. 2, we present the oracle and prove its correctness in Sect. 3. Experiments are reported in Sect. 4, and Sect. 5 contains concluding remarks.

2 Preliminaries

We will define a dynamic oracle for the non-projective parser originally defined by Covington (2001), and implemented by Nivre (2008) under the transition-based parsing framework. For space reasons, we only sketch the parser very briefly, and refer to the above reference for more details.

Parser configurations are of the form $c = \langle \lambda_1, \lambda_2, B, A \rangle$, where λ_1 and λ_2 are lists of partially processed words, B is another list (called the buffer) with currently unprocessed words, and A is the set of dependencies built so far. Suppose that we parse a string $w_1 \cdots w_n$, whose word occurrences will be identified with their indices $1 \cdots n$ for simplicity. Then, the parser starts at an initial configuration $c_s(w_1 \dots w_n) = \langle [], [], [1 \dots n], \emptyset \rangle$, and executes transitions chosen from those in Figure 1 until a terminal configuration of the form $\{\langle \lambda_1, \lambda_2, [], A \rangle \in C\}$ is reached, and the sentence's parse tree is obtained from A.[1]

The transition semantics is very simple, mirroring the double nested loop traversing word pairs in the formulation by Covington (2001). When the algorithm is in a configuration $\langle \lambda_1 | i, \lambda_2, j | B, A \rangle$, we will say that it is considering the **focus words** i and j, located at the end of the first list and at the beginning of the buffer. A decision is then made about whether these two words should be linked with a rightward arc $i \rightarrow j$ (Right-Arc transition), a leftward arc $i \leftarrow j$ (Left-Arc transition) or not linked (No-Arc transition). The first two choices will be unavailable in configurations where the newly-created arc would violate the **single-head constraint** (a node cannot have more than one incoming arc) or the **acyclicity constraint** (cycles are not allowed). In any of these three transitions, i is then moved to the second list to make $i-1$ and j the focus words for the next step. Alternatively, we can choose to read a new word from the string with a Shift transition, so that the focus words in the resulting configuration will be j and $j+1$.

The result is a parser that can generate any possible dependency tree for the input, and runs in quadratic worst-case time. Although in theory this complexity can seem like a drawback compared to linear-time transition-based parsers (e.g. (Nivre, 2003; Gómez-Rodríguez and Nivre, 2013)), it has been shown by Volokh and Neumann (2012) to actually outperform linear algorithms in practice, as it allows for relevant optimizations in feature extraction that cannot be implemented in other parsers. In fact, one of the fastest dependency parsers to date uses this algorithm (Volokh, 2013).

3 The oracle

As sketched in Sect. 1, a dynamic oracle is a training component that, given a configuration c and a gold tree t_G, provides the set of transitions that are applicable in c and lead to trees with minimum Hamming loss with respect to t_G. The Hamming loss between a tree t and t_G, written $\mathcal{L}(t, t_G)$, is the number of nodes that have a different head in t than in t_G. Following Goldberg and Nivre (2013), we say that a set of arcs A is **reachable** from configuration c, written $c \rightsquigarrow A$, if there is some (possibly empty) path of transitions from c to some configuration $c' = \langle \lambda_1, \lambda_2, B, A' \rangle$, with $A \subseteq A'$. Then, we can define the loss of a configuration as

$$\ell(c) = \min_{t | c \rightsquigarrow t} \mathcal{L}(t, t_G),$$

and the set of transitions that must be returned by a correct dynamic oracle is then

$$o_d(c, t_G) = \{\tau \mid \ell(c) - \ell(\tau(c)) = 0\},$$

i.e., the transitions that do not increase configuration loss, and hence lead to the best parse (in terms of loss) reachable from c. Therefore, implementing a dynamic oracle reduces to computing the loss $\ell(c)$ for each configuration c.

Goldberg and Nivre (2013) show that the calculation of the loss is easy for parsers that are **arc-decomposable**, i.e., those where for every configuration c and arc set A that is **tree-compatible** (i.e. that can be a part of a well-formed parse[2]), $c \rightsquigarrow A$ is entailed by $c \rightsquigarrow (i \rightarrow j)$ for every $i \rightarrow j \in A$. That is, if each arc in a tree-compatible set is individually reachable from configuration c, then that

[1] The arcs in A form a forest, but we convert it to a tree by linking any node without a head as a dependent of an artificial node at position 0 that acts as a dummy root. From now on, when we refer to some dependency graph as a tree, we assume that this transformation is being implicitly made.

[2] In the cited paper, tree-compatibility required projectivity, as the authors were dealing with projective parsers. In our case, since the parser is non-projective, tree-compatibility only consists of the single-head and acyclicity constraints.

Shift:	$\langle \lambda_1, \lambda_2, j\|B, A\rangle \Rightarrow \langle \lambda_1 \cdot \lambda_2\|j, [], B, A\rangle$
No-Arc:	$\langle \lambda_1\|i, \lambda_2, B, A\rangle \Rightarrow \langle \lambda_1, i\|\lambda_2, B, A\rangle$
Left-Arc:	$\langle \lambda_1\|i, \lambda_2, j\|B, A\rangle \Rightarrow \langle \lambda_1, i\|\lambda_2, j\|B, A \cup \{j \rightarrow i\}\rangle$ only if $\nexists k \mid k \rightarrow i \in A$ (single-head) and $i \rightarrow^* j \notin A$ (acyclicity).
Right-Arc:	$\langle \lambda_1\|i, \lambda_2, j\|B, A\rangle \Rightarrow \langle \lambda_1, i\|\lambda_2, j\|B, A \cup \{i \rightarrow j\}\rangle$ only if $\nexists k \mid k \rightarrow j \in A$ (single-head) and $j \rightarrow^* i \notin A$ (acyclicity).

Figure 1: Transitions of the Covington non-projective dependency parser.

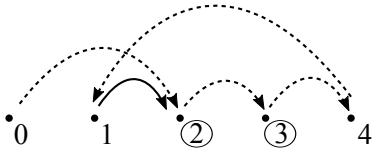


Figure 2: An example of non-arc-decomposability of the Covington parser: graphical representation of configuration $c = \langle [1,2], [], [3,4], A = \{1 \rightarrow 2\}\rangle$. The solid arc corresponds to the arc set A, and the circled indexes mark the focus words. The dashed arcs represent the gold tree t_G.

set of arcs is reachable from c. If this holds, then computing the loss of a configuration c reduces to determining and counting the gold arcs that are *not* reachable from c, which is easy in most parsers.

Unfortunately, the Covington parser is not arc-decomposable. This can be seen in the example of Figure 2: while any of the gold arcs $2\rightarrow3$, $3\rightarrow4$, $4\rightarrow1$ can be reachable individually from the depicted configuration, they are not jointly reachable as they form a cycle with the already-built arc $1\rightarrow2$. Thus, the configuration has only one individually unreachable arc ($0\rightarrow2$), but its loss is 2.

However, it is worth noting that non-arc-decomposability in the parser is exclusively due to cycles. If a set of individually reachable arcs do not form a cycle together with already-built arcs, then we can show that the set will be reachable. This idea is the basis for an expression to compute loss based on counting individually unreachable arcs, and then correcting for the effect of cycles:

Theorem 1 *Let $c = \langle \lambda_1, \lambda_2, B, A\rangle$ be a configuration of the Covington parser, and t_G the set of arcs of a gold tree. We call $\mathcal{I}(c, t_G) = \{x \rightarrow y \in t_G \mid c \rightsquigarrow (x \rightarrow y)\}$ the set of **individually reachable arcs** of t_G; note that this set may overlap A. Conversely, we call $\mathcal{U}(c, t_G) = t_G \setminus \mathcal{I}(c, t_G)$ the set of **individually unreachable arcs** of t_G from c. Finally, let $n_c(G)$ denote the number of cycles in a graph G.*

Then $\ell(c) = |\mathcal{U}(c, t_G)| + n_c(A \cup \mathcal{I}(c, t_G))$. □

We now sketch the proof. To prove Theorem 1, it is enough to show that (1) there is at least one tree reachable from c with exactly that Hamming loss to t_G, and (2) there are no trees reachable from c with a smaller loss. To this end, we will use some properties of the graph $A \cup \mathcal{I}(c, t_G)$. First, we note that no node in this graph has in-degree greater than 1. In particular, each node except for the dummy root has exactly one head, either explicit or (if no head has been assigned in A or in the gold tree) the dummy root. No node has more than one head: a node cannot have two heads in A because the parser transitions enforce the single-head constraint, it cannot have two heads in $\mathcal{I}(c, t_G)$ because t_G must satisfy this constraint as well, and it cannot have one head in A and another in $\mathcal{I}(c, t_G)$ because the corresponding arc in $\mathcal{I}(c, t_G)$ would be unreachable due to the single-head constraint.

This, in turn, implies that the graph $A \cup \mathcal{I}(c, t_G)$ has no overlapping cycles, as overlapping cycles can only appear in graphs with in-degree greater than 1. This is the key property enabling us to exactly calculate loss using the number of cycles.

To show (1), consider the graph $A \cup \mathcal{I}(c, t_G)$. In each of its cycles, there is at least one arc that belongs to $\mathcal{I}(c, t_G)$, as A must satisfy the acyclicity constraint. We arbitrarily choose one such arc from each cycle, and remove it from the graph. Note that this results in removing exactly $n_c(A \cup \mathcal{I}(c, t_G))$ arcs, as we have shown that the cycles in $A \cup \mathcal{I}(c, t_G)$ are disjoint. We call the resulting graph $\mathcal{B}(c, t_G)$. As it has maximum in-degree 1 and it is acyclic (because we have broken all the cycles), $\mathcal{B}(c, t_G)$ is a tree, modulo our standard assumption that headless nodes are assumed to be linked to the dummy root.

This tree $\mathcal{B}(c, t_G)$ is reachable from c and has loss $\ell(c) = |\mathcal{U}(c, t_G)| + n_c(A \cup \mathcal{I}(c, t_G))$. Reachability is shown by building a sequence of trans-

itions that will visit the pairs of words corresponding to remaining arcs in order, and intercalating the corresponding Left-Arc or Right-Arc transitions, which cannot violate the acyclicity or single-head constraints. The term $\mathcal{U}(c, t_G)$ in the loss stems from the fact that $A \cup \mathcal{I}(c, t_G)$ cannot contain arcs in $\mathcal{U}(c, t_G)$, and the term $n_c(A \cup \mathcal{I}(c, t_G))$ from not including the $n_c(A \cup \mathcal{I}(c, t_G))$ arcs that we discarded to break cycles.

Finally, from these observations, it is easy to see that $\mathcal{B}(c, t_G)$ has the best loss among reachable trees, and thus prove (2): the arcs in $\mathcal{U}(c, t_G)$ are always unreachable by definition, and for each cycle in $n_c(A \cup \mathcal{I}(c, t_G))$, the acyclicity constraint forces us to miss at least one arc. As the cycles are disjoint, this means that we necessarily miss at least $n_c(A \cup \mathcal{I}(c, t_G))$ arcs, hence $|\mathcal{U}(c, t_G)| + n_c(A \cup \mathcal{I}(c, t_G))$ is indeed the minimum loss among reachable trees. □

Thus, to calculate the loss of a configuration c, we only need to compute both of the terms in Theorem 1. For the first term, note that if c has focus words i and j (i.e., $c = \langle \lambda_1 | i, \lambda_2, j | B, A \rangle$), then an arc $x \rightarrow y$ is in $\mathcal{U}(c, t_G)$ if it is not in A, and at least one of the following holds:

- $j > \max(x, y)$, as in this case we have read too far in the string and will not be able to get x and y as focus words,
- $j = \max(x, y) \wedge i < \min(x, y)$, as in this case we have $\max(x, y)$ as the right focus word but the left focus word is to the left of $\min(x, y)$, and we cannot move it back,
- there is some $z \neq 0, z \neq x$ such that $z \rightarrow y \in A$, as in this case the single-head constraint prevents us from creating $x \rightarrow y$,
- x and y are on the same weakly connected component of A, as in this case the acyclicity constraint will not let us create $x \rightarrow y$.

All of these arcs can be trivially enumerated in $O(n)$ time (in fact, they can be updated in $O(1)$ if we start from the configuration that preceded c). The second term of the loss, $n_c(A \cup \mathcal{I}(c, t_G))$, can be computed by obtaining $\mathcal{I}(c, t_G)$ as $t_G \setminus \mathcal{U}(c, t_G)$ to then apply a standard cycle-finding algorithm (Tarjan, 1972) which, for a graph with maximum in-degree 1, runs in $O(n)$ time.

Algorithm 1 presents the resulting loss calculation algorithm in pseudocode form, where COUNTCYCLES is a function that counts the number of cycles in the given graph in linear time as mentioned above. Note that the for loop runs in linear time: the condition on line 8 can be computed in constant time by recovering the head of y. The call to WEAKLYCONNECTED in line 9 finds out whether the two given nodes are weakly connected in A, and can also be resolved in $O(1)$, by querying the disjoint set data structure that implementations of the Covington algorithm commonly use for the parser's acyclicity checks (Nivre, 2008).

Algorithm 1 Computation of the loss of a configuration.

```
1:  function LOSS(c = ⟨λ1|i, λ2, j|B, A⟩, t_G)
2:      U ← ∅                    ▷ Variable U is for U(c, t_G)
3:      for each x → y ∈ (t_G \ A) do
4:          left ← min(x, y)
5:          right ← max(x, y)
6:          if j > right ∨
7:          (j = right ∧ i < left) ∨
8:          (∃z > 0, z ≠ x : z → y ∈ A) ∨
9:          WEAKLYCONNECTED(A, x, y) then
10:             U ← u ∪ {x → y}
11:     I ← t_G \ U      ▷ Variable I is for I(c, t_G)
12:     return |U| + COUNTCYCLES(A ∪ I)
```

It is worth noting that the linear-time complexity can also be achieved by a standalone implementation of the loss calculation algorithm, without recurse to the parser's auxiliary data structures (although this is dubiously practical). To do so, we can implement WEAKLYCONNECTED so that the first call computes the connected components of A in linear time (Hopcroft and Tarjan, 1973) and subsequent calls use this information to find out if two nodes are weakly connected in constant time.

On the other hand, a more efficient implementation than the one shown in Algorithm 1 (which we chose for clarity) can be achieved by more tightly coupling the oracle to the parser, as the relevant sets of arcs associated with a configuration can be obtained incrementally from those of the previous configuration.

