



Converting Dependency Treebanks to MALT-XML

Johan Hall
Jens Nilsson

Johan Hall Jens Nilsson

Converting Dependency Treebanks to MALT-XML

Report

Computer Science

2005

Contents

1	Introduction	1
2	Background	1
2.1	The Treebanks	2
2.2	Projectivity	2
2.3	Treebank formats	3
3	A framework for mapping treebanks	4
3.1	Implementation	5
4	From non-projectivity to projectivity	6
5	Results	8
5.1	Converting DDT in TIGER-XML to MALT-XML	8
5.2	Projectivity conversion	8
6	Conclusion	9
	References	11
A	The non-projective to projective algorithm	12
B	A proposal for encoding Dependency Treebanks in TIGER-XML	13
B.1	Corpus header	13
B.2	Corpus body	14

1 Introduction

In data-driven approaches to natural language processing, a common problem is the lack of data for many languages. Within the project *Stochastic Dependency Grammars for Natural Language Parsing* at Växjö University, we (Joakim Nivre, Johan Hall and Jens Nilsson) are developing a deterministic data-driven dependency parser [9], which is language independent. In this project we intend to enlarge the data resources for our parser. For the moment, we have only tested our parser on small Swedish treebank converted to dependency structure, and on English using Penn treebank [5] converted to dependency trees. Since we do not have more Swedish dependency treebanks at hand, we want to broaden our view towards treebanks for other languages, especially the bigger ones, to investigate the influence of data size. Primarily, we are focusing on the Danish Dependency Treebank (DDT) [3] and the Prague Dependency Treebank (PDT) [2]. These treebanks are not in a format that we can use for our parser and therefore we have to convert them to MALT-XML, a format which our parser can handle.

A problem that has to be addressed is the differences in dependency theory. One such theory issue is projectivity. Intuitively, if the words in a sentence are linearly ordered and all arcs from the head to its dependents are drawn above the word, then the sentence is projective if no arcs cross each other and root of the sentence is not covered by any arcs. The parser we are using assumes that the trees in the training data are projective, and it can only produce projective trees as output. This will be more closely described in section 2.2.

Generally, the conversion problem has the following two main goals:

- To develop a generic method for converting dependency treebanks between different formats (section 3).
- To develop a generic method for converting non-projective trees to projective trees (section 4).

The next section will present the background for this project, such as the difference between projectivity and non-projectivity, and will give an overview of DDT and PDT. The section ends with an introduction to the treebank formats we use and motivates the use of a generic framework. Section 3 will discuss the implemented framework, and section 4 will investigate the issue of projectivity. In section 5, the result of our tests will be analyzed. We will end the report with some conclusions.

2 Background

It is hard to say anything in general about treebanks with dependency structure, since the term dependency structure is rather vague. Dependency trees do not have non-terminal nodes, as do phrase structure trees. The hierarchy is instead realized by relations directly between words. Usually, the relations are binary, with one word acting as head or governor and the other as child or dependent. This is what most theories have in common, but there are disagreements about many other things. For example, some dependency theories permit multiple heads, while others do not, and some permit non-projective trees, others do not. All this and other theory dependent decisions are reflected in the few existing dependency treebanks. This section will give the appropriate background for the practical work described in later sections.

2.1 The Treebanks

This section will briefly discuss the Danish Dependency Treebank (DDT) and the Prague Dependency Treebank (PDT). DDT consists of approximately 5050 sentences having 100000 words. We will not go into the details of linguistic theory here (see Kromann [3]), since our data-driven parser is not dependent on the use of a pre-defined theory. As long as the theory does not have non-projective structures, this also means that no conversion between different linguistic frameworks is necessary for our parser.

The original format that DDT was created in has some characteristics which makes it inappropriate for our parser to use. For example, DDT permits non-projective trees. Although the learner would not fail completely when encountering non-projective input during parser training, the information it actually "learns" is not what the non-projective training data says. To achieve higher parsing accuracy, the best approach for DDT, and probably for all non-projective dependency treebanks, is to convert the non-projective trees to projective ones. The number of non-projective sentences in DDT are 853 (15.48%), but the number of arcs causing a sentence to be non-projective are not that frequently occurring. Only 941 arcs are non-projective, which is less than 1% of the total number of arcs. Figure 2.1 shows a non-projective sample sentence from DDT.

Another characteristic of DDT is that the linguistic theory permits a word to have more than governor, one primary governor and zero or more secondary governors. This input is strictly forbidden since our parser is not able to handle more than one governor per head. But since we have access to DDT in TIGER-XML, which clearly distinguishes primary and secondary governors, we do not have to worry about this.

The syntactic layer of the Prague Dependency Treebank is in one way more attractive to our data-driven parser, since it contains roughly 20 times more data than DDT, i.e. just over 100000 sentences. Unlike DDT, PDT does not allow multiple heads in the form of secondary edges. So in this perspective it is more similar to the dependency format our parser expects. However, in line with DDT, it allows non-projective trees.

