Synchronous Parsing of Syntactic and Semantic Structures

Bernd Bohnet
International Computer Science Institute
1947 Center Street
Berkeley 94704, California
bohnet@icsi.Berkeley.edu

Abstract

We describe in this paper an approach for synchronous parsing of syntactic and semantic dependency structures that combines recent advances in the area to get a very high accuracy as well at the same time very good parsing times. The time for parsing, the time for training and the values of the memory footprint are to our knowledge the best results reported while the parsing accuracy are as high as the highest results reported in the 2008 shared task. The corpora used in the shared task are still different to the dependency structures of the Meaning-Text Theory. Therefore, we outline the adaption of the approach to the dependency structures of the Meaning-Text Theory.

1 Introduction

Recently, dependency parsing made large advances. Reasons for this are influential work of some researches and shared tasks for dependency parsing in the years 2006, 2007 (cf. (Buchholz and Marsi, 2006), (Nivre et al., 2007)) and the shared tasks for joint parsing of dependency and semantic structures in the years 2008 and an upcoming one in 2009 (cf. (Surdeanu et al., 2008)). There are two main approaches to dependency parsing: Maximum Spanning Tree (MST) based dependency parsing (Eisner, 1996; McDonald and Pereira, 2006) and transition based dependency parsing, cf. (Nivre and Nilsson, 2004). In this paper, we use the first approach since we could better improve the parsing speed and the MST based dependency parsing approach has a slightly better accuracy. To our knowlege there are only a few MTT parser available and even less attempts have been made to parse dependency trees of different representation levels with statistical trained parser. One of the exceptions is a transition based parser, which was train on the Russian SYNTAGRUS tree-bank, cf. (Nivre et al., 2008).

2 Parsing Algorithm

We adopted the second order MST parsing algorithm as outlined in Eisner (1996). This algorithm has a higher accuracy compared to the first order parsing algorithm since it considers also siblings and grandchildren of a node. Algorithm 1 shows the first order algorithm which is the basis for the second order parsing algorithm. Eisner (1996) first order approach can compute a projective dependency tree within cubic time (O(n^3)). Therefore, the algorithm shown in Algorithm 1 has at most three nested loops.

Both algorithms are bottom up parsing algorithms based on dynamic programming similar to the CKY chart parsing algorithm. The score for a dependency tree is the sum of all edge scores. The following equation describes this formally.

$$score(S,t) = \sum_{\forall(i,j)\in E} score(i,j)$$

The score of the sentence S and a tree t over S is defined as the sum of all edge scores where the words of S are $w_0...w_1$. The tree consists of set of nodes N and set of edges $E = \langle N \times N \rangle$. The word indices $(0..n)$ are the elements of the node set N. The expression $(i,j) \in E$ denotes an edge which is going from the node i to the node j.

The parsing Algorithm 1 searches the tree with the best score bottom up considering only the scores of single edges. Therefore, it is call first order dependency parsing algorithm. The score function in the algorithm scores always a subtree and the algorithm search for the best combination of smaller trees in order to build larger trees which maximize the overall score of the tree.

Algorithm 1: First Order Parsing Algorithm

```
// $S \leftarrow w_0...w_n$ is the sentence with the words $w_i$
// $l_S$ = $n+1$ is the sentence length
// $N$ = $0...n$ the indices of the words
// $E$ = $N \times N$ set of edges
// $C$ = $(C_{left}, C_{right}, E)$ the chart element where $C_{left}$ and $C_{right}$is a pointer to another chart element
// $D$ = $\{0,1\}$ D represents the direction of the edge
// $C_O$ = $N \times N \times D \times C$ the chart with the open sub-trees
// $C_C$ = $N \times N \times D \times C$ the chart with the closed sub-trees
for j ← 1 to $l_S$ do
  for s ← 0 to $l_S$ do
    t ← s + j
    if t > $l_S$ then break
    for r = s to t-1 do
      if (score($C_O$[s][t][1])<score($C_C$[s][r][1]) + score($C_C$[r + 1][t][0])) then
        $C_O$[s][t][1] ← ($C_C$[s][r][1], $C_C$[r + 1][t][0]),[s→t])
      if (score($C_O$[s][t][0])<score($C_C$[s][r][1]) + score($C_C$[r + 1][t][0])) then
        $C_O$[s][t][0] ← ($C_C$[s][r][1], $C_C$[r + 1][t][0]),[s←t])
    end for
    for r = s+1 to t do
      if (score($C_C$[s][t][1])<score(O[s][r][1]) + score(C[r][t][1])) then
        $C_C$[s][t][1] ← ($C_C$[s][r][1], $C_C$[r][t][1]),[s→t])
    end for
    for r = s to t-1 do
      if (score($C_C$[s][t][0])<score($C_C$[s][r][0]) + score($C_O$[r][t][0])) then
        $C_C$[s][t][0] ← ($C_C$[s][r][1], $C_O$[r][t][1]),[s←t])
    end for
  end for
end for
```

