Computing and Informatics, Vol. 35, 2016, 555–585

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, David Vilares

LyS Research Group
Departamento de Computación
Universidade da Coruña
Campus de A Coruña
s/n, 15071 A Coruña, Spain
e-mail: {cgomezr, 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, when the huge amount of raw textual information that computers must process, a vital role has been given to the 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 a natural language processing (NLP).

One of the most ubiquitous and useful NLP processes is syntactic parsing. This consists of mapping a sentence in a 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.

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,

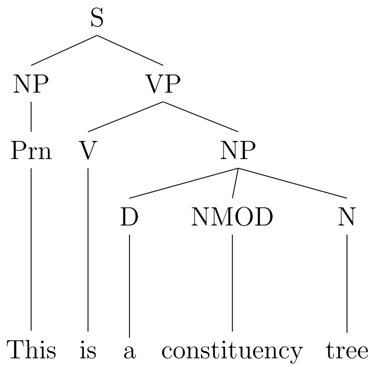

Figure 1. Constituency tree for an English sentence

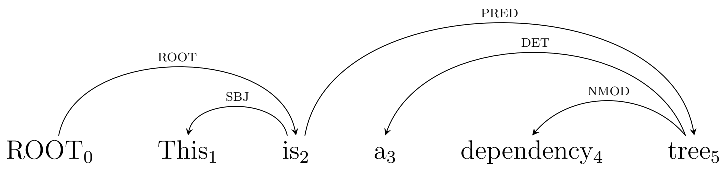

Figure 2. Dependency tree for an English sentence

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 the 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 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

[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.

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 $t : C \to C$,
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 $o : C \to T$.

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 $\nexists k \mid k \rightarrow i \in A$ (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 $\nexists k \mid k \rightarrow j \in A$ (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$_0$]	[ ]	

Figure 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].

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

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$_0$]	[ ]	

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

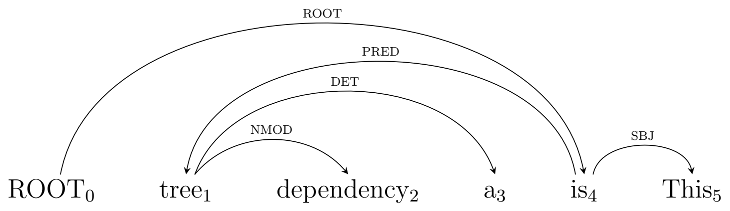

Figure 5. Dependency tree obtained by applying the reverse arc-eager parser on the English sentence in Figure 2

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 the length higher than 15, we 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.

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.

[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 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 7 a) 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 7 b). Note that the analysis made by the reverse parser presents two incorrect arcs: the two leftmost arcs $\mathrm{ROOT}_0 \rightarrow \mathrm{This}_1$ and $\mathrm{This}_1 \rightarrow \mathrm{is}_2$. Secondly, Figure 7 c) presents the dependency tree obtained by the original arc-eager parser. In this tree, there are also two mistakes: the incorrect rightmost arc $\mathrm{dependency}_4 \rightarrow a_3$ and the incorrect label DET in arc $\mathrm{tree}_5 \rightarrow \mathrm{dependency}_4$. 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 7 d) shows the resulting dependency graph after combining the dependency trees in Figure 7 b) and Figure 7 c). 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 7 c)) to assign head nodes to words located before and at position 2 ($\mathrm{ROOT}_0 \rightarrow \mathrm{is}_2$, $\mathrm{is}_2 \rightarrow \mathrm{This}_1$ and $\mathrm{is}_2 \rightarrow \mathrm{tree}_5$), and we complete the new graph with arcs $\mathrm{tree}_5 \rightarrow \mathrm{dependency}_4$ and $\mathrm{tree}_5 \rightarrow a_3$ provided by the reverse dependency tree (Figure 7 b)). Since we rely on the reverse parser to build the $\mathrm{tree}_5 \rightarrow \mathrm{dependency}_4$, the correct label of this arc is taken from the reverse arc-eager dependency tree. As we can see, the output in Figure 7 d) 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.

```
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
```

Figure 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.

