Non-Projectivity in Transition-Based Dependency Parsers

Lena Reisinger

MInf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

2016

Abstract

Many of the most widely used transition-based dependency parsers, such as the arc-eager parser, can only parse projective dependency trees. While most English sentences fall into this category, it is a much too strict constraint for many other languages such as Czech, Dutch, German, Hungarian and Portuguese. That is why in the last years there has been much research into ways to account for non-projectivity in sentences. In the course of my project I implemented two transition systems that can only build projective dependency trees and two that can parse trees with an arbitrary number of crossing dependency arcs, in order to investigate how allowing for non-projectivity influences the attachment accuracy of transition-based dependency parsers. One of the two projective systems is the arc-eager parser, while the other one is a novel transition system that unlike the arc-eager parser does not exhibit spurious ambiguity and is based on the fact that dependency structure can be described by a context-free grammar. The first non-projective parser I implemented is based on a fundamental dependency algorithm by Covington where all possible dependency relations are allowed. I also introduce a new transition system that combines the transitions from Covington's parser with those transitions from the arc-eager parser that remove words from being considered for more arcs. This significantly improves training time compared to Covington's parser and leads to a higher attachment accuracy on all languages I tested my systems on. I also show that it has the highest average performance of all four parsers, leading to the conclusion that allowing for all kinds of dependency arcs gives better results than restricting the output to only projective ones, for languages with a large number of non-projective sentences.

Acknowledgements

I would like to thank my project supervisor, Adam Lopez, for his help, his suggestions and his encouragement.

Special thanks also to my family and friends for their support.

Table of Contents

Chapter 1

Introduction

For the first part of my MInf Project I am looking at how allowing for crossing (non-projective) dependency arcs influences the attachment accuracy of transition-based dependency parsers. To do this I have implemented two parsers that build only projective dependency trees and two that place no restrictions on the dependency arcs allowed and compared their performance on different languages. All parsers use the same machine learning framework to ensure that a fair comparison is possible. Two of the parsers I implemented are based on existing transition systems, whereas the other two were developed in the course of this project after exploring problems with the existing systems. Section 2 of this report will define the concepts of dependency structure, dependency parsing, transition systems and non-projectivity. It will also mention some applications of dependency parsing, give an overview of relevant work in the field and summarise my contributions. In the following section, Section 3, I will introduce the transition systems that I have implemented, followed by a section giving a detailed description of the implementation. After that, in Section 5, I have presented the results of my experiments which consisted of evaluating all parsers on six different languages with a significant number of non-projective sentences. This is followed by Section 6, describing the conclusions that can be drawn from my results. The last section talks about work that still has to be done and proposes goals for Part 2 of my MInf project.

Chapter 2

Background

This section describes what dependency parsing is, what different kinds of dependency parsers there are, why it is an interesting topic which has attracted a lot of research in the last years, some relevant examples of this research and what I have contributed in the course of my project. Part of the material in this chapter has been drawn from (Kübler et al., 2009).

2.1 Dependency Structure

There are two main ways the syntactic structure of language can be viewed. The first one is constituency structure, where sentences are built up from nested constituents. To do this we need rules defining how to make up constituents from other constituents and terminals, which are words. The constituent *S* (which stands for sentence) can for example be created by combining a *VP* (verb phrase) with a *NP* (noun phrase) using the rule $S \rightarrow VP\ NP$.

The other way to view sentence structure is by specifying the words that depend on each word. Dependency structure refers to syntactic relations between words in a sentence, where one is the head and the other the dependent. The dependent can be referred to as the syntactic subordinate of its head. Every word has exactly one head, which means that dependencies can be represented as a rooted tree. Each node in a dependency tree corresponds to a terminal (a word), whereas nodes in a constituency tree can correspond to either constituents or terminals. Advantages of dependency structure are that dependency trees are often simpler than corresponding constituency trees and

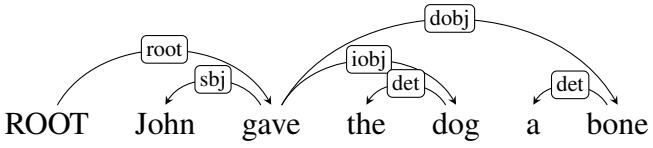


Figure 2.1: Dependency relations of an example sentence.

they are usually better suited to represent sentences of languages with flexible word order.

Dependents are either:

- Arguments: In the example sentence in Figure 2.1, *John*, *dog* and *bone* are arguments of *gave*.
- Modifiers (sometimes called adjuncts): In Figure 2.1, *the* is a modifier of *dog* and *a* is a modifier of *bone*.

As Figure 2.1 shows, a dependency relation is represented as an arrow pointing from the head to the dependent. An artificial word *ROOT* is inserted at the beginning of the sentence, which has an arc to the syntactic head word of the sentence, in this case *gave*. Each dependency arc has a label associated with it, describing the kind of relation between the two words. In the example in Figure 2.1, we have the relation that *John* is the subject of *gave*, therefore the label of the arc connecting the two is *sbj*. *Dog* is the indirect object of *gave*, indicated by label *iobj*, *bone* is its direct object with label *dobj* and *the* and *a* are determiners, which means their dependency label is *det*. Formally we define an arc as (i, l, j), which means that there is a dependency relation from the ith to the jth word in the sentence which is of the kind l.

2.2 Dependency Parsing

The problem of dependency parsing consists of automatically finding the dependency tree of a given input sentence. Dependency parsers can either be data-driven, meaning that machine learning approaches are used to learn a model from annotated data, or grammar-based, where formal grammars describe the kind of dependency trees that are possible.

Data-driven dependency parsing can be split into two sub-problems:

- Learning problem: This entails learning a model from a text corpus where each word is annotated with its head and the type of dependency to its head, that can then be used to find the dependency tree of an input sentence.
- Parsing problem: Given an input sentence, use the model to output its dependency tree.

Data-driven models can be categorised into graph-based parsers and transition-based parsers. For graph-based parsers the learning problem consists of learning a model that can assign likelihood scores to all possible dependency trees for a sentence. Parsing then consists of finding the dependency tree with the highest score, which can be achieved with dynamic programming.

Transition-based parsers build a dependency tree for a sentence by applying a sequence of transitions to it. The learning problem entails computing a model that predicts the

next most likely transition for a given partially parsed sentence and the parsing problem means using this model to predict a sequence of transitions that builds a dependency tree for a given input sentence.

A main difference between the two approaches is that transition-based dependency parsers are greedy and only choose locally optimal next transitions, whereas graph-based ones build the globally most likely parse tree. On the other hand, transition-based parsers can use much richer features and take much more information about the parsing history into account when making parsing decisions than the graph-based parsers . Both kinds of parsers achieve similar parsing accuracies (Kübler et al., 2009). For my project I will be looking at different transition-based parsers, which are described in Section 3.

2.3 Non-Projectivity

Many dependency parsers allow only for projective dependency trees, meaning none of the arcs are crossing each other. An example for a non-projective sentence can be seen in Figure 2.2 where the arcs *saw* $\rightarrow$ *yesterday* and *dog* $\rightarrow$ *was* are crossing each other. Formally, non-projectivity can be defined in the following way:

Definition 2.3.1. Non-projectivity: Let $w = w_1, w_2, .., w_n$ be a sentence and w_0 the artificial root node. It is non-projective if its dependency tree contains arcs (i, l, j) and (i', l', j') (we are assuming w.l.o.g. that $i < j$ and $i' < j'$) where the integers $i, j, i', j' \in [0, n]$ and $i < i' < j < j'$.

Restricting the parser to non-projective structures is a reasonable assumption for languages such as English where the number of crossing arcs is very small, but too constrained for several other languages, especially those permitting free word order to some degree.

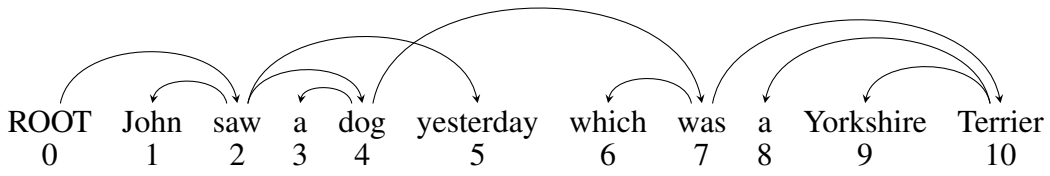


Figure 2.2: Dependency relations of an example non-projective sentence.

In the course of this paper I will look at several dependency parsers, some of which can create non-projective dependency trees and some that are restricted to only projective ones.

My goal is to examine how the projectivity constraint influences the attachment accuracy of the parsers and to observe how this differs across languages with a varying percentage of non-projective sentences.

2.4 Transition-Based Dependency Parsing

Transition-based dependency parsing describes the process of transforming a sentence into its corresponding dependency tree through a sequence of transitions. Such a transition system is defined by an initial configuration, a set of transitions that transform one configuration into the next one, and a set of final configurations which indicate the end of a parse (Nivre, 2008). A configuration typically includes a buffer β with tokens that were not processed yet, a stack σ containing tokens that have been processed already and can still be involved in future arcs, and a set of arcs A containing all dependencies (i, l, j) between tokens i and j with dependency label l that have been added so far. Typical transitions to go from one configuration to the next are *shift*, which moves the item in the beginning of the buffer onto the stack, *reduce*, which removes the item on top of the stack, *left-arc*, which adds an arc from the word in the beginning of the buffer to the word on top of the stack, and *right-arc*, which adds an arc from the word on top of the stack to the word in the beginning of the buffer. There are two stages to transition-based dependency parsing:

- Learning: You need training data in the form of sentences annotated with extra information such as part-of-speech tags, with their corresponding dependency tree. The goal of the learning problem is to train a model that predicts the optimal transition for each configuration, where a configuration is a partial analysis of the sentence. The input to the classifier has to be a sequence of configurations with the optimal transition to apply at each of them. Since there is an infinite number of possible configurations, we define a function f that transforms a configuration into a feature vector representing certain properties of the configuration. Let C be the set of all possible configurations, T the set of all possible transitions and Y the set of all feature vectors. Then we define f as:

  $$f(c) : C \rightarrow Y$$

  Our classifier g will be of the form:

  $$g(y) : Y \rightarrow T$$

  Now the question is how to turn a training example into a sequence of transitions, needed as input to the classifier. For a given configuration there is often more than one legal transition. Therefore we need a mechanism, called oracle, to decide what transition to apply next. During training this is an algorithm that looks at the items on the stack and in the buffer and the dependency tree of the sentence and determines which transition is optimal.

- Parsing: This problem consists of finding the dependency tree for a given input sentence. Our oracle for predicting the next transition during parsing is the model g that we trained during the learning stage. A sentence's parse is finished when a final configuration is reached, which is often defined as an empty buffer. Note that for many transition systems it is possible that the dependency graph found during parsing is not connected and that not all words in a sentence are

assigned a head. One extreme such case is when the only transition performed is the *shift* transition.

In section 3, four transition systems are described in more detail.

2.5 Applications of Dependency Parsing

There are applications for dependency parsing in many different fields. It can be used for **information extraction**, as shown for example by Culotta and Sorensen (2004). A sub-problem of information extraction is finding the relations between entities in a sentence. This can be done using machine learning methods by measuring similarity between labelled training instances and novel testing instances. Culotta et al. showed that if the structures containing the entities are represented as dependency trees, the accuracy is much higher than when representing them as a ”bag-of-words” with no structural information.

Another field where dependency parsing is used is **machine translation**. An example of this is provided by Ding and Palmer (2005). They describe a method to learn a synchronous grammar, which is a grammar that applies to two languages at the same time and whose production rules have two right hand sides (one for each of the languages), from dependency trees. To build this grammar a corpus of parallel dependency trees which represent the same sentence in each of the languages is needed. It is then used to transform a dependency tree representing a sentence in the source language into one in the target language. This is called a tree-to-tree machine translation system. There has also been work for tree-to-string systems, where the source sentence is represented as a tree, and string-to-tree systems, where a dependency tree is produced for the target sentence.

Applications for dependency parsing are also found in the field of **question answering**, for example in (Wang, 2007). When dealing with the problem of selecting an answer to a question, both the question and the possible answers are transformed into dependency trees and used to compute the likelihood of each of the answers. They show that this outperforms other state-of-the-art systems.

2.6 Relevant Work

The first data-driven dependency parsing algorithm was introduced by Eisner (1996). It is a dynamic programming algorithm that builds on the CYK algorithm that is used for parsing context-free grammars by combining the analysis of shorter sub-strings into that of longer sub-strings.

One of the first transition-based dependency parsers was introduced by Yamada and Matsumoto (2003) and is similar to the shift-reduce parser. The shift-reduce parser works in the following way: if the right hand side of a production rule of the form $A \rightarrow BC$ is on top of the stack, replace it with the left hand side (*reduce*), otherwise move the next item in the buffer onto the stack (*shift*). Yamada's parser uses *left* and *right* transitions to add arcs (i, l, j) between words, essentially reducing ij (or ji) and replacing it with the head i. This transition system has the property that right arcs can only be added once the dependent in that relation has all of its dependents attached, since it is removed afterwards.

The arc-eager parser introduced by Nivre (2003) on the other hand attaches right-arcs as soon as possible, without removing the dependent afterwards. This works by adding an additional *reduce* transition that removes the item on top of the stack. This in one of the transition systems I have implemented and a detailed description can be found in Section 3.1.

