An Efficient Algorithm for Easy-First Non-Directional Dependency Parsing

Yoav Goldberg[*] and **Michael Elhadad**
Ben Gurion University of the Negev
Department of Computer Science
POB 653 Be'er Sheva, 84105, Israel
{yoavg|elhadad}@cs.bgu.ac.il

Abstract

We present a novel deterministic dependency parsing algorithm that attempts to create the easiest arcs in the dependency structure first in a non-directional manner. Traditional deterministic parsing algorithms are based on a shift-reduce framework: they traverse the sentence from left-to-right and, at each step, perform one of a possible set of actions, until a complete tree is built. A drawback of this approach is that it is extremely local: while decisions can be based on complex structures on the left, they can look only at a few words to the right. In contrast, our algorithm builds a dependency tree by iteratively selecting the best pair of neighbours to connect at each parsing step. This allows incorporation of features from already built structures both to the left and to the right of the attachment point. The parser learns both the attachment preferences and the order in which they should be performed. The result is a deterministic, best-first, $O(nlogn)$ parser, which is significantly more accurate than best-first transition based parsers, and nears the performance of globally optimized parsing models.

1 Introduction

Dependency parsing has been a topic of active research in natural language processing in the last several years. An important part of this research effort are the CoNLL 2006 and 2007 shared tasks (Buchholz and Marsi, 2006; Nivre et al., 2007), which allowed for a comparison of many algorithms and approaches for this task on many languages.

Current dependency parsers can be categorized into three families: local-and-greedy transition-based parsers (*e.g.*, MALTPARSER (Nivre et al., 2006)), globally optimized graph-based parsers (*e.g.*, MSTPARSER (McDonald et al., 2005)), and hybrid systems (*e.g.*, (Sagae and Lavie, 2006b; Nivre and McDonald, 2008)), which combine the output of various parsers into a new and improved parse, and which are orthogonal to our approach.

Transition-based parsers scan the input from left to right, are fast ($O(n)$), and can make use of rich feature sets, which are based on all the previously derived structures. However, all of their decisions are very local, and the strict left-to-right order implies that, while the feature set can use rich structural information from the left of the current attachment point, it is also very restricted in information to the right of the attachment point: traditionally, only the next two or three input tokens are available to the parser. This limited look-ahead window leads to error propagation and worse performance on root and long distant dependencies relative to graph-based parsers (McDonald and Nivre, 2007).

Graph-based parsers, on the other hand, are globally optimized. They perform an exhaustive search over all possible parse trees for a sentence, and find the highest scoring tree. In order to make the search tractable, the feature set needs to be restricted to features over single edges (first-order models) or edges pairs (higher-order models, e.g. (McDonald and Pereira, 2006; Carreras, 2007)). There are several attempts at incorporating arbitrary tree-based features but these involve either solving an ILP problem (Riedel and Clarke, 2006) or using computa-

[*]Supported by the Lynn and William Frankel Center for Computer Sciences, Ben Gurion University