2.2 Projectivity

As mentioned in the introduction, a sentence is not projective if at least two of the arcs between the words cross or if the root is covered by an arc, provided that all arcs are drawn above the sentence and the words are linearly ordered. A requirement saying that the sentence must be fully connected is often added to the definition of projectivity. In other words, exactly all but one word must have a governor.

There are several good linguistic reasons why a dependency theory should allow non-projectivity for several different languages (Mel'cuk [6]). By examining DDT, we could conclude that most non-projective constructions are implied by the theory, and are therefore annotated quite consistently. But some of the others are easily identified as errors caused by humans, while others are at least linguistically debatable.

However, our parser is constructed in such a way that it can not produce non-projective sentences let aside the fact that sentences may be unconnected. Arcs causing a sentence to be non-projective are in general relatively rare and their frequency depend heavily on the underlying dependency theory. In many cases, reconstructing one or more arcs which violate the projective property does not result

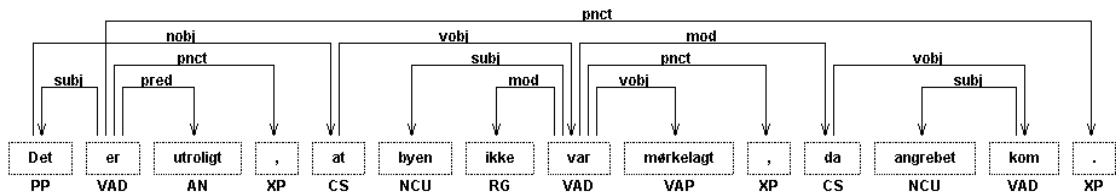


Figure 2.1: Reconstructible non-projective example

in loss or change of information, only the theory has to be somewhat altered. In the example in figure 2.1, the arcs $Det \rightarrow at$ and $er \rightarrow .$ cross each other. No information is lost or changed if the arc $Det \rightarrow at$ is substituted by $er \rightarrow at$, as long as the theory is changed accordingly. These kinds of non-projective structures can automatically be reconverted to the original format. This is a quite commonly occurring phenomenon, so this kind of replacement should in general be valid.

Moreover, even if the arc $er \rightarrow .$ did not exist, the sentence would still be non-projective. The word *er* is the root of the sentence and is covered by the arc $Det \rightarrow at$. By performing the substitution as above, the sentence becomes projective. In other cases, forcing a sentence to be projective will make it impossible to represent it without (semantically) changing the interpretation, see Kromann [4] (e.g. *a hard nut to crack*). Moreover, creating a parser framework that allows crossing arcs and arcs covering the root word will of course be more general than our parser that can not handle non-projectivity. An interesting question is, whether a non-projective dependency parser can correctly recognize non-projective constructions and insert the appropriate arcs accordingly and avoid adding non-projective arcs in all other cases? This is still an open question, but it is partially answered by Nivre and Nilsson [10]. In their study, the non-projective parser produced too many non-projective graphs and had a lower precision than the projective one (although vice versa for recall).

We believe that non-projective arcs are quite rare and come in many different flavors within a linguistic theory. Hence, most parsers that rely on data-driven approaches will have problems restricting themselves to the correct cases only, if they find the correct cases at all. On the other hand, for the non-projective constructions, a projective parser will never analyze the constructions correctly, provided that they cannot be reconstructed to projective ones without information loss.

The choice stands between using a more general non-projective parser that might insert more non-projective arcs in many situations when it should not (thus decreasing accuracy), or using a more restrictive projective parser that in some cases will fail, since it cannot construct the necessary arcs. We have chosen the latter, and since DDT and PDT contain non-projective constructions we have to do something about them. The best but most costly way to do this is to let a linguistically skilled person manually convert the non-projective sentences. This is not an option for us, and we have therefore tried to do this conversion automatically.

2.3 Treebank formats

There are many formats for encoding syntactically annotated corpora or treebanks. In most cases when a treebank is created, a new annotation scheme and encoding format are designed. When we began to develop our parser and supporting tools we also designed our own encoding format, MALT-XML. The TIGER-XML is

not an encoding format for a specific treebank, but rather an interface format for representing various treebanks and corpora, see next section.

TIGER-XML format

The TIGER-XML [7] treebank encoding format is one attempt to create a more generic format for representing various corpora and treebanks. To cope with the problem of various kinds of formats that must be supported by the treebank search tool TIGERSearch, the research group designed an interface format that can act as import and export format for the software package. A user of TIGERSearch first has to transform the encoding of a treebank to TIGER-XML using for example a conversion program or an XSLT stylesheet.