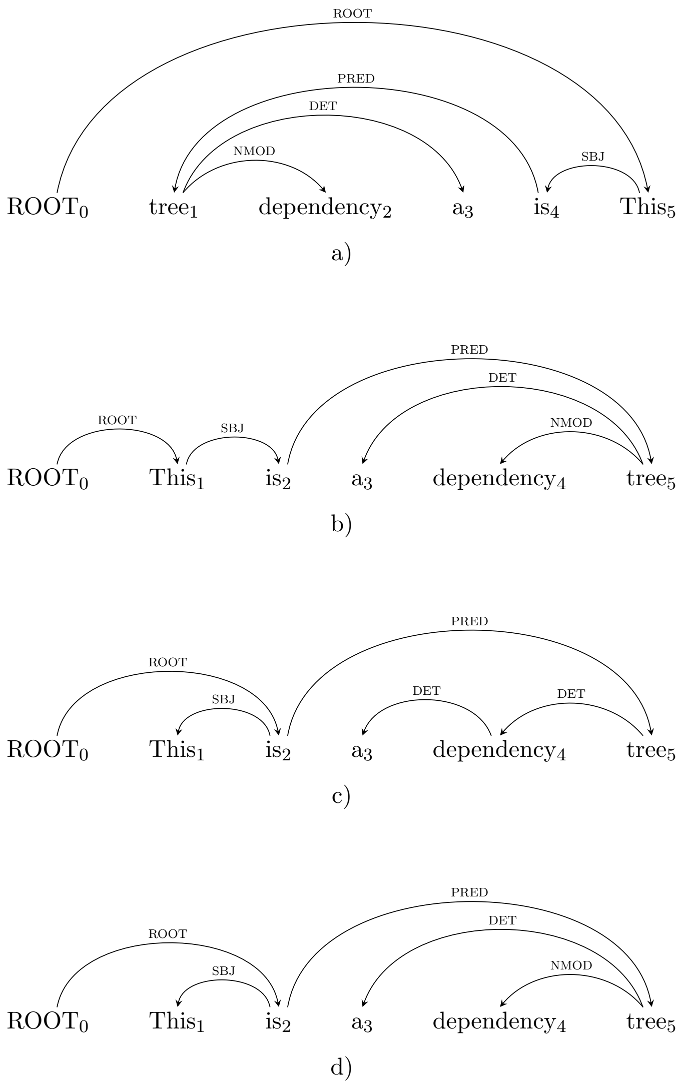
ROOT
PRED
DET
NMOD
SBJ
ROOT$_0$ tree$_1$ dependency$_2$ a$_3$ is$_4$ This$_5$
a)
PRED
DET
ROOT
SBJ
NMOD
ROOT$_0$ This$_1$ is$_2$ a$_3$ dependency$_4$ tree$_5$
b)
PRED
ROOT
SBJ
DET
DET
ROOT$_0$ This$_1$ is$_2$ a$_3$ dependency$_4$ tree$_5$
c)
PRED
DET
ROOT
SBJ
NMOD
ROOT$_0$ This$_1$ is$_2$ a$_3$ dependency$_4$ tree$_5$
d)

Figure 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 7 a). c) Dependency tree derived by the original arc-eager parser. d) Combination of the reverse arc-eager dependency tree in Figure 7 b) and the arc-eager dependency tree in Figure 7 c) using the position-based strategy with $p = 2$.

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

[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/`.

parser with feature models originally optimized for the standard one could introduce 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.

	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

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

	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**		

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		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 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

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.

Further, to 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 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.

[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.

	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

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

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, to see 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 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 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.

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

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

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 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 [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

	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**		

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.

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.

	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**		

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.

	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**		

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.

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

	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**		

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.

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.