4 Experiments

To evaluate the performance of our approach, we conduct experiments on both static and dynamic Covington non-projective oracles. Concretely, we train an averaged perceptron model for 15 iterations on nine datasets from the CoNLL-X shared task (Buchholz and Marsi, 2006) and all data-

Unigrams
L_0w; L_0p; L_0wp; L_0l; $L_{0h}w$; $L_{0h}p$; $L_{0h}l$; $L_{0l'}w$; $L_{0l'}p$; $L_{0l'}l$; $L_{0r'}w$; $L_{0r'}p$; $L_{0r'}l$; $L_{0h2}w$; $L_{0h2}p$; $L_{0h2}l$; $L_{0l}w$; $L_{0l}p$; $L_{0l}l$; $L_{0r}w$; $L_{0r}p$; $L_{0r}l$; L_0wd; L_0pd; L_0wv_r; L_0pv_r; L_0wv_l; L_0pv_l; L_0ws_l; L_0ps_l; L_0ws_r; L_0ps_r; L_1w; L_1p; L_1wp; R_0w; R_0p; R_0wp; $R_{0l'}w$; $R_{0l'}p$; $R_{0l'}l$; $R_{0l}w$; $R_{0l}p$; $R_{0l}l$; R_0wd; R_0pd; R_0wv_l; R_0pv_l;R_0ws_l; R_0ps_l; R_1w; R_1p; R_1wp; R_2w; R_2p; R_2wp; CLw; CLp; $CLwp$; CRw; CRp; $CRwp$;
Pairs
L_0wp+R_0wp; L_0wp+R_0w; L_0w+R_0wp; L_0wp+R_0p; L_0p+R_0wp; L_0w+R_0w; L_0p+R_0p;R_0p+R_1p; L_0w+R_0wd; L_0p+R_0pd;
Triples
R_0p+R_1p+R_2p; L_0p+R_0p+R_1p; $L_{0h}p$+L_0p+R_0p; L_0p+$L_{0l'}p$+R_0p; L_0p+$L_{0r'}p$+R_0p; L_0p+R_0p+$R_{0l'}p$; L_0p+$L_{0l'}p$+$L_{0l}p$; L_0p+$L_{0r'}p$+$L_{0r}p$; L_0p+$L_{0h}p$+$L_{0h2}p$; R_0p+$R_{0l'}p$+$R_{0l}p$;

Table 1: Feature templates. L_0 and R_0 denote the left and right focus words; $L_1, L_2, \ldots$ are the words to the left of L_0 and $R_1, R_2, \ldots$ those to the right of R_0. X_{ih} means the head of X_i, X_{ih2} the grandparent, X_{il} and $X_{il'}$ the farthest and closest left dependents, and X_{ir} and $X_{ir'}$ the farthest and closest right dependents, respectively. CL and CR are the first and last words between L_0 and R_0 whose head is not in the interval $[L_0, R_0]$. Finally, w stands for word form; p for PoS tag; l for dependency label; d is the distance between L_0 and R_0; v_l, v_r are the left/right valencies (number of left/right dependents); and s_l, s_r the left/right label sets (dependency labels of left/right dependents).

sets from the CoNLL-XI shared task (Nivre et al., 2007). We use the same feature templates for all languages, which result from adapting the features described by Zhang and Nivre (2011) to the data structures of the Covington non-projective parser, and are listed in detail in Table 1.

Table 2 reports the accuracy obtained by the Covington non-projective parser with both oracles. As we can see, the dynamic oracle implemented in the Covington algorithm improves over the accuracy of the static version on all datasets except Japanese and Swedish, and most improvements are statistically significant at the .05 level.[3]

In addition, the Covington dynamic oracle achieves a greater average improvement in accuracy than the Attardi dynamic oracle (Gómez-Rodríguez et al., 2014) over their respective static versions. Concretely, the Attardi oracle accomplishes an average improvement of 0.52 percentage points in UAS and 0.71 in LAS, while our approach achieves 0.80 in UAS and 0.74 in LAS.

[3]Note that the loss of accuracy in Japanese and Swedish is not statistically significant.

	s-Covington		d-Covington	
Language	UAS	LAS	UAS	LAS
Arabic	80.03	71.32	**81.47***	**72.77***
Basque	75.76	69.70	**76.49***	**70.27***
Catalan	88.66	83.92	**89.28**	**84.26**
Chinese	83.94	79.59	**84.68***	**80.16***
Czech	77.38	71.21	**78.58***	**72.59***
English	84.64	83.72	**86.14***	**84.96***
Greek	79.33	72.65	**80.52***	**73.67***
Hungarian	77.70	74.32	**78.22**	**74.61**
Italian	83.39	79.66	**83.66**	**79.91**
Turkish	82.14	76.00	**82.38**	**76.15**
Bulgarian	87.68	84.55	**88.48***	**85.32***
Danish	84.07	79.99	**84.98***	**80.85***
Dutch	80.28	77.55	**81.17***	**78.54***
German	86.12	83.93	**87.47***	**85.15***
Japanese	**93.92**	**92.51**	93.79	92.42
Portuguese	85.70	82.78	**86.23**	**83.27**
Slovene	75.31	68.97	**76.76***	**70.35***
Spanish	78.82	75.84	**79.87***	**76.97***
Swedish	**86.78**	**81.29**	86.66	81.21
Average	82.72	78.39	**83.52**	**79.13**

Table 2: Parsing accuracy (UAS and LAS, including punctuation) of Covington non-projective parser with static (s-Covington) and dynamic (d-Covington) oracles on CoNLL-XI (first block) and CoNLL-X (second block) datasets. For each language, we run five experiments with the same setup but different seeds and report the averaged accuracy. Best results for each language are shown in boldface. Statistically significant improvements ($\alpha = .05$) (Yeh, 2000) are marked with *.

5 Conclusion

We have defined the first dynamic oracle for a transition-based parser supporting unrestricted non-projectivity. The oracle is very efficient, computing loss in $O(n)$, compared to $O(n^8)$ for the only previously known dynamic oracle with support for a subset of non-projective trees (Gómez-Rodríguez et al., 2014).

Experiments on the treebanks from the CoNLL-X and CoNLL-XI shared tasks show that the dynamic oracle significantly improves accuracy on many languages over a static oracle baseline.

Acknowledgments

Research partially funded by the Spanish Ministry of Economy and Competitiveness/ERDF (grants FFI2014-51978-C2-1-R, FFI2014-51978-C2-2-R), Ministry of Education (FPU grant program) and Xunta de Galicia (grant R2014/034).

References

Giuseppe Attardi. 2006. Experiments with a multilanguage non-projective dependency parser. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL-X)*, pages 166–170, Morristown, NJ, USA. Association for Computational Linguistics.

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pages 149–164.

Jinho D. Choi and Andrew McCallum. 2013. Transition-based dependency parsing with selectional branching. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1052–1062, Sofia, Bulgaria.

Michael A. Covington. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102, New York, NY, USA. ACM.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of COLING 2012*, pages 959–976, Mumbai, India, December. Association for Computational Linguistics.

Yoav Goldberg and Joakim Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *Transactions of the Association for Computational Linguistics*, 1:403–414.

Yoav Goldberg, Francesco Sartorio, and Giorgio Satta. 2014. A tabular method for dynamic oracles in transition-based parsing. *Transactions of the Association for Computational Linguistics*, 2:119–130.

Carlos Gómez-Rodríguez and Joakim Nivre. 2013. Divisible transition systems and multiplanar dependency parsing. *Computational Linguistics*, 39(4):799–845.

Carlos Gómez-Rodríguez, Francesco Sartorio, and Giorgio Satta. 2014. A polynomial-time dynamic oracle for non-projective dependency parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 917–927. Association for Computational Linguistics.

John Hopcroft and Robert Endre Tarjan. 1973. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June.

Peter Neuhaus 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 (ACL) and the 8th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 337–343.

Joakim Nivre, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. 2007. The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 915–932, June.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, pages 149–160. ACL/SIGPARSE.

Joakim Nivre. 2008. Algorithms for Deterministic Incremental Dependency Parsing. *Computational Linguistics*, 34(4):513–553.

Robert Endre Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160.

Alexander Volokh and Günter Neumann. 2012. Dependency parsing with efficient feature extraction. In Birte Glimm and Antonio Krüger, editors, *KI*, volume 7526 of *Lecture Notes in Computer Science*, pages 253–256. Springer.

Alexander Volokh. 2013. *Performance-Oriented Dependency Parsing*. Doctoral dissertation, Saarland University, Saarbrücken, Germany.

Alexander Yeh. 2000. More accurate tests for the statistical significance of result differences. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING)*, pages 947–953.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers - Volume 2*, pages 188–193.

Parsing as Reduction

Daniel Fernández-González[†*] **André F. T. Martins**[‡#]
[†]Departamento de Informática, Universidade de Vigo, Campus As Lagoas, 32004 Ourense, Spain
[‡]Instituto de Telecomunicações, Instituto Superior Técnico, 1049-001 Lisboa, Portugal
[#]Priberam Labs, Alameda D. Afonso Henriques, 41, 2º, 1000-123 Lisboa, Portugal
danifg@uvigo.es, atm@priberam.pt

Abstract

We reduce phrase-based parsing to dependency parsing. Our reduction is grounded on a new intermediate representation, "head-ordered dependency trees," shown to be isomorphic to constituent trees. By encoding order information in the dependency labels, we show that any off-the-shelf, trainable dependency parser can be used to produce constituents. When this parser is non-projective, we can perform discontinuous parsing in a very natural manner. Despite the simplicity of our approach, experiments show that the resulting parsers are on par with strong baselines, such as the Berkeley parser for English and the best non-reranking system in the SPMRL-2014 shared task. Results are particularly striking for discontinuous parsing of German, where we surpass the current state of the art by a wide margin.

1 Introduction

Constituent parsing is a central problem in NLP—one at which statistical models trained on treebanks have excelled (Charniak, 1996; Klein and Manning, 2003; Petrov and Klein, 2007). However, most existing parsers are slow, since they need to deal with a heavy grammar constant. Dependency parsers are generally faster, but less informative, since they do not produce constituents, which are often required by downstream applications (Johansson and Nugues, 2008; Wu et al., 2009; Berg-Kirkpatrick et al., 2011; Elming et al., 2013). How to get the best of both worlds?

Coarse-to-fine decoding (Charniak and Johnson, 2005) and shift-reduce parsing (Sagae and Lavie, 2005; Zhu et al., 2013) were a step forward to accelerate constituent parsing, but typical runtimes still lag those of dependency parsers. This is only made worse if **discontinuous** constituents are allowed—such discontinuities are convenient to represent wh-movement, scrambling, extraposition, and other linguistic phenomena common in free word order languages. While non-projective dependency parsers, which are able to model such phenomena, have been widely developed in the last decade (Nivre et al., 2007; McDonald et al., 2006; Martins et al., 2013), discontinuous constituent parsing is still taking its first steps (Maier and Søgaard, 2008; Kallmeyer and Maier, 2013).

In this paper, we show that an off-the-shelf, trainable, **dependency parser** is enough to build a highly-competitive constituent parser. This (surprising) result is based on a **reduction**[1] of constituent to dependency parsing, followed by a simple post-processing procedure to recover unaries. Unlike other constituent parsers, ours does not require estimating a grammar, nor binarizing the treebank. Moreover, when the dependency parser is non-projective, our method can perform discontinuous constituent parsing in a very natural way.

Key to our approach is the notion of **head-ordered dependency trees** (shown in Figure 1): by endowing dependency trees with this additional layer of structure, we show that they become isomorphic to constituent trees. We encode this structure as part of the dependency labels, enabling a dependency-to-constituent conversion. A related conversion was attempted by Hall and Nivre (2008) to parse German, but their complex encoding scheme blows up the number of arc labels, affecting the final parser's quality. By contrast, our light encoding achieves a 10-fold decrease in the label alphabet, leading to more accurate parsing.

While simple, our reduction-based parsers are on par with the Berkeley parser for English (Petrov

[*]This research was carried out during an internship at Priberam Labs.

[1]The title of this paper is inspired by the seminal paper of Pereira and Warren (1983) "Parsing as Deduction."

and Klein, 2007), and with the best single system in the recent SPMRL shared task (Seddah et al., 2014), for eight morphologically rich languages. For discontinuous parsing, we surpass the current state of the art by a wide margin on two German datasets (TIGER and NEGRA), while achieving fast parsing speeds. We provide a free distribution of our parsers along with this paper, as part of the TurboParser toolkit.[2]

2 Background

We start by reviewing constituent and dependency representations, and setting up the notation. Following Kong and Smith (2014), we use c-/d- prefixes for convenience (*e.g.*, we write c-parser for constituent parser and d-tree for dependency tree).

2.1 Constituent Trees

Constituent-based representations are commonly seen as derivations according to a context-free grammar (CFG). Here, we focus on properties of the c-trees, rather than of the grammars used to generate them. We consider a broad scenario that permits c-trees with discontinuities, such as the ones derived with linear context-free rewriting systems (LCFRS; Vijay-Shanker et al. (1987)). We also assume that the c-trees are lexicalized.

Formally, let $w_1 w_2 \dots w_L$ be a sentence, where w_i denotes the word in the ith position. A **c-tree** is a rooted tree whose leaves are the words $\{w_i\}_{i=1}^{L}$, and whose internal nodes (constituents) are represented as a tuple $\langle Z, h, \mathcal{I} \rangle$, where Z is a non-terminal symbol, $h \in \{1, \dots, L\}$ indicates the lexical head, and $\mathcal{I} \subseteq \{1, \dots, L\}$ is the node's yield. Each word's parent is a pre-terminal unary node of the form $\langle p_i, i, \{i\} \rangle$, where p_i denotes the word's part-of-speech (POS) tag. The yields and lexical heads are defined so that for every constituent $\langle Z, h, \mathcal{I} \rangle$ with children $\{\langle X_k, m_k, \mathcal{J}_k \rangle\}_{k=1}^{K}$, (i) we have $\mathcal{I} = \bigcup_{k=1}^{K} \mathcal{J}_k$; and (ii) there is a unique k such that $h = m_k$. This kth node (called the head-child node) is commonly chosen applying a handwritten set of head rules (Collins, 1999; Yamada and Matsumoto, 2003).

A c-tree is **continuous** if all nodes $\langle Z, h, \mathcal{I} \rangle$ have a contiguous yield $\mathcal{I}$, and **discontinuous** otherwise. Trees derived by a CFG are always continuous; those derived by a LCFRS may have discontinuities, the yield of a node being a union of spans, possibly with gaps in the middle. Figure 1 shows an example of a continuous and a discontinuous c-tree. Discontinuous c-trees have crossing branches, if the leaves are drawn in left-to-right surface order. An internal node which is not a pre-terminal is called a **proper node**. A node is called unary if it has exactly one child. A c-tree without unary proper nodes is called **unaryless**. If all proper nodes have exactly two children then it is called a **binary** c-tree. Continuous binary trees may be regarded as having been generated by a CFG in Chomsky normal form.

Prior work. There has been a long string of work in statistical c-parsing, shifting from simple models (Charniak, 1996) to more sophisticated ones using structural annotation (Johnson, 1998; Klein and Manning, 2003), latent grammars (Matsuzaki et al., 2005; Petrov and Klein, 2007), and lexicalization (Eisner, 1996; Collins, 1999). An orthogonal line of work uses ensemble or reranking strategies to further improve accuracy (Charniak and Johnson, 2005; Huang, 2008; Björkelund et al., 2014). Discontinuous c-parsing is considered a much harder problem, involving mildly context-sensitive formalisms such as LCFRS or range concatenation grammars, with treebank-derived c-parsers exhibiting near-exponential runtime (Kallmeyer and Maier, 2013, Figure 27). To speed up decoding, prior work has considered restrictons, such as bounding the fan-out (Maier et al., 2012) and requiring well-nestedness (Kuhlmann and Nivre, 2006; Gómez-Rodríguez et al., 2010). Other approaches eliminate the discontinuities via tree transformations (Boyd, 2007; Kübler et al., 2008), sometimes as a pruning step in a coarse-to-fine parsing approach (van Cranenburgh and Bod, 2013). However, reported runtimes are still superior to 10 seconds per sentence, which is not practical. Recently, Versley (2014a) proposed an easy-first approach that leads to considerable speed-ups, but is less accurate. In this paper, we design fast discontinuous c-parsers that outperform all the ones above by a wide margin, with similar runtimes as Versley (2014a).