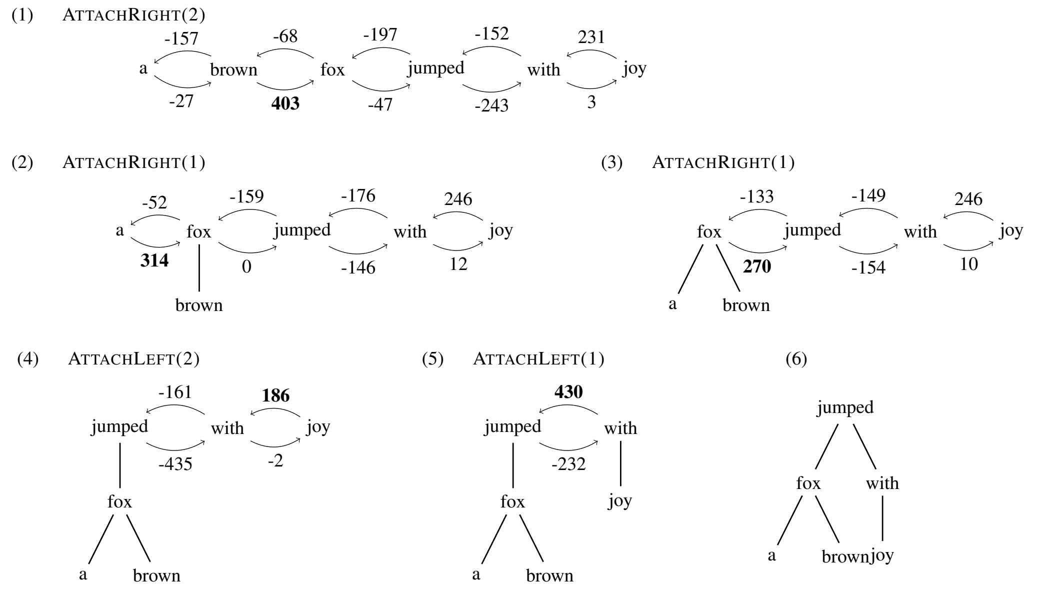

Figure 1: Parsing the sentence “a brown fox jumped with joy”. Rounded arcs represent possible actions.

tionally intensive sampling-based methods (Nakagawa, 2007). As a result, these models, while accurate, are slow ($O(n^3)$ for projective, first-order models, higher polynomials for higher-order models, and worse for richer tree-feature models).

We propose a new category of dependency parsing algorithms, inspired by (Shen et al., 2007): non-directional easy-first parsing. This is a greedy, deterministic parsing approach, which relaxes the left-to-right processing order of transition-based parsing algorithms. By doing so, we allow the explicit incorporation of rich structural features derived from both sides of the attachment point, and implicitly take into account the entire previously derived structure of the whole sentence. This extension allows the incorporation of much richer features than those available to transition- and especially to graph-based parsers, and greatly reduces the locality of transition-based algorithm decisions. On the other hand, it is still a greedy, best-first algorithm leading to an efficient implementation.

We present a concrete $O(nlogn)$ parsing algorithm, which significantly outperforms state-of-the-art transition-based parsers, while closing the gap to graph-based parsers.

2 Easy-first parsing

When humans comprehend a natural language sentence, they arguably do it in an incremental, left-to-right manner. However, when humans consciously annotate a sentence with syntactic structure, they hardly ever work in fixed left-to-right order. Rather, they start by building several isolated constituents by making easy and local attachment decisions and only then combine these constituents into bigger constituents, jumping back-and-forth over the sentence and proceeding from easy to harder phenomena to analyze. When getting to the harder decisions a lot of structure is already in place, and this structure can be used in deciding a correct attachment.

Our parser follows a similar kind of annotation process: starting from easy attachment decisions, and proceeding to harder and harder ones. When making later decisions, the parser has access to the entire structure built in earlier stages. During the training process, the parser learns its own notion of easy and hard, and learns to defer specific kinds of decisions until more structure is available.

3 Parsing algorithm

Our (projective) parsing algorithm builds the parse tree bottom up, using two kinds of actions: **ATTACHLEFT(*i*)** and **ATTACHRIGHT(*i*)** . These actions are applied to a list of partial structures $p_1, \ldots, p_k$, called $pending$, which is initialized with the n words of the sentence $w_1, \ldots, w_n$. Each action connects the heads of two neighbouring structures, making one of them the parent of the other, and removing the daughter from the list of partial structures. **ATTACHLEFT(*i*)** adds a dependency edge (p_i, p_{i+1}) and removes p_{i+1} from the list. **ATTACHRIGHT(*i*)** adds a dependency edge (p_{i+1}, p_i) and removes p_i from the list. Each action shortens the list of partial structures by 1, and after $n-1$ such actions, the list contains the root of a connected projective tree over the sentence.

Figure 1 shows an example of parsing the sentence "a brown fox jumped with joy". The pseudocode of the algorithm is given in Algorithm 1.

Algorithm 1: Non-directional Parsing

```
   Input: a sentence= $w_1 \dots w_n$
   Output: a set of dependency arcs over the
           sentence ($Arcs$)
1  $Acts = \{$ATTACHLEFT, ATTACHRIGHT$\}$
2  $Arcs \leftarrow \{\}$
3  $pending = p_1 \dots p_n \leftarrow w_1 \dots w_n$
4  while $length(pending) > 1$ do
5      $best \leftarrow \arg\max_{act \in Acts, 1 \le i \le len(pending)} score(act(i))$
6      $(parent, child) \leftarrow$ edgeFor($best$)
7      $Arcs$.add( $(parent, child)$ )
8      $pending$.remove($child$)
9  end
10 return $Arcs$
```

$$\text{edgeFor(act(i))} = \begin{cases} (p_i, p_{i+1}) & \text{ATTACHLEFT}(i) \\ (p_{i+1}, p_i) & \text{ATTACHRIGHT}(i) \end{cases}$$