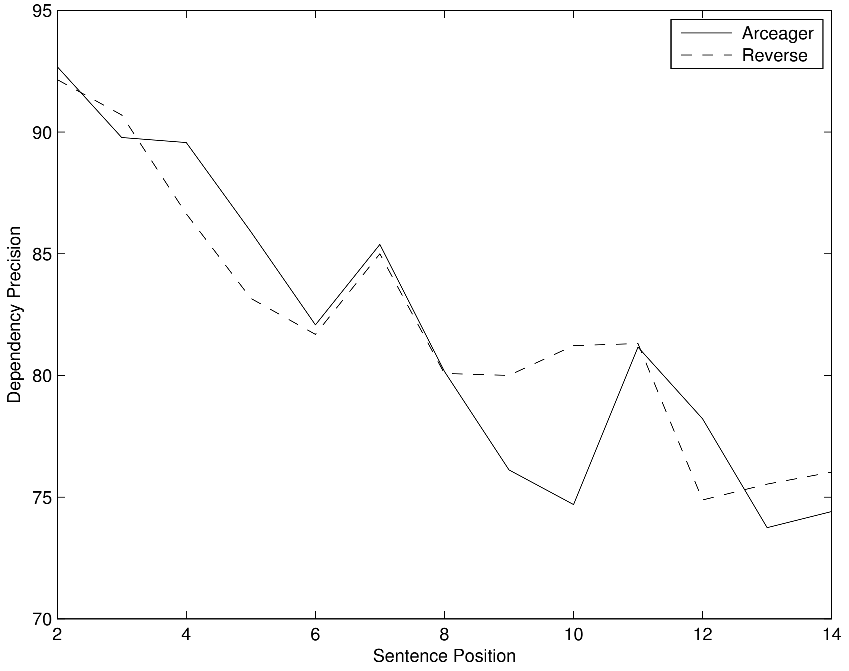

Figure 8. Dependency arc precision relative to position in the sentence, for the arc-eager parser (Arceager) and the reverse arc-eager parser (Reverse), on the Czech dataset

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 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.

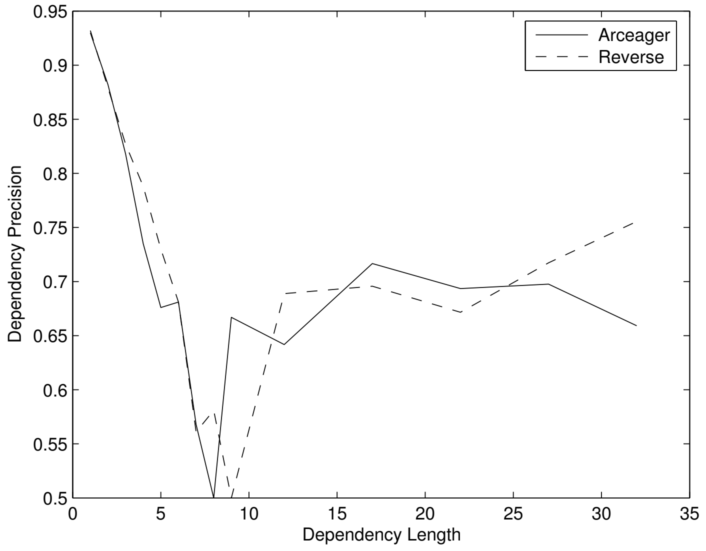

Figure 9. Dependency arc precision relative to predicted dependency length, for the arc-eager parser (Arceager) and the reverse arc-eager parser (Reverse) on the Czech dataset

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 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.

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

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

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 in the reverse variant on the rightmost 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].

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 ERDF (projects TIN2010-18552-C03-01, TIN2010-18552-C03-02, FFI2014-51978-C2-1-R, FFI2014-51978-C2-2-R), Ministry of Education (FPU Grant Program AP2010-1730 and FPU13/01180) and Xunta de Galicia (CN 2012/008, Rede Galega de Procesamento da Linguaxe e Recuperación da Información, Rede Galega de Recursos Lingüísticos para unha Sociedade do Coñecemento, and an Oportunius program grant).

[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.

REFERENCES

[1] Afonso, S.—Bick, E.—Haber, R.—Santos, D.: "Floresta Sintá(c)tica": A Treebank for Portuguese. Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC 2002), ELRA, Paris, France, 2002, pp. 1968–1703.