2.2 Dependency Trees

In this paper, we use d-parsers as a black box to parse constituents. Given a sentence $w_1 \dots w_L$, a **d-tree** is a directed tree spanning all the words in the sentence.[3] Each arc in this tree is a tuple

[2] `http://www.ark.cs.cmu.edu/TurboParser`

[3] We assume throughout that dependency trees have a single root among $\{w_1, \dots, w_L\}$. Therefore, there is no need to

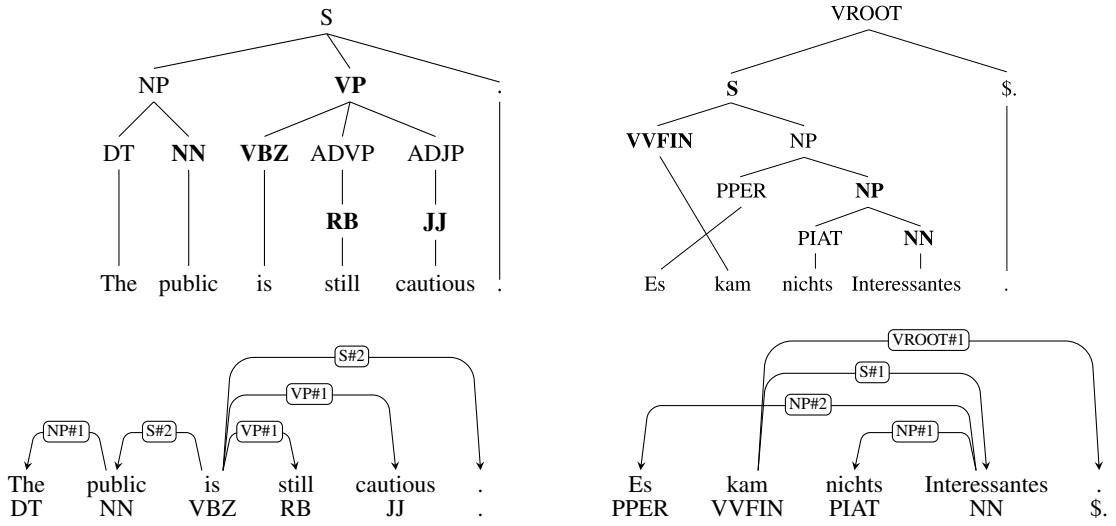


Figure 1: Top: a continuous (left) and a discontinuous (right) c-tree, taken from English PTB §22 and German NEGRA, respectively. Head-child nodes are in bold. Bottom: corresponding head-ordered d-trees. The indices #1, #2, etc. denote the order of attachment events for each head. Note that the English unary nodes ADVP and ADJP are dropped in the conversion.

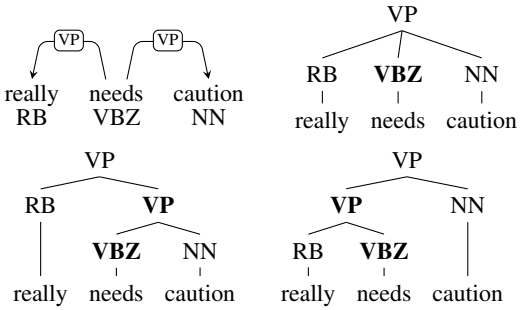


Figure 2: Three different c-structures for the VP "*really needs caution*." All are consistent with the d-structure at the top left.

$\langle h, m, \ell \rangle$, expressing a typed dependency relation ℓ between the head word w_h and the modifier w_m.

A d-tree is **projective** if for every arc $\langle h, m, \ell \rangle$ there is a directed path from h to all words that lie between h and m in the surface string (Kahane et al., 1998). Projective d-trees can be obtained from continuous c-trees by reading off the lexical heads and dropping the internal nodes (Gaifman, 1965). However, this relation is many-to-one: as shown in Figure 2, several c-trees may project onto the same d-tree, differing on their flatness and on left or right-branching decisions. In the next section, we introduce the concept of head-ordered d-trees and express one-to-one mappings between these two representations.

Prior work. There has been a considerable amount of work developing rich-feature d-parsers. While projective d-parsers can use dynamic programming (Eisner and Satta, 1999; Koo and Collins, 2010), non-projective d-parsers typically rely on approximate decoders, since the underlying problem is NP-hard beyond arc-factored models (McDonald and Satta, 2007). An alternative are transition-based d-parsers (Nivre et al., 2006; Zhang and Nivre, 2011), which achieve observed linear time. Since d-parsing algorithms do not have a grammar constant, typical implementations are significantly faster than c-parsers (Rush and Petrov, 2012; Martins et al., 2013). The key contribution of this paper is to reduce c-parsing to d-parsing, allowing to bring these runtimes closer.

consider an extra root symbol, as often done in the literature.

3 Head-Ordered Dependency Trees

We next endow d-trees with another layer of structure, namely **order information**. In this framework, not all modifiers of a head are "born equal." Instead, their attachment to the head occurs as a sequence of "events," which reflect the head's preference for attaching some modifiers before others. As we will see, this additional structure will undo the ambiguity expressed in Figure 2.

3.1 Strictly Ordered Dependency Trees

Let us start with the simpler case where the attachment order is strict. For each head word h with modifiers $M_h = \{m_1, \ldots, m_K\}$, we endow M_h with a **strict order relation** $\prec_h$, so we can organize all the modifiers of h as a chain, $m_{i_1} \prec_h m_{i_2} \prec_h \ldots \prec_h m_{i_K}$. We regard this chain as reflecting the order by which words are attached (*i.e.*, if $m_i \prec_h m_j$ this means that "m_i is attached

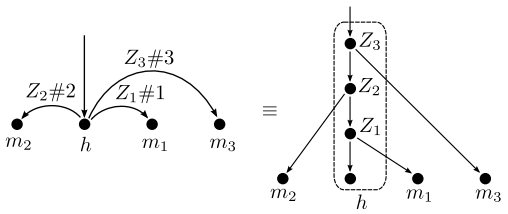


Figure 3: Transformation of a strictly-ordered d-tree into a binary c-tree. Each node is split into a linked list forming a spine, to which modifiers are attached in order.

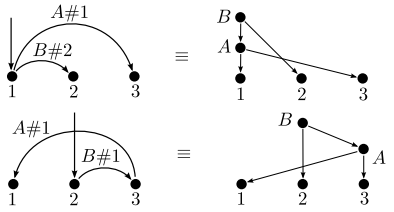


Figure 4: Two discontinuous constructions caused by a non-nested order (top) and a non-projective d-tree (bottom). In both cases node A has a non-contiguous yield.

to h before m_j"). We represent this graphically by decorating d-arcs with indices ($\#1, \#2, \ldots$) to denote the order of events, as we do in Figure 1.

A d-tree endowed with a strict order for each head is called a **strictly ordered d-tree**. We establish below a correspondence between strictly ordered d-trees and binary c-trees. Before doing so, we need a few more definitions about c-trees. For each word position $h \in \{1, \ldots, L\}$, we define $\psi(h)$ as the node higher in the c-tree whose lexical head is h. We call the path from $\psi(h)$ down to the pre-terminal p_h the **spine** of h. We may regard a c-tree as a set of L spines, one per word, which attach to each other to form a tree (Carreras et al., 2008). We then have the following

Proposition 1. *Binary c-trees and strictly-ordered d-trees are isomorphic, i.e., there is a one-to-one correspondence between the two sets, where the number of symbols is preserved.*

Proof. We use the construction in Figure 3. A formal proof is given as supplementary material. □

3.2 Weakly Ordered Dependency Trees

Next, we relax the strict order assumption, restricting the modifier sets $M_h = \{m_1, \ldots, m_K\}$ to be only **weakly ordered**. This means that we can partition the K modifiers into J equivalence classes, $M_h = \bigcup_{j=1}^{J} \bar{M}_h^j$, and define a strict order $\prec_h$ on the quotient set: $\bar{M}_h^1 \prec_h \ldots \prec_h \bar{M}_h^J$. Intuitively, there is still a sequence of events (1 to J), but now at each event j it may happen that multiple modifiers (the ones in the equivalence set $\bar{M}_h^j$) are simultaneously attached to h. A **weakly ordered d-tree** is a d-tree endowed with a weak order for each head and such that any pair m, m' in the same equivalence class (written $m \equiv_h m'$) receive the same dependency label ℓ.

Algorithm 1 Conversion from c-tree to d-tree

Input: c-tree $\mathcal{C}$.
Output: head-ordered d-tree $\mathcal{D}$.
1: Nodes := GETPOSTORDERTRAVERSAL($\mathcal{C}$).
2: Set $j(h) := 1$ for every $h = 1, \ldots, L$.
3: **for** $v := \langle Z, h, \mathcal{I} \rangle \in$ Nodes **do**
4: **for** every $u := \langle X, m, \mathcal{J} \rangle$ which is a child of v **do**
5: **if** $m \neq h$ **then**
6: Add to $\mathcal{D}$ an arc $\langle h, m, Z \rangle$, and put it in $\bar{M}_h^{j(h)}$.
7: **end if**
8: **end for**
9: Set $j(h) := j(h) + 1$.
10: **end for**

We now show that Proposition 1 can be generalized to weakly ordered d-trees.

Proposition 2. *Unaryless c-trees and weakly-ordered d-trees are isomorphic.*

Proof. This is a simple extension of Proposition 1. The construction is the same as in Figure 3, but now we can collapse some of the nodes in the linked list, originating multiple modifiers attaching to the same position of the spine—this is only possible for sibling arcs with the same index and arc label. Note, however, that if we start with a c-tree with unary nodes and apply the inverse procedure to obtain a d-tree, the unary nodes will be lost, since they do not involve attachment of modifiers. In a chain of unary nodes, only the last node is recovered in the inverse transformation. □

We emphasize that Propositions 1–2 hold without blowing up the number of symbols. That is, the dependency label alphabet is exactly the same as the set of phrasal symbols in the constituent representations. Algorithms 1–2 convert back and forth between the two formalisms, performing the construction of Figure 3. Both algorithms run in linear time with respect to the size of the sentence.

3.3 Continuous and Projective Trees

What about the more restricted class of projective d-trees? Can we find an equivalence relation with continuous c-trees? In this section, we give a precise answer to this question. It turns out that we need an additional property, illustrated in Figure 4.

We say that $\prec_h$ has the **nesting property** iff closer words in the same direction are always attached first, *i.e.*, iff $h < m_i < m_j$ or $h > m_i >$

Algorithm 2 Conversion from d-tree to c-tree

Input: head-ordered d-tree $\mathcal{D}$.
Output: c-tree $\mathcal{C}$.

```
Nodes := GetPostOrderTraversal(D).
for h ∈ Nodes do
    Create v := ⟨p_h, h, {h}⟩ and set ψ(h) := v.
    Sort M_h(D), yielding M̄_h^1 ≺_h M̄_h^2 ≺_h ... ≺_h M̄_h^J.
    for j = 1, ..., J do
        Let Z be the label in {⟨h, m, Z⟩ | m ∈ M̄_h^j}.
        Obtain c-nodes ψ(h) = ⟨X, h, I⟩ and ψ(m) = ⟨Y_m, m, J_m⟩ for all m ∈ M̄_h^j.
        Add c-node v := ⟨Z, h, I ∪ ⋃_{m∈M̄_h^j} J_m⟩ to C.
        Set ψ(h) and {ψ(m) | m ∈ M̄_h^j} as children of v.
        Set ψ(h) := v.
    end for
end for
```

m_j implies that either $m_i \equiv_h m_j$ or $m_i \prec_h m_j$. A weakly-ordered d-tree which is projective and whose orders $\prec_h$ have the nesting property for every h is called a **nested-weakly ordered projective d-tree**. We then have the following result.

Proposition 3. *Continuous unaryless c-trees and nested-weakly ordered projective d-trees are isomorphic.*

Proof. See the supplementary material. □

Together, Propositions 1–3 have as corollary that nested-strictly ordered projective d-trees are in a one-to-one correspondence with binary continuous c-trees. The intuition is simple: if $\prec_h$ has the nesting property, then, at each point in time, all one needs to decide about the next event is whether to attach the closest available modifier on the *left* or on the *right*. This corresponds to choosing between left-branching or right-branching in a c-tree. While this is potentially interesting for most continuous c-parsers, which work with binarized c-trees when running the CKY algorithm, our c-parsers (to be described in §4) do not require any binarization since they work with weakly-ordered d-trees, using Proposition 2.

4 Reduction-Based Constituent Parsers

We now invoke the equivalence results established in §3 to build c-parsers when only a trainable d-parser is available. Given a c-treebank provided as input, our procedure is outlined as follows:

1. Convert the c-treebank to dependencies (Algorithm 1).
2. Train a labeled d-parser on this treebank.
3. For each test sentence, run the labeled d-parser and convert the predicted d-tree into a c-tree without unary nodes (Algorithm 2).
4. Do post-processing to recover unaries.

The next subsections describe each of these steps in detail. Along the way, we illustrate with experiments using the English Penn Treebank (Marcus et al., 1993), which we lexicalized by applying the head rules of Collins (1999).[4]

4.1 Dependency Encoding

The first step is to convert the c-treebank to head-ordered dependencies, which we do using Algorithm 1. If the original treebank has discontinuous c-trees, we end up with non-projective d-trees or with violations of the nested property, as established in Proposition 3. We handle this gracefully by training a non-projective d-parser in the subsequent stage (see §4.2). Note also that this conversion drops the unary nodes (a consequence of Proposition 2). These nodes will be recovered in the last stage, as described in §4.4.

Since in this paper we are assuming that only an off-the-shelf d-parser is available, we need to convert head-ordered d-trees to plain d-trees. We do so by encoding the order information in the dependency labels. We tried two different strategies. The first one, **direct encoding**, just appends suffixes #1, #2, etc., as in Figure 1. A disadvantage is that the number of labels grows unbounded with the treebank size, as we may encounter complex substructures where the event sequences are long. The second strategy is a **delta-encoding** scheme where, rather than writing the absolute indices in the dependency label, we write the *differences* between consecutive ones.[5] We used this strategy for the continuous treebanks only, whose d-trees are guaranteed to satisfy the nested property.

For comparison, we also implemented a replication of the encoding proposed by Hall and Nivre (2008), which we call **H&N-encoding**. This strategy concatenates all the c-nodes' symbols in the modifier's spine with the attachment position in the head's spine (*e.g.*, in Figure 3, if the modifier m_2 has a spine with nodes X_1, X_2, X_3, the generated d-label would be $\texttt{X}_1|\texttt{X}_2|\texttt{X}_3\#2$; our direct encoding scheme generates $\texttt{Z}_2\#2$ instead). Since their strategy encodes the entire spines into com-

[4] We train on §02–21, use §22 for validation, and test on §23. We predict automatic POS tags with *TurboTagger* (Martins et al., 2013), with 10-fold jackknifing on the training set.

[5] For example, if $\#1, \#3, \#4$ and $\#2, \#3, \#3, \#5$ are respectively the sequence of indices from the head to the left and to the right, we encode these sequences as $\#1, \#2, \#1$ and $\#2, \#1, \#0, \#2$ (using 3 distinct indices instead of 5).

plex arc labels, many such labels will be generated, leading to slower runtimes and poorer generalization, as we will see.

For the training portion of the English PTB, which has 27 non-terminal symbols, the direct encoding strategy yields 75 labels, while delta encoding yields 69 labels (2.6 indices per symbol). By contrast, the H&N-encoding procedure yields 731 labels, more than 10 times as many. We later show (in Tables 1–2) that delta-encoding leads to a slightly higher c-parsing accuracy than direct encoding, and that both strategies are considerably more accurate than H&N-encoding.

4.2 Training the Labeled Dependency Parser

The next step is to train a labeled d-parser on the converted treebank. If we are doing continuous c-parsing, we train a projective d-parser; otherwise we train a non-projective one.

In our experiments, we found it advantageous to perform labeled d-parsing in two stages, as done by McDonald et al. (2006): first, train an unlabeled d-parser; then, train a dependency labeler.[6] Table 1 compares this approach against a one-shot strategy, experimenting with various off-the-shelf d-parsers: *MaltParser* (Nivre et al., 2007), *MSTParser* (McDonald et al., 2005), *ZPar* (Zhang and Nivre, 2011), and *TurboParser* (Martins et al., 2013), all with the default settings. For *TurboParser*, we used basic, standard and full models.