The first dependency parsing algorithm which was also able to produce non-projective dependency trees was introduced by Covington (2001). It essentially works by looking at each pair of words in a sentence and checking if there is a dependency relation between them. A more detailed description can be found in Section 3.3.

Another way to deal with non-projectivity is introduced by Nivre and Nilsson (2005), which is pseudo-projective parsing. This method makes it possible to produce non-projective dependency trees while using a transition system that is restricted to projective structures. It works by performing operations to turn non-projective dependency trees in the training set into projective ones, while keeping track of what operations were performed in the dependency arc labels. When a word is assigned such an annotated label during parsing, the dependency label can be used as "instructions" on how to turn the dependency tree into a non-projective one.

In recent years there has also been interest in exploring mildly non-projective dependency structures, which allow certain kinds of crossing dependency arcs. An example for this can be found in (Pitler, 2013), which introduces a dependency parser for the class of 1-Endpoint-Crossing trees. This is defined as the class of trees where whenever an arc is crossed, all arcs that cross it share an endpoint.

2.7 Contributions

In this section I will summarise my contributions to this project and explain what implementing the four transition parsers entailed:

- After investigating how the arc-eager parser exhibits spurious ambiguity, meaning that more than one transition sequence leads to the same dependency tree,

and how this could be a problem, I derived a novel projective transition system, which I call attach-complete transition system, that does not have this form of ambiguity. It is based on the fact that dependency structure can be described by a context-free grammar and works by associating a transition with each of the production rules of this grammar. Experiments showed that when looking at the fraction of correctly assigned heads during parsing (unlabelled attachment score), it outperforms the arc-eager system on two of six languages. When evaluating the fraction of complete dependency trees that were parsed correctly (exact match) the attach-complete parser leads to better results than the arc-eager one on five of six languages.

- A critical evaluation of Covington's parser led me to believe that its performance could be improved by adding a mechanism to restrict the arcs that could be added in further parsing steps, while still making sure that all possible non-projective trees could be parsed. That is why I invented a transition system which could be called a hybrid of the transition system based on Covington's non-projective algorithm for dependency parsing and the arc-eager transition system. It behaves equivalently to the arc-eager system on projective sentences but uses Covington's setup to be able to parse all possible non-projective sentences. Since the difference between this new transition system and Covington's transition system is that *reduce* transitions are added, I will refer to it as Covington's parser with reduce transitions in the remainder of this report. My experiments showed that it got the highest number of dependency relations correct on four of the six languages I used for evaluation, with the highest number of correctly assigned non-projective links on all six languages.

- I implemented four transition-based parsers: the arc-eager parser, Covington's parser, the attach-complete parser and Covington's parser with reduce transitions:

  - I wrote code that reads data from the Universal Dependencies treebank files into `Python` data structures, while recording properties of the data such as the number of sentences, the number of non-projective sentences, the number of dependency arcs (equivalent to the number of words), and the number of crossing dependency arcs. This included interpreting an algorithm that turns a non-projective dependency tree into a projective one, since two of the parsers I implemented can only handle projective structures. To do this I also wrote functions that check whether a dependency arc or sentence is non-projective.

  - I created a `Configuration` class for each parser with functions for each of their transitions, which update the configuration to the configuration resulting from applying the transition. These functions check if the preconditions for applying the transitions are fulfilled and return `False` if they are not.

  - I adapted the function for extracting features from the Natural Language Toolkit's `nltk.parse.transitionparser` module so that it can be used for all four parsers and extended it to the set of features described in (Nivre, 2008) with the addition of a distance feature indicating the number of words

between the token on top of the stack and the token in the beginning of the buffer.

  - For each parser I implemented the oracle algorithm used for choosing which transition to apply next during training. This transition is then applied to the current configuration and the type of transition and features of the current configuration are recorded as training data for a classifier.
  - In my implementation, each parser has multiple classifiers associated with it to improve learning time by training each of them in parallel on a subset of the training data. To do this I have coded a `Classifier` class which keeps track of the training examples associated with it and later the trained model. The machine learning setup for training a classifier and using it for parsing is based on the one used in the `nltk.parse.transitionparser` module.
  - I coded an evaluator that takes the results of a parser and the data of the gold standard file as input and computes attachment scores and label accuracy (UAS, LAS, LA) both for all arcs and for non-projective and projective arcs separately, and exact match scores (UEM, LEM) both for all sentences and non-projective and projective ones separately.

- I evaluated each parser's performance on the complete training and test sets from the Universal Dependencies treebank on German, Dutch, Hungarian, Portuguese and Danish and on part of the training set (40,000 of 68,525 sentences) and the complete test set for Czech. Then I used these results to argue about the effect of allowing non-projective dependency arcs on the attachment accuracy of transition-based dependency parsers.
- After theorising about possible reasons for the differences in performance between the parsers, I wrote code analysing several properties of the dependency graphs that were output by the parsers, such as whether they are well-formed (every word has a head) and what the average arc length is.

Chapter 3

Transition Systems

In the following subsections I will introduce the four transition systems I have implemented. First I will look at following two projective transition systems:

- Arc-eager transition system (Nivre, 2003): This is one of the most widely used transition systems for dependency parsing and many transition-based dependency parsers build on it. It is called arc-eager, because arcs are added as soon as possible, unlike in earlier transition systems where right arcs from a head to a dependent were only added after the dependent had all its dependents added.

- Attach-Complete transition system: After showing how the arc-eager parser exhibits spurious ambiguity, meaning that several transition sequences lead to the same dependency tree, I will introduce a transition system that is based on the fact that dependency grammar can be transformed into a context-free grammar where each parse tree corresponds to a unique transition sequence. It works by having two copies of each word and removing one of them when a word is assigned a head and the other one as soon as it has no more dependents. The *attach* transitions are used to add arcs, and *complete* transitions are used to remove the last copy of a word.

Then I will consider two non-projective transition systems that allow for all possible arcs between tokens and place no restraints on the crossing of arcs:

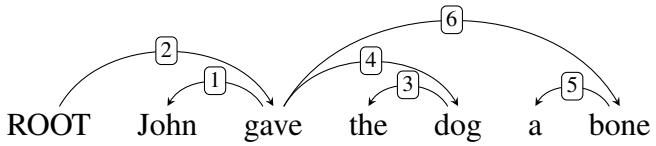
- Covington's non-projective fundamental algorithm for dependency parsing (Covington, 2001): This transition system can build any dependency tree imaginable, under the constraint that each word can only have one head and that it is in fact a tree and therefore does not contain any cycles. For each word in the buffer it checks if any of the words on the stack are its head and also if it is the head of any of the words on the stack that do not have a head yet. No words are ever removed from consideration and therefore arcs between any two words are possible.

- Covington's parser with reduce transitions: After evaluating the performance of Covington's parser compared to the arc-eager one and noticing significantly lower attachment accuracies coupled with a greatly prolonged learning time, I have thought about possible improvements to the parser without loosing its ad-

vantages. Therefore I developed a variant of Covington's transition system that can still parse all possible dependency trees but is more efficient and, as experiments showed, more accurate than the original one. It basically adds transitions from the arc-eager transition system that remove words from the configuration, meaning they cannot be involved in further arcs (e.g. the *reduce* transition), and can therefore be considered a hybrid of the arc-eager and Covington's parser.

3.1 Arc-Eager Parser

The arc-eager transition system only builds projective dependency trees. An arc between the word with index i and the word with index j can only be added after all shorter arcs with both endpoints in the interval $[i, j)$ or $(i, j]$ have been added. There cannot be an arc with only one endpoint between i and j, since it would be crossing the arc between i and j. Parsing the sentence is done in a left-to-right manner, meaning if we have arcs (i,l,j) and (i',l',j') where $max(i,j) < max(i',j')$, then the arc from i to j will be added first. The following dependency graph illustrates the order in which arcs are added for an example sentence:



The arc-eager parser works by keeping track of a buffer, which initially contains all input tokens, and a stack. There are transitions to push tokens from the buffer onto the stack, to add arcs between the token on top of the stack and in the beginning of the buffer, and to pop items off the stack.

Formally, a configuration for the arc-eager parser for sentence $w = w_0, w_1, ..., w_n$ consists of:

- A stack σ containing indices of partially processed words, i.e. tokens $i \leq k$ for some $k \leq n$
- A buffer β containing indices of the words that have not been processed yet, i.e. tokens $j > k$
- A set of arcs A which contains a partial dependency tree of the sentence

Following transitions are allowed for the configuration $(\sigma|s, b|\beta, A)$ (Nivre, 2012):

- LEFT-ARC: Adds arc (b,l,s) from the token in the beginning of the buffer to the token on top of the stack and pops s off the stack since all arcs to left dependents will have already been added and a dependency to a token in the buffer would violate the projectivity constraint. This operation can only be executed if s does not already have a head.

Transition	$(\sigma,$	$\beta,$	$A)$
	$[0],$	$[1,...,10],$	$A = \emptyset$
SHIFT $\Longrightarrow$	$[0,1],$	$[2,...,10],$	A
LEFT-ARC $\Longrightarrow$	$[0],$	$[2,...,10],$	$A \cup 1 \leftarrow 2$
RIGHT-ARC $\Longrightarrow$	$[0,2],$	$[3,...,10],$	$A \cup 0 \rightarrow 2$
SHIFT $\Longrightarrow$	$[0,2,3],$	$[4,...,10],$	A
LEFT-ARC $\Longrightarrow$	$[0,2],$	$[4,...,10],$	$A \cup 3 \leftarrow 4$
RIGHT-ARC $\Longrightarrow$	$[0,2,4],$	$[5,...,10],$	$A \cup 2 \rightarrow 4$
SHIFT $\Longrightarrow$	$[0,2,4,5],$	$[6,...,10],$	A
LEFT-ARC $\Longrightarrow$	$[0,2,4],$	$[6,...,10],$	$A \cup 5 \leftarrow 6$
RIGHT-ARC $\Longrightarrow$	$[0,2,4,6],$	$[7,...,10],$	$A \cup 4 \rightarrow 6$
SHIFT $\Longrightarrow$	$[0,2,4,6,7],$	$[8,9,10],$	A
SHIFT $\Longrightarrow$	$[0,2,4,6,7,8],$	$[9,10],$	A
LEFT-ARC $\Longrightarrow$	$[0,2,4,6,7],$	$[9,10],$	$A \cup 8 \leftarrow 9$
LEFT-ARC $\Longrightarrow$	$[0,2,4,6],$	$[9,10],$	$A \cup 7 \leftarrow 9$
RIGHT-ARC $\Longrightarrow$	$[0,2,4,6,9],$	$[10],$	$A \cup 6 \rightarrow 9$
REDUCE $\Longrightarrow$	$[0,2,4,6],$	$[10]$	A
REDUCE $\Longrightarrow$	$[0,2,4],$	$[10]$	A
REDUCE $\Longrightarrow$	$[0,2],$	$[10]$	A
RIGHT-ARC $\Longrightarrow$	$[0,2,10],$	$[],$	$A \cup 2 \rightarrow 10$

Figure 3.1: Trace of the arc-eager parser for the sentence "John saw a dog which was a Yorkshire terrier yesterday". (1 2 3 4 5 6 7 8 9 10)

- RIGHT-ARC: Adds arc (s,l,b) to A and moves b onto the stack since it cannot have a dependent that is already on the stack without having crossing dependency arcs.

- REDUCE: Pops s off the stack if it has a head already.

- SHIFT: Moves b from the beginning of the buffer to the top of the stack.

Training a transition-based parser means finding a sequence of transitions that leads to the gold standard dependency tree for a set of training sentences. Deciding on what transition to apply next can be done using a static oracle, which first checks if there is an arc between the top of the stack and the beginning of the buffer and if yes performs a LEFT-ARC or RIGHT-ARC transition. If no arc exists, it checks if a token from the stack has a dependency relation with the beginning of the buffer and if yes performs a REDUCE transition. Otherwise the SHIFT transition is chosen. This oracle is described by Algorithm 1. An example of a parse using this oracle can be seen in Figure 3.1.

3.1.1 Spurious Ambiguity in the Arc-Eager Parser

The arc-eager dependency parser exhibits spurious ambiguity, meaning more than one transition sequence can lead to the same dependency tree. When using the transitions defined for the arc-eager parser to parse the sentence "John$_1$ gave$_2$ the$_3$ dog$_4$ a$_5$ bone$_6$", whose dependency structure can be seen in Figure 2.1, there are following two possible transition sequences to arrive at the correct parse tree, where S_i means that word i is shifted onto the buffer, $LA_{j\leftarrow i}$ and $RA_{i\rightarrow j}$ that an arc between word i and word j is added and R_i that word i is reduced:

(1) S_1, $LA_{1\leftarrow 2}$, $RA_{0\rightarrow 2}$, S_3, $LA_{3\leftarrow 4}$, $RA_{2\rightarrow 4}$, S_5, $LA_{5\leftarrow 6}$, R_4, $RA_{2\rightarrow 6}$
(2) S_1, $LA_{1\leftarrow 2}$, $RA_{0\rightarrow 2}$, S_3, $LA_{3\leftarrow 4}$, $RA_{2\rightarrow 4}$, R_4, S_5, $LA_{5\leftarrow 6}$, $RA_{2\rightarrow 6}$

We can see that after adding an arc from *gave* to *dog* we can either shift *a* from the buffer onto the stack or pop *dog* off the stack. When transforming a gold sentence into a sequence of transitions there needs to be a mechanism for deciding what transition to apply next. The algorithm that performs this task is referred to as an oracle, which is defined by Algorithm 1. This oracle is static, meaning that when repeatedly using it to predict the transition sequence to build a dependency tree for the same sentence, the resulting transition sequence will always be the same. The oracle described here will always choose the SHIFT transition over the REDUCE transition when a situation like above arises, meaning it puts off the decision whether or not the word on top of the stack can have more dependents in the buffer. The SHIFT-REDUCE ambiguity is the only spurious ambiguity the arc-eager parser exhibits (Nivre, 2012).

