An Experimental Study of Nivre's Parser

Peter Nilsson

Examensarbete för 30 hp
Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds Universitet

Thesis for a diploma in computer science, 30 ECTS credits
Department of computer science, Faculty of science, Lund University

Abstract

In most fields within computer linguistics it is essential to have access to high-quality syntactic parsing. During the previous 10-15 years great improvements have been made in development of parsing algorithms. Increasingly, the approach is to use algorithms to train the parser from annotated text samples. This report studies one of the most successful parser algorithms and explores methods of how to find optimal training parameters.

Sammanfattning

För de flesta områden inom datalingvistiken är det av avgörande betydelse att ha tillgång till en syntaktisk parser med hög prestanda. De senaste 10–15 åren har stora framgångar uppnåtts inom utvecklandet av parseralgoritmer. I allt högre grad används tillvägagångssättet att träna parsern med exempel av uppmärkt text. I denna rapport studeras en av de mest framgångsrika parseralgoritmerna och undersöker olika sätt att finna de bästa parametrarna för träning.

Contents

Acknowledgements
I would like to thank my supervisor Pierre Nugues for many inspiring discussions and his always encouraging support.

Chapter 1

Introduction

One of the activities that distinguishes man from other living beings is the use of language. Most humans can handle at least one language. When we grow up we gradually build an infallible sense for how our native language is constructed and what constructs belong and does not belong to it.

The linguist tries to describe how this sense works. The common idea is to describe the language constructs using a set of rules called a grammar. These rules shall define how different parts of the language are connected. Usually it is required that, to be called a grammar, the rules should be quite detailed at the level of single and groups of words as parts of a sentence.

Thus, using the rules in grammar we can describe all possible sentences in a language. Ideally such a grammar should be able to describe every possible sentence in the language and at the same time not describe any invalid sentence. In this way we can use the grammar to answer the question “Does this sentence belong to the language?”.

The notion of a grammar in the western culture goes back to ancient Greece and Alexandria. It includes the concept of *part of speech* as constituents of a sentence, e.g. nouns and verbs. There have been different opinions on exactly what those parts are, but the basic set has remained practically unchanged to the present day.

It is of course interesting to compare the difficulty, or impossibility, of constructing a perfect grammar to the ease with which a child intuitively learns its native language without any theoretical aid. It is not known how the human sense for right and wrong in a language works. The construction of a grammar is an attempt to define the exact rule set that seems to be used when using a language. It is comparable to other situations where an observer tries to make a model describing a phenomenon in nature. The grammar constructor doesn’t necessarily have to believe that this is the exact working of the human mind, in particular since a native speaker usually exceeds a grammar in performance.

Much of the refinement of traditional grammar was made when applied to Greek and classical Latin being taught and studied as second language. This was the case in Alexandria around 100 BC and in Rome ca 300 AD (Keith, 2007, p. 11).

The term *parsing* is originally used to denote the act of reading a text and describing how it is constructed using a grammar. The meaning has been slightly generalised to indicate a method of consuming a text and deliver one or more analyses. It is not mandatory that the analysis is based on a grammar. Other means are also possible.

During the latter part of the twentieth century it has been technically feasible to handle very large amounts of text. The mere difference between the size of how much text is accessible and how fast it can be processed today compared to 50 years ago makes a considerable change to the value of statistical analysis of the data. Large sets of text are collected as base for analysis. Such a collection is called a corpus. The text can also be annotated according to a grammar. A text corpus with syntactically annotated sentences is called a *treebank*. If the annotations are made by hand it is also referred to as a *gold standard*.

Besides the corpora there is also a huge amount of text accessible on the Internet which can be used for statistical analysis.

This report presents a study of one of the most successful parser algorithms today.

Chapter 2

Background

2.1 Two grammars

This chapter will briefly describe two of the most common grammars during the previous half century or more. They represent two different approaches on how to describe the sentence structure of a natural language.

The *phrase-structure grammar* considers a sentence to be built from phrases which in turn is built from smaller constituents, where the smallest is the actual word. For this reason it is also called *constituent grammar*.

The *dependency grammar*, on the other hand, is focused on the relations (or dependencies) between the individual words without any intermediate representations.

These are by no means totally incompatible approaches to the language, and there are ways to view one in the perspective of the other, but traditionally there has been a polarization between the two.

2.2 Phrase-structure grammar

The prevailing theory for a grammar during the second half of the previous century is the phrase-structure grammar. One of the most influential linguists in this field is Noam Chomsky. In *Semantic Structure*, Chomsky (1957) presents a grammar of phrase-structure rules. A simple example is the very small grammar in Figure 2.1 (Chomsky, 1957, p. 26).

Each rule with the form $X \rightarrow Y$ should be read as "rewrite X as Y". Thus, according to the first rule, a Sentence can be rewritten as *NP* (a Noun Phrase) and VP (a Verb Phrase).

Figure 2.2 shows what is called a derivation of the sentence *The man hit the ball*. Starting with the Sentence, each following line is a rewriting of the previous

(i)	$Sentence \rightarrow NP + VP$
(ii)	$NP \rightarrow T + N$
(iii)	$VP \rightarrow V + NP$
(iv)	$T \rightarrow$ the
(v)	$N \rightarrow$ man, ball, etc
(vi)	$V \rightarrow$ hit, took, etc

Figure 2.1: A simple grammar.

$Sentence$	
$NP + VP$	(i)
$T + N + VP$	(ii)
$T + N + V + NP$	(iii)
the $+ N + V + NP$	(iv)
the + man $+ V + NP$	(v)
the + man + hit $+ NP$	(vi)
the + man + hit $+ T + N$	(ii)
the + man + hit + the $+ N$	(iv)
the + man + hit + the ball	(v)

Figure 2.2: Derivation of the sentence *The man hit the ball*.

using one of the rules (as indicated in parentheses). It is clear that a sentence is considered being composed of smaller constituents, “phrases”. Using the rewriting rules each constituent is replaced by its constituents and this procedure continues until no more replacement can be done and the actual words in the sentence remains. The bottom row of Figure 2.2 is said to be terminated, and it is called a terminal string. The terminal strings for a language is the (possibly infinite) set of valid strings for that language.

2.3 Parsing phrase-structure grammars

A parser would read the sentence and, using the rules, generate the analysis shown as a graph in Figure 2.3. This can be done in different ways, but they are mainly divided into two categories, top-down and bottom-up.