Our separate d-labeler receives as input a backbone d-structure and predicts a label for each arc. For each head h, we predict the modifiers' labels using a simple sequence model, with features of the form $\boldsymbol{\phi}(h, m, \ell)$ and $\boldsymbol{\phi}(h, m, m', \ell, \ell')$, where m and m' are two consecutive modifiers (possibly on opposite sides of the head) and ℓ and ℓ' are their labels. We use the same arc label features $\boldsymbol{\phi}(h, m, \ell)$ as *TurboParser*. For $\boldsymbol{\phi}(h, m, m', \ell, \ell')$, we use the POS triplet $\langle p_h, p_m, p_{m'} \rangle$, plus unilexical features where each of the three POS is replaced by the word form. Both features are conjoined with the label pair ℓ and ℓ'. Decoding under this model can be done by running the Viterbi algorithm independently for each head. The runtime is almost negligible compared with the time to parse: it took 2.1 seconds to process PTB §22, a fraction of about 5% of the total runtime.

[6] The reason why a two-stage approach is preferable is that one-shot d-parsers, for efficiency reasons, use label features parsimoniously. However, for our reduction approach, d-labels are crucial and strongly interdependent, since they jointly encode the c-structure.

Dependency Parser	UAS	LAS	F_1	# toks/s.
MaltParser	90.93	88.95	86.87	5,392
MSTParser	92.17	89.86	87.93	363
ZPar	92.93	91.28	89.50	1,022
TP-Basic	92.13	90.23	87.63	2,585
TP-Standard	93.55	91.58	90.41	1,658
TP-Full	93.70	91.70	90.53	959
TP-Full + Lab., H&N enc.	93.80	87.86	89.39	871
TP-Full + Lab, direct enc.	93.80	91.99	90.89	912
TP-Full + Lab., delta enc.	**93.80**	**92.00**	**90.94**	912

Table 1: Results on English PTB §22 achieved by various d-parsers and encoding strategies. For dependencies, we report unlabeled/labeled attachment scores (UAS/LAS), excluding punctuation. For constituents, we show F_1-scores (without punctuation and root nodes), as provided by `EVALB` (Black et al., 1992). We report total parsing speeds in tokens per second (including time spent on pruning, decoding, and feature evaluation), measured on a Intel Xeon processor @2.30GHz.

	direct enc.		**delta enc.**	
	# labels	**F_1**	**# labels**	**F_1**
Basque	26	85.04	17	85.17
French	61	79.93	56	80.05
German	66	83.44	59	83.39
Hebrew	62	83.26	43	83.29
Hungarian	24	86.54	15	86.67
Korean	44	79.79	16	79.97
Polish	47	92.39	34	92.64
Swedish	29	77.02	25	77.19

Table 2: Impact of direct and delta encodings on the dev sets of the SPMRL14 shared task. Reported are the number of labels and the F_1-scores yielded by each encoding technique.

4.3 Decoding into Unaryless Constituents

After training the labeled d-parser, we can run it on the test data. Then, we need to convert the predicted d-tree into a c-tree without unaries.

To accomplish this step, we first need to recover, for each head h, the weak order of its modifiers M_h. We do this by looking at the predicted dependency labels, extracting the event indices j, and using them to build and sort the equivalent classes $\{\bar{M}_h^j\}_{j=1}^J$. If two modifiers have the same index j, we force them to have consistent labels (by always choosing the label of the modifier which is the closest to the head). For continuous c-parsing, we also decrease the index j of the modifier closer to the head as much as necessary to make sure that the nesting property holds. In PTB §22, these corrections were necessary only for 0.6% of the tokens. Having done this, we use Algorithm 2 to obtain a predicted c-tree without unary nodes.

4.4 Recovery of Unary Nodes

The last stage is to recover the unary nodes. Given a unaryless c-tree as input, we predict unaries by running independent classifiers at each node in the tree (a simple unstructured task). Each class is either NULL (in which case no unary node is appended to the current node) or a concatenation of unary node labels (*e.g.*, `S->ADJP` for a node `JJ`). We obtained 64 classes by processing the training sections of the PTB, the fraction of unary nodes being about 11% of the total number of nodes. To reduce complexity, for each node symbol we only consider classes that have been observed with that symbol in the training data. In PTB §22, this yields an average of 9.9 candidates per node occurrence.

The classifiers are trained on the original c-treebank, stripping off unary nodes and trained to recover those nodes. We used the following features (conjoined with the class and with a flag indicating if the node is a pre-terminal):

- The production rules above and beneath the node (*e.g.*, `S->NP VP` and `NP->DT NN`);
- The node's label, alone and conjoined with the parent's label or the left/right sibling's label;
- The leftmost and rightmost word/lemma/POS tag/morpho-syntactic tags in the node's yield;
- If the left/right node is a pre-terminal, the word/lemma/morpho-syntactic tags beneath.

This is a relatively easy task: when gold unaryless c-trees are provided as input, we obtain an EVALB F_1-score of 99.43%. This large figure is due to the small amount of unary nodes, making this module have less impact on the final parser than the d-parser. Being a lightweight unstructured task, this step took only 0.7 seconds to run on PTB §22, a tiny fraction (less than 2%) of the total runtime.

Table 1 shows the accuracies obtained with the d-parser followed by the unary predictor. Since two-stage TP-Full with delta-encoding is the best strategy, we use this configuration in the sequel. To further explore the impact of delta encoding, we report in Table 2 the scores obtained by direct and delta encodings on eight other treebanks (see §5.2 for details on these datasets). With the exception of German, in all cases the delta encoding yielded better EVALB F_1-scores with fewer labels.

5 Experiments

To evaluate the performance of our reduction-based parsers, we conduct experiments in a variety of treebanks, both continuous and discontinuous.

Parser	LR	LP	F1	#Toks/s.
Charniak (2000)	89.5	89.9	89.5	–
Klein and Manning (2003)	85.3	86.5	85.9	143
Petrov and Klein (2007)	90.0	90.3	90.1	169
Carreras et al. (2008)	90.7	91.4	91.1	–
Zhu et al. (2013)	90.3	90.6	90.4	1,290
Stanford Shift-Reduce (2014)	89.1	89.1	89.1	655
Hall et al. (2014)	88.4	88.8	88.6	12
This work	89.9	90.4	90.2	957
Charniak and Johnson (2005)*	91.2	91.8	91.5	84
Socher et al. (2013)*	89.1	89.7	89.4	70
Zhu et al. (2013)*	91.1	91.5	91.3	–

Table 3: Results on the English PTB §23. All systems reporting runtimes were run on the same machine. Marked as * are reranking and semi-supervised c-parsers.

5.1 Results on the English PTB

Table 3 shows the accuracies and speeds achieved by our system on the English PTB §23, in comparison to state-of-the-art c-parsers. We can see that our simple reduction-based c-parser surpasses the three Stanford parsers (Klein and Manning, 2003; Socher et al., 2013, and Stanford Shift-Reduce), and is on par with the Berkeley parser (Petrov and Klein, 2007), while being more than 5 times faster.

The best supervised competitor is the recent shift-reduce parser of Zhu et al. (2013), which achieves similar, but slightly better, accuracy and speed. Our technique has the advantage of being flexible: since the time for d-parsing is the dominating factor (see §4.4), plugging a faster d-parser automatically yields a faster c-parser. While reranking and semi-supervised systems achieve higher accuracies, this aspect is orthogonal, since the same techniques can be applied to our parser.

5.2 Results on the SPMRL Datasets

We experimented with datasets for eight languages, from the SPMRL14 shared task (Seddah et al., 2014). We used the official training, development and test sets with the provided predicted POS tags. For French and German, we used the lexicalization rules detailed in Dybro-Johansen (2004) and Rehbein (2009), respectively. For Basque, Hungarian and Korean, we always took the rightmost modifier as head-child node. For Hebrew and Polish we used the leftmost modifier instead. For Swedish we induced head rules from the provided dependency treebank, as described in Versley (2014b). These choices were based on dev-set experiments.

Table 4 shows the results. For all languages ex-

cept French, our system outperforms the Berkeley parser (Petrov and Klein, 2007), with or without prescribed POS tags. Our average F_1-scores are superior to the best non-reranking system participating in the shared task (Crabbé and Seddah, 2014) and to the c-parser of Hall et al. (2014), achieving the best results for 4 out of 8 languages.

5.3 Results on the Discontinuous Treebanks

Finally, we experimented on two widely-used discontinuous German treebanks: TIGER (Brants et al., 2002) and NEGRA (Skut et al., 1997). For the former, we used two different splits: TIGER-SPMRL, provided in the SPMRL14 shared task; and TIGER-H&N, used by Hall and Nivre (2008). For NEGRA, we used the standard splits. In these experiments, we skipped the unary recovery stage, since very few unary nodes exist in the data.[7] We ran *TurboTagger* to predict POS tags for TIGER-H&N and NEGRA, while in TIGER-SPMRL we used the predicted POS tags provided in the shared task. All treebanks were lexicalized using the head-rule sets of Rehbein (2009). For comparison to related work, sentence length cut-offs of 30, 40 and 70 were applied during the evaluation.

Table 5 shows the results. We observe that our approach outperforms all the competitors considerably, achieving state-of-the-art accuracies for both datasets. The best competitor, van Cranenburgh and Bod (2013), is more than 3 points behind, both in TIGER-H&N and in NEGRA. Our reduction-based parsers are also much faster: van Cranenburgh and Bod (2013) report 3 hours to parse NEGRA with $L \leq 40$. Our system parses all NEGRA sentences (regardless of length) in 27.1 seconds in a single core, which corresponds to a rate of 618 tokens per second. This approaches the speed of the easy-first system of Versley (2014a), who reports runtimes in the range 670–920 tokens per second, but is much less accurate.

TIGER-SPMRL	$L \leq 70$	all
V14b, *gold*	76.46 / 41.05	76.11 / 40.94
Ours, *gold*	**80.98 / 43.44**	**80.62 / 43.32**
V14b, *pred*	73.90 / 37.00	– / –
Ours, *pred*	**77.72 / 38.75**	**77.32 / 38.64**
TIGER-H&N	$L \leq 40$	all
HN08, *gold*	79.93 / 37.78	– / –
V14a, *gold*	74.23 / 37.32	– / –
Ours, *gold*	**85.53 / 51.21**	**84.22 / 49.63**
HN08, *pred*	75.33 / 32.63	– / –
CB13, *pred*	78.8– / 40.8–	– / –
Ours, *pred*	**82.57 / 45.93**	**81.12 / 44.48**
NEGRA	$L \leq 30$	$L \leq 40$	all
M12, *gold*	74.5– / –	– / –	– / –
C12, *gold*	– / –	72.33 / 33.16	71.08 / 32.10
KM13, *gold*	75.75 / –	– / –	– / –
CB13, *gold*	– / –	76.8– / 40.5–	– / –
Ours, *gold*	**82.56 / 52.13**	**81.08 / 48.04**	**80.52 / 46.70**
CB13, *pred*	– / –	74.8– / 38.7–	– / –
Ours, *pred*	**79.63 / 48.43**	**77.93 / 44.83**	**76.95 / 43.50**

Table 5: F_1 / exact match scores on TIGER and NEGRA test sets, with gold and predicted POS tags. These scores are computed by the DISCO-DOP evaluator ignoring root nodes and, for TIGER-H&N and NEGRA, punctuation tokens. The baselines are published results by Hall and Nivre 2008 (HN08), Maier et al. 2012 (M12), van Cranenburgh 2012 (C12), Kallmeyer and Maier 2013 (KM13), van Cranenburgh and Bod 2013 (CB13), and Versley 2014a, 2014b (V14a, V14b).

6 Related Work

Conversions between constituents and dependencies have been considered by De Marneffe et al. (2006) in one direction, and by Collins et al. (1999) and Xia and Palmer (2001) in the other, toward multi-representational treebanks (Xia et al., 2008). This prior work aimed at linguistically sound conversions, involving grammar-specific transformation rules to handle the kind of ambiguities expressed in Figure 2. Our work differs in that we are not concerned about the linguistic plausibility of our conversions, but only with the formal aspects that underlie the two representations.

The work most related to ours is Hall and Nivre (2008), who also convert dependencies to constituents to prototype a c-parser for German. Their encoding strategy is compared to ours in §4.1: they encode the entire spines into the dependency labels, which become rather complex and numerous. A similar strategy has been used by Versley (2014a) for discontinuous c-parsing. Both are largely outperformed by our system, as shown in §5.3. The crucial difference is that we encode only the top node's label and its position in the spine—besides being a much lighter representation, ours has an interpretation as a weak ordering, leading to the isomorphisms expressed in Propositions 1–3.

Joint constituent and dependency parsing have been tackled by Carreras et al. (2008) and Rush et al. (2010), but the resulting parsers, while accurate, are more expensive than a single c-parser. Very recently, Kong et al. (2015) proposed a much cheaper pipeline in which d-parsing is performed first, followed by a c-parser constrained to be con-

[7] NEGRA has no unaries; for the TIGER-SPMRL and H&N dev-sets, the fraction of unaries is 1.45% and 1.01%.

Parser	Basque	French	German	Hebrew	Hungar.	Korean	Polish	Swedish	Avg.
Berkeley	70.50	**80.38**	78.30	86.96	81.62	71.42	79.23	79.19	78.45
Berkeley Tagged	74.74	79.76	78.28	85.42	85.22	78.56	86.75	80.64	81.17
Hall et al. (2014)	83.39	79.70	78.43	87.18	**88.25**	**80.18**	90.66	82.00	83.72
Crabbé and Seddah (2014)	85.35	79.68	77.15	86.19	87.51	79.35	**91.60**	82.72	83.69
This work	**85.90**	78.75	**78.66**	**88.97**	88.16	79.28	91.20	**82.80**	**84.22**
Björkelund et al. (2014)	88.24	82.53	81.66	89.80	91.72	83.81	90.50	85.50	86.72

Table 4: F_1-scores on eight treebanks of the SPMRL14 shared task, computed with the provided `EVALB_SPMRL` tool, which takes into account all tokens except root nodes. Berkeley Tagged is a version of Petrov and Klein (2007) using the predicted POS tags provided by the organizers. Crabbé and Seddah (2014) is the best non-reranking system in the shared task, and Björkelund et al. (2014) the ensemble and reranking-based system which won the official task. We report their published scores.

sistent with the predicted d-structure. Our work differs in which we do not need to run a c-parser in the second stage—instead, the d-parser already stores constituent information in the arc labels, and the only necessary post-processing is to recover unary nodes. Another advantage of our method is that it can be readily used for discontinuous parsing, while their constrained CKY algorithm can only produce continuous parses.

7 Conclusion

We proposed a reduction technique that allows to implement a c-parser when only a d-parser is given. The technique is applicable to any d-parser, regardless of its nature or kind. This reduction was accomplished by endowing d-trees with a weak order relation, and showing that the resulting class of head-ordered d-trees is isomorphic to constituent trees. We showed empirically that the our reduction leads to highly-competitive c-parsers for English and for eight morphologically rich languages; and that it outperforms the current state of the art in discontinuous parsing of German.