At each step the algorithm chooses a specific action/location pair using a function $score($**ACTION(*i*)**$)$, which assign scores to action/location pairs based on the partially built structures headed by p_i and p_{i+1}, as well as neighbouring structures. The $score()$ function is learned from data. This scoring function reflects not only the correctness of an attachment, but also *the order* in which attachments should be made. For example, consider the attachments (brown,fox) and (joy,with) in Figure (1.1). While both are correct, the scoring function prefers the (adjective,noun) attachment over the (prep,noun) attachment. Moreover, the attachment (jumped,with), while correct, receives a negative score for the bare preposition "with" (Fig. (1.1) - (1.4)), and a high score once the verb has its subject and the PP "with joy" is built (Fig. (1.5)). Ideally, we would like to score easy and reliable attachments *higher than* harder less likely attachments, thus performing attachments in order of confidence. This strategy allows us both to limit the extent of error propagation, and to make use of richer contextual information in the later, harder attachments. Unfortunately, this kind of ordering information is not directly encoded in the data. We must, therefore, learn how to order the decisions.

We first describe the learning algorithm (Section 4) and a feature representation (Section 5) which enables us to learn an effective scoring function.

4 Learning Algorithm

We use a linear model $score(x) = \vec{w} \cdot \phi(x)$, where $\phi(x)$ is a feature representation and $\vec{w}$ is a weight vector. We write $\phi_{act(i)}$ to denote the feature representation extracted for action act at location i. The model is trained using a variant of the structured perceptron (Collins, 2002), similar to the algorithm of (Shen et al., 2007; Shen and Joshi, 2008). As usual, we use parameter averaging to prevent the perceptron from overfitting.

The training algorithm is initialized with a zero parameter vector $\vec{w}$. The algorithm makes several passes over the data. At each pass, we apply the training procedure given in Algorithm 2 to every sentence in the training set.

At training time, each sentence is parsed using the parsing algorithm and the current $\vec{w}$. Whenever an invalid action is chosen by the parsing algorithm, it is not performed (line 6). Instead, we update the parameter vector $\vec{w}$ by decreasing the weights of the features associated with the *invalid* action, and increasing the weights for the currently highest scoring *valid* action.[1] We then proceed to parse the sentence with the updated values. The process repeats until a valid action is chosen.

Note that each single update does not guarantee that the next chosen action is valid, or even different than the previously selected action. Yet, this is still an aggressive update procedure: we do not leave a sentence until our parameters vector parses it cor-

[1] We considered 3 variants of this scheme: (1) using the highest scoring valid action, (2) using the leftmost valid action, and (3) using a random valid action. The 3 variants achieved nearly identical accuracy, while (1) converged somewhat faster than the other two.

rectly, and we do not proceed from one partial parse to the next until $\vec{w}$ predicts a correct location/action pair. However, as the best ordering, and hence the best attachment point is not known to us, we do not perform a single aggressive update step. Instead, our aggressive update is performed incrementally in a series of smaller steps, each pushing $\vec{w}$ away from invalid attachments and toward valid ones. This way we integrate the search of confident attachments into the learning process.

Algorithm 2: Structured perceptron training for direction-less parser, over one sentence.

```
Input: sentence,gold arcs,current $\vec{w}$,feature
       representation $\phi$
Output: weight vector $\vec{w}$

1  $Arcs \leftarrow \{\}$
2  $pending \leftarrow sent$
3  while $length(pending) > 1$ do
4      $allowed \leftarrow \{act(i) | isValid(act(i), Gold, Arcs)\}$
5      $choice \leftarrow \underset{act \in Acts,\ 1 \le i \le len(pending)}{\arg\max} \vec{w} \cdot \phi_{act(i)}$
6      if $choice \in allowed$ then
7          $(parent, child) \leftarrow$ edgeFor($choice$)
8          $Arcs$.add( $(parent, child)$ )
9          $pending$.remove($child$)
10     else
11         $good \leftarrow \underset{act(j) \in allowed}{\arg\max} \vec{w} \cdot \phi_{act(j)}$
12         $\vec{w} \leftarrow \vec{w} + \phi_{good} - \phi_{choice}$
13 end
14 return $\vec{w}$
```