As indicated by the name TIGER-XML, the format uses the eXtended Markup Language (XML) to encode corpora and treebanks. A TIGER-XML document is divided into two parts: header and body. The header contains general information about the corpus or treebank. The header should also declare all features used, e.g. word, part-of-speech, and features of nonterminal nodes. The body part supports the data model based on syntax graphs, which are directed acyclic graphs with a single root node. To encode this in XML, all terminal and nonterminal nodes are listed and edges are explicitly encoded. There is also support for including subcorpora and defining queries for TIGERSearch.

The format is designed to be theory-independent, but it is especially suited for treebanks using phrase structure annotation scheme, where the encoding uses terminals and nonterminals for creating the syntactic structure. Using TIGER-XML as an exchange format with an annotation schema for a dependency treebank, the encoding has to include nonterminal nodes which is not the most suitable way to encode a dependency treebank. The Nordic Treebank Network has decided to use TIGER-XML as an exchange format. To address the problem of represent treebanks using dependency structure, a working group in the network has designed a proposal to encode dependency treebanks in TIGER-XML. The proposal is documented in appendix B.

MALT-XML format

The MALT-XML is a format for encoding dependency treebanks and is designed at Växjö University to be used as an intermediate format for different treebank resources. The format's original purpose was to convert old Swedish treebanks, e.g. Talbanken [1], into dependency structures represented in a more modern encoding format like XML.

The format is inspired by the TIGER-XML format in the way that it divides the document into two parts: header and body. In the header, features are declared as in TIGER-XML. The body part segments the treebank into sentences. Each sentence is divided into terminal nodes or lexical nodes. A lexical node is described with word form, part-of-speech and other syntactic information.

3 A framework for mapping treebanks

In this section, a framework for mapping treebanks from one encoding formats to another is presented. It should be possible to include support for converting new treebanks with different annotation schemes, languages and format with minimum effort. The general architecture of the framework is presented in figure 3.1.

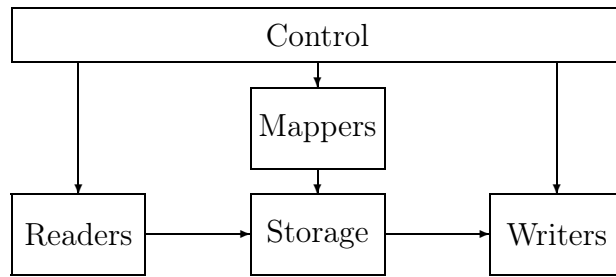


Figure 3.1: Architecture

A treebank is usually stored in one or several files and in a specific format, for example XML, SGML or tab-separated or sometimes in a database. One of the tasks for a program which implements the framework must be to read from a file or a database into some kind of storage with data structures, supporting the annotation scheme. The storage can be implemented in different ways, for example different data structures for dependency structure and phrase structure. In some cases, the framework has to cope with the mapping from phrase structure to dependency structure and therefore store both.

To handle different features, e.g. word forms and parts-of-speech with different tagsets, the framework supports what we call a mapper. This reads a mapping file, containing the feature names in the input-format, and the corresponding names in the output-format. For example, in one format the part-of-speech is called *msd* and in the other *postag*. In the mapping file it should also be possible to define mappings between different tagsets, if this is required. The features and tagsets are stored in suitable data structures in the storage unit and are integrated with the other data structures.

To be able to write to the output-format the framework needs different writers. A writer defines how the particular format should be created in a file or a database. The writer interacts with the storage unit. Finally, the framework needs a control unit which manages the whole process and interacts with the user. It can be a simple terminal interface or a more advanced graphical user interface.

3.1 Implementation

As a test case we implemented a program that converts the Danish Dependency Treebank in TIGER-XML format to MALT-XML and the other way around.

The treebank mapping framework is partly implemented in the Java programming language. To maintain generality of the framework the program uses different packages for the different parts of the framework. Furthermore, each package has an interface to interact with other parts of the program. In other words, if the program should support a specific format it has to be extended with a new Reader class which implements the Reader interface. The specific Reader class reads from the input file and must be able to distinguish the parts in the particular format and then saves the interesting parts in the storage. If the program should be able to store a tree or a graph in a different way, the program has to implement the storage interface. The same strategy goes for the writer and mapper interfaces.

Every part of the framework is merged into a toolkit, which defines different configurations of the system. An instance of a toolkit, for example converting from TIGER-XML to MALT-XML, extends an abstract toolkit which must have at least

three methods: *setInput()*, *setMap()* and *setOutput()*. Every method takes a file as an argument. Each toolkit configures its own set-up of the framework which is appropriate for a specific configuration. For example, if the program is to handle the conversion from TIGER-XML to MALT-XML, the toolkit creates an instance of the TIGER-XML reader and requires dependency structure storage (if we use the TIGER-XML to store dependency structure). Furthermore, the toolkit needs a MALT-XML writer and finally it needs an instance of a mapping class, which reads a mapping file. When we use the TIGER-XML to MALT-XML toolkit we only have to supply the program with an input file in TIGER-XML (dependency structure), a mapping file in a specific format (how features and tagsets should be mapped) and an output file.