Acknowledgments

We would like to thank the three reviewers for their insightful comments, and Slav Petrov, Djamé Seddah, Yannick Versley, David Hall, Muhua Zhu, Lingpeng Kong, Carlos Gómez-Rodríguez, and Andreas van Cranenburgh for valuable feedback and help in preparing data and running software code. This research has been partially funded by the Spanish Ministry of Economy and Competitiveness and FEDER (project TIN2010-18552-C03-01), Ministry of Education (FPU Grant Program) and Xunta de Galicia (projects R2014/029 and R2014/034). A. M. was supported by the EU/FEDER programme, QREN/POR Lisboa (Portugal), under the Intelligo project (contract 2012/24803), and by the FCT grants UID/EEA/50008/2013 and PTDC/EEI-SII/2312/2012.

References

Taylor Berg-Kirkpatrick, Dan Gillick, and Dan Klein. 2011. Jointly learning to extract and compress. In *Proc. of Annual Meeting of the Association for Computational Linguistics*.

Anders Björkelund, Özlem Çetinoğlu, Agnieszka Faleńska, Richárd Farkas, Thomas Mueller, Wolfgang Seeker, and Zsolt Szántó. 2014. Introducing the ims-wrocław-szeged-cis entry at the spmrl 2014 shared task: Reranking and morpho-syntax meet unlabeled data. In *Proc. of the First Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages*.

Ezra Black, John Lafferty, and Salim Roukos. 1992. Development and evaluation of a broad-coverage probabilistic grammar of english-language computer manuals. In *Proc. of Annual Meeting on Association for Computational Linguistics*.

Adriane Boyd. 2007. Discontinuity revisited: An improved conversion to context-free representations. In *Proc. of Linguistic Annotation Workshop*.

Sabine Brants, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius, and George Smith. 2002. The TIGER treebank. In *Proc. of the workshop on treebanks and linguistic theories*.

Xavier Carreras, Michael Collins, and Terry Koo. 2008. TAG, Dynamic Programming, and the Perceptron for Efficient, Feature-rich Parsing. In *Proc. of the International Conference on Natural Language Learning*.

Eugene Charniak and Mark Johnson. 2005. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proc. of Annual Meeting of the Association for Computational Linguistics*.

Eugene Charniak. 1996. Tree-bank grammars. In *Proc. of the National Conference on Artificial Intelligence*.

Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *Proc. of the North American Chapter of the Association for Computational Linguistics Conference*.

Michael Collins, Lance Ramshaw, Jan Hajič, and Christoph Tillmann. 1999. A Statistical Parser for Czech. In *Proc. of the Annual Meeting of the Association for Computational Linguistics on Computational Linguistics*.

Michael Collins. 1999. *Head-driven statistical models for natural language parsing*. Ph.D. thesis, University of Pennsylvania.

Benoit Crabbé and Djamé Seddah. 2014. Multilingual discriminative shift reduce phrase structure parsing for the SPMRL 2014 shared task. In *Proc. of the First Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages*.

Marie-Catherine De Marneffe, Bill MacCartney, Christopher D Manning, et al. 2006. Generating typed dependency parses from phrase structure parses. In *Proc. of the Meeting of the Language Resources and Evaluation Conference*.

Ane Dybro-Johansen. 2004. Extraction automatique de Grammaires d'Arbres Adjoints à partir d'un corpus arboré du français. Master's thesis, Université Paris 7.

Jason Eisner and Giorgio Satta. 1999. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proc. of Annual Meeting of the Association for Computational Linguistics*.

Jason Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proc. of International Conference on Computational Linguistics*.

Jakob Elming, Anders Johannsen, Sigrid Klerke, Emanuele Lapponi, Hector Martinez Alonso, and Anders Søgaard. 2013. Down-stream effects of tree-to-dependency conversions. In *Proc. of the Annual Conference of the Human Language Technologies - North American Chapter of the Association for Computational Linguistics*.

Haim Gaifman. 1965. Dependency systems and phrase-structure systems. *Information and control*.

Carlos Gómez-Rodríguez, Marco Kuhlmann, and Giorgio Satta. 2010. Efficient parsing of well-nested linear context-free rewriting systems. In *Proc. of the Annual Conference of the North American Chapter of the Association for Computational Linguistics*.

Johan Hall and Joakim Nivre. 2008. A dependency-driven parser for german dependency and constituency representations. In *Proc. of the Workshop on Parsing German*.

David Hall, Greg Durrett, and Dan Klein. 2014. Less grammar, more features. In *Proc. of the Annual Meeting of the Association for Computational Linguistics*.

Liang Huang. 2008. Forest reranking: Discriminative parsing with non-local features. In *Proc. of the Annual Meeting of the Association for Computational Linguistics*.

Richard Johansson and Pierre Nugues. 2008. Dependency-based Semantic Role Labeling of PropBank. In *Empirical Methods for Natural Language Processing*.

Mark Johnson. 1998. PCFG models of linguistic tree representations. *Computational Linguistics*.

Sylvain Kahane, Alexis Nasr, and Owen Rambow. 1998. Pseudo-projectivity: a polynomially parsable non-projective dependency grammar. In *Proc. of the International Conference on Computational Linguistics*.

Laura Kallmeyer and Wolfgang Maier. 2013. Data-driven parsing using probabilistic linear context-free rewriting systems. *Computational Linguistics*.

Dan Klein and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In *Proc. of Annual Meeting on Association for Computational Linguistics*.

Lingpeng Kong and Noah A Smith. 2014. An empirical comparison of parsing methods for stanford dependencies. *arXiv preprint arXiv:1404.4314*.

Lingpeng Kong, Alexander M. Rush, and Noah A. Smith. 2015. Transforming dependencies into phrase structures. In *Proc. of the Conference of the North American Chapter of the Association for Computational Linguistics*.

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proc. of Annual Meeting of the Association for Computational Linguistics*.

Sandra Kübler, Wolfgang Maier, Ines Rehbein, and Yannick Versley. 2008. How to compare treebanks. In *Proc. of the Meeting of the Language Resources and Evaluation Conference*.

Marco Kuhlmann and Joakim Nivre. 2006. Mildly non-projective dependency structures. In *Proc. of the joint conference of the International Committee on Computational Linguistics and the Association for Computational Linguistics*.

Wolfgang Maier and Anders Søgaard. 2008. Treebanks and mild context-sensitivity. In *Proc. of Formal Grammar*.

Wolfgang Maier, Miriam Kaeshammer, and Laura Kallmeyer. 2012. Data-driven plcfrs parsing revisited: Restricting the fan-out to two. In *Proc. of the Eleventh International Conference on Tree Adjoining Grammars and Related Formalisms*.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*.

André F. T. Martins, Miguel B. Almeida, and Noah A. Smith. 2013. Turning on the turbo: Fast third-order non-projective turbo parsers. In *Proc. of the Annual Meeting of the Association for Computational Linguistics*.

Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. 2005. Probabilistic CFG with latent annotations. In *Proc. of the Annual Meeting of the Association for Computational Linguistics*.

Ryan McDonald and Giorgio Satta. 2007. On the complexity of non-projective data-driven dependency parsing. In *Proc. of International Conference on Parsing Technologies*.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajic. 2005. Non-projective dependency pars-

ing using spanning tree algorithms. In *Proc. of Empirical Methods for Natural Language Processing*.

Ryan McDonald, Kevin Lerman, and Fernando Pereira. 2006. Multilingual dependency analysis with a two-stage discriminative parser. In *Proc. of International Conference on Natural Language Learning*.

Joakim Nivre, Johan Hall, Jens Nilsson, Gülsen Eryiğit, and Svetoslav Marinov. 2006. Labeled pseudo-projective dependency parsing with support vector machines. In *Proc. of International Conference on Natural Language Learning*.

Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülsen Eryiğit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. 2007. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*.

Fernando C. N. Pereira and David H. D. Warren. 1983. Parsing as Deduction. In *Proc. of the Annual Meeting of the Association for Computational Linguistics*.

Slav Petrov and Dan Klein. 2007. Improved inference for unlexicalized parsing. In *Proc. of the North American Chapter of the Association for Computational Linguistics*.

Ines Rehbein. 2009. *Treebank-Based Grammar Acquisition for German*. Ph.D. thesis, School of Computing, Dublin City University.

Alexander M Rush and Slav Petrov. 2012. Vine pruning for efficient multi-pass dependency parsing. In *Proc. of the North American Chapter of the Association for Computational Linguistics*.

Alexander Rush, David Sontag, Michael Collins, and Tommi Jaakkola. 2010. On dual decomposition and linear programming relaxations for natural language processing. In *Proc. of Empirical Methods for Natural Language Processing*.

Kenji Sagae and Alon Lavie. 2005. A classifier-based parser with linear run-time complexity. In *Proc. of the Ninth International Workshop on Parsing Technology*.

Djamé Seddah, Sandra Kübler, and Reut Tsarfaty. 2014. Introducing the spmrl 2014 shared task on parsing morphologically-rich languages. In *Proc. of the First Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages*, August.

Wojciech Skut, Brigitte Krenn, Thorsten Brants, and Hans Uszkoreit. 1997. An annotation scheme for free word order languages. In *Proc. of the Fifth Conference on Applied Natural Language Processing ANLP-97*.

Richard Socher, John Bauer, Christopher D Manning, and Andrew Y Ng. 2013. Parsing with compositional vector grammars. In *Proc. of Annual Meeting of the Association for Computational Linguistics*.

Andreas van Cranenburgh and Rens Bod. 2013. Discontinuous parsing with an efficient and accurate dop model. *Proc. of International Conference on Parsing Technologies*.

Andreas van Cranenburgh. 2012. Efficient parsing with linear context-free rewriting systems. In *Proc. of the Conference of the European Chapter of the Association for Computational Linguistics*.

Yannick Versley. 2014a. Experiments with easy-first nonprojective constituent parsing. In *Proc. of the First Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages*.

Yannick Versley. 2014b. Incorporating semi-supervised features into discontinuous easy-first constituent parsing. *CoRR*, abs/1409.3813.

Krishnamurti Vijay-Shanker, David J Weir, and Aravind K Joshi. 1987. Characterizing structural descriptions produced by various grammatical formalisms. In *Proc. of the Annual Meeting on Association for Computational Linguistics*.

Yuanbin Wu, Qi Zhang, Xuanjing Huang, and Lide Wu. 2009. Phrase dependency parsing for opinion mining. In *Proc. of Empirical Methods for Natural Language Processing*.

Fei Xia and Martha Palmer. 2001. Converting dependency structures to phrase structures. In *Proc. of the First International Conference on Human Language Technology Research*.

Fei Xia, Owen Rambow, Rajesh Bhatt, Martha Palmer, and Dipti Misra Sharma. 2008. Towards a multi-representational treebank. *LOT Occasional Series*.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proc. of International Conference on Parsing Technologies*.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proc. of the Annual Meeting of the Association for Computational Linguistics*.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *Proc. of Annual Meeting of the Association for Computational Linguistics*.

A Supplementary Material

A.1 Proof of Proposition 1

We will show that, given an arbitrary strictly-ordered d-tree $\mathcal{D}$, we can perform an invertible transformation to turn it into a binary c-tree $\mathcal{C}$; and vice-versa. Let $\mathcal{D}$ be given. We visit each node $h \in \{1, \ldots, L\}$ and split it into $K + 1$ nodes, where $K = |M_h|$, organized as a linked list, as Figure 3 illustrates (this will become the spine of h in the c-tree). For each modifier $m_k \in M_h$ with $m_1 \prec_h \ldots \prec_h m_K$, move the tail of the arc $\langle h, m_k, Z_k \rangle$ to the $(K + 1 - k)$th node of the linked list and assign the label Z_k to this node, letting h be its lexical head. Since the incoming and outgoing arcs of the linked list component are the same as in the original node h, the tree structure is preserved. After doing this for every h, add the leaves and propagate the yields bottom up. It is straightforward to show that this procedure yields a valid binary c-tree. Since there is no loss of information (the orders $\prec_h$ are implied by the order of the nodes in each spine), this construction can be inverted to recover the original d-tree. Conversely, if we start with a binary c-tree, traverse the spine of each h, and attach the modifiers $m_1 \prec_h \ldots \prec_h m_K$ in order, we get a strictly ordered d-tree (also an invertible procedure).

A.2 Proof of Proposition 3

We need to show that (i) Algorithm 1, when applied to a continuous c-tree $\mathcal{C}$, retrieves a head ordered d-tree $\mathcal{D}$ which is projective and has the nesting property, (ii) vice-versa for Algorithm 2. To see (i), note that the projectiveness of $\mathcal{D}$ is ensured by the well-known result of Gaifman (1965) about the projection of continuous trees. To show that it satisfies the nesting property, note that nodes higher in the spine of a word h are always attached by modifiers farther apart (otherwise edges in $\mathcal{C}$ would cross, which cannot happen for a continuous $\mathcal{C}$). To prove (ii), we use induction. We need to show that every created c-node in Algorithm 2 has a contiguous span as yield. The base case (line 3) is trivial. Therefore, it suffices to show that in line 8, assuming the yields of (the current) $\psi(h)$ and each $\psi(m)$ are contiguous spans, the union of these yields is also contiguous. Consider the node v when these children have been appended (line 9), and choose $m \in \bar{M}_h^j$ arbitrarily. We only need to show that for any d between h and m, d belongs to the yield of v. Since $\mathcal{D}$ is projective and there is a d-arc between h and m, we have that d must descend from h. Furthermore, since projective trees cannot have crossing edges, we have that h has a unique child a, also between h and m, which is an ancestor of d (or d itself). Since a is between h and m, from the nesting property, we must have $\langle h, m, \ell \rangle \not\prec_h \langle h, a, \ell' \rangle$ Therefore, since we are processing the modifiers in order, we have that $\psi(a)$ is already a descendent of v after line 9, which implies that the yield of $\psi(a)$ (which must include d, since d descends from a) must be contained in the yield of v.

APPENDIX A

Resumen

El objetivo final del *procesamiento del lenguaje natural* (PLN) es transformar texto bruto escrito en lenguaje natural en una representación que una máquina sea capaz de manejar. De esta forma, información textual sin tratar puede ser fácilmente utilizada por un ordenador para acometer tareas más complejas como traducción automática, extracción de información o búsqueda de respuestas.

El análisis sintáctico es uno de los procesamientos del PLN más utilizados y extendidos. Éste consiste en determinar la estructura gramatical de una oración en lenguaje natural: dada una oración de entrada, un analizador creará su representación sintáctica. Esta estructura subyacente puede ser representada en diferentes formatos dependiendo de la teoría sintáctica que guíe al analizador.

Hay dos formalismos sintácticos ampliamente extendidos para este proposito: representaciones de *constituyentes* [2, 7] y de *dependencias* [44]. En el primer caso, las oraciones son analizadas sintácticamente mediante su descomposición en partes con significado, denominadas *constituyentes*, creando relaciones entre éstas y las palabras para, finalmente, construir un *árbol de constituyentes*, como el descrito en la Figura A.1. Por otro lado, en el análisis de dependencias, la estructura sintáctica de una oración es representada mediante un *grafo de dependencias*. Éste está compuesto por un conjunto de relaciones binarias denominadas *dependencias* que unen pares de palabras de una oración para describir una relación sintáctica entre ellas, donde una actúa como *padre* y la otra como *dependiente*. Decimos que es un *árbol de dependencias* si cada una de las palabras de la oración tiene un único padre, la estructura es acíclica y únicamente tiene una raíz, como el presentado en la Figura A.2.

En las dos últimas decadas, el análisis sintáctico de dependencias ha llegado a ser muy popular en la comunidad del PLN, en detrimento de su rival, el análisis sintáctico de constituyentes. Esto se ha debido principalmente a que el primero tiene algunas ventajas irrefutables sobre el segundo. La falta de nodos intermedios en los grafos de

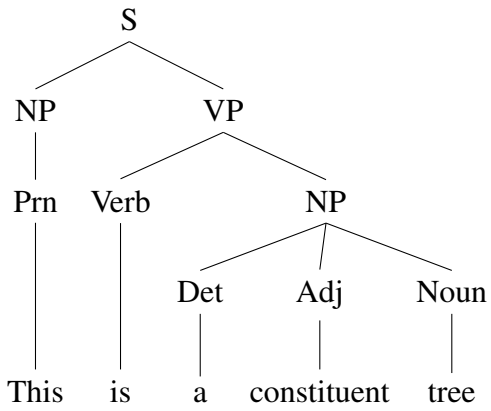


Figura A.1: Árbol de constituyentes para una oración en inglés.