[2] Atalay, N. B.—Oflazer, K.—Say, B.: The Annotation Process in the Turkish Treebank. Proceedings of EACL Workshop on Linguistically Interpreted Corpora (LINC-03), ACL, USA, 2003, pp. 243–246.

[3] Attardi, G.—Dell'Orletta, F.: Reverse Revision and Linear Tree Combination for Dependency Parsing. NAACL-Short 09, 2009, pp. 261–264.

[4] Ballesteros, M.—Nivre, J.: Going to the Roots of Dependency Parsing. Computational Linguistics, Vol. 39, 2013, No. 1, pp. 5–13.

[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.): Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics (EACL), ACL, 2012, pp. 77–87.

[7] Bohnet, B.—Nivre, J.: A Transition-Based System for Joint Part-of-Speech Tagging and Labeled Non-Projective Dependency Parsing. Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL '12), ACL, 2012, pp. 1455–1465. `http://dl.acm.org/citation.cfm?id=2390948.2391114`.

[8] Brants, S.—Dipper, S.—Hansen, S.—Lezius, W.—Smith, G.: The Tiger Treebank. Proceedings of the Workshop on Treebanks and Linguistic Theories, September 20–21, 2002, Sozopol, Bulgaria. `http://www.coli.uni-sb.de/~sabine/tigertreebank.pdf`.

[9] Buchholz, S.—Marsi, E.: CoNLL-X Shared Task on Multilingual Dependency Parsing. Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL), 2006, pp. 149–164.

[10] Chang, C. C.—Lin, C. J.: LIBSVM: A Library for Support Vector Machines. 2001. Software available on: `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, Chapter 13, Kluwer, 2003, pp. 231–248.

[12] Chomsky, N.: Three Models for the Description of Language. IRE Transactions on Information Theory IT-2, 1956, pp. 113–124.

[13] Ding, Y.—Palmer, M.: Synchronous Dependency Insertion Grammars: A Grammar Formalism for Syntax Based Statistical MT. Proceedings of the Workshop on Recent Advances in Dependency Grammar, 2004, pp. 90–97.

[14] Eisner, J. M.: Three New Probabilistic Models for Dependency Parsing: An Exploration. Proceedings of the 16th International Conference on Computational Linguistics (COLING), 1996, pp. 340–345.

[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, Vol. 9, 2008, pp. 1871–1874.

[16] Fishel, M.—Nivre, J.: Voting and Stacking in Data-Driven Dependency Parsing. In: Jokinen, K., Bick, E. (Eds.): Proceedings of the 17th Nordic Conference of Computational Linguistics (NODALIDA 2009), NEALT Proceedings Series, Vol. 4, 2009, pp. 219–222.

[17] Gómez-Rodríguez, C.—Nivre, J.: Divisible Transition Systems and Multiplanar Dependency Parsing. Computational Linguistics, Vol. 39, 2013, No. 4, pp. 799–845.

[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. 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. Proceedings of the CoNLL Shared Task of EMNLP-CoNLL 2007, 2007, pp. 933–939.

[21] Huang, L.—Sagae, K.: Dynamic Programming for Linear-Time Incremental Parsing. Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL '10), ACL, 2010, pp. 1077–1086. `http://portal.acm.org/citation.cfm?id=1858681.1858791`.

[22] Koo, T.—Collins, M.: Efficient Third-Order Dependency Parsers. Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL '10), 2010, pp. 1–11.

[23] Kromann, M. T.: The Danish Dependency Treebank and the Underlying Linguistic Theory. Proceedings of the 2nd Workshop on Treebanks and Linguistic Theories (TLT), Växjö University Press, Växjö, Sweden, 2003, pp. 217–220.

[24] Marcus, M. P.—Santorini, B.—Marcinkiewicz, M. A.: Building a Large Annotated Corpus of English: The Penn Treebank. Computational Linguistics, Vol. 19, 1993, pp. 313–330.

[25] Martins, A.—Smith, N.—Xing, E.: Concise Integer Linear Programming Formulations for Dependency Parsing. 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), 2009, pp. 342–350.