We compute the edge score ($score(i,j)$) as the scalar product of a feature vector representation of each edge $\overrightarrow{f_S}(i,j)$ with a weight vector $\overrightarrow{w}$ where i,j are the indices of the words in a sentence. The feature vector f_S might take into account not only the words with indices i and j but also additional values such as the words before and after the words w_i and w_j. The following equation shows the score function.

$$score(i,j) = \overrightarrow{f_S}(i,j) * \overrightarrow{w}$$

Most of the features are built out of the properties from the words such as part-of-speech, morphologic features, lemmas, and forms. For instance, some of the features for the words 2 (bought) and 5 (computer) of the sentence below would be {`VB+N, VB+N+Distance_3, buy+N, VB+computer, buy+computer`}. The features are frequently encoded as strings and mapped to a number. The number becomes the index of the feature in the feature vector and weight vector. Therefore, a feature vector looks like the following expression $f_S(2,5) = \{0,0,0,1,0,0,..,0,1,..,1,..1,..\}$ [1] and the weight vector looks

[1] A implementation of such a sparse vector has to store the values more efficient.

similar, e.g. $w = \{0.1258, -0.2554, 0, 0.333, -0.0125, ...\}$.

He$_1$ bought$_2$ a$_3$ new$_4$ computer$_5$ which$_6$ was$_7$ very$_8$ expensive$_9$.$_{10}$

In order to compute the weight vector, we use a support vector machine which has proven to be efficient for dependency parsing. The support vector machine implements online large margin multi-class learning, cf. (Crammer et al., 2003; McDonald et al., 2005). We provide more details in Section 3.

2.1 Second-Order Dependency Parsing

The first order dependency parsing algorithm takes only into account the parent and one dependent. While the second order algorithm uses information of the composition of the subtrees namely the edges of grandchildren and siblings. This improves the parsing accuracy since for instance an edge to a preposition has to be followed mostly by an edge to a noun to complete the prepositional part and also coordination consist always of more than one edge.

2.2 Labeled Dependency Parsing

Algorithm 1 builds an unlabeled dependency tree. However, all new dependency tree banks have trees with labeled edges. The following two approaches are common to solve this problem: The first approach uses an additional algorithm to label the edges in a post-processing step. The second approaches extends the parsing algorithm and integrates the labeling algorithm into the parsing algorithm.

McDonald et al. (2006) use an additional algorithm. Their two stage model has a good computational complexity since the labeling algorithm contributes again only a cubic time complexity to the algorithm (O(n^3)) and keeps therefore the joint algorithm still cubic. This solution computes the edge labels for each possible edge separate from the unlabeled dependency tree. The algorithm has three loops. The first two loops iterate over the words of the sentence and they build a matrix which refer to all possible edges i, j. The third loop iterates over all possible labels and selects the highest scored label due to the score function $score(w_i, label) + score(w_j, label)$ and inserts the highest scored label into the matrix. The scores are also used in the parsing algorithms and added to the edge scores which improves the overall parsing results as well. In the first order parsing scenario, this procedure is sufficient since no combination of edges are considered by the parsing algorithm. However, in the second order parsing scenario where more than one edge are considered by the parsing algorithm, combinations of two edges might be more accurate than two single edges with the highest score.