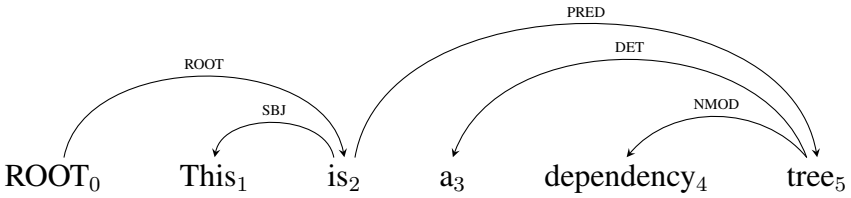


Figura A.2: Árbol de dependencias para una oración en inglés.

dependencias les conceden la sencillez necesaria para representar fenómenos lingüísticos más complejos, tales como las discontinuidades causadas por el orden libre de las palabras, y ha permitido el desarrollo de analizadores sintácticos más eficientes. Como consecuencia, este tipo de analizadores han sido utilizados en una gran variedad de aplicaciones de forma exitosa que van desde la traducción automática [12, 24, 41, 49] y extracción de relaciones [11, 14, 23, 33] hasta la búsqueda de respuestas [8, 10] y minería de opiniones [22, 46].

En contraste con los analizadores sintácticos de dependencias *guiados por una gramática*, un nuevo paradigma centrado en los datos ha emergido en los últimos veinte años. La representación explícita de conocimiento mediante el uso de reglas gramaticales [27, 43] ha sido reemplazada por el paradigma *guiado por los datos*, apoyado por el emergente campo del *aprendizaje automático* y la disponibilidad de una gran cantidad de datos. La creciente disponibilidad de recursos manualmente anotados, tal como el Penn Treebank [28] o corpora proporcionado por la CoNLL-X [5] Shared Task y la CoNLL-XI [37] Shared Task, han hecho posible la aplicación de técnicas de aprendizaje automático, capaces de extraer de forma automática modelos estadísticos a partir de los datos sin necesidad de una gramática explícita, para el desarrollo de analizadores sintácticos de dependencias que producen análisis precisos de forma muy eficiente.

Los analizadores sintácticos de dependencias guiados por los datos han sido un campo

muy fructífero dentro del PLN, resultando en algunos de los sistemas más precisos y eficientes como aquellos desarrollados por Nivre et al. [38], McDonald et al. [32], Titov y Henderson [45], Martins et al. [29], Huang y Sagae [20], Koo y Collins [26], Zhang y Nivre [51], Bohnet y Nivre [3] o Goldberg y Nivre [15]. Prácticamente todos estos sistemas pueden ser clasificados en dos familias, comúnmente denominadas analizadores *basados en grafos* y *basados en transiciones* [31, 50].

Los analizadores basados en grafos aprenden un modelo global para puntuar posibles grafos de dependencias para una oración dada y, a continuación, el proceso de análisis sintáctico consiste en buscar el grafo con mayor puntuación. Las propuestas de McDonald et al. [32] y Martins et al. [29] son dos analizadores basados en grafos ampliamente conocidos. El principal inconveniente de esta técnica es que el proceso de análisis sintáctico se lleva a cabo bajo una complejidad temporal, en el mejor de los casos, cuadrática.

Por otro lado, los analizadores basados en transiciones han demostrado ser más rápidos y eficientes, ya que muchos de ellos son capaces de realizar el análisis de una oración con una complejidad temporal lineal; aún siendo capaces de ofrecer precisiones a la altura del estado del arte. Dada una oración, un analizador de esta naturaleza construye incrementalmente un grafo de dependencias de izquierda a derecha mediante la elección *voraz* de la transición permitida con mayor puntuación en cada estado del proceso. De este modo, la oración de entrada es analizada sintácticamente por una secuencia de transiciones compuesta por las acciones con mayor puntuación. El conjunto de transiciones que el analizador puede utilizar son individualmente puntuadas por un modelo estadístico previamente entrenado y que se denomina *oráculo*. Además, existe una variante de analizadores basados en transiciones que incluye la técnica del *beam search* para la selección de la mejor secuencia de transiciones [50], en lugar de hacerlo de forma voraz. De hecho, uno de los sistemas basados en transiciones que ostenta el estado del arte es el analizador con *beam search* de Zhang y Nivre [51].

Desafortunadamente, la naturaleza voraz que les concede su eficiencia también se convierte en su principal debilidad. McDonald y Nivre [30] muestran que la principal razón de perdida de precisión en los analizadores basados en transiciones es la propagación de errores: una transición seleccionada de forma errónea podría posicionar el analizador en un estado incorrecto, causando a continuación más errores en la secuencia de transiciones. Como la oración es analizada en un proceso secuencial, un error en un estado temprano del proceso podría llevar a cometer más errores en futuras etapas. En particular, una de las fuentes de propagación de errores es el cumplimiento de la restricción de padre único necesaria para formar un árbol de dependencias. Por ejemplo, si un sistema basado en transiciones está analizando la oración de la Figura A.2 y erróneamente aplica la transición que crea un arco de a_3 a $tree_5$ en lugar de la correcta dependencia de $tree_5$ a a_3; no fallaría únicamente en la creación de esta dependencia sino también en aquella que va de is_2 a $tree_5$, ya que estaría en un estado donde la restricción de padre único no permitiría la creación de dos arcos entrantes sobre

el nodo $tree_5$. A mayores, oraciones y arcos de gran longitud se ven especialmente afectados por la propagación de errores, ya que es necesario utilizar una secuencia de transiciones más larga para su análisis. En esta tesis, centramos todos nuestros esfuerzos en conseguir que los analizadores sintácticos basados en transiciones sean más precisos mediante la reducción de la propagación de errores, pero sin penalizar su ventajosa eficiencia. Además, algunas de las mejoras aplicadas con éxito en los sistemas basados en transiciones también podrían ser utilizadas en cualquier otro tipo de analizadores de dependencias.

A pesar del hecho de que el análisis sintáctico de dependencias ha estado en el foco de atención en los últimos veinte años, las representaciones de constituyentes han generado recientemente un gran interés entre los investigadores de PLN. Esto ha sido facilitado por el hecho de que algunas aplicaciones, como el análisis de sentimientos y la minería de opiniones [1, 13, 21, 48], requieren de formalismos sintácticos más informativos que las representaciones de dependencias. Sin embargo, el principal problema es que la mayoría de los analizadores de constituyentes disponibles son significativamente lentos [6, 25, 39], puesto que necesitan lidiar con una pesada constante gramatical. Por lo tanto, la comunidad de PLN demanda un analizador de constituyentes que sea tan eficiente como sus homónimos de dependencias. Para satisfacer esta necesidad, proponemos un nuevo enfoque que mezcla lo mejor de ambos mundos: construir una estructura de constituyentes más informativa mediante un eficiente analizador de dependencias.

Por lo tanto, esta tesis pretende hacer aportaciones tanto al campo del análisis sintáctico de constituyentes como al de dependencias. Nuestra intención es mejorar el rendimiento de los analizadores de dependencias (concretamente, los sistemas basados en transiciones) y utilizarlos para llevar a cabo un análisis sintáctico de constituyentes de forma eficiente.

A.1 | Preliminares

A continuación, introducimos algunas definiciones y notaciones básicas sobre analizadores de dependencias basados en transiciones, que nos van a servir como base para presentar todas nuestras contribuciones.

A.1.1 | Análisis sintáctico de dependencias

Un *grafo de dependencias* es un grafo dirigido y etiquetado que representa la estructura sintáctica de una oración dada. Más formalmente, puede ser definido como:

Definición 1 *Sea $w = w_1 \dots w_n$ una cadena de entrada. Un* grafo de dependencias *para $w_1 \dots w_n$ es un grafo dirigido y etiquetado $G = (V_w, A)$, donde $V_w = \{0, \dots, n\}$ es el conjunto de nodos, y $A \subseteq V_w \times L \times V_w$ es el conjunto de arcos etiquetados y dirigidos.*

■

Además de cada palabra de la oración con índice i tal que $1 \leq i \leq n$, el conjunto V_w incluye un nodo especial con índice 0 denominado ROOT, el cual no se corresponde con ninguna palabra de la oración y siempre será la raíz del grafo de dependencias.

Cada arco en A codifica una relación de dependencia entre dos palabras. Denominamos una arista (w_i, l, w_j) en un grafo de dependencias G a un *enlace de dependencia* de w_i a w_j con etiqueta l, reprensentado como $w_i \overset{l}{\rightarrow} w_j$. Decimos que w_i es el *padre* de w_j e, inversamente, que w_j es el *dependiente* de w_i. Las etiquetas sobre los enlaces de dependencia son habitualmente utilizados para representar funciones sintácticas, tal como SBJ para el sujeto en la dependencia $is_2 \rightarrow This_1$ de la Figura A.2.

Por conveniencia, escribimos $w_i \rightarrow w_j \in G$ si el arco (w_i,w_j) existe (sin importar su etiqueta) y $w_i \rightarrow^* w_j \in G$ si hay un (posiblemente vacío) camino dirigido de w_i a w_j.

La mayoría de los formalismos sintácticos basados en dependencias están restringidos a grafos acíclicos donde cada nodo tiene como máximo un padre. Tales grafos de dependencias se denominan *bosques de dependencias*.

Definición 2 *Un grafo de dependencias G se dice que es un* bosque de dependencias *si cumple las siguientes restricciones:*

1. Restricción de padre único*: si $w_i \rightarrow w_j$, entonces no existe ningún $w_k \neq w_i$ tal que $w_k \rightarrow w_j$.*
2. Restricción de aciclicidad*: si $w_i \rightarrow^* w_j$, entonces no existe ningún arco $w_j \rightarrow w_i$.*

■

Los nodos que no tienen padre en un bosque de dependencias se denominan *raíces*. Además de las dos restricciones previamente descritas, algunos formalismos de dependencias añaden la condición adicional de que el bosque únicamente debe tener una raíz (o, equivalentemente, que todos los nodos del grafo estén conectados). Un bosque con estas características se conoce como un *árbol de dependencias*.

Un *analizador de dependencias* es un sistema encargado de analizar una oración dada produciendo un grafo de dependencias. En esta tesis, se ha trabajado con analizadores que construyen árboles de dependencias. Esto significa que cumplen con las restricciones de padre único y de aciclicidad, así como que todas las raíces del grafo sean enlazadas como dependientes del nodo artificial ROOT.

Finalmente, muchos analizadores están restringidos a trabajar con estructuras de dependencias *proyectivas* para preservar su eficiencia computacional. Se trata de grafos de dependencias donde el conjunto de nodos alcanzables al atravesar cero o más arcos desde

cualquier nodo k se corresponde con una subcadena continua de la entrada, esto es, un intervalo $\{x \in V_w \mid i \leq x \leq j\}$. Para identificar si un grafo de dependencias es proyectivo se puede utilizar su representación gráfica, como la de la Figura A.2, donde la ausencia de arcos que se cruzan confirma la proyectividad de la estructura. Para analizar fenómenos sintácticos más complejos, es necesario utilizar grafos de dependencias no proyectivos (con arcos cruzados), los cuales permiten la representación de discontinuidades causadas por el orden libre de las palabras.

A.1.2 | Sistema de transiciones

El entorno de trabajo propuesto por Nivre [36] ofrece los componentes necesarios para desarrollar un analizador basado en transiciones. Éste es un analizador de dependencias determinista definido por un *sistema de transiciones* no determinista. Un sistema de transiciones especifica el conjunto de operaciones elementales que son aplicadas de forma determinista por un oráculo en cada estado del proceso de análisis. Más formalmente, se define como:

Definición 3 *Un* sistema de transiciones *para análisis de dependencias es una tupla* $S = (C, T, c_s, C_t)$*, donde*

1. C *es un conjunto de posibles* configuraciones*,*
2. T *es un conjunto finito de* transiciones*, que son funciones parciales* $t : C \rightarrow C$*,*
3. c_s *es una función de inicialización que representa cada cadena de entrada* w *como una única* configuración inicial $c_s(w)$*, y*
4. $C_t \subseteq C$ *es un conjunto de* configuraciones terminales*.*

■

Definición 4 *Un* oráculo *para un sistema de transiciones es una función* $o : C \rightarrow T$*.*

■

Una oración de entrada w puede ser analizada sintácticamente utilizando un sistema de transiciones $S = (C, T, c_s, C_t)$ y un oráculo o, empezando en la configuración inicial $c_s(w)$, llamando a la función oráculo en la configuración actual c, y desplazandose a la siguiente configuración mediante el uso de la transición seleccionada por el oráculo. Este proceso es repetido hasta que se alcanza una configuración terminal. Cada secuencia de configuraciones que el analizador puede recorrer desde la configuración inicial hasta la terminal para una entrada w se denomina una *secuencia de transiciones*.

Transition	Stack (σ)	Buffer (β)	Added Arc
	[ROOT$_0$]	[This$_1$, ... , tree$_5$]	
SHIFT	[ROOT$_0$, This$_1$]	[is$_2$, ... , tree$_5$]	
LA$_{SBJ}$	[ROOT$_0$]	[is$_2$, ... , tree$_5$]	(2, SBJ, 1)
RA$_{ROOT}$	[ROOT$_0$, is$_2$]	[a$_3$, ... , tree$_5$]	(0, ROOT, 2)
SHIFT	[ROOT$_0$, is$_2$, a$_3$]	[dependency$_4$, tree$_5$]	
SHIFT	[ROOT$_0$, is$_2$, a$_3$, dependency$_4$]	[tree$_5$]	
LA$_{NMOD}$	[ROOT$_0$, is$_2$, a$_3$]	[tree$_5$]	(5, NMOD, 4)
LA$_{DET}$	[ROOT$_0$, is$_2$]	[tree$_5$]	(5, DET, 3)
RA$_{PRED}$	[ROOT$_0$, is$_2$, tree$_5$]	[ ]	(2, PRED, 5)

Figura A.3: Secuencia de transiciones necesaria para analizar sintácticamente la oración de la Figura A.2 utilizando el algoritmo Arc-eager (LA=LEFT-ARC, RA=RIGHT-ARC).

En la práctica, el oráculo se implementa mediante un modelo estadístico previamente entrenado en un *banco de árboles* [38] y su cometido es el de seleccionar la transición con mayor puntuación del conjunto T para ser aplicada en cada configuración. Un banco de árboles está formado por una gran cantidad de oraciones manualmente anotadas con sus respectivos grafos de dependencias. En particular, en esta tesis hemos trabajado con el English Penn Treebank [28] y bancos de árboles disponibles en la CoNLL-X Shared Task [5] y la CoNLL-XI Shared Task [37] .