The current version of the Java implementation supports conversion from dependency structures encoded in TIGER-XML to MALT-XML and vice versa. It is also possible to take either of these formats as input and generate a tab-separated file. There are mapping files for the Danish Dependency Treebank and Talbanken. In the near future we will also have support for the Prague Dependency Treebank. For the moment we have done a quick conversion from the original PDT format in SGML to MALT-XML. The information in their dependency representation enables us to more or less just make a mapping between the formats. We will soon be able to parse Czech data, but this requires some modifications in the parser.

4 From non-projectivity to projectivity

Non-projective constructions can be divided into two categories, although they formally can be captured in the same definition. The first one is crossing arcs and the other is arcs covering the root word of a sentence. See Nivre [8] for a formal definition of projectivity and well-formed trees.

Crossing arcs

For the first category, the main idea behind the automatic conversion is simply to identify crossing arcs and move one of them so that they do not cross each other anymore. The algorithm is constructed in a bottom-up fashion by considering the arcs with the smallest spans first, and then examining the arcs with larger spans gradually. If an arc between word w_i and word w_{i+k} , and another arc between $w_a, i < a < i + k$ and $w_b, \neg(i \leq b \leq i + k)$ is found, then they are identified as two crossing arcs. To look at some cases of crossing arcs, see figure 4.1. Depending on the properties of the arcs (explained below), this local projectivity is removed by moving one the arcs. The idea is that each changed arc will locally always produce non-projective constructions. Our hypothesis is that this will lead to projectivity globally.

The crossing arcs have been classified by a number of questions enumerated below. The first two results in two cases each, while the indented questions result in three cases. This gives in total $2 \cdot 2 \cdot 3 = 12$ cases for the algorithm to take care of. The three features are:

- If the smaller arc headed to the left or to the right?
- If the head node of the other arc inside or outside the smaller arc? (IN or -IN)
 - If the smaller arc ancestor to the other arc? (SANC)

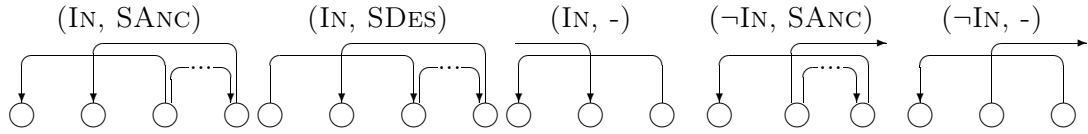


Figure 4.1: before pictures of all right-headed cases

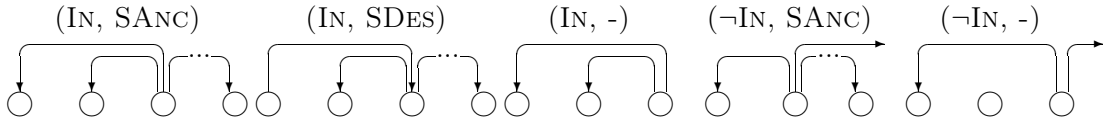


Figure 4.2: after pictures of all right-headed cases

- Else, if the smaller arc a descendent of the other arc? (SDES)
- Else, they only have a common ancestor (-)

The underlying principle applied, if one of the arcs has been identified as an ancestor of the other, is to "move up" the descendant arc so that the arcs no longer cross each other using as few steps as possible, i.e. (IN, SANC), (IN, SDES) and (¬IN, SANC) in figures 4.1 and 4.2. In the cases when neither arc is an ancestor of the other, (¬IN, -) and (IN, -)), the decision is a little bit more tricky, since they are more irregular and probably contain more annotation errors than the previous cases. The idea here is to let the dependent of the longer arc become dependent of the governor of the smaller arc. This is based on the rather weak assumption that an arc with smaller span is probably more reliable than an arc with longer span.

The questions above result in twelve cases, but can be reduced to ten, since (¬In, SDes) and (¬In, -) entail in the same change, both from now on denoted (¬In, -). The five cases where the smaller arc of the two crossing arcs is headed to the left are depicted in figure 4.1. The other five cases, where the smaller arc is right-headed, are not shown but can be illustrated as mirror cases. Figure 4.2 shows the trees after the change according to each case. Moreover, the direction of the arc crossing the smaller arc does not matter. The incoming arc for example in (IN, SANC) is drawn as a left-headed arc, but could just as well be right-headed. The same holds for all other cases. Another thing worth mentioning is that whenever there is an arc with ellipsis in the figures, this means that the connected nodes do not have to be in a direct relation. They could instead be a chain consisting of two or more nodes (thus not denoted father but ancestor).