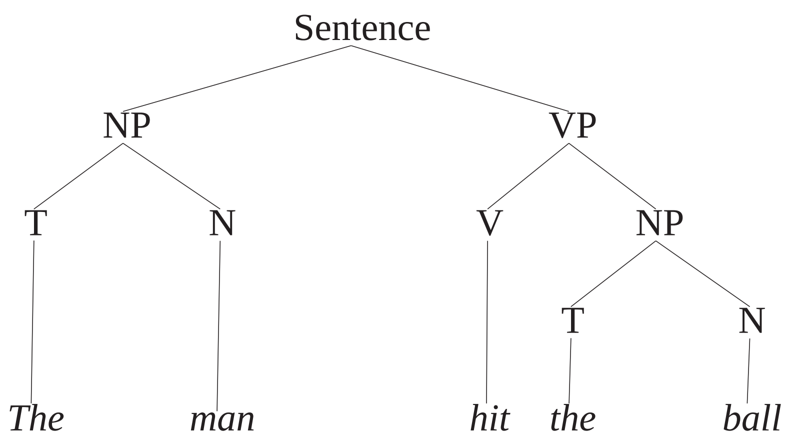

Figure 2.3: Tree structure of *The man hit the ball*.

The behavior for the top-down parser is to take the rule that describes the full sentence and then gradually try to unfold the structure by applying the rewriting rules. Another way of saying this is that a top-down parser searches the rules from the left side of the arrow to the right.

In the example sentence *the man hit the ball* above, the top-down parser will start with the first rule where Sentence can be rewritten as $NP+VP$. It will then look for all possible rewritings of *NP* and find rule (ii). The right side of rule (ii) is $T+N$. Once again it will take the first part, T, and try to find rewritings of this. It will find rule (iv) and see that T can be rewritten as the word *the*. This matches the first word in the sentence to be parsed, so this is a match. The parser will then go back to rule (ii) and try to find rewritings of the *N* in $T+N$. In a similar manner it will find that *N* can be the word *man* and there is once again a match. Since all of the right side of rule (ii) now is matched, the NP in rule (i) is matched and the parser continues to try to find a match for the remaining part, VP.

The bottom-up parser goes the other way. It starts with the first word in the sentence and tries to match the rewriting rules backwards from the right side to the left. In that process it will group smaller parts into larger units.

With our example sentence it starts with the word *the* and looks for a rule where a rewriting can be a *the*. It finds rule (iv) where $T \rightarrow$ *the*. The parser will search for a rule that can be rewritten as T, but will not find any. Continuing with the word *man* it finds that it can be an N. Now it has $T + N$, and will find rule (ii) where they can be collapsed to NP. Eventually it will have parsed the full sentence and arrive at the final reducing of $NP + VP$ to $Sentence$ in rule (i).

In both cases with this very small grammar the parser will find a match at once, but in a real world situation there can be a large amount of rules, and several rules that have the same left part of the rule, that is alternative rewritings of the same expression. In that case the parser will have to try many rules and backtrack on failure. If the full sentence is parsed successfully, the sentence is said to be part of the language described by the grammar, and the parser will have generated an analysis.

The analysis can also be illustrated graphically as in Figure 2.3 and is in that case for obvious reasons called a *parse tree*. It differs from the derivation in Figure 2.2 in that a parse tree doesn't show the order in which the rules are applied. The same tree could have been the result of top-down and bottom-up parsing or yet another parsing method.

A grammar is also intended to be used for generating all possible sentences of the language. For instance, in addition to our example sentence the grammar in Figure 2.1 can generate, among others, *the man hit the man*, *the ball hit the ball* and *the ball hit the man*.

This report presents only the basic idea of the phrase-structure grammar with a simple example. To make it a useful theory it has to be extended in several ways. Among other things the rules must be adjusted so that the different parts of a sentence match regarding gender, number and time. Another problem is to handle the fact that the complexity of a grammar can grow immensely if every possible rewriting is covered.

As can be seen in the grammar, a symbol on the left can always be rewritten irrespective of where it occurs in a rewriting rule. For that reason this particular kind of grammar is called a context-free grammar. There are also other kinds, for instance the context-sensitive grammar that contains rules like

$$a + NP + b \rightarrow a + T + N + b$$

meaning that in this case a NP can be rewritten as $T + N$ only if it is surrounded by the symbols a and b.

2.4 Ambiguity

(i)	$Sentence \rightarrow NP + VP$
(ii)	$NP \rightarrow T + N$
(iii)	$NP \rightarrow T + N + PP$
(iv)	$VP \rightarrow V + NP$
(v)	$VP \rightarrow V + NP + PP$
(vi)	$PP \rightarrow P + NP$
(vii)	$T \rightarrow$ the
(viii)	$N \rightarrow$ man, ball, etc
(ix)	$V \rightarrow$ hit, took, etc
(x)	$P \rightarrow$ with, of, etc

Figure 2.4: An extended simple grammar.

One of the hardest problems in parsing is how to handle ambiguity. In Figure 2.4 our simple grammar has been slightly extended to contain rules for prepositional phrases *PP* and prepositions *P*. With this grammar we can construct the sentence “The man hit the ball with a bat”. This can be parsed as the analysis in Figure 2.5 and also as the analysis in Figure 2.6. While most humans wouldn’t even consider any interpretation other than Figure 2.5 both analyses are correct according to the grammar. Similarly if we exchange “bat” for “dot” and get “the man hit the ball with a dot”, the analysis in Figure 2.6 seems natural to most people.

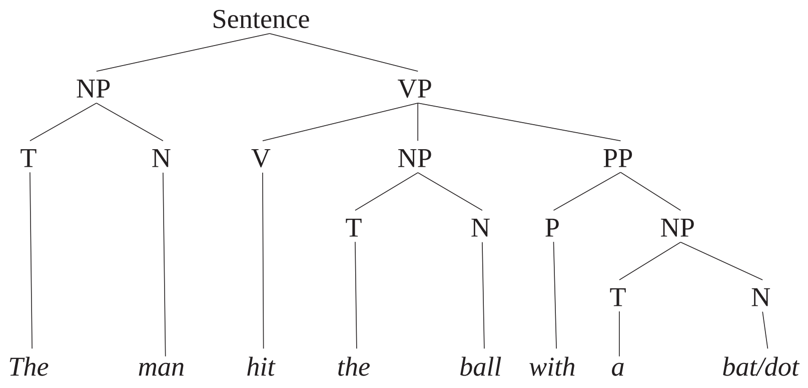

Figure 2.5: Tree structure of *The man hit the ball with a bat*.