A.1.3 | Analizador sintáctico Arc-eager

El analizador de dependencias *Arc-eager* desarrollado por Nivre [35] es uno de los sistemas basados en transiciones más conocidos y utilizados. Concretamente, el sistema de transiciones del Arc-eager (C, T, c_s, C_t) se define como:

1. C es el conjunto de todas las configuraciones de la forma $c = \langle \sigma, \beta, A \rangle$, donde σ y β son listas disjuntas de nodos de V_w (para alguna entrada w), y A es un conjunto de arcos de dependencias sobre V_w. La lista β, denominada *buffer*, se utiliza para almacenar los nodos correspondientes a las palabras entrantes que todavía no han sido leídas. La lista σ, denominada *pila*, contiene los nodos de las palabras que ya han sido leídas, pero que todavía tienen arcos pendientes de ser creados. Por conveniencia, se usará la notación $\sigma|w_i$ para denotar que la palabra w_i está en la cima de la pila σ, y la notación $w_j|\beta$ para indicar que la palabra w_j se encuentra en la primera posición del buffer β. El conjunto A de arcos de dependencias contiene la parte del análisis construido hasta el momento.

2. La configuración inicial es $c_s(w_1 \dots w_n) = \langle [], [w_1 \dots w_n], \emptyset \rangle$, donde el buffer

inicialmente contiene toda la cadena de entrada y la pila está vacía (o $c_s(w_1 \dots w_n) = \langle [ROOT_0], [w_1 \dots w_n], \emptyset \rangle$, si la pila contiene el nodo artificial ROOT).

3. El conjunto de configuraciones terminales es $C_t = \{\langle \sigma, [], A \rangle \in C\}$, donde las configuraciones finales son aquellas con el buffer vacío, sin importar el contenido de la pila.

4. El conjunto T incluye las siguientes transiciones:

   - SHIFT : $\langle \sigma, w_i|\beta, A \rangle \Rightarrow \langle \sigma|w_i, \beta, A \rangle$

   - REDUCE : $\langle \sigma|w_i, \beta, A \rangle \Rightarrow \langle \sigma, \beta, A \rangle$

   - LEFT-ARC$_l$:
     $\langle \sigma|w_i, w_j|\beta, A \rangle \Rightarrow \langle \sigma, w_j|\beta, A \cup \{w_j \xrightarrow{l} w_i\} \rangle$
     sólo si $\nexists w_k \mid w_k \rightarrow w_i \in A$ (padre único)

   - RIGHT-ARC$_l$:
     $\langle \sigma|w_i, w_j|\beta, A \rangle \Rightarrow \langle \sigma|w_i|w_j, \beta, A \cup \{w_i \xrightarrow{l} w_j\} \rangle$
     sólo si $\nexists w_k \mid w_k \rightarrow w_j \in A$ (padre único)

La transición SHIFT se utiliza para leer palabras de la cadena de entrada, al desplazar el siguiente nodo en el buffer a la cima de la pila. La transición LEFT-ARC$_l$ crea arcos etiquetados con l hacia la izquierda desde el primer nodo del buffer hasta el nodo en la cima de la pila, eliminando este último de la pila. Inversamente, la transición RIGHT-ARC$_l$ construye una dependencia etiquetada con l hacia la derecha desde el nodo en la cima de la pila hasta el primer nodo del buffer y desplaza este último a la pila. La transición REDUCE se utiliza para eliminar los nodos de la cima de la pila que ya no van a estar implicados en la creación de ningún arco. La Figura A.3 muestra la secuencia de transiciones que necesita el sistema Arc-eager para producir el grafo de dependencias descrito en la Figura A.2.

Nótese que el analizador Arc-eager es un algoritmo de complejidad temporal lineal, ya que garantiza su terminación después de $2n$ transiciones (siendo n la longitud de la cadena de entrada), y está restringido a árboles de dependencias proyectivos.

Otros sistemas basados en transiciones populares son: los analizadores *Arc-standard* y *Covington* [9, 36], así como, los sistemas *Planar* y *Two-planar* [18].

A.2 | Contribuciones

A continuación resumimos las principales aportaciones de la presente tesis.

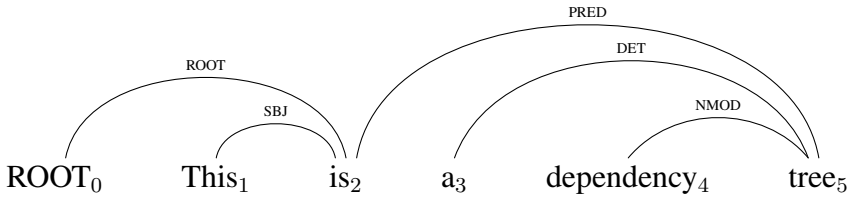


Figura A.4: Grafo de dependencias no dirigido para la oración en inglés de la Figura A.2.

A.2.1 | Análisis sintáctico de dependencias no dirigido

En nuestro primer intento por reducir el impacto de la propagación de errores hemos presentado un nuevo enfoque: el análisis sintáctico de dependencias no dirigido. Hasta la fecha, todos los sistemas de dependencias existentes producían grafos dirigidos (Figura A.2), donde las dependencias tienen una dirección desde la palabra padre a la dependiente. Nosotros hemos desarrollado analizadores basados en transiciones capaces de trabajar con grafos de dependencias no dirigidos como el representado en la Figura A.4. Esto implica que la restricción de padre único no debe ser cumplida durante el proceso de análisis sintáctico, ya que la noción de dirección de padre a dependiente se pierde en los grafos no dirigidos. Como consecuencia, el analizador tiene mayor libertad, pudiendo prevenir situaciones donde el cumplimiento de esta restricción daba lugar a la propagación de errores.

Tras un proceso de análisis sintáctico no dirigido, es necesario realizar un paso de pos-procesado con el fin de recuperar la dirección de las dependencias, generando una estructura de dependencias válida. Por consiguiente, parte de la complejidad se traslada del proceso de análisis a la etapa de pos-procesado. Esta simplificación tiene como resultado que los analizadores no dirigidos cometan menos errores que la versión original, logrando un incremento en la precisión en prácticamente todos los experimentos realizados.

Concretamente, se han implementado las variantes no dirigidas de los analizadores basados en transiciones Planar, Two-planar [18] y Covington [9, 36], y han sido testados sobre el English Penn Treebank [28] y diferentes conjuntos de datos de la CoNLL-X Shared Task [5]. Los resultados han confirmado la utilidad del análisis sintáctico de dependencias no dirigido. Además, posteriores análisis de los resultados han corroborado que, efectivamente, este nuevo enfoque reduce la influencia de la propagación de errores en los sistemas basados en transiciones.

Esta técnica puede ser aplicada en cualquier analizador de dependencias, ofreciendo más información sobre la misma en los Artículos 4.1 y 4.2. Por favor, nótese que el Artículo 4.2 es una versión extendida del Artículo 4.1, el cual añade más contenido incluyendo nuevos experimentos y un detallado análisis de errores.

A.2.2 | Transiciones de buffer

Debido a que el análisis basado en transiciones se produce por medio de una secuencia de transiciones, el impacto de la propagación de errores es directamente proporcional a la longitud de dicha secuencia. En otras palabras, es más probable cometer errores (y, por consiguiente, propagarlos) si el número de decisiones que el analizador tiene que tomar es mayor. Ésta fue la intuición inicial que nos llevó a plantear esta estrategia: intentar reducir el número de transiciones necesarias para analizar una oración dada. Para conseguir esto, es necesario diseñar nuevas transiciones cuyo efecto reemplace a dos o más transiciones originales. También sería deseable que estas nuevas transiciones fuesen aplicadas en circunstancias fácilmente identificables por el analizador, sin sobrecargar la labor del clasificador.

En particular, se han desarrollado cuatro diferentes transiciones para el popularmente conocido analizador Arc-eager [35], detallado previamente en la Sección A.1.2, denominadas *transiciones de buffer*. Éstas se definen como:

- LEFT-BUFFER-ARC$_l$:
  $(\sigma, w_i|w_j|\beta, A) \Rightarrow (\sigma, w_j|\beta, A \cup \{w_j \xrightarrow{l} w_i\})$.
- RIGHT-BUFFER-ARC$_l$:
  $(\sigma, w_i|w_j|\beta, A) \Rightarrow (\sigma, w_i|\beta, A \cup \{w_i \xrightarrow{l} w_j\})$.
- LEFT-NONPROJ-BUFFER-ARC$_l$:
  $(\sigma|w_i, w_j|w_k|\beta, A) \Rightarrow (\sigma, w_j|w_k|\beta, A \cup \{(w_k \xrightarrow{l} w_i)\})$. Precondiciones: $i \neq 0$ y $\nexists w_m, l' \mid (w_m, l', w_i) \in A$ (padre único)
- RIGHT-NONPROJ-BUFFER-ARC$_l$:
  $(\sigma|w_i, w_j|w_k|\beta, A) \Rightarrow (\sigma|w_i, w_j|\beta, A \cup \{w_i \xrightarrow{l} w_k\})$.

Se trata de transiciones que crean un arco de dependencia sobre algún nodo del buffer que no está disponible para las transiciones estándar. De esta forma, son capaces de construir algunas dependencias *sencillas* a priori, antes de que los nodos implicados sean desplazados a la pila, simplificando, en consecuencia, el trabajo del clasificador a la hora de decidir qué transición estándar debe aplicar. A mayores, las transiciones LEFT-NONPROJ-BUFFER-ARC y RIGHT-NONPROJ-BUFFER-ARC amplian la cobertura del analizador Arc-eager. Ambas transiciones permiten la creación de un conjunto limitado de arcos no proyectivos, ya que actúan sobre nodos no contiguos de la pila y del buffer.

También es necesario destacar que las transiciones LEFT-BUFFER-ARC y RIGHT-BUFFER-ARC son equivalentes a aplicar la secuencia de transiciones estándar SHIFT + LEFT-ARC y SHIFT + RIGHT-ARC + REDUCE, respectivamente, dando lugar a un acortamiento de la secuencia final de transiciones. Por otro lado, el efecto de las transiciones LEFT-NONPROJ-BUFFER-ARC y RIGHT-NONPROJ-BUFFER-ARC no puede ser representado mediante transiciones estándar debido a que añaden una

funcionalidad no presente en el analizador Arc-eager, aunque también suponen una reducción de la longitud de la secuencia de transiciones.

Experimentos realizados sobre distintos conjuntos de datos de la CoNLL-X Shared Task [5] apoyan nuestra hipótesis y muestran que, mediante el uso de las transiciones de buffer, la precisión del analizador Arc-eager se ve mejorada en prácticamente todos los casos.

Esta técnica se detalla en el Artículo 4.3 y puede ser aplicada en analizadores de dependencias basados en transiciones que presenten un buffer y una pila en sus configuraciones.

A.2.3 | Análisis sintáctico en sentido inverso

Los sistemas basados en transiciones analizan una oración de izquierda a derecha. Debido a la propagación de errores, la posibilidad de elegir una transición incorrecta tiende a ser más alta a medida que nos acercamos al final de la oración. Como resultado, los arcos situados en la parte derecha del grafo de dependencias sufren de una mayor pérdida de precisión. Parece razonable pensar que aplicar un sistema basado en transiciones que analice sintácticamente la oración en el sentido invertido, de derecha a izquierda, podría ser más preciso en los arcos situados en la parte derecha del grafo. De hecho, ésta es la idea principal que nos llevó a proponer el uso del análisis sintáctico en *sentido inverso*. Concretamente, presentamos un sistema compuesto, donde un analizador de izquierda a derecha se combina con su variante en sentido contrario (de derecha a izquierda).

Se ha demostrado que analizar una oración en orden inverso no mejora la precisión global de los sistemas basados en transiciones [34]. Sin embargo, un análisis en sentido inverso es capaz de producir correctamente algunos arcos del grafo de dependencias que la versión original crea de forma errónea. En particular, hemos descubierto que, además de mostrar un mejor rendimiento en los arcos ubicados a la derecha del grafo, el análisis en orden inverso es capaz de conseguir mayor precisión en arcos de cierta longitud. Para sacar ventaja de ello, hemos propuesto un sistema que combina de forma eficiente el grafo resultante, tanto del analizador original, como del invertido, dando lugar a un nuevo grafo de dependencias mejorado. Este sistema usa dos estrategias diferentes para acometer la combinación: una estrategia basada en la posición y otra en la longitud de los arcos.

Hemos llevado a cabo diferentes experimentos para comprobar la efectividad de este nuevo enfoque sobre diferentes conjuntos de datos (English Penn Treebank [28] y corpora de la CoNLL-X Shared Task [5]) y distintos sistemas basados en transiciones. Los resultados obtenidos muestran que la técnica propuesta produce mejoras de precisión en todas las pruebas realizadas (véanse más detalles en el Artículo 4.4).

A.2.4 | Analizador Arc-eager con restricción arbórea

La investigación recogida en el Artículo 4.5 concierne exclusivamente al analizador basado en transiciones Arc-eager [35] descrito en la Sección A.1.2.

El algoritmo Arc-eager puede alcanzar una configuración terminal sin llegar a vaciar la pila (exceptuando el nodo artificial ROOT). Debido a errores producidos durante el proceso de análisis, parte de las palabras en la pila podrían no estar conectadas al grafo de dependencias resultante (o, equivalentemente, podrían no tener padre). Esto da lugar a un grafo fragmentado como resultado del proceso de análisis sintáctico. La salida deseada sería un árbol de dependencias que cumpliese las restricciones de conectividad, aciclicidad, padre único y que tuviese solamente una raíz. Para conseguir esto, la solución estándar es convertir, al final del proceso de análisis, este grafo fragmentado en un árbol, conectando todas las palabras sin padre en la pila al nodo artificial ROOT. Esta heurística produce un árbol de dependencias bien formado, pero no reduce la perdida de precisión causada por los errores cometidos durante el análisis.

Como alternativa, proponemos una modificación del algoritmo Arc-eager que garantice que, después del análisis, el grafo resultante sea un árbol de dependencias. Concretamente, se ha añadido la transición determinista UNSHIFT que, si el buffer está vacío, desplaza palabras sin padre de la pila al buffer, para que puedan ser procesadas otra vez. Además, se ha establecido una nueva configuración terminal que implica que, no sólo el buffer esté vacío, sino que también ha de estarlo la pila (exceptuando el nodo artificial ROOT). De este modo, al deshabilitar la transición SHIFT, forzamos el analizador Arc-eager a crear arcos sobre estas palabras hasta alcanzar la nueva configuración terminal, ya que sólo puede utilizar las transiciones LEFT-ARC, RIGHT-ARC y REDUCE (esta última únicamente esta permitida sobre palabras que tengan un nodo padre y se utiliza de forma determinista cuando el buffer está vacío). Como resultado, el algoritmo Arc-eager tiene la oportunidad de rectificar algunos de los errores cometidos durante el proceso de análisis.

Evaluaciones empíricas efectuadas en todos los idiomas de la CoNLL-X Shared Task [5] concluyeron que la restricción arbórea implementada en el analizador Arc-eager mejora consistentemente la precisión obtenida por la heurística estándar que conecta todas las palabras sin padre al nodo artificial ROOT.

A.2.5 | Oráculo dinámico no proyectivo

Con el fin de reducir el impacto de la propagación de errores en los analizadores basados en transiciones, Goldberg y Nivre [15] han desarrollado una nueva estrategia denominada *oráculos dinámicos*. Ésta consiste en dotar a los oráculos de los sistemas basados en transiciones con la capacidad necesaria para tolerar errores producidos durante el proceso de análisis, mitigando su impacto en el resultado final. Estos nuevos oráculos están diseñados para sobreponerse a la presencia de errores cometidos en decisiones previas e intentar perder el mínimo número de arcos en estados posteriores.

A diferencia de los oráculos estándar, que son entrenados con la secuencia de transiciones necesarias para analizar una oración dada, los oráculos dinámicos se entrenan con secuencias no óptimas. Durante la etapa de entrenamiento, algunas transiciones son seleccionadas de forma aleatoria simulando los errores cometidos durante el análisis. Esto permite preparar a los oráculos dinámicos para situaciones que se van a encontrar en la etapa de análisis.