Algorithm 1: Oracle for the arc-eager parser

Data: Configuration $(\sigma|s,b|\beta,A)$; set of gold standard arcs for the sentence, A_{gold}
Result: Next transition t

```
if (b,l,s) ∈ A_gold then
    t = LEFT-ARC;
else
    if (s,l,b) ∈ A_gold then
        t = RIGHT-ARC;
    else
        if ∃s' ∈ σ so that (s',l,b) ∈ A_gold ∨ (b,l,s') ∈ A_gold then
            t = REDUCE;
        else
            t = SHIFT;
        end
    end
end
```

Because of the static nature of the oracle used, when training the parser the same transition sequence will be chosen in all cases even if different ones are possible and we

cannot be sure that this is the transition sequence that is easiest to learn. Another problem of having spurious ambiguity in a transition system and using a static oracle to predict the transition sequence, is that if we deviate from the optimal transition sequence while parsing a sentence then we might arrive at a configuration that was not encountered during training, which leads to error propagation since the trained classifier does not know how to deal with it. This problem can be solved by using a dynamic oracle, which is however quite complicated (Nivre, 2012). Another way to remove spurious ambiguity is described in (Cohen et al., 2012). They define a transition system that, like the transition system developed in the course of my project, builds a different parse tree for each distinct transition sequence. My transition system however adds arcs between words as soon as they are the first two items on top of the stack, whereas their transition system only adds arc (i,l,j) once j has all its dependents added.

In the next subsection I will explore an alternate way of removing spurious ambiguity, by making use of the fact that dependency grammars can be transformed into context-free grammars where a sentence corresponds to exactly one parse tree. A transition system can then be developed where each transition corresponds to a production rule of the CFG.

3.2 Attach-Complete Transition System

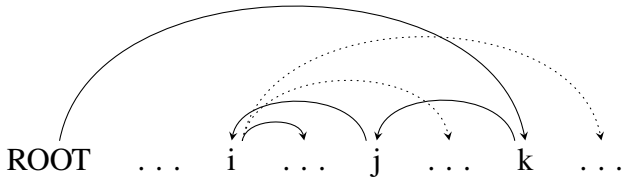
The main idea behind the attach-complete parser is to define dependency structure in terms of a context-free grammar (CFG), which we can then find a transition system for. The intuition for these transitions is that for production rules $A \rightarrow BC$ and $A \rightarrow a$ of the CFG where A, B, C are non-terminals and a is a terminal (a word), whenever the next item in the buffer is a we push A onto the stack and whenever we have BC on top of the stack we replace it with A. A valid parse has been found if the stack contains only the start symbol S in the end. When applying this to dependency parsing, some of the production rules $A \rightarrow BC$ can only be applied if certain arcs between words exist.

The transition system I describe here is based on Eisner's dynamic programming algorithm for parsing context-free grammars (Eisner, 1999). To utilise this algorithm we make use of the fact that projective dependency grammars can be transformed into the CFG described in Figure 3.2 (Johnson, 2007). For each of the CFG production rules, we define a transition performing the equivalent operation that allows us to build the same dependency tree.

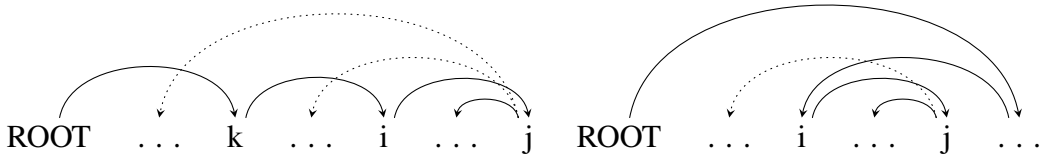
(1) $S \rightarrow L_u \; {}_uR$ where $0 \rightarrow u$
(2) $L_u \rightarrow L_v \; {}_vM_u$ where $v \leftarrow u$
(3) ${}_uR \rightarrow {}_uM_v \; {}_vR$ where $u \rightarrow v$
(4) ${}_xM_y \rightarrow {}_xR \; L_y$

Figure 3.2: This figure shows the CFG equivalent to projective dependency grammar.

Informally, we can say that the transition system works by adding arcs with the ATTACH operations and using the COMPLETE operations as soon as a token that already has a head has no more dependents in the buffer. To accomplish this we add two copies of each word to the buffer, a left copy and a right copy. The ATTACH transitions delete one copy of the dependent token and the COMPLETE transitions delete the second one. As can be inferred when looking at the formal descriptions of the transitions below, the left copy of the word is used for adding arcs to and from words to the left of it and the right copy for adding arcs to and from words to its right. After adding a left arc $i \leftarrow j$ from j to i, the right copy of i is deleted, meaning no more arcs to words to the right of it can be added. This is in accordance with the projectivity constraint, since any arcs to words to the right of j that do not have a head yet would cross an ancestor arc of j, as the dotted arcs in the following illustration show:



It also holds that any arcs from i to words between i and j would have been added already. Similarly, when adding a right arc from i to j, the left copy of j is deleted, meaning j cannot have any more arcs to words on its left. Any such projective arcs to words between i and j would have to have been added already, since the sentence is parsed left-to-right. Two examples of how any arcs from j to words to the left of i are non-projective, can be seen below, where the dotted arcs indicate illegal non-projective arcs:



The configuration (σ, β, A) for this parser consists of a stack σ, a buffer β and a set of arcs A. Let the sentence to be parsed be defined as $w = w_1, w_2, ..., w_n$. In the initial configuration $([], \beta, [])$, $\beta = [1_L, 1_R, ..., n_L, n_R]$.
Transitions:

- ATTACH-LEFT: $(\sigma|i_R|j_L, \beta, A) \Rightarrow (\sigma|j_L, \beta, A \cup (j, l, i))$ if $(j, l, i) \in A_{gold}$
- ATTACH-RIGHT: $(\sigma|i_R|j_L, \beta, A) \Rightarrow (\sigma|i_R, \beta, A \cup (i, l, j))$ if $(i, l, j) \in A_{gold}$
- COMPLETE-LEFT: $(\sigma|i_L|j_L, \beta, A) \Rightarrow (\sigma|j_L, \beta, A)$ if $(j, l, i) \in A_{gold} \wedge \nexists k \exists l' [(i, l', k) \in A_{gold} \wedge k \in \beta]$
- COMPLETE-RIGHT: $(\sigma|i_R|j_R, \beta, A) \Rightarrow (\sigma|i_R, \beta, A)$ if $(i, l, j) \in A_{gold} \wedge \nexists k \exists l' [(j, l', k) \in A_{gold} \wedge k \in \beta]$
- FINISH: $([]|i_L|i_R, [], A) \Rightarrow ([], [], A \cup (0, l, i))$ if $(0, l, i) \in A_{gold}$

- SHIFT: $(\sigma, i|\beta, A) \Rightarrow (\sigma|i, \beta, A)$

These transitions describe the oracle of the transition system during training, where the conditions for applying each of the ATTACH and COMPLETE transitions are checked and if they are not fulfilled, SHIFT is applied if there are still elements in the buffer and FINISH otherwise.

The transition system builds the same parse tree that is derived from the CFG in Figure 3.2 from the bottom up. The CFG rule (4) corresponds to the ATTACH-LEFT and ATTACH-RIGHT transitions, rule (3) to the COMPLETE-RIGHT transition, rule (2) to the COMPLETE-LEFT transition and rule (1) to the FINISH transition. An example for this can be seen in Figure 3.3. It shows the productions of the CFG in the geometric shapes and the corresponding transitions in bold next to them.

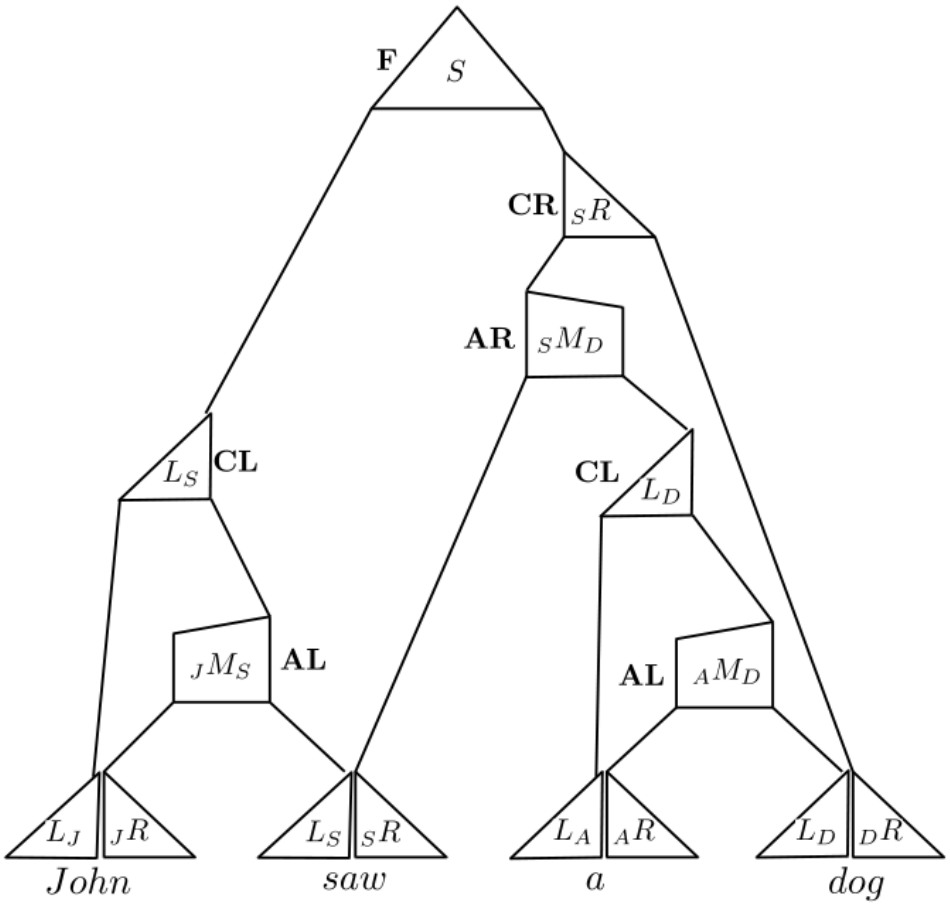


Figure 3.3: This figure shows the connection between the context free grammar that is equivalent to dependency grammar and the ATTACH and COMPLETE transitions. AL = ATTACH-LEFT; AR = ATTACH-RIGHT; CL = COMPLETE-LEFT; CR = COMPLETE-RIGHT; F = FINISH

Since the transition system parses the sentence left-to-right it holds that after an ATTACH-

LEFT transition adding an arc from j to i, all the allowed arcs from i to its dependents have been added already (since its right copy is removed it can only add arcs to words to the left of it which would have been added in previous parsing steps). Therefore we can merge the ATTACH-LEFT and the COMPLETE-LEFT transitions into one transition I will call ATTACH-COMPLETE-LEFT. Formally it is defined as follows:

ATTACH-COMPLETE-LEFT: $(\sigma|i_L|i_R|j_L,\beta,A) \Rightarrow (\sigma|j_L,\beta,A\cup(j,l,i))$ if $(j,l,i) \in A_{gold}$

The transition parser would parse the sentence "John saw a dog." (whose dependency relations can be seen in Figure 2.2) in the following way (leaving SHIFT operations out):

- The buffer would be initialised to $\beta = [John_L, John_R, saw_L, saw_R, a_L, a_R, dog_L, dog_R]$ (in reality the indices of the words are used, but to make understanding easier we are using the words themselves in this example)
- The ATTACH-COMPLETE-LEFT transition will add an arc from *saw* to *John* and delete $John_R$ and $John_L$.
- The ATTACH-COMPLETE-LEFT operation will add an arc from *dog* to *a* and delete a_R and a_L.
- The ATTACH-RIGHT operation will add an arc from *saw* to *dog* and delete dog_L.
- Since *dog* has no more dependents, the COMPLETE-RIGHT transition will delete dog_R.
- The FINISH operation ends the parse and adds an arc from the artificial root word to *saw*.

The full transition sequence for the sentence in Figure 2.2, after the sentence has been restructured into a projective one, can be seen in Figure 3.4.