One way to handle disambiguation is to use probability. In probabilistic context-free grammar, PCFG, each rule is assigned a probability (Charniak, 1993). Prob-

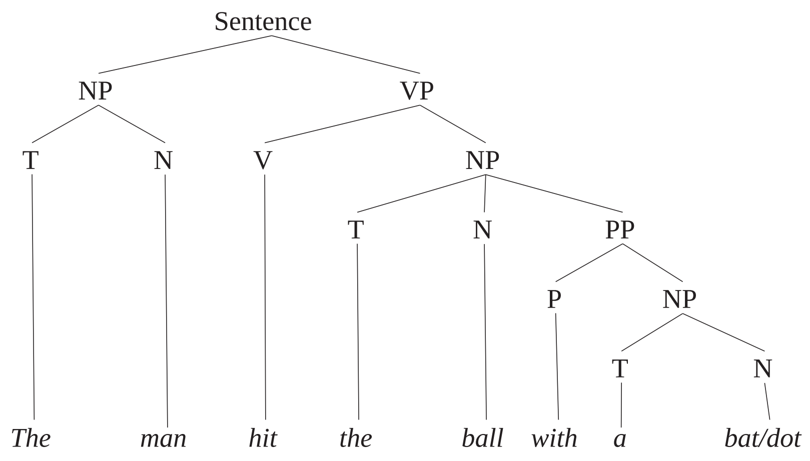

Figure 2.6: Tree structure of *The man hit the ball with a dot*.

abilities are collected from a treebank. There are different ways to do this, but one simple example is this:

1. Parse the text and follow the annotations to construct analyses for each sentence.
2. Count the total number of times each rule is used in all the analyses.
3. For every group of rules that has the same left side (e.g. $NP \rightarrow \ldots$): Sum the total and give each rule a probability that corresponds to its relative proportion of the sum.

Parsing is the performed in the same way as described above, with the addition that each time a rule is applied it is counted. The probability for a particular parse tree is then the joint probability of each rule used in the tree, where each rule is counted as many times at it occurs. If a parse results in more than one possible analysis, the most probable analysis is selected.

While being an improvement to the non-probabilistic context-free grammar in some areas, PCFG has some weaknesses. By its nature it is based on phrase structure statistics and does not account for the lexical content, which could be a better guide for disambiguation.

2.5 Dependency grammar

Historically there are many names from different times associated with a the concept of a dependency grammar. There has been a diversity of ideas about what

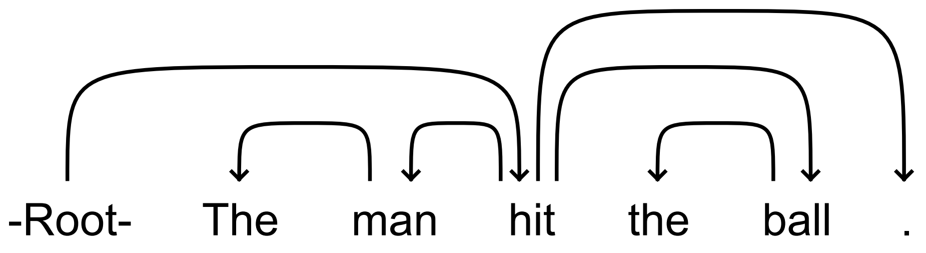

Figure 2.7: Dependency graph for *The man hit the ball*.

comprises the grammar. Nevertheless there are a few fundamental ideas which are mostly agreed on.

The structure of a sentence consists of elements and their relations. The only elements are the terminal words. The relations between the words are asymmetrical connections, arcs, where one edge is called *head* and the other *dependent* or *modifier*. With one exception, every word has exactly one head and can have zero or more dependents. One word in the sentence has no head, but a special relation called *Root*. This is the main word of the sentence.

A typical graphical representation is illustrated in Figure 2.7. Analogous to the phrase-structure parse tree this is called a *dependency graph* or sometimes *dependency tree* and is equivalent to an analysis of the sentence.

There are four properties which are required of a dependency graph:

Single head. Every node shall have exactly one relation to a head.

Acyclic. The graph must be acyclic. That is, the head for a node can not be depending on that node directly or indirectly. Put in another way, if we start at any node and traverse the relations from dependent to head it shall not be possible to arrive at any node a second time. In graph theory this is known as a directed acyclic graph, DAG.

Connected. The graph must be connected. This means that all nodes except one must have a connection to a head.

Projective. A graph is projective if no connections are crossed. This means that for any two words that are connected in a sentence, all words between them must be connected to each other or any of the two words. There is some disagreement over this constraint, partly because a various percent of sentences in a language must be described by a graph that is non-projective. This is dealt with in different ways as described later, and generally the requirement for a graph to be projective is accepted.

2.6 Parsing dependency grammars

Parsing a text using a dependency grammar is usually a more straightforward procedure than the corresponding ways to parse using a phrase-structure grammar. This is because there is no need to build an intermediate formal structure but simply find a connection for each word. Parsing can be done in different ways, but similar to phrase-structure parsing we can discern two approaches, attacking the problem from different sides.

1. Parsing the words in a sentence one by one and try to add a connection with one or more of the previously parsed words until a dependency graph is built.

2. Start by making every possible connection between all words, and then remove them one by one until a dependency graph remains.

Just as in the case of phrase-structure parsing there is the problem of selecting the best among many possible analyses, and the solution is the same: using hand crafted rules or statistics or a mixture of both.

2.7 Why dependency parsing?

According to Covington (2001) constituent grammar appears to have been invented only once, by the ancient Stoics, and has been passed through formal logic to linguists of modern times. On the other hand, dependency grammar seems to have been invented many times in many places (Covington, 2001, p. 95). Nevertheless the constituent based view has for the major part of the previous century overshadowed every other view of syntactic representation.

Mel'čuk makes an argument that this can be partially explained by the fact that the phrase-structure view is particularly suitable for English, and this is the mother tongue of the founding fathers (Mel'čuk, 1988, p. 4).

Furthermore Mel'čuk summarizes a few reasons why the dependency model is preferrable:

1. A phrase-structure tree focus on grouping of the words, which words go together in the sentence, but does not give a representation of the relations between the words.

2. A dependency tree is based on relations. It shows which words are related and in what way. The sentence is "built out of words, linked by dependencies". The relations could be described in more detail by giving them meaningful labels.

3. A dependency tree also represents grouping. A phrase is represented by a word and its entire sub-tree of dependents.

4. In a phrase-structure tree usually most nodes are nonterminal, representing intermediate groupings. A dependency tree consists of only terminal nodes. There is no need for abstract representation of grouping.

5. In a phrase-structure tree the linear order of the nodes is relevant. It must be kept to retain the meaning of the sentence. In a dependency tree this is not important. All information is preserved in the, possibly labeled, connections.

2.8 Grammar-griven vs. data-driven parsing

We have described previously that parsing can be driven by a grammar, but this method can be extended. For instance, in probabilistic context-free grammar the disambiguation is resolved by using statistics from a treebank. The use of treebanks and other large corpora can be drawn much further as to become the main guide for the parser. In this way the rules can be said to be "discovered" by analyzing the annotated text. These two categories for performing parsing are usually called grammar-driven parsing and data-driven parsing respectively. These are the extremes. Many parsing methods use a mix of the two.

2.9 Machine learning

For data-driven parsing, the "discovery" part is made using methods in the field of machine learning. This is a large field, and only a brief description will be made here since it will be referred to later.