The algorithm for resolving crossing arcs is shown in figure A.1, Appendix A. It starts by considering the smaller arcs first, going from the left to the right in the sentence, looking for any arcs crossing it. After one pass the span is increased by one and a new pass from left to right is performed. If two crossing arcs are found, the case is determined, and the proper arc is moved. In the algorithm, as opposed to figure 4.1 and 4.2, all ten cases are shown.

The algorithm is generic enough to completely convert any non-projective dependency treebank in MALT-XML to a projective one. So far, only DDT has been converted in this way, and a brief evaluation of the conversion is found in section 5.2.

Arcs covering the root

The other non-projective case not handled so far is arcs covering the root node of the sentence. The left part of figure 4.3 gives an intuitive understanding of the concept.

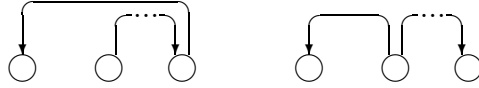


Figure 4.3: Before (left) and after (right) pictures of an arc covering the root

The middle node is the root node of the sentence. The root w_i is ancestor in one or more steps to a word $w_j, i < j$, which in turn is ancestor to a word $w_k, k < i$, according to the picture, or vice versa, i.e. $j < i$ and $i < k$.

As mentioned in the beginning of this section, some of the arcs are non-projective due to errors in the treebank. By looking at several sentences, having arcs covering the root node, we can with confidence state that most of them are not theory related.

Therefore, the sentences are a more heterogeneous collection than the sentences having crossing arcs. Nevertheless, something has to be done about them in an automatic way. The idea here is first of all to identify the arcs covering its head (i.e. usually the only root node of the sentence). The assumption made in this situation is that the root node shall be kept as root node, and that the dependent node of the covering arc will have the root node as its new head node. This is depicted to the right in figure 4.3.

5 Results

5.1 Converting DDT in TIGER-XML to MALT-XML

TIGER-XML have more expressive power than MALT-XML. In general, when converting from TIGER-XML to MALT-XML, there can be loss of information like meta information about the corpora, included subcorpora, encoded queries for TIGERSearch and secondary edges. There is no loss of information when converting from MALT-XML to TIGER-XML.

We used a TIGER-XML version of the DDT to test the program. The DDT uses an annotation schema based on the computational dependency formalism Discontinuous Grammar [4], which uses secondary edges to annotate reference, filler and replacement edges. This information will be lost in the conversion to MALT-XML. The non-projective structures in DDT can be encoded in MALT-XML without any problem. The conversion resulted in 26 warnings. All of them were caused by the fact that those sentences had more than root. Since MALT-XML allows more than one root in a sentence, it is not a fatal error in the conversion.

5.2 Projectivity conversion

An exhaustive evaluation of the conversion from a linguistic point of view will not be done in this report. Instead, some more objective measurements will be presented. As mentioned in section 2, approximately 860 of 5050 sentences in DDT were non-projective before the conversion. After the conversion, no sentences contained any non-projective structures. In total 972 arcs were changed. Ten of these were arcs covering the root, and the others were distributed over the ten cases described previously, according to table 5.1. The number of right-headed cases exceeds the number of left-headed cases, which is probably caused by the fact that there are usually a greater number of right-headed arcs compared to left-headed arcs. The only disturbing thing to note here is that the right headed (\neg IN, -) are so numerous.

	(IN, SANC)	(IN, SDES)	(IN, -)	(-IN, SANC)	(-IN, -)
Left-headed	24	81	0	55	40
Right-headed	140	47	40	114	421

Table 5.1: Number of non-projective cases in DDT

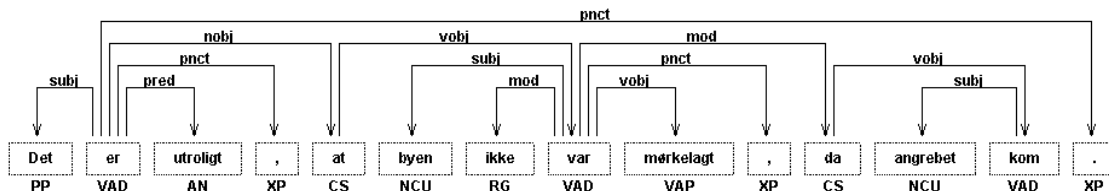


Figure 5.1: after picture of subordinate clause arc

The cases $(-IN, -)$ and $(IN, -)$ contain most of the linguistically questionable arcs present in DDT, but an extensive evaluation of the reason for this high number has not been done.

Some examples of converted sentences

In this section, we will look at some sentences from DDT in more detail and analyze what actually has happened with the non-projective arcs. A few sampled sentences will be used to illustrate this. Let us start by looking at the sentence from figure 2.1. As already mentioned, we want the arc $Det \rightarrow at$ to be replaced by $er \rightarrow at$. Of the two crossing arcs, $Det \rightarrow at$ has the smallest span and will be categorized as right-headed with $(-IN, SANC)$. According to the algorithm, this substitution is actually conducted, resulting in the well-formed tree in figure 5.1. Subjects having dependents to the right of the finite verb, which is the case in figure 2.1, does generally not pose a problem for the algorithm. In fact, among the non-projective constructions in DDT, this type occurs quite frequently. As mentioned in section 4, the idea of the algorithm is that the crossing arcs shall be “moved up” to the head of the arcs it crosses, which figure 5.1 nicely illustrates.