Carreras (2007) as well as Johansson and Nugues (2008) combine edge labeling with the second order parsing algorithm. This adds an additional loop over the edge labels to the parsing algorithm. The complexity is therefore O(n^4). This increases in our experiments the parsing accuracy from about 0.86 labeled accuracy score to 0.88 what is a relative high improvement. We got the first value for our partly reimplementation of the McDonald and Pereira (2006) parser and the second value for the parsing algorithm that includes labels. For our experiments, we used the English dependency tree bank as provided in the CoNLL shared task 2008.

2.3 Non-Projective Dependency Parsing

The dependency parsers developed in the last few years use two different techniques for non-projective dependency parsing. The most common technique uses tree rewriting and was invented by Kahane et al. (1998). This technique was taken up again by Nivre and Nilsson (2005) in Nivre‘s transition based dependency parser which performed second best in the 2006 shared task which has therefore become well known.

By using pseudo-projective dependency parsing, the training data for the parser is first projectivized by applying a minimal number of lifting operations to the non-projective edges and encoding information about these lifts in edge labels. After this operations, the trees are projective and therefore a projective dependency parser can be applied. During the train, the parser learns also to built trees with the lifted edges and so indirect to built non-projective dependency trees since after the projective dependency parsing the inverse operations to the lifting are performed and by this operation the edges are moved downwards the tree and non-projective trees are built.

McDonald and Pereira (2006) developed a technique to rearrange edges in the tree in a postprocessing step after the projective parsing has taken place. They call the algorithm approximation non-projective dependency parsing. It searches first the highest scoring projective parse tree and then it rearranges edges in the tree, in each step one, until the rearrangements does not anymore increase the score for the tree. This technique is computationally expensive for trees with a large number of non-projective edges since it considers to reattach all edges to any other node until no higher scoring trees can be found. Their argument for the algorithm is that most edges in a tree even in language with lot of non-projective sentences, the portion of non-projective edges are still small and therefore by starting with the highest scoring projective tree, typically the highest scoring non-projective tree is only a small number of transformations away from.

Threshold	Labeled Accuracy Score (LAS)	Unlabeled Accuracy Score (UAS)
projective	0.86711	0.90350
0.0	0.86653	0.90127
0.8	0.86727	0.90416
1.0	0.86852	0.90619
1.1	0.86880	0.90530
1.2	0.86790	0.90421

Table 1: Accuracy Scores of different thresholds for the approximation non-projective dependency parsing.

We found out in our experiments with the non-projective approximation algorithm that with a threshold higher then about 0.7, the parsing accuracy even for English slightly improves. With a threshold of 1.1, we got the best improvement. The results of this experiment are summarized in Table 1. Since we opt for a high labeled accuracy score, we selected a threshold of 1.1 also if the unlabeled score for 1.0 is higher.

3 Parsing Framework

One of the main goals of the paper is to show how such a parser can be implemented fast without loosing accuracy. This is very important for the applications which use a parser. Parsing is in most cases only one component of a Natural Language Processing application such as summarization, dialog system or machine translation. Many applications have a tight time schedule and a parser can not take more than a second or even only some milliseconds to parse a sentence. The same holds true for the memory footprint since the parser has to share the memory with other components of a system or it has to run on a small device which might provide only a very limited amount of memory. During the development of a parser, it is very important as well that it does not take too long to train the parser since otherwise experiments take too long and it becomes impossible to improve the parser in a give time.

3.1 Online Learning

As learning technique, we use Margin Infused Relaxed Algorithm (MIRA) as developed by Crammer et al. (2003) and applied to dependency parsing by McDonald et al. (2005). The Algorithm in Figure 2 processes one training instance on each iteration, and updates the parameters due to the currently processed instance.

The inner loop iterates over all sentences x of the training set while the outer loop repeats the train i times. The algorithm returns an averaged weight vector and uses an auxiliary weight vector v that accumulates the values of w after each iteration. At the end, the algorithm computes the average of all weight vectors by dividing it by the number of training iterations and sentences. This helps to avoid overfitting, cf. (Collins, 2002).