Machine learning could be described as "any computer program that improves its performance through experience" (Mitchell, 1997, p. 2). To generate experience the program is usually constructed to make a *hypothesis* about the solution to a problem, and is given the possibility to compare that hypothesis to the expected, correct, one. This type of learning is called *supervised learning*, because the program can supervise its progress.

A common use of this method, and the one of current interest in this report, is when the hypothesis is a classification of the sample. In this case the sample should be classified from a finite set of classifications. By comparing the hypothesis with the correct classification in the training sample the program can adjust parameters to improve its performance. This is called the *training phase*.

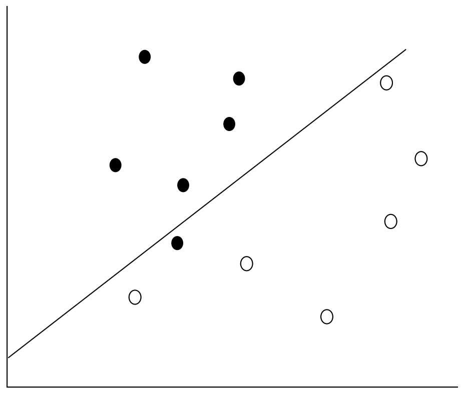

Figure 2.8: Hypothesis for classification as a straight line.

When the degree of classification is considered good enough, this method can be used to classify unknown samples.

A simple instructive example is visualized in Figure 2.8. The figure shows a set of 12 sample dots in a coordinate system. The relation between the color of a dot and its position is made according to a system which is unknown to us. The collection of 12 dots is the training set. The learning algorithm is presented the training data in the form of a list containing the data $\{x, y, color\}$ for each of the 12 dots. The training task is to learn to classify new dots given only the coordinates x and y of the dot.

Since we don't know the system for coloring the dots, we have to make an attempt to approximate it using some assumption. This assumption is crucial to the success of the task. If we suspected that the colors were assigned at random we could do no better than to guess with a 50% chance to be correct. The only thing we know is that there are equally many black and white dots in the training set, and that seems to be the distribution. The assumption thus limits how we can express our hypothesis for the classification. The collection of all possible hypotheses that are possible to express based on such an assumption is called the *hypothesis space*.

In Figure 2.8 is shown the assumption that the dots can be classified depending on whether they are positioned above or below a straight line. The possible hypotheses can then be expressed as different values for the slope and y-intercept of the line. During training the algorithm will use the data from the train set to adjust these values until all or a sufficiently large part of the train set are classified correctly.

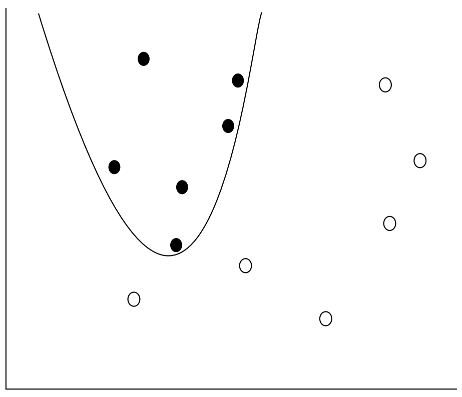

Figure 2.9: Hypothesis for classification as a curve.

As we can observe by looking at Figure 2.8 it will not be possible to make a perfect classification of the samples using a straight line. To achieve this we will need to be able to make a more expressive hypothesis, for instance a combination of lines or a curve. Depending on the nature of the problem we try to solve we can either change our assumption to get better train results or accept the results from using the straight line. Perhaps we know that the training data may contain a certain degree of errors.

Figure 2.9 shows the same training set where the hypothesis is a curve. In this case the curve has been trimmed to fit the area of the black dots very well. Without any other information there is no way to know if this is a very good approximation of the unknown coloring system or not. To get an indication how good approximation a hypothesis is the usual method is to set aside part of the training set to a *development set*. During training optimal hypothesis is searched for and then the hypothesis is applied to the development set. The expected behavior is that as the results for the training set improves the results for the development set will follow but be a bit lower. At a certain point the results for the development set will decrease while the training set will still increase. At that point the training hypotheses will be optimized for the particular instances in the training set and cease to be generally useful. This phenomenon is called *overfitting*.

As illustrated above a more expressive hypothesis space is more inclined to overfitting. We could consider an even more expressive hypothesis space where the hypothesis was a collection of coordinates and color for a position. Such hypotheses would be able to adapt perfectly to any training set, but would be useless for anything else since it would not give us any clue of how to classify a new dot. The hypotheses would contain exactly the same information as we already have in the training set. The strength of a good hypothesis space is the ability to learn from specific instances and make generalizations that apply well to unknown instances.

Chapter 3

Previous work

In this chapter we briefly present a few well-known dependency parsers. Together they serve as examples of the variety of approaches that is applied to the parsing problem.

3.1 MiniPar

MiniPar (Lin, 1998) uses a hand-crafted grammar, represented as a network with grammatical categories as nodes and types of dependency relations as links. Grammatical rules are implemented as constraints associated with nodes and links. The parser contains a huge lexicon (approximately 130000 entries), where each entry contains all possible part of speech of the word. This is a way to handle lexical ambiguities.

During parsing all possible analyses are generated and the one with the highest ranking is selected. While the grammar is constructed manually, the ranking is based on statistics acquired from parsing a very large (1GB) corpus.

3.2 Covington

Covingtons parser (Covington, 2001) presents some strategies to improve a brute-force search of head-dependency pairs. For instance, one strategy is to set aside all words that already are dependents and not consider those when searching for dependents of a word.

For this purpose the parser maintains two list during parsing, *WordList* and *HeadList*. The *WordList* contains all words parsed so far, and the *HeadList* contains all of those words which lack a head. For both lists words are inserted at the front, meaning that when reading the list the most recent inserted words are encountered first.

The parser reads words one by one from the beginning of the sentence. For each word W the following is done:

(i) Add the word to the front of the *WordList*. Search the rest of *WordList* for a word that can be the head of W. If such a word is found, add a dependency arc from that word to W, otherwise add W to the front of *HeadList*.

(ii) Look through the *HeadList* for an words that can have W as head. For every such word, add a dependency arc from W to that word, and remove that word from *HeadList*.

The efficiency of this method is then further improved by adding limitations enforced by the requirement of projectivity.

For item (i) the search for a possible head for W ends when an unbounded word is encountered. This is because all words between W and its head must have arcs between W and its head. Otherwise the arcs would cross.

For item (ii) the search for a dependent to W is ended as soon as a word is found that is not dependent of W. The reason for this is the same as for item (i).

3.3 MSTParser

It is probably safe to say that the majority of the most successful parsers or parsing methods used for dependency parsing today are based on one of two state-of-the-art parsing algorithms. One is the main topic of this study, Nivre's Parser, and the other is McDonalds MSTParser (McDonald, 2006).

The approach of the MST parser is to view the problem of finding the right dependency graph for a sentence as the problem of finding the *maximum spanning tree* for the graph.