3.3 Covington's Non-Projective Fundamental Algorithm for Dependency Parsing

Covington's non-projective algorithm works by going through the tokens in a sentence and keeping track of the ones that have been processed so far and the ones that do not have a head yet. When processing the next token t in the sentence, it first looks at the list of tokens without head and checks if any of them are dependents of t and then checks if any of the tokens that have already been processed are t's head (Covington, 2001). The transition system, with configuration (σ,λ,β,A), works by adding arcs between the top of the stack and the first element in the buffer, moving elements from the buffer to the stack, and pushing tokens from the top of the stack onto the list λ if there are no arcs to add between them and the first element of the buffer. When the next token in the sentence is processed, all elements of λ are added to the stack again

Transition	$(\sigma,$	$\beta,$	$A)$
	$[1_l, 1_r]$,	$[2_l, 2_r, ..., 10_l, 10_r]$,	$A = \emptyset$
SHIFT $\Longrightarrow$	$[1_l, 1_r, 2_l]$,	$[2_r, 3_l, 3_r, ..., 10_l, 10_r]$,	A
ATTACH-COMPLETE-LEFT $\Longrightarrow$	$[2_l]$,	$[2_r, 3_l, 3_r, ..., 10_l, 10_r]$,	$A \cup 1 \leftarrow 2$
SHIFT $\Longrightarrow$	$[2_l, 2_r]$,	$[3_l, 3_r, ..., 10_l, 10_r]$,	A
SHIFT $\Longrightarrow$	$[2_l, 2_r, 3_l]$,	$[3_r, 4_l, 4_r..., 10_l, 10_r]$,	A
SHIFT $\Longrightarrow$	$[2_l, 2_r, 3_l, 3_r]$,	$[4_l, 4_r, ..., 10_l, 10_r]$,	A
SHIFT $\Longrightarrow$	$[2_l, 2_r, 3_l, 3_r, 4_l]$,	$[4_r, 5_l, r_r, ..., 10_l, 10_r]$,	A
ATTACH-COMPLETE-LEFT $\Longrightarrow$	$[2_l, 2_r, 4_l]$,	$[4_r, 5_l, r_r, ..., 10_l, 10_r]$,	$A \cup 3 \leftarrow 4$
ATTACH-RIGHT $\Longrightarrow$	$[2_l, 2_r]$,	$[4_r, 5_l, r_r, ..., 10_l, 10_r]$,	$A \cup 2 \rightarrow 4$
SHIFT $\Longrightarrow$	$[2_l, 2_r, 4_r]$,	$[5_l, 5_r..., 10_l, 10_r]$,	A
SHIFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 5_l]$,	$[5_r, 6_l, 6_r..., 10_l, 10_r]$,	A
SHIFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 5_l, 5_r]$,	$[6_l, 6_r..., 10_l, 10_r]$,	A
SHIFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 5_l, 5_r, 6_l]$,	$[6_r, 7_l, 7_r, ..., 10_l, 10_r]$,	A
ATTACH-COMPLETE-LEFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 6_l]$,	$[6_r, 7_l, 7_r, ..., 10_l, 10_r]$,	$A \cup 5 \leftarrow 6$
ATTACH-RIGHT $\Longrightarrow$	$[2_l, 2_r, 4_r]$,	$[6_r, 7_l, 7_r, ..., 10_l, 10_r]$,	$A \cup 4 \rightarrow 6$
SHIFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 6_r]$,	$[7_l, 7_r, ..., 10_l, 10_r]$,	A
SHIFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 6_r, 7_l]$,	$[7_r, 8_l, 8_r, ..., 10_l, 10_r]$,	A
SHIFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 6_r, 7_l, 7_r]$,	$[8_l, 8_r, ..., 10_l, 10_r]$,	A
SHIFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 6_r, 7_l, 7_r, 8_l]$,	$[8_r, 9_l, 9_r, 10_l, 10_r]$,	A
SHIFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 6_r, 7_l, 7_r, 8_l, 8_r]$,	$[9_l, 9_r, 10_l, 10_r]$,	A
SHIFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 6_r, 7_l, 7_r, 8_l, 8_r, 9_l]$,	$[9_r, 10_l, 10_r]$,	A
ATTACH-COMPLETE-LEFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 6_r, 7_l, 7_r, 9_l]$,	$[9_r, 10_l, 10_r]$,	$A \cup 8 \leftarrow 9$
ATTACH-COMPLETE-LEFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 6_r, 9_l]$,	$[9_r, 10_l, 10_r]$,	$A \cup 7 \leftarrow 9$
ATTACH-RIGHT $\Longrightarrow$	$[2_l, 2_r, 4_r, 6_r]$,	$[9_r, 10_l, 10_r]$,	$A \cup 6 \rightarrow 9$
SHIFT $\Longrightarrow$	$[2_l, 2_r, 4_r, 6_r, 9_r]$,	$[10_l, 10_r]$,	A
COMPLETE-RIGHT $\Longrightarrow$	$[2_l, 2_r, 4_r, 6_r]$,	$[10_l, 10_r]$,	A
COMPLETE-RIGHT $\Longrightarrow$	$[2_l, 2_r, 4_r]$,	$[10_l, 10_r]$,	A
COMPLETE-RIGHT $\Longrightarrow$	$[2_l, 2_r]$,	$[10_l, 10_r]$,	A
SHIFT $\Longrightarrow$	$[2_l, 2_r, 10_l]$,	$[10_r]$,	A
ATTACH-RIGHT $\Longrightarrow$	$[2_l, 2_r]$,	$[10_r]$,	$A \cup 2 \rightarrow 10$
SHIFT $\Longrightarrow$	$[2_l, 2_r, 10_r]$,	$[]$,	A
COMPLETE-RIGHT $\Longrightarrow$	$[2_l, 2_r]$,	$[]$,	A
FINISH $\Longrightarrow$	$[]$,	$[]$,	$A \cup 0 \rightarrow 2$

Figure 3.4: Trace of the attach-complete parser for the sentence "John saw a dog which was a Yorkshire terrier yesterday".
1 2 3 4 5 6 7 8 9 10

Transition	$(\sigma,$	$\lambda,$	$\beta,$	$A)$
	$[0],$	$[],$	$[1,...,10],$	$A = \emptyset$
SHIFT $\Longrightarrow$	$[0,1],$	$[],$	$[2,...,10],$	A
LEFT-ARC $\Longrightarrow$	$[0],$	$[1],$	$[2,...,10],$	$A \cup 1 \leftarrow 2$
RIGHT-ARC $\Longrightarrow$	$[],$	$[0,1],$	$[2,...,10],$	$A \cup 0 \rightarrow 2$
SHIFT $\Longrightarrow$	$[0,1,2],$	$[],$	$[3,...,10],$	A
SHIFT $\Longrightarrow$	$[0,...3],$	$[],$	$[4,...,10],$	A
LEFT-ARC $\Longrightarrow$	$[0,1,2],$	$[3],$	$[4,...,10],$	$A \cup 3 \leftarrow 4$
RIGHT-ARC $\Longrightarrow$	$[0,1],$	$[2,3],$	$[4,...,10],$	$A \cup 2 \rightarrow 4$
SHIFT $\Longrightarrow$	$[0,...,4],$	$[],$	$[5,...,10],$	A
NO-ARC $\Longrightarrow$	$[0,...,3],$	$[4],$	$[5,...,10],$	A
NO-ARC $\Longrightarrow$	$[0,1,2],$	$[3,4],$	$[5,...,10],$	A
RIGHT-ARC $\Longrightarrow$	$[0,1],$	$[2,3,4],$	$[5,...,10],$	$A \cup 2 \rightarrow 5$
SHIFT $\Longrightarrow$	$[0,...,5],$	$[],$	$[6,...,10],$	A
SHIFT $\Longrightarrow$	$[0,...,6],$	$[],$	$[7,...,10],$	A
LEFT-ARC $\Longrightarrow$	$[0,...,5],$	$[6],$	$[7,...,10],$	$A \cup 6 \leftarrow 7$
NO-ARC $\Longrightarrow$	$[0,...,4],$	$[5,6],$	$[7,...,10],$	A
RIGHT-ARC $\Longrightarrow$	$[0,...,3],$	$[4,5,6],$	$[7,...,10],$	$A \cup 4 \rightarrow 7$
SHIFT $\Longrightarrow$	$[0,...,7],$	$[],$	$[8,9,10],$	A
SHIFT $\Longrightarrow$	$[0,...,8],$	$[],$	$[9,10],$	A
SHIFT $\Longrightarrow$	$[0,...,9],$	$[],$	$[10],$	A
LEFT-ARC $\Longrightarrow$	$[0,...,8],$	$[9],$	$[10],$	$A \cup 9 \leftarrow 10$
LEFT-ARC $\Longrightarrow$	$[0,...,7],$	$[8,9],$	$[10],$	$A \cup 8 \leftarrow 10$
RIGHT-ARC $\Longrightarrow$	$[0,...,6],$	$[7,8,9],$	$[10],$	$A \cup 7 \rightarrow 10$
SHIFT $\Longrightarrow$	$[0,...,10],$	$[],$	$[],$	A

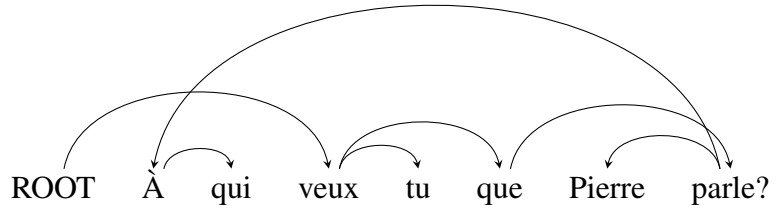
Figure 3.5: Trace for Covington's parser for the sentence "John saw a dog yesterday which was a Yorkshire terrier".
1 2 3 4 5 6 7 8 9 10

to allow for all possible arcs (Nivre, 2008). Here is the list of allowed transitions for configuration $(\sigma|s,\lambda,b|\beta,A)$:

- LEFT-ARC: Adds arc (b,l,s) from the beginning of the buffer to the top of the stack to A, pops s off the stack and prepends it to list λ. The precondition for this transition is that s does not have a head already.
- RIGHT-ARC: Adds arc (s,l,b) to A, pops s off the stack and prepends it to λ.
- NO-ARC: Moves s from the stack to the beginning of λ. This allows for more arcs to be added between the element at the beginning of the buffer and the remaining elements on the stack.
- SHIFT: This transition moves b onto the stack, meaning no more arcs between b and the remaining elements of the stack can be added. It has as consequence that it is not always the case that arcs are considered between all possible combinations of words since some of them can be skipped with this transition.

We can see the trace of a parse using Covington's transition system in Figure 3.5.

It is interesting to note that for this transition system there can also be multiple ways of getting to the same parse tree. Whenever a SHIFT operation is performed when there are no more arcs between the first item in the buffer and the remaining items on the stack, we can first perform NO-ARC transitions until the stack is empty. Looking at the oracle used to predict the next transition, which is described by Algorithm 2, we see that we always perform the SHIFT transition as soon as the first item in the buffer has no more arcs to and from the items on the stack. Using a similar trick to remove this spurious ambiguity as the attach-complete parser of having two copies of each word, a left copy for adding arcs to words to its left and a right copy for adding arcs to words to its right, and deleting one of the copies with an ATTACH transition and the second one with a COMPLETE transition will not be possible, since after adding an arc (i,l,j) while parsing a non-projective sentence, j can still have arcs that are not added yet to words both on its left and on its right. An example of this is following French sentence:



If adding the arc *que* $\rightarrow$ *parle* deletes the left copy of *parle*, then the arc $\grave{A} \leftarrow$ *parle* cannot be added anymore.

Algorithm 2: Oracle for Covington's parser

Data: Configuration $(\sigma|s,\lambda,b|\beta,A)$; set of gold standard arcs for the sentence, A_{gold}
Result: Next transition t

```
if (b,l,s) ∈ A_gold then
    t = LEFT-ARC;
else
    if (s,l,b) ∈ A_gold then
        t = RIGHT-ARC;
    else
        if ∃s' ∈ σ so that (s',l,b) ∈ A_gold ∨ (b,l,s') ∈ A_gold then
            t = NO-ARC;
        else
            t = SHIFT;
        end
    end
end
```

3.4 Covington's Parser with Reduce Transitions

Covington's algorithm for dependency parsing is inefficient in the sense that a sentence of length n is turned into a transition sequence of size $O(n^2)$ since in the worst case every word is compared to every other word when looking for dependency links. The much more efficient arc-eager parser only allows transition sequences of length $O(n)$, leading to much shorter learning and parsing times. Most dependency arcs, even for languages like Dutch with a relative high proportion of non-projective sentences, still do not cross other arcs, meaning that adding all previous words to the stack whenever we look at dependency links to and from the next word in the buffer like Covington's parser does, could lead to wrong arcs being added and an unnecessarily high number of transitions being performed. Therefore it would be useful to give Covington's parser a mechanism of removing words that will not appear in any more dependency relations, both for a reduction in training time as fewer transitions mean fewer training examples which has as consequence that the classifier can be trained faster, and hopefully an improvement in attachment accuracy.

The development of the transition system in this section was motivated by my observations that during some initial experiments, Covington's parser took much longer to train the classifier and had a significantly lower number of correctly added projective dependency arcs than the arc-eager parser. I theorised that allowing Covington's parser to behave like the arc-eager one on projective sentences and to only use the transitions that enable it to add crossing arcs when necessary, would improve both its training time and its attachment accuracy. To do this I have implemented a version of Covington's parser that allows for REDUCE operations. Two versions of the LEFT-ARC transition are needed, one that allows for further non-projective links to and from the child in the dependency relation, LEFT-ARC-KEEP, and one that removes it from consideration, LEFT-ARC-REDUCE. My experiments in Section 5 show that these changes indeed lead to a faster training time and higher attachment scores for projective arcs. What was unexpected is that they also improved the parser's accuracy on non-projective dependency arcs. My assumption that restricting the arcs that could be added in the future would lead to a decreased attachment score for arcs that are crossed in the gold dependency tree was refuted. Section 5 includes explanations on why this might be the case.