Function `isValid`(*action,Gold,Arcs*)

```
1  $(p, c) \leftarrow$ edgeFor($action$)
2  if $(\exists c' : (c, c') \in Gold \wedge (c, c') \notin Arcs)$
   $\vee (p, c) \notin Gold$ then
3      return false
4  return true
```

The function $isValid(act(i), gold, arcs)$ (line 4) is used to decide if the chosen action/location pair is valid. It returns True if two conditions apply: (a) (p_i, p_j) is present in $gold$, (b) all edges $(\square, p_j)$ in $gold$ are also in $arcs$. In words, the function verifies that the proposed edge is indeed present in the gold parse and that the suggested daughter already found all its own daughters.[2]

[2]This is in line with the Arc-Standard parsing strategy of shift-reduce dependency parsers (Nivre, 2004). We are currently experimenting also with an Arc-Eager variant of the non-directional algorithm.

5 Feature Representation

The feature representation for an action can take into account the original sentence, as well as the entire parse history: $\phi_{act(i)}$ above is actually $\phi(act(i), sentence, Arcs, pending)$.

We use binary valued features, and each feature is conjoined with the type of action.

When designing the feature representation, we keep in mind that our features should not only direct the parser toward desired actions and away from undesired actions, but also provide the parser with means of choosing between several desired actions. We want the parser to be able to defer some desired actions until more structure is available and a more informed prediction can be made. This desire is reflected in our choice of features: some of our features are designed to signal to the parser the presence of possibly "incomplete" structures, such as an incomplete phrase, a coordinator without conjuncts, and so on.

When considering an action ACTION(i), we limit ourselves to features of partial structures around the attachment point: $p_{i-2}, p_{i-1}, p_i, p_{i+1}, p_{i+2}, p_{i+3}$, that is the two structures which are to be attached by the action (p_i and p_{i+1}), and the two neighbouring structures on each side[3].

While these features encode local context, it is local in terms of syntactic structure, and not purely in terms of sentence surface form. This let us capture some, though not all, long-distance relations.

For a partial structure p, we use w_p to refer to the head word form, t_p to the head word POS tag, and lc_p and rc_p to the POS tags of the left-most and right-most child of p respectively.

All our prepositions (IN) and coordinators (CC) are lexicalized: for them, t_p is in fact $w_p t_p$.

We define *structural*, *unigram*, *bigram* and *pp-attachment* features.

The *structural* features are: the length of the structures (len_p), whether the structure is a word (contains no children: nc_p), and the surface distance between structure heads ($\Delta_{p_i p_j}$). The *unigram* and *bigram* features are adapted from the feature set for left-to-right Arc-Standard dependency parsing de-

[3]Our sentences are padded from each side with sentence delimiter tokens.

Structural	
for p in $p_{i-2}, p_{i-1}, p_i, p_{i+1}, p_{i+2}, p_{i+3}$	len_p , nc_p
for p,q in (p_{i-2}, p_{i-1}),(p_{i-1}, p_i),(p_i, p_{i+1}),$(p_{i+1}, pi+2)$,(p_{i+2}, p_{i+3})	Δ_{qp} , $\Delta_{qp} t_p t_q$
Unigram	
for p in $p_{i-2}, p_{i-1}, p_i, p_{i+1}, p_{i+2}, p_{i+3}$	t_p , w_p , $t_p lc_p$, $t_p rc_p$, $t_p rc_p lc_p$
Bigram	
for p,q in (p_i, p_{i+1}),(p_i, p_{i+2}),(p_{i-1}, p_i),(p_{i-1}, p_{i+2}),(p_{i+1}, p_{i+2})	$t_p t_q$, $w_p w_q$, $t_p w_q$, $w_p t_q$ $t_p t_q lc_p lc_q$, $t_p t_q rc_p lc_q$ $t_p t_q lc_p rc_q$, $t_p t_q rc_p rc_q$
PP-Attachment	
if p_i is a preposition	$w_{p_{i-1}} w_{p_i} rc_{p_i}$, $t_{p_{i-1}} w_{p_i} rcw_{p_i}$
if p_{i+1} is a preposition	$w_{p_{i-1}} w_{p_{i+1}} rc_{p_{i+1}}$, $t_{p_{i-1}} w_{p_{i+1}} rcw_{p_{i+1}}$ $w_{p_i} w_{p_{i+1}} rc_{p_{i+1}}$, $t_{p_i} w_{p_{i+1}} rcw_{p_{i+1}}$
if p_{i+2} is a preposition	$w_{p_{i+1}} w_{p_{i+2}} rc_{p_{i+2}}$, $t_{p_{i+1}} w_{p_{i+2}} rcw_{p_{i+2}}$ $w_{p_i} w_{p_{i+2}} rc_{p_{i+2}}$, $t_{p_i} w_{p_{i+2}} rcw_{p_{i+2}}$