The initial stage is a graph with words of a sentence as nodes, and dependency connections in both directions from every node to every other. Each connection has been assigned a score. By removing connections until we have a connected graph with no cycles we get a spanning tree. The sum of the scores for each connection in the score of the spanning tree. There are many possible spanning trees for each sentence, and the one with the hightest score is the maximum spanning tree.

The problem with this method is of course to find the tree with the highest score. If we simply select the connection with the highest score coming to each node it could result in a spanning tree (which then would be the maximum), but it could also result in a graph containing cycles.

MST uses the Chu-Liu-Edmonds algorithm for finding the maximum spanning tree. It will not be decribed in detail here. In short, it works as described above by first greedily selecting the incoming connection with the highest score for each node. If there is a cycle it will be contracted into a new single node, and in the new node the connection to remove while keeping the best score will be found. This could possibly be a recursive process. The end result will be the maximum spanning tree.

Chapter 4

Nivre's Parser

4.1 Description

Nivre's Parser (Nivre, 2003) is an algorithm for extracting the dependency graph for a sentence. The algorithm uses a stack and an input string. Initially the input string consists of tokens representing the words, delimiters and punctuations of the sentence. The algorithm then performs a series of well-defined operations on the first token in the input string and the top token of the stack, if any. These operations are: moving the first input token to the top of the stack, removing the token on top of the stack and creating a dependency arc between the first input token and the token on top of the stack. In this way the dependency graph is created.

More formally, the parser configuration is described as a triple $<S,I,A>$ where S is a stack, I a string of input tokens and A the set of arcs between the tokens. The initial state of the configuration is $<nil,W,\emptyset>$ meaning an empty stack, an input string of tokens, W, and an empty set of arcs. The parsing process consists of a series of four possible transitions, described below, and the parsing terminates when the configuration state is $<S,nil,A>$.

The transitions involve possible operations on the token on top of the stack and the next token in the input string. Each transition is equivalent to a change of configuration state, thus transition t_i is equivalent to the change from configuration c_{i-1} to c_i.
The transitions are:

Left-Arc. Adds an arc $i \leftarrow j$ between the token on top of the stack, i, and the next input token, j. The token on top of the stack is then popped.

This transition is possible only if the stack is not empty and the set of arcs does not contain an arc making i dependent of another token.

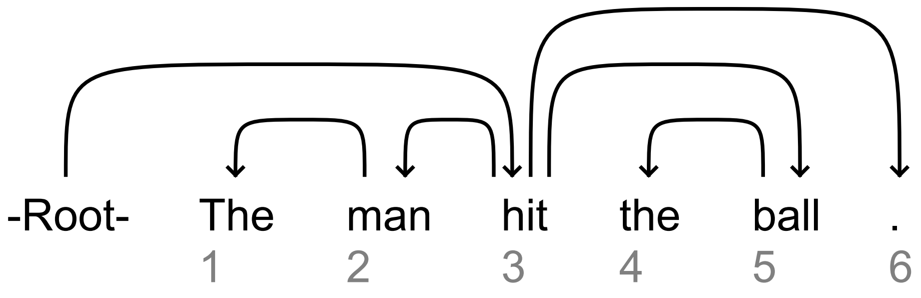

Figure 4.1: Dependency graph for *The man hit the ball*.

Right-Arc. Adds an arc $i \rightarrow j$ between the token on top of the stack, i, and the next input token, j. The token j is then removed from the input string and pushed onto the stack.

This transition is possible only if the stack is not empty.

Reduce. Pops the token on top of the stack, i. This transition is possible only if the stack is not empty and the set of arcs contains an arc making i dependent of another token.

Shift. Removes the next input token, j, from the input string and pushes it on the stack. This transition is always possible providing the input string is not empty.

The transitions Left-Arc and Reduce reduces the size of the stack, and the transitions Right-Arc and Shift reduces the size of the input string. Because of the conditions for the different transitions the termination state will always be reached, and it can be easily proven that the algorithm terminates after at most $2n-1$ transitions (Nivre, 2005, p. 79), where n is the number of tokens in the input string.

As an example, the graph in Figure 4.1 is created using the transitions in Table 4.1. Each word and punctuation in the sentence is represented by a node numbered from 1 and upwards. The first row in Table 4.1 shows the initial configuration, an empty stack and an input string containing the nodes 1-6. The following rows shows the sequence of transitions, with the operations abbreviated as LA, RA, RA, and SH, and the arc created, if any. For visibility reasons the stack is shown right to left, with the top token on the right, and the input string is shown left to right, with the first token to the left.

The first operation is inevitably Shift, since the stack is initially empty. Now the stack contains the first node and the rest are in the input string.

In the second step a LeftArc is performed, so the top stack node is popped and the created arc $1 \leftarrow 2$ is registered. Now the stack is empty again, so another Shift

	Operation	Stack	Input	Created arc
		()	(1, 2, 3, 4, 5, 6)	
1	SH	(1)	(2, 3, 4, 5, 6)	
2	LA	()	(2, 3, 4, 5, 6)	(1←2)
3	SH	(2)	(3, 4, 5, 6)	
4	LA	()	(3, 4, 5, 6)	(2←3)
5	SH	(3)	(4, 5, 6)	
6	SH	(3, 4)	(5, 6)	
7	LA	(3)	(5, 6)	(4←5)
8	RA	(3, 5)	(6)	(3→5)
9	RE	(3)	(6)	
10	RA	(3, 6)	()	(3→6)

Table 4.1: Sequence of transitions to create the graph in Figure 4.1.

is performed. Following the sequence we see how the input string is gradually shortened and that the sequence is ended when the input string is empty. The last column in the table contains the collection of arcs created.

It might be intuitively realized that the graph in Figure 4.1 can be described by the sequence of transitions in Table 4.1. That is, a sequence of transitions $T = t_1 \ldots t_n$ describe exactly one graph G.

Since the move from transition t_x to t_{x+1} corresponds to a change from configuration c_x to c_{x+1}, the sequences $c_1 \ldots c_n$ and $t_1 \ldots t_n$ are just two ways to describe the same graph G.

4.1.1 Nondeterminism

Most configurations make it possible to choose between more than one transition. In particular, since Shift is always applicable, every configuration without an empty stack allows multiple transitions. Given only this, the parsing process would be nondeterministic. To make it deterministic there need to be some unambiguous way to make the selection.

One simple way is to order the transitions with different priorities, e.g. always considering them in the order Left-Arc, Right-Arc, Reduce, Shift and select the first one that applies. By itself this rule would lead to the sequence Shift, Left-Arc, Shift, Left-Arc ... until the end of the input string is reached. For a meaningful result there need to be more rules to override it.