Algorithm 2: MIRA

```
$\tau = \{S_x, T_x\}_{x=1}^{X}$ // The set of training data consists of sentences and the corresponding dependency trees
$\overrightarrow{w}^{(0)} = 0, \overrightarrow{v} = 0$
**for** n = 1 **to** $N$
    **for** x = 1 **to** X
        $w^{i+1}$ = update $w^i$ according to instance $(S_x, T_x)$
        $v = v + w^{i+1}$
        $i = i + 1$
    **end for**
**end for**
$w = v/(N * X)$
```

The update function computes the update to the weight vector w^i during the training so that wrong classified edges of the training instances are possibly classified correct. This is computed by increasing the weight for the correct features and decreasing the weight for wrong features of the vectors for the tree of the training set $\overrightarrow{f_{T_x}} * w^i$ and the vector for the predicted dependency tree $\overrightarrow{f_{T'_x}} * w^i$.

The update function tries to keep the change to the parameter vector w^i as small as possible for correctly classifying the current instance with a difference at least as large as the loss of the incorrect classifications. The update of the algorithm in Figure 2 can be formalized by following update function.

$$\min ||w(i+1) - w(i)||$$
$$s.t.\ score(S_x, T_x) - score(S_x, T'_y) \geq L(T_x, T'x))$$

3.2 Selected Parsing Features

Table 2 and 3 give an overview of the selected features for the reimplementation of McDonald and Pereira (2006) (System A) and the extended version with the integrated edge labels (System B), cf. Johansson and Nugues (2008). Both parser versions shared the same code and training algorithm except from two parts: the parsing algorithm itself which are only about 100 lines of code and the code which extract the features. The reason for this is that the parsing accuracy of both algorithms are sensitive to the selected features. For the parsing and training speed, most important is a fast feature extraction beside of a fast parsing algorithm.

3.3 Implementation Aspects

In this subsection, we provide some implementation details that concern all the speed of the parser and distinguish this implementation from others. The learning architecture determines the architecture of the parser.

The training has three passes. The goal of the first two passes is to collect the set of possible features for all elements of the training set. In the first pass, the feature extractor collects all attributes that the features can contain since our goal is to determine the minimal description length for each attribute. The reason for this is to save memory and computational time during the feature creation. For instance, when the feature extractor builds features due to the pattern `label, h-pos, d-form`[2] then in the first pass the attributes edge labels, part-of-speech tags (pos) and word forms are included into collection procedure. For each category (labels, pos, etc.), the extractor builds a mapping to a number which is continuos from 1 to the count of elements without duplicates. The following equation shows this formally.

We enumerate in the same way the feature patterns and obtain the function $f_{feature-type}(value)$, e.g. $f_{feature-type}$(label,h-pos,d-form)=7. Now, we can calculate the minimal description length in bits for each

[2]We combine all the elements (label, pos, form) of this feature pattern to a single feature (label+pos+form).

Standard Features				**Linear Features**	
Feature	System	Feature	System	Feature	System
h-form	A,B	h-form, d-pos	A,B	h-pos, d-pos, h-pos + 1	A,B
h-pos	A,B	h-pos, d-form	A,B	h-pos, d-pos, h-pos - 1	A,B
d-form	A,B	h-form, d-form	A,B	h-pos, d-pos, d-pos + 1	A,B
d-pos	A,B	h-pos, d-pos	A,B	h-pos, d-pos, d-pos - 1	A,B
h-form,h-pos	A,B	h-form, d-form, h-pos	A,B	h-pos, d-pos, h-pos - 1, d-pos - 1	A,B
d-form,d-pos	A,B	h-form, d-form, d-pos	A,B	h-pos, d-pos, h-pos - 1, d-pos + 1	A,B
		h-pos, d-pos, h-form	A,B	h-pos, d-pos, h-pos + 1, d-pos - 1	A,B
		h-pos, d-pos, d-form	A,B	h-pos, d-pos, h-pos + 1, d-pos + 1	A,B
		h-pos, d-pos, h-form, d-form	A,B	h-pos - 1, d-pos, d-pos + 1	A
				h-pos + 1, d-pos, d-pos - 1	A
				h-pos - 1, h-pos, d-pos + 1	A
				h-pos + 1, h-pos, d-pos - 1	A