Figure 2: Feature Templates

scribed in (Huang et al., 2009). We extended that feature set to include the structure on both sides of the proposed attachment point.

In the case of unigram features, we added features that specify the POS of a word and its left-most and right-most children. These features provide the non-directional model with means to prefer some attachment points over others based on the types of structures already built. In English, the left- and right-most POS-tags are good indicators of constituency.

The *pp-attachment* features are similar to the bigram features, but fire only when one of the structures is headed by a preposition (IN). These features are more lexicalized than the regular bigram features, and include also the word-form of the right-most child of the PP (rcw_p). This should help the model learn lexicalized attachment preferences such as (hit, with-bat).

Figure 2 enumerate the feature templates we use.

6 Computational Complexity and Efficient Implementation

The parsing algorithm (Algorithm 1) begins with $n+1$ disjoint structures (the words of the sentence + ROOT symbol), and terminates with one connected structure. Each iteration of the main loop connects two structures and removes one of them, and so the loop repeats for exactly n times.

The argmax in line 5 selects the maximal scoring action/location pair. At iteration i, there are $n - i$ locations to choose from, and a naive computation of the argmax is $O(n)$, resulting in an $O(n^2)$ algorithm.

Each performed action changes the partial structures and with it the extracted features and the computed scores. However, these changes are limited to a fixed local context around the attachment point of the action. Thus, we observe that the feature extraction and score calculation can be performed once for each action/location pair in a given sentence, and reused throughout all the iterations. After each iteration we need to update the extracted features and calculated scores for only k locations, where k is a fixed number depending on the window size used in the feature extraction, and usually $k \ll n$.

Using this technique, we perform only $(k+1)n$ feature extractions and score calculations for each sentence, that is $O(n)$ feature-extraction operations per sentence.

Given the scores for each location, the argmax can then be computed in $O(logn)$ time using a heap, resulting in an $O(nlogn)$ algorithm: n iterations, where the first iteration involves n feature extraction operations and n heap insertions, and each subsequent iteration involves k feature extractions and heap updates.

We note that the dominating factor in polynomial-time discriminative parsers, is by far the feature-extraction and score calculation. It makes sense to compare parser complexity in terms of these operations only.[4] Table 1 compares the complexity of our

[4]Indeed, in our implementation we do not use a heap, and opt instead to find the argmax using a simple $O(n)$ *max* operation. This $O(n^2)$ algorithm is faster in practice than the heap based one, as both are dominated by the $O(n)$ feature extraction, while the cost of the $O(n)$ *max* calculationis negligible compared to the constants involved in heap maintenance.

parser to other dependency parsing frameworks.

Parser	Runtime	Features / Scoring
MALT	$O(n)$	$O(n)$
MST	$O(n^3)$	$O(n^2)$
MST2	$O(n^3)$	$O(n^3)$
BEAM	$O(n * beam)$	$O(n * beam)$
NONDIR (This Work)	$O(nlogn)$	$O(n)$

Table 1: Complexity of different parsing frameworks. MST: first order MST parser, MST2: second order MST parser, MALT: shift-reduce left-to-right parsing. BEAM: beam search parser, as in (Zhang and Clark, 2008)

In terms of feature extraction and score calculation operations, our algorithm has the same cost as traditional shift-reduce (MALT) parsers, and is an order of magnitude more efficient than graph-based (MST) parsers. Beam-search decoding for left-to-right parsers (Zhang and Clark, 2008) is also linear, but has an additional linear dependence on the beam-size. The reported results in (Zhang and Clark, 2008) use a beam size of 64, compared to our constant of $k = 6$.

Our Python-based implementation[5] (the perceptron is implemented in a C extension module) parses about 40 tagged sentences per second on an Intel based MacBook laptop.