4.1.2 Manually improving the rule set

Studying Table 4.1, we see that in step 6 the repetition of Shift, Left-Arc is broken by another Shift operation instead of Left-Arc as otherwise would be. The tokens for which this operation is performed is word 3 and 4. In Figure 4.1 we see that these are the words ‘hit’ and ‘the’. Thus, we could have step 6 performed if we added a rule that no arc may be created between the words ‘hit’ and ‘the’. With this rule, when we come to step 6 and see ‘hit’ on the stack and ‘the’ as the next input token we must not select Left-Arc or Right-Arc. The next operation in priority order is Reduce, but that can not be performed as it requires the token on the stack to have a dependency arc. So the first (and only) operation that applies is Shift, just as we intended.

Since there would be many rules for many combinations of words, we could try to generalize by looking at the part of speech of the words instead of their lexical values. So the rule would be that no dependency arc may be created between a verb and a determiner. This could make sense, since it seems counterintuitive that a determiner and a verb should be directly dependent on each other.

Going further down Table 4.1, to step 8, we find another deviation from our method of simple priority order selection of transitions. Since the stack is not empty, and the top token has no head node, a Left-Arc would apply, but a Right-Arc is performed. Again, looking at Figure 4.1 we see that the two tokens in question are 3 and 5, ‘hit’ and ‘ball’. These two words represent the main action of the sentence, so intuitively it seems right that these two words would have a dependency. The action is ‘hit’, a verb, which is performed on the ball, a noun, so we make another generalized rule that if a dependency arc can be made between a verb and noun token, it should be an arc from the verb to the noun, i.e. the noun should be dependent of the verb.

We now have set of 3 rules:

(i) (det, noun): No arc. Apply first of RE, SH

(ii) (verb, noun): Arc from verb to noun

(iii) Apply first of LA, RA , RE, SH

With this list at hand, and using only the first applicable when parsing the sentence in Figure 4.1, we get the sequence in Table 4.2.

In this case we managed to parse most of the sentence in the right way. This is not surprising since the rules were made by analysing the correct sequence of transitions for this sentence. If we added an overriding rule for the last step, verb and punctuation, the parse would be perfect. This set of rules would probably not suffice to parse any English sentence, and the question is if it would be possible to extend this method of extracting more rules by analysing more sentences.

	Operation	Stack	Input	Created arc	Rule used
		()	(1, 2, 3, 4, 5, 6)		
1	SH	(1)	(2, 3, 4, 5, 6)		3
2	LA	()	(2, 3, 4, 5, 6)	(1←2)	3
3	SH	(2)	(3, 4, 5, 6)		3
4	LA	()	(3, 4, 5, 6)	(2←3)	2
5	SH	(3)	(4, 5, 6)		3
6	SH	(3, 4)	(5, 6)		1
7	LA	(3)	(5, 6)	(4←5)	3
8	RA	(3, 5)	(6)	(3→5)	2
9	RE	(3)	(6)		3
10	LA	()	(6)	(3←6)	3

Table 4.2: Sequence of transitions using the small rule set.

In a more general sense, we can view the selection of transition as a function taking as input the current parser configuration and perhaps some other state information and output one of the four transitions. The perfect, never failing, version of this function could be called *oracle*. Since this function most likely is unattainable we could try to approximate it with a function called *guide* with the goal to be as close to an oracle as possible. A minimal version of the algorithm in pseudo code is.

```
while more_input()
  transition = select_transition(configuration, state)
  perform_transition(transition, configuration)
```

As previously mentioned a configuration with an empty stack is the only one with a single choice of transition. The algorithm could thus be elaborated to:

```
while more_input()
  if stack_empty()
    transition = Shift
  else
    transition = select_transition(configuration, state)
  perform_transition(transition, configuration)
```

The implementation of the function `select_transition()` need some strategy to make a correct choice for every configuration and state. As long as the directions for when each transition is possible are followed this strategy will not change the fact that the parser will always generate a connected, projective graph.

4.2 Inductive dependency parsing

The example above can be viewed as a very simple example of data-driven parsing. Starting with a simple assumption, rules are added or modified to improve the match between the analysis and annotations. However, to make it a strong, real world parser it is useful to take a step back and start with some theoretical ground. A solid foundation for the algorith is presented in Nivre (2005), and most of the formal reasoning in this section is extracted from that work.

The parsing method for Nivre's Parser is called Inductive Dependency Parsing. The term *inductive* refers to the possibility to make generalized decisions from a finite set of samples. In this case methods of mapping the text $T_t = (x_1,...,x_n)$ from the language L to the right analysis, using only annotated sample text from the language.

In general, for data-driven parsing, the parser can be divided into three components:

(i) A formal model M defining permissible analyses for sentences in L.

(ii) A sample of text $T_t = (x_1,\ldots,x_n)$ from L, with or without the correct analyses $A_t = (y_1,\ldots,y_n)$.

(iii) An inductive inference scheme I defining actual analyses for the sentences of any text $T = (x_1,\ldots,x_n)$ in L, relative to M and T_t (and possibly A_t).

The model M could actually be a grammar. In that case the analyses are constrained by a formal grammar. This is the case for PCFG which only will generate analyses according to the rules of a context-free grammar. But in data-driven parsing a formal grammar is not needed. In our case the model should be any system that guarantees that the result is a valid dependency graph compliant with the constraints single head, acyclic, connected and possibly projective.

The sample text is usually called training data or training corpus. It is usually taken from a treebank, which also contains annotations. The program is trained from a sample of the treebank and is then used to classify unannotated text.

Regardless of how the inductive inference scheme is implemented it must be able to make a selection among several possible analyses by assigning each a probability score. Thus, the training phase must consist of a method to trim some set of parameters which then will be used to assign the score to unseen text.

More formally the inductive inference scheme can be considered to consist of three main elements:

(i) A parameterized *stochastic model* M_Θ , assigning a score, $S(x,y)$ to each analysis y of sentence x, relative to a set of parameters Θ.

(ii) A *parsing method*, computing the best analysis y of x according to S (and an instantiation of Θ).

(iii) A *learning method*. A method for instantiating Θ based on inductive inference from the training sample T_t.

A common kind of parameterized stochastic model is the history-based model. In such a model there is a mapping of each pair (x,y) of an input string x, and an analysis y to a sequence of decisions, $D = (d_1, d_2..d_n)$. This sequence uniquely defines the analysis, and each decision has a probability. The joint probability is then by the chain rule:

$$P(y \mid x) = P(d_1, \ldots, d_n \mid x) = \prod_{i=1}^{n} P(d_i \mid d_1 \ldots, d_{i-1,x})$$

Each decision has a probability conditioned on the decisions up to that point in the sequence. This conditioning context, $d_1 \ldots d_{i-1}$, is called the history. Usually histories are then grouped into a set of equivalence classes by a function Φ:

$$P(y \mid x) = P(d_1, \ldots d_n \mid x) = \prod_{i=1}^{n} P(d_i \mid \Phi(d_1 \ldots d_{i-1}, x))$$

For Nivre's Parser the decision sequence correspond to the transition sequence $C_{0,n} = (c_0, \ldots, c_n)$ where each sequence represents one analysis which defines exactly one dependency graph G.