Grandchild Features		**Sibling Features**	
Feature	System	Feature	System
h-form, d-pos, g-pos	A	d-form, s-form $\oplus$ dir(d,s) $\oplus$dist(d,s)	A
h-form, d-pos, g-pos, dir(h,d)	A	d-pos, s-form $\oplus$ dir(d,s) $\oplus$dist(d,s)	A
		d-pos, s-form $\oplus$ dir(d,s)+ $\oplus$ dist(d,s)	A
		d-pos, s-pos $\oplus$dir(d,s)$\oplus$dist(d,s)	A
h-pos, d-pos, g-pos, dir(h,d), dir(d,g)	B	h-pos, d-pos, s-pos, dir(h,d), dir(d,s) $\oplus$dist(h,s)	B
h-form, g-form, dir(h,d), dir(d,g)	B	h-form, s-form, dir(h,d), dir(d,s)$\oplus$dist(h,s)	B
d-form, g-form, dir(h,d), dir(d,g)	B	d-form, s-form, dir(h,d), dir(d,s) $\oplus$dist(h,s)	B
h-pos, g-form, dir(h,d), dir(d,g)	B	h-pos, s-form, dir(h,d), dir(d,s)$\oplus$dist(h,s)	B
d-pos, g-form, dir(h,d), dir(d,g)	B	d-pos, s-form, dir(h,d), dir(d,s)$\oplus$dist(h,s)	B
h-form, g-pos, dir(h,d), dir(d,g)	B	h-form, s-pos, dir(h,d), dir(d,s)$\oplus$dist(h,s)	B
d-form, g-pos, dir(h,d), dir(d,g)	B	d-form, s-pos, dir(h,d), dir(d,s)$\oplus$dist(h,s)	B

Table 2: Selected Features. h stands for head, d for dependent, g for grandchild, and s for sibling. System A builds additional features by adding the **dir**ection and a feature that has additional the **dist**ance plus the direction. The direction is left if the dependent is left of the head otherwise right. The distance is the number of words between the head and the dependent, if $\leq$5, 6 if $>$5 and 11 if $>$10. In some cases, we could also improve system B by adding this features as well. In this cases, we list this explicit. System B has always the edge label included in the features which is not indicated in order to make to compare easier. $\oplus$ means that an additional feature is build with the previous part plus the next part.

$f_{attribute}(value) \rightarrow N$, e.g. let be $f_{labels}(value) \rightarrow \{$(punc,0),(sbj,1),(obj,2),(mod,3) ..$\}$ then $f_{label}(sbj) = 1$ [3]

of the attributes with the following equation:

$bits(attribute) = ceil(log_2(max(N_{attribute})))$

In the second pass, the extractor builds the features for all training examples which occur in the train set but not for all combination, i.e., the extractor builds feature for all in the training set contained edges. In other words, only for the positive examples and not for the negative cases that do not occur. However, these features of the *wrong edges* could improve the parser accuracy since the parser considers also this edges during the creation of the parse tree. This would lead to a much larger number of features. Therefore, most of the implementation do not consider these features.

We create the features with a function that maps iteratively the attributes of a feature to a number represented with 64 bit and then enumerates and maps these numbers to 32 bit numbers to save even more memory. This is computed by the equation $l(value, start, value_{previous}) = last_{previous} + (value << start)$. The expression `number<<n` means shift the binary representation of the number by n-bits to the left. For instance, let *mod,N* be the set of attribute of the feature pattern `label, h-pos`, $bits$(labels)=7, $bits$(pos)=6, $bits$(feature-type)=6, f_{labels}(mod)= 3 (11b), f_{pos}(N)=6(110b), and $f_{feature-type}(label + h - pos)$=8