7 Experiments and Results

We evaluate the parser using the WSJ Treebank. The trees were converted to dependency structures with the Penn2Malt conversion program,[6] using the head-finding rules from (Yamada and Matsumoto, 2003).[7]

We use Sections 2-21 for training, Section 22 for development, and Section 23 as the final test set. The text is automatically POS tagged using a trigram HMM based POS tagger prior to training and parsing. Each section is tagged after training the tagger on all other sections. The tagging accuracy of the tagger is 96.5 for the training set and 96.8 for the test set. While better taggers exist, we believe that the simpler HMM tagger overfits less, and is more representative of the tagging performance on non-WSJ corpus texts.

Parsers We evaluate our parser against the transition-based MALT parser and the graph-based MST parser. We use version 1.2 of MALT parser[8], with the settings used for parsing English in the CoNLL 2007 shared task. For the MST parser[9], we use the default first-order, projective parser settings, which provide state-of-the-art results for English. All parsers are trained and tested on the same data. Our parser is trained for 20 iterations.

Evaluation Measures We evaluate the parsers using three common measures:

(unlabeled) Accuracy: percentage of tokens which got assigned their correct parent.

Root: The percentage of sentences in which the ROOT attachment is correct.

Complete: the percentage of sentences in which all tokens were assigned their correct parent.

Unlike most previous work on English dependency parsing, we *do not* exclude punctuation marks from the evaluation.

Results are presented in Table 2. Our non-directional easy-first parser significantly outperforms the left-to-right greedy MALT parser in terms of accuracy and root prediction, and significantly outperforms both parsers in terms of exact match. The globally optimized MST parser is better in root-prediction, and slightly better in terms of accuracy.

We evaluated the parsers also on the English dataset from the CoNLL 2007 shared task. While this dataset is also derived from the WSJ Treebank, it differs from the previous dataset in two important aspects: it is much smaller in size, and it is created using a different conversion procedure, which is more linguistically adequate. For these experiments, we use the dataset POS tags, and the same parameters as in the previous set of experiments: we train the non-directional parser for 20 iterations, with the same feature set. The CoNLL dataset contains some non-projective constructions. MALT and MST deal with non-projectivity. For the non-directional parser, we projectivize the training set prior to training using the procedure described in (Carreras, 2007).

Results are presented in Table 3.

[5]http://www.cs.bgu.ac.il/~yoavg/software/

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

[7]While other and better conversions exist (see, *e.g.*, (Johansson and Nugues, 2007; Sangati and Mazza, 2009)), this conversion heuristic is still the most widely used. Using the same conversion facilitates comparison with previous works.

[8]http://maltparser.org/dist/1.2/malt-1.2.tar.gz

[9]http://sourceforge.net/projects/mstparser/

Parser	Accuracy	Root	Complete
MALT	88.36	87.04	34.14
MST	90.05	93.95	34.64
NONDIR (this work)	89.70	91.50	37.50

Table 2: Unlabeled dependency accuracy on PTB Section 23, automatic POS-tags, including punctuation.

Parser	Accuracy	Root	Complete
MALT	85.82	87.85	24.76
MST	89.08	93.45	24.76
NONDIR (this work)	88.34	91.12	29.43

Table 3: Unlabeled dependency accuracy on CoNLL 2007 English test set, including punctuation.

While all models suffer from the move to the smaller dataset and the more challenging annotation scheme, the overall story remains the same: the non-directional parser is better than MALT but not as good as MST in terms of parent-accuracy and root prediction, and is better than both MALT and MST in terms of producing complete correct parses.

That the non-directional parser has lower accuracy but more exact matches than the MST parser can be explained by it being a deterministic parser, and hence still vulnerable to error propagation: once it erred once, it is likely to do so again, resulting in low accuracies for some sentences. However, due to the easy-first policy, it manages to parse many sentences without a single error, which lead to higher exact-match scores. The non-directional parser avoids error propagation by not making the initial error. On average, the non-directional parser manages to assign correct heads to over 60% of the tokens before making its first error.

The MST parser would have ranked 5^{th} in the shared task, and NONDIR would have ranked 7^{th}. The better ranking systems in the shared task are either higher-order global models, beam-search based systems, or ensemble-based systems, all of which are more complex and less efficient than the NONDIR parser.