The sentence in figure 5.2 is more complex than the previous one because the number of crossing arcs is more than one (three to be precise), but it still falls under the same principle. Firstly, the left-headed arc $er \rightarrow \text{“is”}$ is found, having $Det \rightarrow har$ as its crossing arc. Although, it is a different case according to the algorithm, the correct arc is moved to the right place. Also, the two other cases of crossing arcs are resolved at the same time.

Another non-projective discontinuous case occurring quite frequently is shown in figure 5.3. The sentence has a discontinuous coordination *Både ... og*. The direct connection between *Både* and *og* is lost and is substituted by letting the intermediate word act as “hub”. This might be a case where information is inevitably lost when the arc is moved.

6 Conclusion

Our initial goal was to convert DDT from the TIGER-XML format to MALT-XML, and PDT, in their format, to MALT-XML. Furthermore, since DDT and PDT allow non-projective trees, we implemented an automatic converter from non-projectivity

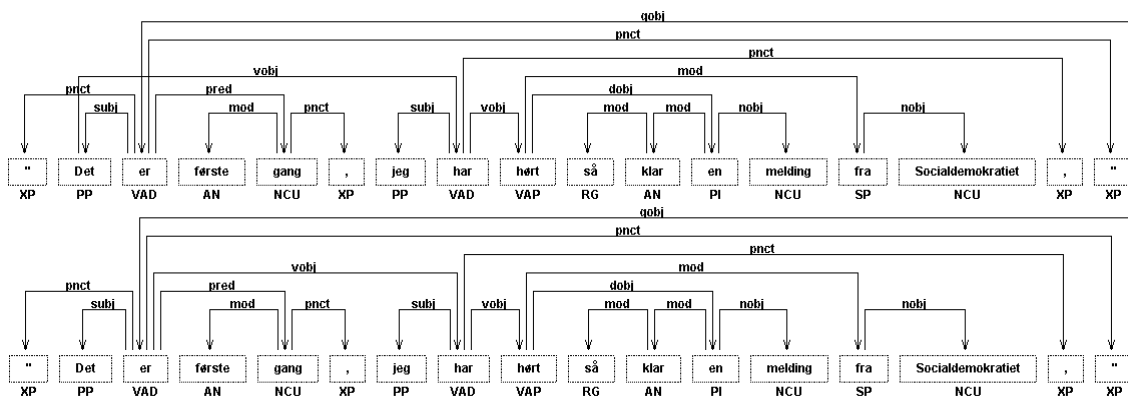


Figure 5.2: before and after picture of multiple crossing arcs

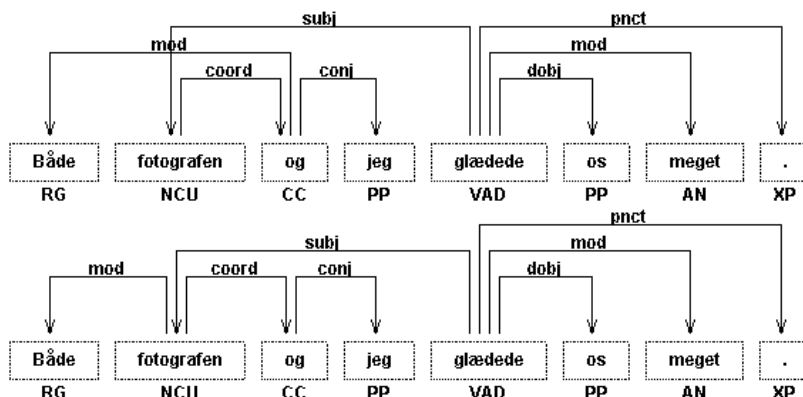


Figure 5.3: before and after picture of a discontinuous coordination arc

to projectivity. We also implemented a generic framework for converting from any dependency treebank in TIGER-XML to MALT-XML and a corresponding converter in the reverse direction.

Since the TIGER-XML format contains more information than the MALT-XML format, one will inevitably lose information when going from TIGER-XML to MALT-XML. Although there is a disagreement concerning the underlying theory of dependency grammar, a big advantage using a common encoding format is that it is easier to exchange data between research groups.

The implementation of the converter from non-projective trees to projective ones seems, according to section 5.2, to work in an acceptable way. The converter does of course make mistakes, and a more detailed evaluation could be needed.