Label Features			
Feature	System	Feature	System
label, pos $\oplus$ child $\oplus$ dir(h,d)	A	label, pos, pos + 1 $\oplus$ child $\oplus$ dir(h,d)	A
label, pos, pos-1, pos-2 $\oplus$ child $\oplus$ dir(h,d)	A	label, pos, pos -1, pos-2, pos+1 $\oplus$ child $\oplus$ dir(h,d)	A
label, pos, pos-2, $\oplus$ child $\oplus$ dir(h,d)	A	label, pos, pos +2, $\oplus$ child $\oplus$ dir(h,d)	A
label, pos, pos-1 $\oplus$ child $\oplus$ dir(h,d)	A	label, pos, pos+1 $\oplus$ child $\oplus$ dir(h,d)	A
label, pos, pos+1, pos-1 $\oplus$ child $\oplus$ dir(h,d)	A	label, form $\oplus$ child $\oplus$ dir(h,d)	A

Table 3: System A uses a boolean flag **child** to indicate that it is the head or dependent and adds these feature once for the head and once for the dependent. $\oplus$ means that an additional feature is build with the previous part plus the next part.

(1000b). Then the value for the feature type is computed by $l(8,0,0) = 8$ (1000b) and start=$bits$(feature-type)=6; l(3,6,8)= 1000b + (11b<<6) = 1000b + 11000000b = 11001000b =200 and start=6+$bits$(label)=12; $l(12,6,200)$= 11001000b + (11b<<12)= 11001000b + 11000000000000 = 11000011001000b.

The following list shows an overview of the most important implementation details that improve the speed:

1. We use as feature vector within the support vector machine only a list of the features without any additional (double floating point) value.

2. We store the feature vectors for $f(label, w_i, w_j)$, $f(label, w_i, w_j, w_g)$, $f(label, w_i, w_j, w_s)$ etc. in a **compressed** file. The reason for storing vectors in a file is that it is faster to compute the values once and then to load them in each of the training iterations (6-10 times).

3. We zip the file with the option for fast compression and decompression. The reason for this is that otherwise the IO to the hard disk drive becomes the bottleneck.

4. After the training, we store only the parameters of the support vector machine which are not zero.

	McDonald and Pereira (2006)	System A	Johansson and Nugues (2008)	System B
Type	2nd order	2nd order	2nd order integrated labels	2nd order integrated labels
training time memory usage	70 hours 7 GB	4 hours 1.5 GB	60 hours not reported	14 hours 3 GB
parsing time memory usage	2 seconds 1.5 GB	0.05 seconds 700 MB	1.49 seconds not reported	0.6 seconds 1 GB
LAS	0.86	0.87	0.88	0.88

Table 4: Performance Comparison

Table 4 gives an overview of different parsing systems and their performance and memory usage. We use the training and development set of the 2008 shared task and for the speed comparison a 2.8 Ghz Mac Pro and the values reported in Johansson and Nugues (2008) based on 3.2 Ghz Mac Pro.

4 Semantic Role Labeling

Semantic Role Labeling (SRL) as well as Dependency Parsing has been topics of CoNLL shared tasks, cf. (Carreras and Màrques, 2004; Carreras and Màrquez, 2005). The first two shared task 2004 and 2005 used phrase structures trees as input to the semantic role labeler while the last shard task (2008) and the upcoming CoNNL shared task (2009) uses dependency trees. We use a pipeline architecture for semantic role labeling. The components of the pipeline are predicate identification (PI), argument identification (AI), argument classification (AC), and word sense disambiguation (WSD). For training and testing, we use the English corpus of the shared task 2008. The corpus is in addition to dependency trees annotated with predicates and semantic roles of the NomBank and PropBank, cf. (Meyers et al., 2004; **?**).