Following transitions are allowed for the configuration $(\sigma|s,\lambda,b|\beta,A)$:

- LEFT-ARC-KEEP: Behaves like the LEFT-ARC transition of Covington's parser and adds arc (b,l,s) to A, pops s off the stack and prepends it to list λ. The precondition for this transition is that s does not have a head already. This transition allows for non-projective arcs between s and the remaining items in the buffer to be added.
- LEFT-ARC-REDUCE: This transition is equivalent to the LEFT-ARC transition of the arc-eager parser. It adds arc (b,l,s) from the beginning of the buffer to the top of the stack and pops s off the stack. It can only be executed it s does not have a head already. After applying this transition, no more arcs between s

and the remaining elements of the buffer can be added. Any such arc would be non-projective.

- RIGHT-ARC: Adds arc (s,l,b) to A, pops s off the stack and prepends it to λ. This is exactly the same as the RIGHT-ARC transition of Covington's parser. The RIGHT-ARC transition of the arc-eager parser also does not reduce s, meaning it is essentially equivalent.
- NO-ARC: This behaves like the NO-ARC transition of Covington's parser. It moves s from the stack to the beginning of λ, allowing non-projective arcs to be added between s and the remaining items in the buffer.
- REDUCE: This transition behaves like the REDUCE transition of the arc-eager parser. It pops s off the stack if the precondition that it has a head already is fulfilled. After applying this transition, no more arcs can be added between s and the remaining elements on the buffer.
- SHIFT: This transition moves b onto the stack.

The transition sequence to parse a projective sentence will be the same as the one by the arc-eager parser, except that the arc-eager parser shifts b from the buffer onto the stack immediately after a RIGHT-ARC transition, while the SHIFT operation has to be performed explicitly in the hybrid between the arc-eager and Covington's parser. Figure 3.6 shows the sequence of transitions to parse the non-projective sentence "John saw a dog yesterday which was a Yorkshire Terrier". The non-projective arc is handled with the NO-ARC transition, which adds the word "dog" onto the list λ, so that further arcs between it and the remaining items in the buffer can be added. Again an oracle is used to decide which transition to perform at each configuration, which is described by Algorithm 3.

Transition	$(\sigma,$	$\lambda,$	$\beta,$	$A)$
	$[0],$	$[],$	$[1,...,10],$	$A = \emptyset$
SHIFT $\Longrightarrow$	$[0,1],$	$[],$	$[2,...,10],$	A
LEFT-ARC-REDUCE $\Longrightarrow$	$[0],$	$[],$	$[2,...,10],$	$A \cup 1 \leftarrow 2$
RIGHT-ARC $\Longrightarrow$	$[],$	$[0],$	$[2,...,10],$	$A \cup 0 \rightarrow 2$
SHIFT $\Longrightarrow$	$[0,2],$	$[],$	$[3,...,10],$	A
SHIFT $\Longrightarrow$	$[0,2,3],$	$[],$	$[4,...,10],$	A
LEFT-ARC-REDUCE $\Longrightarrow$	$[0,2],$	$[],$	$[4,...,10],$	$A \cup 3 \leftarrow 4$
RIGHT-ARC $\Longrightarrow$	$[0],$	$[2],$	$[4,...,10],$	$A \cup 2 \rightarrow 4$
SHIFT $\Longrightarrow$	$[0,2,4],$	$[],$	$[5,...,10],$	A
NO-ARC $\Longrightarrow$	$[0,2],$	$[4],$	$[5,...,10],$	A
RIGHT-ARC $\Longrightarrow$	$[0],$	$[2,4],$	$[5,...,10],$	$A \cup 2 \rightarrow 5$
SHIFT $\Longrightarrow$	$[0,2,4,5],$	$[],$	$[6,...,10],$	A
SHIFT $\Longrightarrow$	$[0,2,4,5,6],$	$[],$	$[7,...,10],$	A
LEFT-ARC-REDUCE $\Longrightarrow$	$[0,2,4,5],$	$[],$	$[7,...,10],$	$A \cup 6 \leftarrow 7$
REDUCE $\Longrightarrow$	$[0,2,4],$	$[],$	$[7,...,10],$	A
RIGHT-ARC $\Longrightarrow$	$[0,2],$	$[4],$	$[7,...,10],$	$A \cup 4 \rightarrow 7$
SHIFT $\Longrightarrow$	$[0,2,4,7],$	$[],$	$[8,9,10],$	A
SHIFT $\Longrightarrow$	$[0,2,4,7,8],$	$[],$	$[9,10],$	A
SHIFT $\Longrightarrow$	$[0,2,4,7,8,9],$	$[],$	$[10],$	A
LEFT-ARC-REDUCE $\Longrightarrow$	$[0,2,4,7,8],$	$[],$	$[10],$	$A \cup 9 \leftarrow 10$
LEFT-ARC-REDUCE $\Longrightarrow$	$[0,2,4,7],$	$[],$	$[10],$	$A \cup 8 \leftarrow 10$
RIGHT-ARC $\Longrightarrow$	$[0,2,4],$	$[7],$	$[10],$	$A \cup 7 \rightarrow 10$
SHIFT $\Longrightarrow$	$[0,2,4,7,10],$	$[],$	$[],$	A

Figure 3.6: Trace of Covington's parser with reduce transitions for the sentence "John saw a dog yesterday which was a Yorkshire terrier".
1 2 3 4 5 6 7 8 9 10

Algorithm 3: Oracle for Covington's parser with reduce transitions

Data: Configuration $(\sigma|s,\lambda,b|\beta,A)$; set of gold standard arcs for the sentence, A_{gold}
Result: Next transition t

```
if (b,l,s) ∈ A_gold then
    if ∃b' ∈ β so that (s,l,b') ∈ A_gold ∨ (b',l,s) ∈ A_gold then
        t = LEFT-ARC-KEEP
    else
        t = LEFT-ARC-REDUCE
    end
else
    if (s,l,b) ∈ A_gold then
        t = RIGHT-ARC;
    else
        if ∃s' ∈ σ so that (s',l,b) ∈ A_gold ∨ (b,l,s') ∈ A_gold then
            if ∃b' ∈ β so that (s,l,b') ∈ A_gold ∨ (b',l,s) ∈ A_gold then
                t = NO-ARC
            else
                t = REDUCE
            end
        else
            t = SHIFT;
        end
    end
end
```

Chapter 4

Parser Implementation

4.1 Data

The data set used for evaluating the dependency parsers is the Universal Dependencies treebank, version 1.2, released on November 15, 2015. Each word in a sentence is annotated with its lemma, part-of-speech tags, features detailing additional lexical or grammatical properties, its head and the dependency relation to its head.

Table 4.1 shows properties of the sentences in the UD treebanks for Danish, German, Hungarian, Dutch, Portuguese and Czech. The Dutch data set has with 27.46% an especially high number of non-projective sentences in the test set, which is why we might expect that a dependency parser allowing for crossing dependencies will obtain better results than a solely projective one.

Each line in a Universal Dependencies data file corresponds to a word and an empty line indicates the end of the sentence. The training and testing data is prepared by turning each sentence into a list of `Python` dictionaries, where each dictionary corresponds to a word and represents its properties as *key:value* pairs, which are for example the

	training				testing			
	sentences		arcs		sentences		arcs	
Language	#	% NP	#	% NP	#	% NP	#	% NP
Danish	4,868	22.70	88,979	4.92	322	22.98	5,884	4.47
German	14,118	11.21	269,626	2.42	1,000	22.10	16,609	5.11
Hungarian	1,032	24.81	20,764	5.61	138	24.81	2,725	4.84
Dutch	13,000	30.83	188,882	10.10	386	27.46	5,585	9.67
Portuguese	8,800	18.61	201,845	3.39	288	17.01	5,867	3.22
Czech	68,495	12.49	1,173,282	2.94	10,148	12.82	173,737	2.97

Table 4.1: Number of sentences, number of dependency arcs and the percentage of non-projective sentences and non-projective arcs for both training and testing data in the Universal Dependencies (UD) treebanks for several languages.

index of the word, the word itself, additional information about the word, and the index of the word it depends on. A "root dictionary" is added at the beginning of the list, which corresponds to the root of the dependency tree and has index 0. Several properties of the data are recorded, such as the number of non-projective sentences and the total number of arcs, and how many of them are non-projective. To check if an arc (i,l,j) is non-projective I have implemented an algorithm `IsArcProjective` that goes through the list of all arcs in the dependency tree and checks if there is an arc that has one endpoint between i and j and the other either before $min(i,j)$ or after $max(i,j)$. When checking whether a sentence is non-projective I apply `IsArcProjective` to every arc in the dependency tree of the sentence and output that it is non-projective if any arc is non-projective. See algorithm 4 for a description of the function to check whether an arc is projective.

Algorithm 4: IsArcProjective

Data: Set of arcs A, arc (i,l,j)
Result: True or False
for *arc* $(i_2,l_2,j_2) \in A$ **do**
  $i' = min(i,j)$;
  $j' = max(i,j)$;
  $i'_2 = min(i_2,j_2)$;
  $j'_2 = max(i_2,j_2)$;
  if $i' < i'_2 < j' < j'_2$ **then**
    return False;
  end
  if $i'_2 < i' < j'_2 < j'$ **then**
    return False;
  end
  return True;
end

When trying to train the arc-eager parser on a set of sentences, I encountered the problem that the oracle got stuck on a non-projective sentence. The issue was that the *reduce* transition could only be applied if the item on top of the stack s already had a head. If the arc to s is non-projective, the oracle can reach a state of always predicting *reduce* without the transition being applied since the preconditions are not fulfilled. This showed me that when preparing training data for parsers that can only handle projective sentences, a method is needed to deal with non-projective ones. Since just skipping these training examples would mean that the projective parsers would be trained on less data and therefore have a disadvantage over the non-projective ones when their performance is evaluated, I have implemented an algorithm described by Nivre (2005) that lifts non-projective links until the sentence is projective. We recursively call the `Lift` function on the shortest crossed arc until the sentence is projective. If this shortest crossed arc is between i and j then it attaches j to i's head. This procedure is designed to preserve as much of the original sentence's structure as possible. The `Lift` function performs operation:

Language	Average Sentence Length	Average Number of Lifts
Danish	18.28	5.18
German	19.10	4.47
Hungarian	20.12	4.78
Dutch	14.53	4.65
Portuguese	22.94	4.68
Czech	17.13	3.98

Table 4.2: Shows the average number of lift operations performed to projectivise a non-projective sentence and the average sentence length for different languages.

$$(i,l,j) \Rightarrow (head(i),l,j)$$

Table 4.2 shows the average number of `Lift` operations performed to projectivise a sentence and the average sentence length. We can see that it takes approximately 4-5 operations to turn a non-projective sentence into a projective one. Danish sentences take the most operations to projectivise, even though the average sentence length is shorter than for most other languages. The behaviour of the `MakeProjective` function can be seen in Algorithm 5. The non-projective links are stored in a Priority Queue according to their link length, meaning the `ShortestCrossedLink` function was trivial to implement.

Algorithm 5: MakeProjective

Data: Set of arcs A
Result: Set of projective arcs
while *not IsProjective(A)* **do**
    $a \leftarrow ShortestCrossedLink(A)$;
    $A \leftarrow \{A - \{a\}\} \cup Lift(a)$;
end
return A

4.2 Feature Extraction

The features extracted from each configuration have to be equivalent for all four parsers to fulfil my goal of evaluating the effect of allowing for crossing arcs, since otherwise changes in performance of the parsers could be caused by the different features used. Most features consist of an attribute of a word at a specific position in the configuration. We can for example say that one of the features we look at is the part-of-speech of the word on top of the stack. An advantage of transition-based parsing is that we can also add features related to the parsing history. This can be done by also looking at the set of arcs A, which represents the partially constructed parse tree at the current

configuration. When specifying the position of the word for the *(position, attribute)* pair that represents a feature, we can for example specify ”left-most dependent of the word in the beginning of the buffer” or ”head of the word on top of the stack”. It is of course possible that such a word does not exist (yet), in which case no feature will be added.

It is infeasible to add features for every attribute of every word in the configuration since our feature vector will get extremely big, leading to very long training times. Therefore the features extracted for my implementation of the parsers are based on suggestions from Nivre (2008). Table 4.3 shows the features extracted for each configuration. The first column shows the words we are looking at when extracting features, which are the first four words in the buffer, the left-most dependent of the word in the beginning of the buffer of the current configuration, the two top words on the stack, the head of the word on top of the stack if it has one already, the left- and right-most dependent of the word on top of the stack, the first and last word in list λ if it is part of the configuration and the number of words between the word in the beginning of the buffer and the top of the stack. There are several features we can extract for each of the words we are looking at. In Table 4.3, *Form* refers to the word itself, *Lemma* to the stem of the word, *UPOS* to the universal part-of-speech tag, *XPOS* to certain language specific POS tags, *Features* are additional grammatical and structural properties of the word, such as tense and case and *Dependency* describes the kind of dependency label the dependency arc leading to the word has, if one has been assigned already.