Diferentes investigaciones han contribuido con oráculos dinámicos para analizadores proyectivos basados en transiciones como el Arc-eager [15, 16, 17], así como, para sistemas como el analizador de Attardi [19], que soporta un conjunto limitado de arcos no proyectivos. No obstante, la falta de un oráculo dinámico general para arcos no proyectivos, fue la motivación necesaria para aplicar esta estrategia sobre el algoritmo de Covington [9, 36]. Se trata de un analizador que tiene cobertura completa sobre estructuras no proyectivas y es considerado, en la práctica, uno de los sistemas basados en transiciones más rápidos [47].

Concretamente, hemos implementado un oráculo dinámico eficiente específicamente adaptado al analizador de Covington y hemos evaluado su rendimiento sobre los conjuntos de datos disponibles en la CoNLL-X Shared Task [5] y en la CoNLL-XI [37] Shared Task. Los resultados obtenidos prueban que los oráculos dinámicos son también beneficiosos para el algoritmo de Covington, incrementando su precisión significativamente.

En el Artículo 4.6, presentamos los detalles de esta contribución.

A.2.6 | Reducción del análisis de constituyentes a dependencias

Como objetivo final de esta tesis, pretendemos utilizar el análisis de dependencias, reconocido por su eficiencia, para producir representaciones de constituyentes. Para abordar esta tarea, fue necesario desarrollar un formalismo intermedio que permitiese reducir el análisis de constituyentes a dependencias. De este modo, cualquier analizador de dependencias es suficiente para construir un analizador de constituyentes preciso y eficiente.

Este enfoque está basado en la novedosa noción de árboles de dependencias *ordenados por las palabras padre*. Después de determinar las *palabras padre* de cada constituyente, la estructura de un árbol de constituyentes es codificada en un árbol de dependencias como muestra la Figura A.5. Más en detalle, utilizamos las etiquetas de las dependencias salientes de las palabras padre para almacenar cada nodo del árbol de constituyentes (concretamente, el nodo principal del constituyente donde estas palabras actúan como padre) junto con un índice que indica el orden de acoplamiento en la estructura. Por ejemplo, en la Figura A.5 la palabra is actúa como padre en los constituyentes cuyos nodos principales son `VP` y `S`, por lo tanto, ambos nodos son codificados con las etiquetas `VP#1` y `S#2`, respectivamente, donde los índices `#1` y `#2` indican que `VP` es acoplado antes que `S` en el árbol de constituyentes. Si ignoramos el nodo unario `NP` (perdido durante

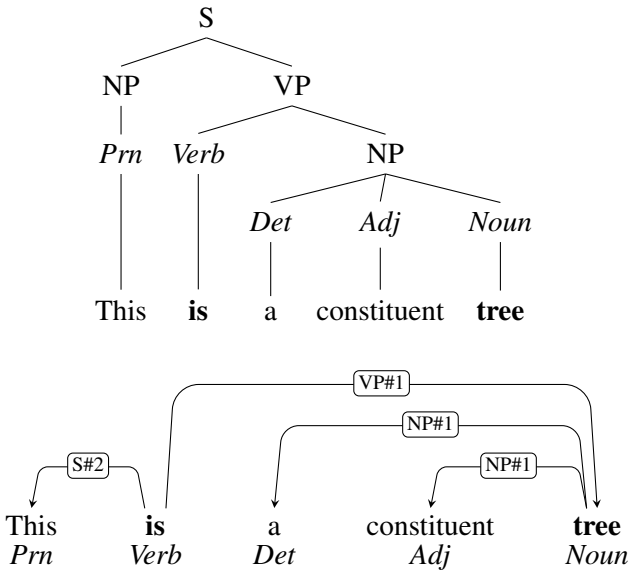


Figura A.5: Un árbol de constituyentes (arriba) y su correspondiente codificación en un árbol de dependencias (abajo). La palabra padre de cada constituyente y las etiquetas morfológicas están marcadas en negrita y cursiva, respectivamente.

la conversión) y las etiquetas morfológicas, veremos como ambos árboles son isomorfos. Esto hace posible una conversión de constituyentes a dependencias (y viceversa) necesaria para producir árboles de constituyentes mediante analizadores de dependencias. Además, los nodos unarios pueden ser eficientemente recuperados en una etapa posterior.

El sistema propuesto ha obtenido resultados a la par con el analizador de constituyentes Berkeley Parser [39] sobre el English Penn Treebank [28], y con el mejor sistema individual de la reciente SPMRL14 Shared Task [40]. También llevamos a cabo experimentos sobre bancos de árboles discontinuos del alemán, concretamente el Tiger [4] y el Negra [42], sobrepasando el actual estado del arte por un amplio margen.

Una detallada descripción de esta técnica puede encontrarse en el Artículo 4.7.

A.3 | Conclusiones

A lo largo de esta tesis hemos presentado novedosas técnicas para mejorar la precisión de los analizadores sintácticos de dependencias basados en transiciones. En particular, las propuestas abordan la principal causa de pérdida de precisión en este tipo de sistemas: la propagación de errores.

Experimentos en diferentes idiomas del English Penn Treebank [28], CoNLL-X Shared Task [5] y CoNLL-XI Shared Task [37] han demostrado que todas las contribuciones son beneficiosas para el rendimiento de los sistemas basados en

transiciones. En todos los casos, conseguimos incrementar su precisión sin penalizar su eficiencia. Además, en nuestra investigación también incluimos detallados análisis de errores que corroboran que las técnicas presentadas alivian el impacto de la propagación de errores.

Queremos también destacar que las aportaciones descritas aquí son completamente compatibles con sistemas basados en transiciones con beam-search y con cualquier otra técnica que mejore la precisión del análisis. De hecho, nuestras cinco contribuciones podrían ser conjuntamente aplicadas en el mismo sistema. Además, algunas de las estrategias presentadas pueden ser trasladadas a otros analizadores de dependencias: por ejemplo, el análisis no dirigido podría ser también beneficioso para técnicas basadas en grafos.

A mayores, hemos ampliado el campo de aplicación de los analizadores de dependencias más allá de la construcción de formalismos de dependencias. Presentamos una nueva técnica para acometer análisis de constituyentes mediante sistemas de dependencias. De esta forma, el análisis sintáctico puede ser llevado a cabo de forma eficiente por un analizador de dependencias en cualquiera de los dos formalismos más ampliamente utilizados. Obviamente, el analizador de dependencias puede incorporar nuestras mejoras en precisión, para que éstas se extiendan también a los constituyentes.

En conclusión, podemos afirmar que las contribuciones desarrolladas durante la etapa de tesis han enriquecido el campo del análisis basado en transiciones con nuevas técnicas, además de proporcionar un novedoso enfoque para producir estructuras de constituyentes de forma eficiente.

References

[1] Taylor Berg-Kirkpatrick, Dan Gillick, and Dan Klein. Jointly learning to extract and compress. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT 2011)*, pages 481–490, Portland, Oregon, 2011.

[2] Leonard Bloomfield. *Language*. University of Chicago Press, 1933.

[3] Bernd Bohnet and Joakim Nivre. A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL 2012)*, pages 1455–1465, Jeju Island, Korea, 2012.

[4] Sabine Brants, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius, and George Smith. The TIGER treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories (TLT 2002)*, Sozopol, Bulgaria, 2002.

[5] Sabine Buchholz and Erwin Marsi. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL 2006)*, pages 149–164, New York City, New York, 2006.

[6] Eugene Charniak. Tree-bank grammars. In *In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996)*, pages 1031–1036, Portland, Oregon, 1996.

[7] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, IT-2:113–124, 1956.

[8] Pere R. Comas, Jordi Turmo, and Lluís Márquez. Using dependency parsing and machine learning for factoid question answering on spoken documents. In *Proceedings of the 13th International Conference on Spoken Language Processing (ICSLP 2010)*, pages 1–4, Makuhari, Japan, 2010.

[9] Michael A. Covington. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102, Athens, Georgia, 2001.

[10] Hang Cui, Renxu Sun, Keya Li, Min-Yen Kan, and Tat-Seng Chua. Question answering passage retrieval using dependency relations. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2005)*, pages 400–407, Salvador, Brazil, 2005.

[11] Aron Culotta and Jeffrey Sorensen. Dependency tree kernels for relation extraction. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL 2004)*, pages 423–429, Barcelona, Spain, 2004.

[12] Yuan Ding and Martha Palmer. Machine translation using probabilistic synchronous dependency insertion grammars. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL 2005)*, pages 541–548, Ann Arbor, Michigan, USA, 2005.

[13] Jakob Elming, Anders Johannsen, Sigrid Klerke, Emanuele Lapponi, Hector Martinez Alonso, and Anders Søgaard. Down-stream effects of tree-to-dependency conversions. In *Proceedings of the Annual Conference of the Human Language Technologies - North American Chapter of the Association for Computational Linguistics (HLT-NAACL 2013)*, pages 617–626, Atlanta, Georgia, 2013.

[14] Katrin Fundel, Robert Küffner, and Ralf Zimmer. RelEx—Relation extraction using dependency parse trees. *Bioinformatics*, 23(3):365–371, 2006.

[15] Yoav Goldberg and Joakim Nivre. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of the 24th International Conference on Computational Linguistics (CoNLL 2012)*, pages 959–976, Mumbai, India, 2012.

[16] Yoav Goldberg and Joakim Nivre. Training deterministic parsers with non-deterministic oracles. *Transactions of the Association for Computational Linguistics*, 1:403–414, 2013.

[17] Yoav Goldberg, Francesco Sartorio, and Giorgio Satta. A tabular method for dynamic oracles in transition-based parsing. *Transactions of the Association for Computational Linguistics*, 2:119–130, 2014.

[18] Carlos Gómez-Rodríguez and Joakim Nivre. A transition-based parser for 2-planar dependency structures. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL 2010)*, pages 1492–1501, Uppsala, Sweden, 2010.

[19] Carlos Gómez-Rodríguez, Francesco Sartorio, and Giorgio Satta. A polynomial-time dynamic oracle for non-projective dependency parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, pages 917–927, Doha, Qatar, 2014.

[20] Liang Huang and Kenji Sagae. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL 2010)*, pages 1077–1086, Uppsala, Sweden, 2010.

[21] Richard Johansson and Pierre Nugues. Dependency-based semantic role labeling of PropBank. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP 2008)*, pages 69–78, Honolulu, Hawaii, 2008.

[22] Mahesh Joshi and Carolyn Penstein-Rosé. Generalizing dependency features for opinion mining. In *Proceedings of the Joint conference of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing (ACL-IJCNLP 2009)*, pages 313–316, Suntec, Singapore, 2009.

[23] Sophia Katrenko, Pieter Adriaans, and Maarten van Someren. Using local alignments for relation recognition. *Journal of Artificial Intelligence Research*, 38(1):1–48, 2010.

[24] Jason Katz-Brown, Slav Petrov, Ryan McDonald, Franz Och, David Talbot, Hiroshi Ichikawa, Masakazu Seno, and Hideto Kazawa. Training a parser for machine translation reordering. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP 2011)*, pages 183–192, Edinburgh, United Kingdom, 2011.

[25] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics (ACL 2003)*, pages 423–430, Sapporo, Japan, 2003.

[26] Terry Koo and Michael Collins. Efficient third-order dependency parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL 2010)*, pages 1–11, Uppsala, Sweden, 2010.

[27] Vincenzo Lombardo and Leonardo Lesmo. An Earley-type recognizer for dependency grammar. In *Proceedings of the 16th International Conference on Computational Linguistics (Coling 1996)*, pages 723–728, Copenhagen, Denmark, 1996.

[28] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19:313–330, 1993.

[29] André F. T. Martins, Noah A. Smith, and Eric P. Xing. Concise integer linear programming formulations for dependency parsing. In *Proceedings of the Joint conference of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing (ACL-IJCNLP 2009)*, pages 342–350, Suntec, Singapore, 2009.

[30] Ryan McDonald and Joakim Nivre. Characterizing the errors of data-driven dependency parsing models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL 2007)*, pages 122–131, Prague, Czech Republic, 2007.

[31] Ryan McDonald and Joakim Nivre. Analyzing and integrating dependency parsers. *Computational Linguistics*, 37:197–230, 2011.

[32] Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing (HLT-EMNLP 2005)*, pages 523–530, Vancouver, British Columbia, Canada, 2005.

[33] Yusuke Miyao, Kenji Sagae, Rune Sætre, Takuya Matsuzaki, and Jun'ichi Tsujii. Evaluating contributions of natural language parsers to protein-protein interaction extraction. *Bioinformatics*, 25(3):394–400, 2009.

[34] Jens Nilsson. *Transformation and Combination in Data-Driven Dependency Parsing*. PhD thesis, Växjö University, 2009.

[35] Joakim Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 2003)*, pages 149–160, Nancy, France, 2003.

[36] Joakim Nivre. Algorithms for Deterministic Incremental Dependency Parsing. *Computational Linguistics*, 34(4):513–553, 2008.

[37] Joakim Nivre, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 915–932, Prague, Czech Republic, 2007.

[38] Joakim Nivre, Johan Hall, and Jens Nilsson. Memory-based dependency parsing. In *Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL 2004)*, pages 49–56, Boston, Massachusetts, 2004.

[39] Slav Petrov and Dan Klein. Improved inference for unlexicalized parsing. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL 2007)*, pages 404–411, Rochester, New York, 2007.

[40] Djamé Seddah, Sandra Kübler, and Reut Tsarfaty. Introducing the SPMRL 2014 shared task on parsing morphologically-rich languages. In *Proceedings of the First Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages (SPMRL-SANCL 2014)*, pages 103–109, Dublin, Ireland, 2014.

[41] Libin Shen, Jinxi Xu, and Ralph Weischedel. A new string-to-dependency machine translation algorithm with a target dependency language model. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL: HLT 2008)*, pages 577–585, Columbus, Ohio, 2008.

[42] Wojciech Skut, Brigitte Krenn, Thorsten Brants, and Hans Uszkoreit. An annotation scheme for free word order languages. In *Proceedings of the Fifth Conference on Applied Natural Language Processing (ANLC 1997)*, pages 88–95, Washington, DC, 1997.

[43] Pasi Tapanainen and Timo Järvinen. A non-projective dependency parser. In *Proceedings of the Fifth Conference on Applied Natural Language Processing (ANLC 1997)*, pages 64–71, Washington, DC, 1997.

[44] Lucien Tesnière. *Éléments de syntaxe structurale*. Editions Klincksieck, 1959.

[45] Ivan Titov and James Henderson. A latent variable model for generative dependency parsing. In *Proceedings of the 10th International Conference on Parsing Technologies (IWPT 2007)*, pages 144–155, Prague, Czech Republic, 2007.

[46] David Vilares, Miguel A. Alonso, and Carlos Gómez-Rodríguez. A syntactic approach for opinion mining on spanish reviews. *Natural Language Engineering*, 21:139–163, 2015.

[47] Alexander Volokh. *Performance-oriented Dependency Parsing*. Saarbrücken dissertations in language sciences and technology, 2013.

[48] Yuanbin Wu, Qi Zhang, Xuanjing Huang, and Lide Wu. Phrase dependency parsing for opinion mining. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing (EMNLP 2009)*, pages 1533–1541, Singapore, 2009.

[49] Peng Xu, Jaeho Kang, Michael Ringgaard, and Franz Och. Using a dependency parser to improve SMT for subject-object-verb languages. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL 2009)*, pages 245–253, Boulder, Colorado, 2009.

[50] Yue Zhang and Stephen Clark. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP 2008)*, pages 562–571, Honolulu, Hawaii, 2008.

[51] Yue Zhang and Joakim Nivre. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL: HLT 2011)*, pages 188–193, Portland, Oregon, 2011.