Algorithm 3:Attribute Identification

```
// $S_x \leftarrow w_0...w_n$ is the sentence x with the words $w_i$
// $P_x \leftarrow p_0...p_m$ is the set of predicates $p_j$ of sentence x
$A_j^x$ is the set of arguments of predicate j in sentence x.
for all $p_j \in P_x$
    for all $w_i \in S_x$
        if score($p_j, w_i$)$\geq 0$ then $A_j^x \leftarrow A_j^x \cup \{i\}$
```

In order to identify the predicates, we look up the lemmas in the PropBank and NomBank. For all other components, we use the same learning technique and architecture as for the dependency parser. We use the same technique because we want to be able to use the scores of the components to rerank other results and the used support vector machine allows a very large number of features that standard decision trees, neural nets, etc. can not handel.

Algorithm 3 identifies the arguments of each predicate. Its two loops iterate over the predicates and over the words of a sentence in the case that the score function is large or equal to zero the argument is added to the set of arguments of the predicate in question.

The argument classification algorithm labels each argument identified in the previous step with a semantic role label. The argument classification algorithm selects with a beam search algorithm the combination of arguments with the highest score. The algorithm allows only one core argument of the same type such (A0 to A5).

The last component of our pipeline is the word sense disambiguation. We put this against the intuition at the end of our pipeline since experiments showed that other components could not profit from disambiguated word senses but on the other hand the word sense disambiguation could profit from the argument identification and argument classification. In order to disambiguate, we iterate over the words in the corpus that have more than one sense and take the sense with the highest score. Due to space restrictions, we can not list all the features that we used in our systems. A lot of good combinations can be found in Che et al. (2008).

The accuracy and scores of our system are only a bit lower than the best reported results (80.4) on the development data of the 2008 shared task, cf. Johansson and Nugues (2008). The Attribute Identification component has a accuracy of 91.6 and the attribute classification applied on the output of the AI a F1 score of 77.5 Since we consider at this stage that all word sense have the first sense 01, the Word Sense Disambiguation can improve the results to 79.2. The average time to execute the SRL pipeline on a sentence is less than 0.15 seconds.

5 Application to the Meaning-Text Theory

The above technique could be directly applied to a MTT corpus. The dependency trees converted from the phrase structure annotation of the Penn Treebank have become much more similar to the surface syntactic trees for instance the coordinations are no longer flat and attached to the conjunction. In a lot of cases, only the edge labels are different. Therefore, the described techniques could be applied to a corpus annotated with MTT surface syntactic dependency trees.

The mapping of surface syntactic dependency trees to deep syntactic dependency trees can be addressed with similar techniques. In this step the main task is to leave out the function words, to introduce lexical functions and to label the edges with deep syntactic dependency labels.

The semantic graphs of the MTT are mostly comparable to the PropBank annotation. The exceptions are the communicative structures which is missing and predicates which form lexical functions are represent different on the semantic stratum.

We hope that MTT Corpora annotated with dependency representation of all levels become available: surface syntactic structures, deep syntactic structures and semantic representations including the communicative structure (Mel'čuk, 2001). We are sure that this would be one of the most valuable linguistic resource. One of the most recent initiative towards this direction is a corpus for Spanish, cf. (Mille et al., 2009).

6 Conclusion

In this paper, we have described an algorithm for synchronous parsing of syntactic and semantic structures. Our implementation has scores that are comparable good as the best so far reported results. Moreover, the implementations and techniques introduced in this paper provide a much better parsing and training times. Also the memory footprint are lower so that the parsers can be trained on standard computer and used on devices which have less memory.

We integrated the synchronous parser into the Meaning-Text Development Environment (Mate) (Bohnet et al., 2000) which can be trained now on MTT corpora so that it is possible to obtain surface syntactic dependency trees and the semantic actants with the above technique when trained on a corpus annotated with MTT structures. This can help also to set up corpora in a boots trap approach.

References

Bohnet, B., A. Langjahr, and L. Wanner. 2000. A Development Environment for an MTT-Based Sentence Generator. In *Proceedings of the First International Natural Language Generation Conference*.

Buchholz, S. and E. Marsi. 2006. Conll-x shared task on multilingual dependency parsing. In *In Proc. of CoNLL*, pages 149–164.