Parse Diversity The parses produced by the non-directional parser are different than the parses produced by the graph-based and left-to-right parsers. To demonstrate this difference, we performed an Oracle experiment, in which we combine the output of several parsers by choosing, for each sentence, the parse with the highest score. Results are presented in Table 4.

Combination	Accuracy	Complete
Penn2Malt, Train 2-21, Test 23		
MALT+MST	92.29	44.03
NONDIR+MALT	92.19	45.48
NONDIR+MST	92.53	44.41
NONDIR+MST+MALT	93.54	49.79
CoNLL 2007		
MALT+MST	91.50	33.64
NONDIR+MALT	91.02	34.11
NONDIR+MST	91.90	34.11
NONDIR+MST+MALT	92.70	38.31

Table 4: Parser combination with Oracle, choosing the highest scoring parse for each sentence of the test-set.

A non-oracle blending of MALT+MST+NONDIR using Sagae and Lavie's (2006) simplest combination method assigning each component the same weight, yield an accuracy of 90.8 on the CoNLL 2007 English dataset, making it the highest scoring system among the participants.

7.1 Error Analysis / Limitations

When we investigate the POS category of mistaken instances, we see that for all parsers, nodes with structures of depth 2 and more which are assigned an incorrect head are predominantly PPs (headed by 'IN'), followed by NPs (headed by 'NN'). All parsers have a hard time dealing with PP attachment, but MST parser is better at it than NONDIR, and both are better than MALT.

Looking further at the mistaken instances, we notice a tendency of the PP mistakes of the NONDIR parser to involve, before the PP, an NP embedded in a relative clause. This reveals a limitation of our parser: recall that for an edge to be built, the child must first acquire all its own children. This means that in case of relative clauses such as "I saw the boy [who ate the pizza] with my eyes", the parser must decide if the PP "with my eyes" should be attached to "the pizza" or not *before* it is allowed to build parts of the outer NP ("the boy who…"). In this case, the verb "saw" and the noun "boy" are both outside of the sight of the parser when deciding on the PP attachment, and it is forced to make a decision in ignorance, which, in many cases, leads to mistakes. The globally optimized MST does not suffer as much from such cases. We plan to address this deficiency in future work.

8 Related Work

Deterministic shift-reduce parsers are restricted by a strict left-to-right processing order. Such parsers can rely on rich syntactic information on the left, but not on the right, of the decision point. They are forced to commit early, and suffer from error propagation. Our non-directional parser addresses these deficiencies by discarding the strict left-to-right processing order, and attempting to make easier decisions before harder ones. Other methods of dealing with these deficiencies were proposed over the years:

Several Passes Yamada and Matsumoto's (2003) pioneering work introduces a shift-reduce parser which makes several left-to-right passes over a sentence. Each pass adds structure, which can then be used in subsequent passes. Sagae and Lavie (2006b) extend this model to alternate between left-to-right and right-to-left passes. This model is similar to ours, in that it attempts to defer harder decisions to later passes over the sentence, and allows late decisions to make use of rich syntactic information (built in earlier passes) on both sides of the decision point. However, the model is not explicitly trained to optimize attachment ordering, has an $O(n^2)$ runtime complexity, and produces results which are inferior to current single-pass shift-reduce parsers.

Beam Search Several researchers dealt with the early-commitment and error propagation of deterministic parsers by extending the greedy decisions with various flavors of beam-search (Sagae and Lavie, 2006a; Zhang and Clark, 2008; Titov and Henderson, 2007). This approach works well and produces highly competitive results. Beam search can be incorporated into our parser as well. We leave this investigation to future work.

Strict left-to-right ordering is also prevalent in sequence tagging. Indeed, one major influence on our work is Shen *et.al.*'s bi-directional POS-tagging algorithm (Shen et al., 2007), which combines a perceptron learning procedure similar to our own with beam search to produce a state-of-the-art POS-tagger, which does not rely on left-to-right processing. Shen and Joshi (2008) extends the bidirectional tagging algorithm to LTAG parsing, with good results. We build on top of that work and present a concrete and efficient greedy non-directional dependency parsing algorithm.

Structure Restrictions Eisner and Smith (2005) propose to improve the efficiency of a globally optimized parser by posing hard constraints on the lengths of arcs it can produce. Such constraints pose an explicit upper bound on parser accuracy.[10] Our parsing model does not pose such restrictions. Shorter edges are arguably easier to predict, and our parses builds them early in time. However, it is also capable of producing long dependencies at later stages in the parsing process. Indeed, the distribution of arc lengths produced by our parser is similar to those produced by the MALT and MST parsers.