Finally, we can mention some preliminary results regarding parsing using DDT. We used 80% of the data for training and 10% for validation (we never used the other 10%). The Danish tagset was reduced to 33 tags and the dependency relations were kept at the original 55. Running our best parser on this configuration, we achieved an accuracy of 87.1% disregarding the dependency relation, whereas 81.1% was the result taking into account both the head and the relation. Even though these figures are based on the correct tags and the Swedish data we have worked with so far has been tagged with an HMM-tagger, the result is more or less in line with the results we got on the Swedish data or even slightly better. Some fine tuning will probably increase accuracy a little bit more.

References

- [1] Jan Einarsson. Talbankens skriftspråkskonkordans. Lund University, 1976.
- [2] Jan Hajic, Eva Hajicova, Petr Pajas, Jarmila Panevova, Petr Sgall, and Barbora Vidova Hladka. The Prague Dependency Treebank, 2001.
- [3] M. Kromann, L. Mikkelsen, and S. Lynge. Danish Dependency Treebank. Annotation manual. Technical report, Dept. of Comp. Ling., Copenhagen Business School, 2003.
- [4] Matthias T. Kromann. Danish dependency treebank annotation guide, 2004. <http://www.id.cbs.dk/~mtk/treebank/>.
- [5] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. In *Computational Linguistics*, page 19:313 330. 1993.
- [6] Igor Mel’cuk. *Dependency Syntax: Theory and Practice*. State University of New York Press, 1988.
- [7] Andreas Mengel and Wolfgang Lezius. An XML-based representation format for syntactically annotated corpora. In *Proceedings of the 2nd International Conference on Language Resources and Evaluation (LREC2000)*, volume 1, pages 121–126, Athens, Greece, May 2000. ELRA.
- [8] Joakim Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies, Nancy, France*, volume IWPT 03, 2003.
- [9] Joakim Nivre, Johan Hall, and Jens Nilsson. Memory-based dependency parsing. In *Proceedings of the Conference on Computational Natural Language Learning (CoNLL), Boston, MA*, 2004.
- [10] Joakim Nivre and Jens Nilsson. Three algorithms for deterministic dependency parsing. In *Proceedings of NoDaLiDa*, 2003.

A The non-projective to projective algorithm

This is the algorithm converting a non-projective sentence in MALT-XML into a projective one.

```
NONPROJ2PROJ(string of words  $[w_1, \dots, w_n]$ )
1  for span  $k = 2$  to  $n - 1$  loop
2    for position  $i = 1$  to  $n - k$  loop
3      if  $w_{i+k} \rightarrow w_i$  exists then // left-headed
2        for all arcs  $w_a \rightarrow w_b$  and  $i < b < i + k$  and  $\neg(i \leq a \leq i + k)$  loop
3          if  $w_b$  is an ancestor of  $w_{i+k}$  then
4            make  $w_i$  a dependent to  $w_b$  //(IN, SANC)
3          else if  $w_{i+k}$  is an ancestor of  $w_b$  then
4            make  $w_b$  a dependent to  $w_{i+k}$  //(IN, SDES)
5          else
6            make  $w_b$  a dependent to  $w_i$  //(IN, -)
7        for all arcs  $w_a \rightarrow w_b$  and  $i < a < i + k$  and  $\neg(i \leq b \leq i + k)$  loop
8          if  $w_a$  is an ancestor of  $w_{i+k}$  then
9            make  $w_i$  a dependent to  $w_a$  //(-IN, SANC)
10         else
11           make  $w_b$  a dependent to  $w_{i+k}$  //(-IN, -)
12      if  $w_i \rightarrow w_{i+k}$  exists then // right-headed
13        for all arcs  $w_a \rightarrow w_b$  and  $i < b < i + k$  and  $\neg(i \leq a \leq i + k)$  loop
14          if  $w_b$  is an ancestor of  $w_i$  then
15            make  $w_{i+k}$  a dependent to  $w_b$  //(IN, SANC)
14          else if  $w_{i+k}$  is an ancestor of  $w_b$  then
15            make  $w_{i+k}$  a dependent to  $w_b$  //(IN, SDEC)
16          else
17            make  $w_b$  a dependent to  $w_i$  //(IN, -)
18        for all arcs  $w_a \rightarrow w_b$  and  $i < a < i + k$  and  $\neg(i \leq b \leq i + k)$  loop
19          if  $w_a$  is an ancestor of  $w_i$  then
20            make  $w_a$  a dependent to  $w_{i+k}$  //(-IN, SANC)
21          else
22            make  $w_b$  a dependent to  $w_i$  //(-IN, -)
```

Figure A.1: The algorithm

B A proposal for encoding Dependency Treebanks in TIGER-XML

On the Nordic Treebank Network meeting we discussed a format for encoding dependency treebanks in the TIGER-XML encoding format and this appendix tries to document the outcome of this discussion. The most of the work is done by Matthias Trautner Kromann and the examples are taken from the Danish Dependency Treebank [3] (DDT). The documentation also includes some details about the TIGER-XML format in general.