Carreras, X. and L. Màrques. 2004. Introduction to the conll-2004 shared task: Semantic role labeling. In *Proceedings of CoNLL-2004*, pages 89–97. Boston, MA, USA.

Carreras, X. and L. Màrquez. 2005. Introduction to the CoNLL-2005 shared task: Semantic role labeling. In *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL-2005)*, pages 152–164, Ann Arbor, Michigan, June. Association for Computational Linguistics.

Carreras, Xavier. 2007. Experiments with a Higher-Order Projective Dependency Parser. In *Proceedings of the EMNLP-CoNLL 2007 Shared Task*.

Che, Wanxiang, Zhenghua Li, Yuxuan Hu, Yongqiang Li, Bing Qin, Ting Liu, and Sheng Li. 2008. A Cascaded Syntactic and Semantic Dependency Parsing System. In *CoNLL 2008: Twelfth Conference on Computational Natural Language Learning*, pages 238–242, Manchester, England. Coling.

Collins, M. 2002. Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *EMNLP*.

Crammer, K., O. Dekel, S. Shalev-Shwartz, and Y. Singer. 2003. Online Passive-Aggressive Algorithms. In *Sixteenth Annual Conference on Neural Information Processing Systems (NIPS)*.

Eisner, J. 1996. Three New Probabilistic Models for Dependency Parsing: An Exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*, pages 340–345, Copenhaen.

Johansson, R. and P. Nugues. 2008. Dependency-based syntactic–semantic analysis with PropBank and NomBank. In *Proceedings of the Shared Task Session of CoNLL-2008*, Manchester, UK.

Kahane, S., A. Nasr, and O. Rambow. 1998. Pseudo-projectivity: A polynomially parsable non-projective dependency grammar. In *COLING-ACL*, pages 646–652.

McDonald, R. and F. Pereira. 2006. Online Learning of Approximate Dependency Parsing Algorithms. In *In Proc. of EACL*, pages 81–88.

McDonald, R., K. Crammer, and F. Pereira. 2005. Online Large-margin Training of Dependency Parsers. In *Proc. ACL*, pages 91–98.

McDonald, R., K. Lerman, F. Pereiraand, K. Crammer, and F. Pereira. 2006. Multilingual Dependency Parsing with a Two-Stage Discriminative Parser. In *Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 91–98.

Mel'čuk, I.A. 2001. *Communicative Organization in Natural Language : The Semantic-Communicative Structure of Sentences*. John Benjamins Publishing, Philadelphia.

Meyers, A., R. Reeves, C. Macleod, R. Szekely, V. Zielinska, B. Young, and R. Grishman. 2004. The nombank project: An interim report. In Meyers, A., editor, *HLT-NAACL 2004 Workshop: Frontiers in Corpus Annotation*, pages 24–31, Boston, Massachusetts, USA, May 2 - May 7. Association for Computational Linguistics.

Mille, S., V. Vidal, A. Burga, and L. Wanner. 2009. Creating an MTT Tree Bank of Spanish. In *Proceedings ot the Fourth International Conference on Meaning-Text Theory*, Montréal.

Nivre, J., Hall J. and J. Nilsson. 2004. Memory-Based Dependency Parsing. pages 49–56, Boston, Massachusetts.

Nivre, J. and J. Nilsson. 2005. Pseudo-Projective Dependency Parsing. In *In Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 99–106.

Nivre, J., J. Hall, S. Kübler, R. McDonald, J. Nilsson, S. Riedel, and D. Yuret. 2007. The conll 2007 shared task on dependency parsing. In *Proc. of the CoNLL 2007 Shared Task. Joint Conf. on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, June.

Nivre, Joakim, Igor M. Boguslavsky, and Leonid L. Iomdin. 2008. Parsing the SYNTAGRUS Treebank of Russian. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*, pages 641–648, Manchester.

Surdeanu, M., R. Johansson, A. Meyers, L. Màrquez, and J. Nivre. 2008. The CoNLL-2008 shared task on joint parsing of syntactic and semantic dependencies. In *Proceedings of the 12th Conference on Computational Natural Language Learning (CoNLL-2008)*.