9 Discussion

We presented a *non-directional* deterministic dependency parsing algorithm, which is not restricted by the left-to-right parsing order of other deterministic parsers. Instead, it works in an *easy-first* order. This strategy allows using more context at each decision. The parser learns both *what* and *when* to connect. We show that this parsing algorithm significantly outperforms a left-to-right deterministic algorithm. While it still lags behind globally optimized parsing algorithms in terms of accuracy and root prediction, it is much better in terms of exact match, and much faster. As our parsing framework can easily and efficiently utilize more structural information than globally optimized parsers, we believe that with some enhancements and better features, it can outperform globally optimized algorithms, especially when more structural information is needed, such as for morphologically rich languages.

Moreover, we show that our parser produces different structures than those produced by both left-to-right and globally optimized parsers, making it a good candidate for inclusion in an ensemble system. Indeed, a simple combination scheme of graph-based, left-to-right and non-directional parsers yields state-of-the-art results on English dependency parsing on the CoNLL 2007 dataset.

We hope that further work on this non-directional parsing framework will pave the way to better understanding of an interesting cognitive question: which kinds of parsing decisions are hard to make, and which linguistic constructs are hard to analyze?

[10]In (Dreyer et al., 2006), constraints are chosen "to be the minimum value that will allow recovery of 90% of the left (right) dependencies in the training corpus".

References

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proc. of CoNLL*.

Xavier Carreras. 2007. Experiments with a higher-order projective dependency parser. In *Proc. of CoNLL Shared Task, EMNLP-CoNLL*.

Michael Collins. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proc of EMNLP*.

Markus Dreyer, David A. Smith, and Noah A. Smith. 2006. Vine parsing and minimum risk reranking for speed and precision. In *Proc. of CoNLL*, pages 201–205, Morristown, NJ, USA. Association for Computational Linguistics.

Jason Eisner and Noah A. Smith. 2005. arsing with soft and hard constraints on dependency length. In *Proc. of IWPT*.

Liang Huang, Wenbin Jiang, and Qun Liu. 2009. Bilingually-constrained (monolingual) shift-reduce parsing. In *Proc of EMNLP*.

Richard Johansson and Pierre Nugues. 2007. Extended constituent-to-dependency conversion for english. In *Proc of NODALIDA*.

Ryan McDonald and Joakim Nivre. 2007. Characterizing the errors of data-driven dependency parsing models. In *Proc. of EMNLP*.

Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *Proc of EACL*.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proc of ACL*.

Tetsuji Nakagawa. 2007. Multilingual dependency parsing using global features. In *Proc. of EMNLP-CoNLL*.

Joakim Nivre and Ryan McDonald. 2008. Integrating graph-based and transition-based dependency parsers. In *Proc. of ACL*, pages 950–958, Columbus, Ohio, June. Association for Computational Linguistics.

Joakim Nivre, Johan Hall, and Jens Nillson. 2006. MaltParser: A data-driven parser-generator for dependency parsing. In *Proc. of LREC*.

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 *Proc. of EMNLP-CoNLL*.

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Incremental Parsing: Bringing Engineering and Cognition Together, ACL-Workshop*.

Sebastian Riedel and James Clarke. 2006. Incremental integer linear programming for non-projective dependency parsing. In *Proc. of EMNLP 2006*, July.

Kenji Sagae and Alon Lavie. 2006a. A best-first probabilistic shift-reduce parser. In *Proc of ACL*.

Kenji Sagae and Alon Lavie. 2006b. Parser combination by reparsing. In *Proc of NAACL*.

Federico Sangati and Chiara Mazza. 2009. An english dependency treebank à la tesnière. In *Proc of TLT8*.

Libin Shen and Aravind K. Joshi. 2008. Ltag dependency parsing with bidirectional incremental construction. In *Proc of EMNLP*.

Libin Shen, Giorgio Satta, and Aravind K. Joshi. 2007. Guided learning for bidirectional sequence classification. In *Proc of ACL*.

Ivan Titov and James Henderson. 2007. Fast and robust multilingual dependency parsing with a generative latent variable model. In *Proc. of EMNLP-CoNLL*.

Yamada and Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proc. of IWPT*.

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proc of EMNLP*.