The TIGER-XML encoding format starts with a `corpus` tag and consists of an `id` attribute, which identifies the treebank (used in the TIGERSearch). The format consists of two parts: header and body.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<corpus id="DDT-1.0">
  corpus/treebank header-part
  corpus/treebank body-part
</corpus>
```

B.1 Corpus header

The corpus header starts with a `<head>` tag and contains meta and annotation information. The meta information, not mandatory, that can be specified is name of the corpus or the treebank, date, author, short description, format and history of the corpus.

```
<meta>
  <name>Danish Dependency Treebank v. 1.0</name>
  <date>Fri Feb 20 15:30:49 MET 2004</date>
  <author>Matthias Trautner Kromann (mtk@id.cbs.dk)</author>
  <description>The Danish Dependency Treebank v. 1.0 ...
</description>
  <format>The annotations ...
</format>
  <history></history>
</meta>
```

The second part of the corpus header contains information about the annotation. First the feature is enumerated, e.g. word form, part-of-speech and lemma. If the feature has a tagset, the tags are enumerated. The edge labels and secondary edge labels should also be enumerated.

```
<annotation>
  <feature name="word" domain="FREC" />
  <feature name="cat" domain="FREC">
    <value name="--">undefined value</value>
    <value name="AC">cardinal adjective</value>
    <value name="AN">normal adjective</value>
    ...
    <value name="XX">other</value>
  </feature>
```

```

<feature name="msd" domain="FREC">
  <value name="AC---U---"></value>
  <value name="ANA---=-R"></value>
  ...
  <value name="XX"></value>
</feature>
<feature name="lemma" domain="FREC"></feature>
<edgelabel>
  <value name="--"></value>
  <value name="&lt;avobj&gt;">adverbial object of
gapped conjunct</value>
  ...
  <value name="xtop">external topic</value>
</edgelabel>
<secedgelabel>
  <value name="[aobj]">adjectival object filler</value>
  <value name="[avobj]">adverbial object filler</value>
  ...
  <value name="ref">coreference</value>
</secedgelabel>
</annotation>

```

The use of the value -- is for the dependency structures only, and indicates that there are no dependency relation between the terminal and the nonterminal (they are the same node in the dependency structure) and the label will not be displayed in TIGERSearch

B.2 Corpus body

The corpus or the treebank is segmented into sentences (denoted <s> in the format) and each sentence is identified by an attribute id. The tag <graph> with the attribute root identifies the root in the graph. Each graph is divided into terminals and nonterminals. The terminal specifies the lexical node in the structure with the features enumerated in the corpus header, like word form, lemma and part-of-speech.

```

<s id="s10">
  <graph root="p10_866">
    <terminals>
      <t id="w10_865" word="Illusioner" msd="NCCPU==I"
        cat="NC" lemma="illusion"/>
      <t id="w10_866" word="er" msd="VADR=----A-" cat="VA"
        lemma="være"/>
      <t id="w10_867" word="farlige" msd="ANP[CN]PU=[DI]U"
        cat="AN" lemma="farlig"/>
      <t id="w10_868" word="." msd="XP" cat="XP"
        lemma="."/>
    </terminals>
    <nonterminals>
      <nt id="p10_865" word="Illusioner" msd="NCCPU==I"
        cat="NC" lemma="illusion">
        <edge idref="w10_865" label="--" />
    </nonterminals>
  </graph>
</s>

```

```

</nt>
<nt id="p10_866" word="er" msd="VADR=----A-" cat="VA"
  lemma="være">
  <edge idref="w10_866" label="--" />
  <edge idref="p10_865" label="subj" />
  <edge idref="p10_867" label="pred" />
  <edge idref="p10_868" label="pnct" />
</nt>
<nt id="p10_867" word="farlige" msd="ANP[CN]PU=[DI]U"
  cat="AN" lemma="farlig">
  <edge idref="w10_867" label="--" />
</nt>
<nt id="p10_868" word="." msd="XP" cat="XP"
  lemma=".">
  <edge idref="w10_868" label="--" />
</nt>
</nonterminals>
</graph>
</s>

```

A dependency treebank does not have nonterminals, but with TIGER-XML we are forced to build up some structure with the nonterminals. Each nonterminal node corresponds to a terminal with the same information specified in the terminal node. The information is specified twice, once in the terminal and once in the corresponding nonterminal.

Each nonterminal node consists of at least one edge to the terminal and with the label --, which points out there is no label between the terminal and nonterminal. In the example, the verb *være* governs all the other nodes in the graph. The noun *Illusioner* is a dependent and a subject to the verb and the adjective *farlige* is the predicate.

In the DDT there are also filler relations and references and these relations are annotated with secondary edges together with the terminals, not shown in the example.



Växjö
universitet

Matematiska och systemtekniska institutionen
SE-351 95 Växjö

tel 0470-70 80 00, fax 0470-840 04
www.msi.vxu.se