	Form	Lemma	UPOS	XPOS	Features	Dependency
$\beta[0]$	X	X	X	X	X	X
$\beta[1]$	X		X			
$\beta[2]$			X			
$\beta[3]$			X			
$ld(\beta[0])$						X
$\sigma[0]$	X	X	X	X	X	X
$\sigma[1]$			X			
$hd(\sigma[0])$	X					
$ld(\sigma[0])$						X
$rd(\sigma[0])$						X
$\lambda[0]$			X			
$\lambda[n]$			X			

Table 4.3: Features for training the classifier; β = buffer; ld = left-most dependent; rd = right-most dependent; σ = stack; hd = head; λ = list for Covington’s parsers

During the course of my project I also tested other features to see if they would improve performance, such as a feature indicating whether an arc is crossed for the non-

projective parsers. This was implemented by checking for a configuration $(\sigma|s,\lambda,b|\beta,A)$ if a potential arc between s and b is crossed by an arc in A and if it is, adding the feature *ARC-CROSSED* to the feature vector. This feature however only slightly increased attachment accuracy on one language while decreasing it on all other languages.

The classifier I am using in my implementation can only deal with numerical features, not canonical ones. Therefore I have to convert the features consisting of *(position of word, attribute)* pairs into numerical features. To do this, while performing the learning step I keep track of all previously encountered features in a Python dictionary and whenever an unseen feature is extracted it is assigned a distinct integer, starting from 0 and increasing by 1 for every new feature. If during parsing a feature is extracted that is not encountered during training, it is ignored.

For the *(position of word, attribute)* pair *(top of the stack, universal part-of-speech)*, you could have following features associated with it:

s0_UPOS_VERB
s0_UPOS_NOUN
s0_UPOS_DET
s0_UPOS_ADP
.
.
.

Of the positions in the feature vector corresponding to those features only one can be set to 1, whereas all others have to be 0. This leads to a very large and sparse feature vector.

4.3 Learning

Since I am interested in investigating whether allowing for non-projective arcs improves the attachment accuracy of a dependency parser, I am using the same machine learning setup for each parser.

The two types of classifiers most commonly used for transition-based dependency parsing are memory-based classifiers and support vector machines (SVM) (Kübler et al., 2009):

- Memory-based: These classifiers parse a sentence by looking for the examples encountered during training that are the most similar to it. Learning is a very trivial task and just means storing the training instances. All the computations are executed during parsing when the most similar training instances have to be found for the current testing instance.

- SVM: These classifiers try to find hyperplanes to separate the data (represented by feature vectors) according to what class they have (which are the transitions in our case). If the data is not linearly separable, kernel functions can be used to map it into higher dimensional space where it can be separated. In this case the learning stage is much more computationally expensive than the parsing stage.

The justification for choosing a support vector machine as the classifier in my implementation is that they lead to consistently higher attachment accuracies for transition-based dependency parsers (Nivre et al., 2006). The classifier is trained using the Support Vector Classifier from the Python *scikit-learn* library, which is based on the LIBSVM library for Support Vector Machines. Parameters for the classifier are chosen according to Nivre (2008), who chose values resulting in good performance during previous experiments. I have not spent time trying to optimise parameters for the classifier since the aim of my experiments is not to improve the performance of the parsers. I am using a polynomial kernel function of degree 2 with kernel coefficient $\gamma = 0.2$ and penalty parameter $C = 0.5$. Training data for the SVM is obtained by finding a transition sequence leading to an optimal dependency tree for each training sentence and extracting features from the current configuration after each transition. The current configuration refers to the state of the stack, buffer and set of arcs during a transition parse. To find such a transition sequence I implemented the oracle algorithms that are described in Section 3 that predict the next transition for a configuration. I also implemented functions for each parser that when given a configuration and a transition, result in a new configuration that is reached by applying the transition to the old configuration. The list of transitions for each parser, as can be seen in Section 3, describes the behaviour of each of those functions. The input to the SVM is a file describing the transition sequences and the features extracted from the current configuration in the following way:

- Associate each transition that occurs during training with a distinct integer value. If the transition is one that adds arcs, it is defined both by whether it is a right arc or a left arc and the dependency label. An example for this is *LEFT-ARC-subj*. Let us call the set of transitions T. There are 40 different dependency labels, which means that each transition adding an arc can add 40 entries to T. For the arc-eager parser the size of the set T is bounded by 82, since there are at most 40 different left-arc and right-arc transitions and 1 shift and 1 reduce transition. In practice I observed a maximum size of 57 during the learning phase of the arc-eager parser on the Hungarian data set and 47 on the Dutch data set.

- Associate each feature that occurs during training with an integer value, as described in the previous section. An example for such a feature is the part-of-speech tag of the word on top of the stack, which could for example be *s0-POS-verb*. Let us call the set of all possible features F. The size of F can get very large since the features I implemented include the form of the word on top of the stack and in the beginning of the buffer for every configuration. While training the arc-eager parser on Hungarian the maximum size of F was 16,271 and on Dutch it was 36,431.

- For each transition performed during training, add following entry to the training

file:
$t\ f_1 : 1.0, f_2 : 1.0, ..., f_k : 1.0$, where $t \in T$ and $f_i \in F$ and each f_i is a feature of the current configuration.

The resulting SVM is a model that given a vector describing all features of a configuration can tell you the most likely transition.

After some initial experiments where I tried training a classifier for Covington's parser on 10,000 sentences from the Dutch data set which took almost 6 days, I have realised that I needed to find a way to speed up training. This problem was solved by splitting the training data according to a property of the feature vector and training a separate classifier on each of those data sets. During parsing we look at the same property of the feature vector and use this to pick the relevant SVM to predict the next transition. The feature that splits the data set has to have several properties. It is important that the set of different values it could take on is not too large, as that would mean each of the SVMs would only be trained on a small number of examples. The attribute that seems best suited for this is the universal part-of-speech tag, which can only take on 17 different values. Another important property of the feature is that when using it to retrieve the SVM for predicting the next transition during parsing, then each training example that is relevant for the prediction has to have been passed to that SVM during the learning step. A feature well-suited for this purpose is the part-of-speech tag of the first element in the buffer. When deciding on whether or not to add an arc between the first element of the buffer and the element on top of the stack during parsing, it is unlikely that looking at examples of what transition was applied when the word in the beginning of the buffer had a different part-of-speech tag than the current one will help much.

4.4 Parsing

When parsing a sentence s, the initial configuration of the transition system is set to $(\sigma = [0], \beta = s, A = [])$. Features are extracted from this configuration and the most likely transition given the features is predicted using one of the trained classifiers (depending on the part-of-speech tag of the word in the beginning of the buffer). Only features that were encountered during training are added to the feature vector that is given to the classifier as input when predicting the next transition. If the preconditions for the predicted transition are fulfilled, we apply it to the configuration, which is in turn updated. If the preconditions are not fulfilled, we try applying the next most likely transition and so on. It is important that for each possible configuration a transition exists that can be applied to it. For the arc-eager and Covington's parsers this transition is the SHIFT transition. For the attach-complete parser it is the SHIFT transition until the buffer is empty and then the FINISH transition. This transition has been adapted so that it can be applied even if not all preconditions (such that only two items are left on the stack) are fulfilled to ensure that there always exists a legal transition. After applying a transition, we let the classifier predict the most likely transition for the new configuration. This process continues until an end condition is reached, which is an

empty buffer for Covington's and the arc-eager parser and an empty buffer and stack for the attach-complete parser.

4.5 Evaluation Metrics

I will evaluate the performance of the parsers using following metrics:

- Unlabelled attachment score (UAS): The percentage of words that have been assigned the correct head by the parser.
- Labelled attachment score (LAS): The percentage of words that have been assigned the correct head and the correct dependency label.
- Label accuracy (LA): The percentage of words that have been assigned the correct dependency label.
- Unlabelled exact match (UEM): The fraction of sentences that have been transformed into the correct dependency tree, disregarding dependency labels.
- Labelled exact match (LEM): The fraction of sentences that have been transformed into the correct dependency tree, including dependency labels.

I am also interested in evaluating the parsers' performance when it comes to correctly predicting heads for words where the arc connecting them is crossed, since that will allow me to argue about the exact effect attachment accuracy of non-projective arcs has on overall performance. Therefore I compute the UAS for all crossed and all uncrossed arcs and the UEM for all projective and all non-projective sentences.

Chapter 5

Evaluation

5.1 Experiment Results

Table 5.1 shows the UAS and LAS of all parsers that were trained on the complete training sets and tested on the complete test sets for languages Danish, German, Hungarian, Dutch and Portuguese and trained on 40,000 training sentences of the training set and tested on the complete test set for Czech. We can see that both the UAS and the LAS averaged over all languages are the highest for Covington's parser with reduce transitions. It has the best performance on the German, Dutch, Portuguese and Czech data sets. The attach-complete parser gives the best results on the Danish data set and the arc-eager one on the Hungarian data set. Looking at Table 4.1 shows that those two languages have the least number of training examples out of all six languages. This suggests that the non-projective parsers need more training data until their performance converges. Further evidence of this is found in Table 5.2, which shows the results of the parsers on the Czech data set when they are trained on an increasing number of sentences. We can observe that the arc-eager parser shows the best UAS for 625 and 2,500 training sentences, whereas Covington's parser with reduce transitions has the highest attachment scores for all larger training sets.

	UAS (LAS)			
Language	arc-eager	attach-complete	cov	cov+reduce
Danish	80.76 (77.62)	**81.00** (**78.20**)	77.09 (74.80)	80.83 (77.94)
German	79.63 (74.27)	78.81 (73.85)	74.83 (70.70)	**79.90** (**74.69**)
Hungarian	**79.38** (**75.67**)	78.83 (75.56)	75.45 (72.51)	78.09 (74.86)
Dutch	72.05 (69.10)	71.10 (68.40)	71.26 (69.11)	**73.89** (**71.01**)
Portuguese	81.56 (78.95)	81.98 (79.39)	79.82 (77.72)	**83.02** (**80.55**)
Czech	84.10 (81.30)	82.24 (79.49)	80.82 (78.55)	**84.62** (**81.84**)
Average	79.58 (76.15)	78.99 (75.82)	76.55 (73.90)	**80.06** (**76.82**)

Table 5.1: The Unlabeled Attachment Score and the Labeled Attachment Score for the four parsers I implemented on six different languages.

	UAS (LAS)			
# training sentences	arc-eager	attach-complete	cov	cov+reduce
625	**72.40** (**67.82**)	71.35 (66.95)	67.57 (64.35)	71.48 (67.25)
1,250	75.14 (70.91)	74.00 (69.90)	70.81 (67.72)	**75.42** (**71.43**)
2,500	**77.97** (74.00)	76.43 (72.56)	73.58 (70.63)	77.86 (**74.07**)
5,000	79.83 (76.13)	78.15 (74.54)	75.69 (72.80)	**79.95** (**76.39**)
10,000	81.35 (78.02)	79.60 (76.33)	77.47 (74.86)	**81.56** (**78.32**)
20,000	82.98 (79.93)	81.03 (78.03)	79.30 (76.83)	**83.32** (**80.33**)
40,000	84.10 (81.30)	82.24 (79.49)	80.82 (78.55)	**84.62** (**81.84**)

Table 5.2: Shows the UAS and LAS of all parsers on the Czech data set for increasing amounts of training data.

	Words with head			
Language	arc-eager	attach-complete	cov	cov+reduce
Danish	98.64	100	88.46	97.67
German	97.68	100	87.12	97.50
Hungarian	97.54	100	85.65	96.18
Dutch	96.38	100	83.49	93.30
Portuguese	98.01	100	89.14	97.82
Czech	98.19	100	88.89	97.67
Average	97.74	100	87.12	96.67

Table 5.3: The fraction of words that have been assigned a head by the parsers.

It is interesting to note, as Table 5.1 shows, that Covington's parser only outperforms the arc-eager one in one instance, which is the LAS for Dutch. This is somewhat surprising since all languages used for my experiments have a relatively high portion of crossing arcs, leading one to believe that a parser that can handle non-projective structures would outperform one that cannot.

One possible explanation for Covington's parser's relatively poor performance can be found in Table 5.3, which shows the fraction of words that are assigned a head by each of the parsers. The arc-eager and Covington's parsers do not guarantee that a well-formed dependency tree is output. An example for this is the case when only SHIFT transitions are performed. This is a valid transition sequence and ends in the terminating condition that the buffer is empty. However, no arcs are added. A way to ensure that complete trees are output by the dependency parser has been introduced for the arc-eager parser (Nivre et al., 2014). An UNSHIFT transition is added, which once the buffer is empty adds items from the stack back into the buffer until every word is assigned a head. It would be an interesting extension to implement this method for all parsers and see how performance improves.