In the transition sequence each move from configuration c_{i-1} to c_i is represented by transition t_i, thus $c_i = t_i(c_{i-1})$.

The equation can now be expressed as:

$$P(G \mid x) = P(c_0, \ldots, c_n \mid x) = \prod_{i=1}^{n} P(t_i \mid c_0, \ldots, c_{i-1}, x)$$

As mentioned previously the sample text is usually annotated. In that case we also would like to have that knowledge about the text, i.e. the annotations A_x, available as condition variables during the training:

$$P(G \mid A_x) = P(c_0, \ldots, c_n \mid A_x) = \prod_{i=1}^{n} P(t_i \mid c_0, \ldots, c_{i-1}, A_x)$$

Just as in the general case with decision sequences above, we would like to make the set of parameters manageable by grouping them into equivalence classes. We also make the simplifying assumption that a transition is conditioned on the current configuration state only, not the full sequence. In other words

$$P(t_i \mid c_0, \ldots, c_{i-1}) = P(t_i \mid c_{i-1}).$$

Since we know that each configuration contains the current state of the stack, input string and all arcs so far, representing the partially constructed graph, this might be sufficient history anyway. Our equivalence classes will be pairs of (c, A_x) grouped by the function Φ:

$$P(G \mid A_x) = P(c_0, \ldots, c_n \mid A_x) = \prod_{i=1}^{n} P(t_i \mid \Phi(c_{i-1}, A_x))$$

In the terminology of Nivre’s Parser the pair (c, A_x) is called *parser condition*. The equivalence function Φ defines equivalence classes of parser states from these parser conditions.

The model parameters, Θ, are the conditional probabilities $P(t \mid \Phi(c, A_x))$ for each combination of possible transition and parser state.

The function `select_transition()` above would be able to use these parameters to select, from the possible transitions T_R, the best transition $t \in T_R$ for every parser state $\Phi(c, A_x)$. The ideal implementation of this function will always select the correct transition while building the graph representing the correct analysis. This is what is called the oracle function above. While this would be the best solution, we don’t know how to implement such a function. The best we can do is to approximate is as well as possible.

4.3 Gold standard parsing

We don’t know how the oracle function is constructed but we have some output from it. A gold standard is a hand annotated text. This can be considered being a sample of text analysed by an oracle function. If we have a large enough sample we can try to extract parameters from it to approximate the function.

As mentioned previously this is a common case for machine learning. Since the transitions are a discrete set and the parser states are also a discrete set, it is useful to view the problem as a classification problem, selecting one of the transitions $t \in T_R$ as classifier for each parser state, $\Phi(c, A_x)$.

We have an annotated sample text, but what we really need for training the classification is a way to extract samples of transitions for different parser states from these samples. This would represent the selections made by the oracle function. In this way we could calculate the probabilities $P(t \mid \Phi(c, A_x))$. At this point we might realize that a major part of approximating the oracle function is how to implement the equivalence function Φ. This will be addressed below.

Formally, we need to extract the set of pairs (c, t) from the sample text T_t:

$$D_t = \{(c, t) \mid oracle(c, A_x) = t, c \in C\}$$

where C is the set of configurations occuring in the training set.

We can then use different implementations/parameterizations of Φ to get training sets:

$$D_\Phi = \{(\Phi(c, A_x), t) \mid (c, t) \in D_t\}$$

Thus

(i) Derive the set D_t from the training corpus T_t.

(ii) Define the parameterization of Φ and derive the training sets D_Φ from D_t.

(iii) Induce a guide function from D_Φ using inductive learning.

Deriving the set D_t from the training corpus can be done with an algorithm quite similar to the guide function in the previous section:

```
gold_standard_parsing(W, Ax)
  configuration = <nil, W, []>
  while more_input()
    if stack_empty()
      transition = Shift
    else
      transition = oracle_select_transition(configuration, Ax)
    add_to_training_instances(transition, configuration, Ax)
    perform_transition(transition, configuration)
```

The change is that we store each pair of transition and configuration, and that the transition function is replaced by an oracle-version. This function predicts the next transition using the function `head-of()` which is part of A_x.

```
oracle_select_transition (configuration, Ax)

  // configuration = (stack | j, i | input, h)
  // j = top of stack, i = first input token
  // Ax = {head-of(),...)

  if head-of(i) = j
    transition = LEFT-ARC
   else if head-of(j) = i
    transition = RIGH-ARC
   else if is_in_stack(head-of(j)) or is_head_for_stack_member(j)
     transition = REDUCE
```

```
else
  transition = SHIFT
return transition
```

Given the information in the configuration and the function `head-of()`, which has the extracted information from the hand annotated sample text, the function can always tell the correct transition. We know that the function is called as long as there is some input left in the input string, and we also know that the function is not called when the stack is empty, thus there are always one token on top of the stack and at least one input token.

By looking at the top token on the stack, i, and the next input token, j, the function can decide that if j is the head of i then the transition should be LEFT-ARC. On the other hand if i is the head if j then the transition should be RIGHT-ARC. If neither is the case, then the function has to look in the stack to see if it contains a k which is the head of j or has j as its head. If such a k is found the transition is REDUCE. Because we know that the annotated dependency graph is connected and projective we can be sure that the nodes in the stack above k can safely be reduced without violating the requirement that the graph must be connected. In this case each node above k must be on the stack as a result of a RIGHT-ARC transition and has a dependency connection to the node immediately below.

If none of the above apply, we know that we have not yet seen the head of either i or j. They must be further back in the input string so the transition must be SHIFT.

4.4 Dependency labels

The annotations in a gold standard could contain not only information about head-dependent relations. These relations can also be labeled with the type of the dependency. The labeling is made according to systems defined by the annotators and could theoretically follow any direction since their meaning is transparent to the parser. However, usually the labels are used to depict a syntactic relation between the nodes. In light of this it can be useful to think of another, less common, name for dependent: *modifier*. This name indicates that the head is determining the behavior, and the dependent is the modifier, object or complement (Covington, 2001, p96).

Adding dependency types, the parser configuration is now `(stack, input, h, d)`.

4.5 Features

When training a program using supervised learning, it has to have information on a number of parameters from the training sample and the correct classification for each occurring sample. The task is then to, in different ways, extrapolate from the training samples and correctly classify new samples which are not in the training set. Different learning methods uses different algorithms for this, but in all cases there needs to be a collection of parameters to train from. These parameters are also known as features.

In our case the classification is the transition to be selected for a certain parser configuration, and the features are extracted from that parser configuration at that point. There are two kinds of features, static and dynamic.