[26] McDonald, R.—Nivre, J.: Characterizing the Errors of Data-Driven Dependency Parsing Models. Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), 2007, pp. 122–131.

[27] McDonald, R.—Nivre, J.: Analyzing and Integrating Dependency Parsers. Computational Linguistics, Vol. 37, 2011, pp. 197–230.

[28] McDonald, R.—Pereira, F.—Ribarov, K.—Hajič, J.: Non-Projective Dependency Parsing Using Spanning Tree Algorithms. Proceedings of the Human Language Technology Conference and the Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP), 2005, pp. 523–530.

[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. Proceedings of the 8th International Workshop on Parsing Technologies (IWPT), 2003, pp. 149–160.

[32] Nivre, J.: Algorithms for Deterministic Incremental Dependency Parsing. Computational Linguistics, Vol. 34, 2008, pp. 513–553.

[33] Nivre, J.: Algorithms for Deterministic Incremental Dependency Parsing. Computational Linguistics, Vol. 34, 2008, No. 4, pp. 513–553. `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. Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL-2004), ACL, 2004, pp. 49–56.

[35] Nivre, J.—Hall, J.—Nilsson, J.: Maltparser: A Data-Driven Parser-Generator for Dependency Parsing. Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC), 2006, pp. 2216–2219.

[36] Nivre, J.—McDonald, R.: Integrating Graph-Based and Transition-Based Dependency Parsers. Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL), 2008, pp. 950–958.

[37] Nivre, J.—Nilsson, J.: Pseudo-Projective Dependency Parsing. Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL), 2005, pp. 99–106.

[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. Kluwer, 2003, pp. 261–277.

[39] Sagae, K.—Lavie, A.: Parser Combination by Reparsing. Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers, 2006, pp. 129–132.

[40] Samuelsson, Y.—Eklund, J.—Täckström, O.—Velupillai, S.—Saers, M.: Mixing and Blending Syntactic and Semantic Dependencies. Proceedings of CoNLL-2008 Shared Task, 2008, pp. 248–252.

[41] Snow, R.—Jurafsky, D.—Ng, A. Y.: Learning Syntactic Patterns for Automatic Hypernym Discovery. Advances in Neural Information Processing Systems (NIPS), 2005.

[42] Surdeanu, M.—Manning, C. D.: Ensemble Models for Dependency Parsing: Cheap and Good? Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguis-

tics (HLT '10), ACL, 2010, pp. 649–652. 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. Proceedings of the 10th International Conference on Parsing Technologies (IWPT), 2007, pp. 144–155.

[45] Yamada, H.—Matsumoto, Y.: Statistical Dependency Analysis with Support Vector Machines. Proceedings of the 8th International Workshop on Parsing Technologies (IWPT), 2003, pp. 195–206.

[46] Zeman, D.—Žabokrtský, Z.: Improving Parsing Accuracy by Combining Diverse Dependency Parsers. Proceedings of the Ninth International Workshop on Parsing Technology (Parsing '05), ACL, 2005, pp. 171–178. 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. Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), 2008, pp. 562–571.

[48] Zhang, Y.—Clark, S.: Syntactic Processing Using the Generalized Perceptron and Beam Search. Computational Linguistics, Vol. 37, 2011, No. 1, pp. 105–151.

[49] Zhang, Y.—Nivre, J.: Transition-Based Parsing with Rich Non-Local Features. 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), Indian Institute of Technology Bombay, 2012, pp. 1391–1400.

Daniel Fernández-González received his M.Sc. degree in computer science from the University of Vigo (UVIGO) in 2010, and his Ph.D. degree in computer science from the University of A Coruña (UDC) in 2015. He is currently working as a researcher at UDC, but he was affiliated with UVIGO when this work was performed. His main research interest is data-driven dependency parsing, especially transition-based parsing.

Carlos Gómez-Rodríguez received his M.Sc. and Ph.D. degrees in computer science from the University of A Coruña in 2005 and 2009, respectively, and is currently Associate Professor in 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 the University of A Coruña in 2012. He is currently Ph.D. student at the Computer Science Department of the University of A Coruña. His research interests include sentiment analysis and natural language processing.