We can see that the fraction of words that are assigned a head is lowest for Covington's parser, where on average this is the case for only 87.12% of words, compared to

	Arc length (Difference to gold standard)				
Language	gold standard	arc-eager	attach-complete	cov	cov+reduce
Danish	3.56	3.64 (+0.08)	3.72 (+0.16)	2.93 (-0.63)	3.51 (-0.05)
German	3.91	3.91 (+0.00)	4.01 (+0.10)	3.37 (-0.54)	3.87 (-0.04)
Hungarian	3.63	3.47 (-0.16)	3.96 (+0.33)	2.92 (-0.71)	3.28 (-0.35)
Dutch	3.55	3.46 (-0.09)	3.98 (+0.43)	3.00 (-0.55)	3.13 (-0.42)
Portuguese	3.51	3.62 (+0.11)	4.00 (+0.49)	2.93 (-0.58)	3.52 (+0.01)
Czech	3.44	3.51 (+0.07)	3.71 (+0.27)	2.93 (-0.51)	3.43 (-0.01)
Average	3.60	3.60 (+0.00)	3.90 (+0.30)	3.01 (-0.59)	3.46 (-0.14)

Table 5.4: The average arc length (in words) in the gold standard dependency tree and in the dependency trees output by the four parsers for all six languages. The value in brackets describes the difference between the average arc length of the parser and the gold standard.

close to 100% for all other parsers. Let us observe that adding reduce operations to Covington's parser increases the fraction of words with a head by almost 10%, which could explain the increased performance. It is likely that the reason for a lower number of words that are assigned a head by Covington's parser is that during training NO-ARC operations are performed between two words that in different circumstances could have a dependency relation between them. One such example is in Figure 3.5, where a NO-ARC operation is performed between *yesterday* and *was*, which in many cases would have a dependency arc between them.

Table 5.3 also shows that the attach-complete parser assigns heads to all words. This is due to the nature of my implementation of the transition system, which only allows a parse to be finished once both buffer and stack are empty, and words can only be removed after they have been assigned a head. This implementation however is forced to allow multiple words to be attached to the artificial root word, leading to trees that are not well-formed dependency trees. The reason for this is that a transition is needed that can always be applied and once the buffer is empty this transition cannot be the SHIFT transition. Therefore the FINISH transition was adapted so that it can add an arc from the root to any word in the buffer, without checking that this is the last word left.

After considering other possible explanations for Covington's parser's low attachment scores, I theorised that perhaps the fact that whenever a SHIFT operation is performed all previously processed tokens are moved onto the stack leads to a bias for shorter arcs. After a SHIFT transition, we have configuration $(\sigma = [0,1,...,i], \lambda = [], \beta = [i+1,...,n], A)$. The next step now is comparing whether there is an arc between token $i+1$ and token i, then between $i+1$ and $i-1$ and so on, until another SHIFT transition is performed. Since every word is compared to those immediately before it first, there is a higher chance of short arcs being added to the dependency graph than long ones. Many of the longer arcs will not even be considered, since another SHIFT transition will be performed first. For the other parsers the stack only contains a subset of the

	Crossed / Uncrossed			
Language	arc-eager	attach-complete	cov	cov+reduce
Danish	38.40 / 82.74	39.54 / **82.94**	43.35 / 78.67	**46.01** / 82.46
German	40.57 / 81.73	35.14 / 81.16	35.97 / 76.91	**42.10** / **81.93**
Hungarian	35.61 / **81.60**	34.85 / 81.06	34.85 / 77.52	**39.40** / 80.06
Dutch	35.37 / **75.98**	32.59 / 75.22	60.00 / 72.47	**66.85** / 74.65
Portuguese	33.33 / 83.16	25.40 / 83.86	47.09 / 80.91	**57.67** / **83.87**
Czech	49.32 / 85.17	42.52 / 83.45	56.71 / 81.56	**64.53** / **85.23**
Average	38.77 / **81.73**	35.00 / 81.28	46.33 / 78.01	**52.76** / 81.36

Table 5.5: The fraction of correctly assigned heads to words that have crossed and uncrossed dependency relations in the gold standard.

previously processed tokens, in fact only those that are likely to still be involved in further arcs. Therefore I analysed the average arc length (in words) in the dependency graphs that were output by the parsers and compared them to the gold standard values. The results of this analysis can be found in Table 5.4. It shows that it is indeed the case that the average arc length of Covington's parser's dependency graphs is significantly shorter than that of the gold standard. To be precise, it goes down from 3.60 to 3.01, whereas the arc-eager parser and Covington's parser with reduce transitions have an average arc length that is very close to the gold standard and the attach-complete parser one that is on average 0.3 words higher. This is evidence that part of the reason for Covington's parser's poorer performance is that it is more likely for short arcs to be added.

We can see the proportion of words that have been assigned the correct head in Table 5.5, split according to whether their dependency relation in the gold standard is projective or non-projective. For words that are connected to their heads with a dependency arc that crosses other dependency arcs, Covington's parser with reduce transition outperforms all other parsers on all languages. When looking solely at words with uncrossed dependency relations we observe that the arc-eager parser has the highest average UAS. Since we know from Table 5.1 that overall Covington's parser with reduce transitions has the highest average UAS, we see how allowing for non-projective dependency arcs makes an improvement in attachment scores.

Looking at the LA in Table 5.6 shows some unexpected results. We see that the arc-eager parser has the highest LA for non-projective words for all six languages. Curiously, this is even the case for German and Portuguese, where Covington's parser with reduce transitions has the highest LA scores overall. This shows that in many cases the correct dependency label is predicted for a word, even if it is attached to the wrong head. This is further evidenced by the fact that the LA is higher than the UAS in all cases. This raises the question of how hard the problem of assigning correct dependency labels to words is if it is viewed as a tagging problem. To investigate this I have computed the LA for the sentences in the Dutch test set when using the primitive method of looking at the part-of-speech tag of each word and assigning it the dependency label that is most common for words with that tag. This method resulted in a

	LA total uncrossed / crossed			
Language	arc-eager	attach-complete	cov	cov+reduce
Danish	89.46 90.18 / **74.14**	**89.58** **90.32** / 73.76	82.55 83.35 / 65.40	89.33 90.07 / 73.38
German	86.54 86.88 / **80.31**	86.33 86.96 / 74.53	78.88 79.80 / 61.79	**86.67** **87.06** / 79.48
Hungarian	**87.34** **88.35** / **67.42**	86.17 87.43 / 61.36	78.90 80.60 / 45.45	85.80 87.08 / 60.61
Dutch	**83.92** **84.42** / **79.26**	81.50 83.21 / 65.56	75.54 76.73 / 64.44	82.56 83.23 / 76.30
Portuguese	90.73 91.16 / **77.78**	89.53 90.56 / 58.73	83.72 84.43 / 62.43	**90.85** **91.41** / 74.07
Czech	**90.58** **90.89** / **80.47**	88.68 89.20 / 71.67	83.63 84.14 / 66.94	90.37 90.68 / 80.08
Average	**88.10** **88.65** / **76.54**	86.96 87.95 / 67.60	80.54 81.51 / 61.07	87.60 88.26 / 73.99

Table 5.6: The Label Accuracy for all arcs, all uncrossed arcs and all crossed arcs.

label accuracy of 66.62%, which is only 8.92% lower than the one computed by Covington's parser. When looking closer at the dependency labels of words with certain POS tags, I observed that determiners had dependency label *det* in 98.43% of cases, punctuation symbols the label *punct* in all cases, adpositions the label *case* 88.44% of the time and pronouns the label *nmod* in 65.6% of cases. Nouns had four main types of dependency labels, depending on whether they are subjects of verbs, objects of verbs, nominal modifiers or conjuncts. My primitive method only used information about the POS of a word in order to assign dependency labels, but the dependency parsers also have information about surrounding words, which would make differentiating between the different types of nouns much more accurate. These experiments showed that computing the label accuracy is not a very hard problem, since extremely basic methods can achieve reasonably high results.

It is also interesting to look at the fraction of sentences that were parsed into the completely correct dependency trees. Table 5.7 shows that on average the attach-complete parser has the highest scores. It far outperforms the arc-eager and Covington's parser in many cases and only has lower UEM than Covington's parser with reduce transitions for Dutch and Czech. One might assume that part of the reason for this is that it assigns a head to all words unlike the other parsers, as can be seen in Table 5.3. However, further investigation shows that on average 26.02% of dependency trees output by the attach-complete parser have arcs from the artificial root to more than one word. A well-formed dependency tree cannot have more than one word as an immediate dependent of the artificial root word, meaning it is impossible for those trees to be correct dependency trees. Of the trees output by the arc-eager parser, on average only 18.38%

	UEM (LEM)			
Language	arc-eager	attach-complete	cov	cov+reduce
Danish	25.16 (21.43)	**30.12** (**25.78**)	21.43 (19.25)	24.53 (21.74)
German	24.30 (13.40)	**29.20** (**18.30**)	17.50 (11.30)	25.00 (14.10)
Hungarian	24.64 (16.67)	**30.43** (**21.74**)	15.22 (10.14)	25.36 (15.94)
Dutch	20.98 (18.13)	23.58 (**21.76**)	18.39 (17.61)	**24.35** (19.95)
Portuguese	26.74 (19.44)	**27.43** (**23.61**)	22.92 (17.01)	**27.43** (19.79)
Czech	36.15 (28.29)	35.07 (27.58)	28.82 (24.24)	**38.30** (**29.82**)
Average	26.33 (19.56)	**29.30** (**23.13**)	20.71 (16.59)	27.49 (20.22)

Table 5.7: The Unlabeled Exact Match and Labeled Exact Match for all parsers I implemented on six different languages.

	Proj. / Non-Proj			
Language	arc-eager	attach-complete	cov	cov+reduce
Danish	32.66 / 0.00	**39.11** / 0.00	27.02 / 2.70	30.65 / **4.05**
German	31.19 / 0.00	**37.48** / 0.00	22.46 / 0.00	31.96 / **0.45**
Hungarian	32.69 / 0.00	**40.38** / 0.00	19.23 / **2.94**	33.65 / 0.00
Dutch	28.93 / 0.00	**32.50** / 0.00	22.14 / 8.49	27.86 / **15.09**
Portuguese	32.22 / 0.00	**33.05** / 0.00	26.78 / 4.08	30.96 / **10.20**
Czech	41.46 / 0.00	40.23 / 0.00	32.50 / 3.84	**42.34** / **10.84**
Average	33.19 / 0.00	**37.12** / 0.00	25.03 / 3.67	32.91 / **6.77**

Table 5.8: The UEM scores for projective and non-projective sentences.

do not meet the constraint that every word has a head. Since therefore the number of well-formed trees output by the attach-complete parser is lower than that of the arc-eager parser while it still has significantly higher exact match scores, we see that its increased UEM and LEM scores are not due to the fact that all words are attached a head. This shows that the attach-complete parser is better at getting complete trees right, but since its average UAS and LAS are slightly lower than those of the arc-eager parser it must be the case that once a mistake has been made during parsing it is not as good at recovering and still achieving a parse tree close to the gold standard as the arc-eager parser.

Looking at Table 5.8 shows that the UEM for the arc-eager and attach-complete parser is 0 for non-projective sentences. This is obvious, since both parser can only output projective dependency trees. That is why it is even more surprising that the attach-complete parser exhibits the overall best average exact match scores. Looking at the UEM for non-projective sentences, we can see that it is quite low even for the non-projective parsers, where Danish, German and Hungarian have significantly lower scores than Dutch, Czech and Portuguese.

Table 5.9 looks at the amount of spurious ambiguity in the arc-eager parser. The values indicate the average number of distinct transition sequences leading to the correct

	Danish	German	Hungarian	Dutch	Portuguese	Czech
Amount of spurious amb.	3.41	4.29	7.73	2.62	4.79	2.99

Table 5.9: The average number of transition sequences leading to the correct parse tree for the arc-eager parser.

parse tree during the training stage of the arc-eager parser. It is computed by looking at the average number of times a SHIFT operation is performed even though a REDUCE operation would have led to the correct dependency tree too for a given sentence. There does not seem to be a correlation between the amount of spurious ambiguity in a sentence and the difference in UAS and LAS between the arc-eager and the attach-complete parser. The arc-eager parser has higher attachment accuracy on both the language with the least amount of spurious ambiguity (Dutch) and the language with the highest amount of spurious ambiguity (Hungarian). On the other hand, when considering the exact match scores in Table 5.7 there does seem to be evidence for a correlation between how much spurious ambiguity a sentence has and how much better the attach-complete parser is at producing the completely correct dependency tree. The two languages with an average of less than three distinct transition sequences leading to the same dependency tree, Dutch and Czech, either show very little improvement in UEM scores (2.6% for Dutch) or even a decrease (for Czech). In contrast, for Hungarian, which has on average over seven transition sequences leading to the same dependency tree, the attach-complete parser's UEM is almost 6% higher.

I also explored the differences in learning and parsing time between the four parsers, as can be seen in Table 5.10. We observe that the arc-eager parser is the fastest, followed by Covington's parser with reduce transitions, then the attach-complete one and Covington's parser as the slowest. Table 5.11 shows that the reason for such big differences in training and parsing time can be attributed to the number of transitions needed to find the dependency tree of a sentence. Covington's parser needs about twice as many transitions on average as the arc-eager parser, explaining its slower runtime. Adding reduce transitions means that the number of transitions is decreased drastically, which leads to a significant improvement in learning and parsing times.

5.2 Critical Evaluation of Results

The main objective of my project was to look at how allowing for non-projectivity influences the accuracy of parsers and not to achieve state-of-the-art parsing accuracies. Nevertheless, it is hard to argue about the significance of my novel transition system that combines transitions from the arc-eager and Covington's parser, without being able to show that it manages to match accuracies of state-of-the-art dependency parsers. Results of a dependency parser using Mate tools which "combine the advantages of graph-based and transition-based dependency parsers" (Bohnet and Kuhn, 2012), on the Universal Dependencies data for German, Czech and Hungarian show that they

	Learning Time			
Language	arc-eager	attach-complete	cov	cov+reduce
Danish	2,973	4,347	19,825	3,794
German	13,086	24,051	66,845	21,979
Hungarian	90	169	290	120
Dutch	9,864	17,263	55,037	10,706
Portuguese	21,289	35,663	113,785	15,804
Czech	200,066	725,968	722,860	304,197
Average	41,228	134,576	163,107	59,433
	Parsing Time			
Language	arc-eager	attach-complete	cov	cov+reduce
Danish	212	345	474	389
German	812	1,443	3,199	1,420
Hungarian	22	45	64	30
Dutch	416	483	622	353
Portuguese	364	572	1,331	874
Czech	39,345	79,268	96,425	53,036
Average	6,862	13,693	17,019	9,350

Table 5.10: The runtime in seconds for the learning and parsing stage of all four parsers for the six languages.

	# Transitions			
Language	arc-eager	attach-complete	cov	cov+reduce
Danish	34.61	59.58	63.46	43.83
German	36.21	61.29	68.30	44.45
Hungarian	38.34	64.34	73.58	47.03
Dutch	27.17	45.87	51.13	34.17
Portuguese	43.90	76.24	86.80	55.76
Czech	32.27	56.23	61.36	41.30
Average	35.42	60.59	67.44	44.43

Table 5.11: The average number of transitions needed to parse a sentence during training for each parser across six languages.

achieve better performance than the parsers I implemented (Tiedemann, 2015). The UAS of their parser compared to my best result was +5.42% for Czech, +4.48% for German and +5.90% for Hungarian. In order to argue about the overall value of Covington's transition system with reduce transitions, more work has to be done on evaluating whether a performance matching that of state-of-the-art dependency parsers can be achieved.

Adding more features would be a way to improve performance of the parsers, though also negatively impacting training and parsing times. A feature model that has been shown to lead to higher attachment scores for transition-based dependency parsers is presented in (Zhang et al., 2011). It includes several features describing the distance between words, the number of dependents a word already has, and third-order features such as the head of the head of the token on top of the stack. Implementing these features for my parsers would help with evaluating if state-of-the-art accuracies can be achieved.

A possible reason for the fact that Covington's parser has lower scores is that less arcs are added during parsing, probably due to too many training examples for the NO-ARC transition. In order to fairly compare its performance to the version with added reduce transitions it is necessary to implement a variation of Covington's transition system that ensures that the dependency graphs it outputs are well-formed trees. Only if that parser still gives lower attachment accuracies can I reliably argue about the superiority of my novel transition system. A method of ensuring that the parser outputs only well-formed trees where every node has a head, is described in (Nivre et al., 2014) for the arc-eager transition system. Implementing a similar method for Covington's parser and Covington's parser with reduce transitions would be the next step in exploring how restricting the further arcs that can be added improves attachment accuracy.

This report also proposes a projective transition system that does not exhibit spurious ambiguity, which has been found to give the best average EM score of the four parsers I evaluated. However, it is not clear if the removal of spurious ambiguity is part of the cause for these improved exact match scores, even though that might be evidenced by the fact that for the language with the most spurious ambiguity (Hungarian) the UEM of the attach-complete parser increases by almost 6% compared to the arc-eager parser, while for the language with the least amount of spurious ambiguity (Dutch) an improvement of only 2.6% can be observed.

Since the attach-complete parser only has higher attachment scores than the arc-eager parser for two of the six languages, this report cannot present evidence that removing the spurious ambiguity found in the arc-eager parser improves attachment scores. It is possible that spurious ambiguity is not necessarily a downside, since if there are multiple transition sequences that lead to the correct dependency tree then either one of them can be predicted during parsing and we can still get the correct result.

Chapter 6

Conclusion

My project was set out to explore how allowing for non-projective arcs changes the attachment accuracy of transition-based dependency parsers. To do this I first implemented two existing parsers, the well-known arc-eager parser producing projective dependency trees and Covington's non-projective parser. After investigating several problems with the parsers, two novel transition systems were introduced and implemented. The first one is the attach-complete transition system that also only deals with projective trees but does not exhibit spurious ambiguity. It is based on the fact that dependency structure can be described by a context-free grammar that gives rise to an unambiguous transition system. The second novel transition system combines transitions from the arc-eager and Covington's parser and can therefore parse all possible dependency trees while at the same time having a mechanism of reducing tokens so they cannot be involved in any further arcs. It was motivated by the observation that Covington's parser was inefficient when it comes to training times and has lower attachment scores than the other parsers. Part of the reason for this is that the average arc length of the dependency trees it outputs is significantly shorter than that of the gold standard dependency trees and that over 10% of words were not assigned a head at all.

Evaluation of the four parsers on six languages with a significant portion of non-projective sentences showed that superior attachment scores could be achieved by not restricting the class of dependency trees to those without crossing arcs. Especially the result that the projective arc-eager parser on average assigns a higher fraction of correct heads to words whose dependency relation is not crossed in the gold standard than both non-projective parsers while Covington's parser with reduce transitions has the highest average attachment scores overall, shows that a parser's accuracy on non-projective arcs plays a significant role in performance.

A downside of using non-projective transition-based dependency parsers is that they are often significantly slower than projective ones, as can be seen by the fact that the training time for Covington's parser is on average four times as long and the parsing time three times as long as that of the arc-eager parser. This is another reason why adding reduce transitions to Covington's parser is an improvement, since it reduces

training time to a third and parsing time to half that of Covington's parser.

The fact that Covington's parser has poorer attachment scores than the arc-eager one in all cases except for the LAS for Dutch shows that allowing for any and all dependency relations between words might be too unrestricted and not realistic for the natural languages I used for evaluation. Adding reduce transitions, which provide a way of restricting the dependency arcs that can be added in the future, leads to higher attachment scores for all languages. Covington's parser with reduce transitions behaves like the arc-eager one on projective sentences without loosing the ability to add non-projective arcs. This explains why the attachment accuracy of projective arcs of the arc-eager parser and Covington's parser with reduce transitions is very similar in all cases. The UAS for non-projective arcs however is up to 31.48% (for Dutch) higher for the non-projective parser, explaining the overall better performance.

When it comes to attachment scores, the attach-complete parser is slightly behind both the arc-eager parser and Covington's parser with reduce transitions. That is why it is especially surprising that it has the highest fraction of completely correctly parsed dependency trees. From these facts it follows that once the attach-complete parser arrives at a configuration from which the gold standard parse tree cannot be reached anymore, it is not as good as the other parsers at recovering from its error and still arriving at a near-optimal dependency tree.

Chapter 7

Future Work

In the course of the second part of my MInf project I am planning on implementing the method described in (Nivre et al., 2014) to ensure only well-formed trees are output by the dependency parser, for my three transition systems that do not enforce a tree constraint. This will most likely improve attachment accuracies of those parsers and help me argue about whether adding reduce transitions to Covington's transition system leads to a better parser.

There are many more forms of dealing with non-projectivity during transition-based dependency parsing than Covington's algorithm and I am interested in exploring how they affect attachment accuracies of dependency parsers. Another possibility is to use a parser that can only output projective dependency trees, like the arc-eager parser, for a technique called pseudo-projective dependency parsing (Nivre et al., 2005). While projectivising the training sentences by applying `Lift` operations that transform a non-projective arc between a head and a dependent into an arc from the head of the head to the dependent, we now include information on this transformation in the dependency label. We lift the arc from i to j, where $(i,l,j),(head(i),l',i) \in A$, by transforming it into $(head(i),l' \rightarrow l,j)$. Both the dependency relation between i and its head, l', and the dependency relation between i and j, l, are now recorded in the new dependency label. When such a label is assigned during parsing, the information is used to transform the arc into the multiple arcs corresponding to it.

Nivre (2009) introduces an online reordering transition system allowing all possible non-projective structures, that works by adding a *swap* transition to the arc-standard parser. This transition moves the second word on the stack to the beginning of the buffer, essentially swapping the first two words on top of the stack. Since this allows you to arbitrarily change the the order of the words in the sentence it is possible to arrive at an order where none of the dependency arcs are crossed, meaning the correct dependency tree can be built by a projective parser.

In (Pitler et al., 2015) a transition system for parsing a class of mildly-non-projective

dependency trees called Crossing Interval Trees is proposed. This subclass of all possible dependency trees covers 96.7% of sentences when evaluated on 10 languages, compared to projective trees which cover only 79.4%. Two registers are added to the configuration of the transition system and non-projective arcs can be added between the words on the stack and in these registers.

My goal is to implement the pseudo-projective transition system, the online reordering transition system and a mildly non-projective transition system and compare their performance to the parsers I have implemented this year.

Several interesting papers on using neural networks for transition-based dependency parsing have been published recently. Chen and Manning (2014) propose a way to use neural networks to train dependency parsers that shows improved attachment accuracies while being 20 times faster than previous systems. Unlike my parsers, which use tens of thousands of indicator features that are set to 1 if they appear in the current configuration and to 0 if they do not, they use only about 200 dense features that can be learnt efficiently by neural networks. The paper (Dyer et al., 2015) shows a way to use three recurrent neural networks with long short-term memory units representing the stack, the buffer and the history of parsing decisions to achieve even higher accuracies. As part of my MInf 2 project I want to implement this for my transition parsers and evaluate how their performance changes.

Bibliography

Bohnet, B. and Kuhn, J. (2012). The best of both worlds: A graph-based completion model for transition-based parsers. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, EACL '12, pages 77–87, Stroudsburg, PA, USA. Association for Computational Linguistics.

Chen, D. and Manning, C. D. (2014). A fast and accurate dependency parser using neural networks. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Cohen, S. B., Gómez-Rodríguez, C., and Satta, G. (2012). Elimination of spurious ambiguity in transition-based dependency parsing. *CoRR*, abs/1206.6735.

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

Culotta, A. and Sorensen, J. (2004). Dependency tree kernels for relation extraction. In *Proceedings of the 42Nd Annual Meeting on Association for Computational Linguistics*, ACL '04, Stroudsburg, PA, USA. Association for Computational Linguistics.

Ding, Y. and Palmer, M. (2005). Machine translation using probabilistic synchronous dependency insertion grammars. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 541–548, Stroudsburg, PA, USA. Association for Computational Linguistics.

Dyer, C., Ballesteros, M., Ling, W., Matthews, A., and Smith, N. A. (2015). Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 334–343, Beijing, China. Association for Computational Linguistics.

Eisner, J. and Satta, G. (1999). Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics on Computational Linguistics*, ACL '99, pages 457–464, Stroudsburg, PA, USA. Association for Computational Linguistics.

Eisner, J. M. (1996). Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th Conference on Computational Linguistics - Volume 1*, COLING '96, pages 340–345, Stroudsburg, PA, USA. Association for Computational Linguistics.

Goldberg, Y. and Nivre, J. (2012). A dynamic oracle for arc-eager dependency parsing.

In *Proceedings of COLING 2012*, pages 959–976, Mumbai, India. The COLING 2012 Organizing Committee.

Johnson, M. (2007). Transforming projective bilexical dependency grammars into efficiently-parsable cfgs with unfold-fold. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 168–175, Prague, Czech Republic. Association for Computational Linguistics.

Kübler, S., McDonald, R., and Nivre, J. (2009). Dependency parsing. *Synthesis Lectures on Human Language Technologies*, 2(1):1–127.

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

Nivre, J. (2008). Algorithms for deterministic incremental dependency parsing. *Comput. Linguist.*, 34(4):513–553.

Nivre, J. (2009). Non-projective dependency parsing in expected linear time. In *ACL 2009, Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the AFNLP, 2-7 August 2009, Singapore*, pages 351–359.

Nivre, J. and Fernández-González, D. (2014). Arc-eager parsing with the tree constraint. *Comput. Linguist.*, 40(2):259–267.

Nivre, J., Hall, J., Nilsson, J., Eryiğit, G., and Marinov, S. (2006). Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL)*, pages 221–225.

Nivre, J. and Nilsson, J. (2005). Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 99–106, Stroudsburg, PA, USA. Association for Computational Linguistics.

Pitler, E., Kannan, S., and Marcus, M. (2013). Finding optimal 1-endpoint-crossing trees. *TACL*, 1:13–24.

Pitler, E. and McDonald, R. (2015). A linear-time transition system for crossing interval trees. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 662–671, Denver, Colorado. Association for Computational Linguistics.

Tiedemann, J. (2015). Cross-lingual dependency parsing with universal dependencies and predicted pos labels. In *Proceedings of the Third International Conference on Dependency Linguistics (Depling 2015)*, pages 340–349.

Wang, M., Smith, N. A., and Mitamura, T. (2007). What is the jeopardy model? a quasi-synchronous grammar for qa. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 22–32, Prague, Czech Republic. Association for Computational Linguistics.

Yamada, H. and Matsumoto, Y. (2003). Statistical dependency analysis with support vector machines. In *Proceedings of 8th International Workshop on Parsing Technologies*, pages 195–206.

Zhang, Y. and Nivre, J. (2011). Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193.