Static features are features which are unchanged during the parsing process. During annotation of the sample text every word and punctuation is assigned some information. This can be lexical features such as the word itself, the word form, but also suffixes, lemmatization and normalized word forms. Annotations usually also contains a part of speech with some granuality. These features are constant and bound to every particular node during parsing.

Dynamic features, on the other hand, are features that are likely to change after each transition. These features are collected from the partially construted dependency graph with, possibly labeled, head-dependent relations. Examples of these features are the dependency label of a node, the part of speech of the head for a node or the lexikal name of the leftmost child of a node.

4.6 Pseudo-projective dependency parsing

As mentioned previously the assumption that a dependency graph always is projective is an idealized constraint. Depending on the language there is a varying percent of nonprojective constructs occurring in a typical text. This makes it impossible for dependency parsing based on the requirement of projectivity to reach the ideal 100% match since, by definition, no resulting analysis graph will be non-projective. Furthermore many treebanks are annotated with a certain amount of nonprojective sentences. This will disturb the training phase for the parser. While many machine learning algorithms are well suited to handle a certain degree of errors in the training data, others are not.

One simple way to deal with the disturbance of the training phase is to ignore every nonprojective sentence in the training data. This only solves one of the problems, though.

Another solution is pseudo-projective dependency parsing (Nivre and Nilsson, 2005). The idea is to transform a nonprojective graph to a projective before training and invert the transformation after parsing. To be able to restore a transformed graph, traces will be left in the dependency labels. For that purpose the set of labels are extended with additional labels containing information about both the original label and the transformation that has been performed.

4.7 CoNLL

SIGNLL (pronounced signal) is the Special Interest Group on Natural Language Learning of the Association for Computer Linguistics, ACL (http://www.aclweb.org).

The aim for SIGNLL is "the answering of fundamental scientific questions about the nature of the human language acquisition process and the development of practical NLL techniques for solving current problems across the full range of Computational Linguistics, whilst admitting the widest possible range of computational approaches."

Among other thing, this manifests itself in the aim to promote research in automated acquisition of syntax, morphology, phonology, semantic / ontological structure and inter-linguistic correspondences.

SIGNLL emphasizes paradigms which can be exploited automatically, such as corpus based analysis including automated tagging and testing, learning in interactive environments, unsupervised and implicitly supervised techniques etc.

CoNLL, Conference on Natural Language Learning, is SIGNLLs yearly meeting. Although there were previous meetings with other names, the first CoNLL meeting was in Madrid 1997. Starting with the 1999 meeting in Bergen, CoNLL included in the conference a shared task among the participants. The organizers provided training and test data so that the participating systems could be evaluated and compared in a systematic way. Descriptions of the systems and an evaluation their performances are presented both at the conference and in the proceedings.

The tenth meeting in New York 2006, for anniversary reasons named CoNLL-X, included the shared task Multi-lingual Dependency Parsing. The task was to assign labeled dependency structures for a range of languages by means of a fully automatic dependency parser.

One reason for the CoNLL shared task was to make it possible to compare the performance of different parser solutions on the same data and on many languages. Until a few years before, most parsers were usually tested on only one or two languages, one of them being English.

Training and test data for 13 languages was provided. This would allow participants to tune their applications by adjusting parameters for particular languages. The same applications should be used for all languages. The train data contained sample text annotated with, among other things, wordform, lemma, part of speech, head for each word and dependency label. The data format was unified but the systems for part of speech, dependency labels etc. were strictly different for each language. 2 months later new test data for the languages was released and the participants were expected to parse the data and submit the result to CoNLL. The results were then evaluated using a script which measured the score for correct head-dependent relations (unlabeled attachment score) and in addition correct dependency labels (labeled attachment score). This script was published at the same time as the initial training data.

For most languages the top two results for both labeled and unlabeled attachment score were submitted by Ryan McDonald with MSTParser and Joakim Nivre with MALT Parser, an implementation of Nivre's Parser. MALT Parser obtained the second best overall score and achieved top results for 9 languages.

Chapter 5

Method

The aim for this study is to make an investigation of the feature set, and try methods of finding an optimal set. The investigation will follow two different paths of semi intuitive search and systematic search. When searching for the optimal set, the search for a correctly connected graph is the primary goal and the search for a correctly labeled graph is secondary. The rationale for this is the idea that if we can achieve a correct graph in one pass it can be labeled in a second pass. This idea is not pursued in this study, though.

5.1 The parser implementation

The parser used in this study is a rather straight forward implementation in C++. It has been compiled and run on both Microsoft Windows and Linux. The majority of the tests were run on a computer cluster. For training and testing the library LibSVM (Chang and Lin, 2001) was used.

The parser can be run in a few modes, the most important ones being ‘train’ and ‘test’. All common parameters for these modes are supplied in a configuration file.

In train mode a train corpus file is parsed using the gold standard parsing described above, and three training files are composed, the action, left arc and right arc file. The action file consists of one of the four actions/transitions and parameters from the parser configuration which are listed in the train/test configuration file. The left and right arc files consist of the dependency label for an arc when the respective arc action is selected and also the same parameters chosen for the action file.

The parser is then trained on these three files which results in three model-files for LibSVM.

In test mode a test corpus is parsed and the actions and arc labels are predicted using the model files.

It is possible for the prediction of actions to suggest an action which could not be performed. More precisely, this includes any action other than Shift when the stack is empty. In that case the parser will perform a Shift action irrespectively on what is suggested.

Other actions are possible to perform, but would lead to the final graph not being connected. These actions are not prevented by the parser. Instead, special parameters are possible to use as hints during training and testing. This is described below.

To construct the parameters the parser keeps name tables for the possible values of part-of-speech, dependency labels and lexical tokens. The first two consist of a finite set of values as defined in the annotations of the corpus. The collection of lexical tokens is constructed from the train corpus. This collection is extended with a pseudo token, UNKNOWN, which will be used for any token appearing during testing which were not present in the train corpus and thus is not found in the name table.

The name table for dependency labels is extended with the pseudo token NOT_SET which means that there is a node as described by the parameter definition, but it is not bound by a dependency arc and thus it has not yet a dependency label. An example of this is if the stack is not empty but the top node has no head node then the part-of-speech will be known but the dependency label will be NOT_SET.

Furthermore, all three name tables are extended with the pseudo token NOTHING which will be used when a parameter value doesn't exist in the current parser configuration, e.g. if the stack is empty, the part-of-speech value of the top of the stack is NOTHING.

5.2 Parameter set

The parsers parameter format is derived in relevant parts from the format for MaltParser 0.4 and extensions have been added.

The basis are the feature type, data structure and indexes. These are used to indicate a particular feature of a certain node in the parser configuration.

Feature types are POS (part-of-speech), LEX (the lexical value) and DEP (dependency label). There are two data structures STACK and INPUT. A data structure combined with a zero based index is a starting point for finding a node. Thus STACK 1 indicates the second node on the stack and INPUT 0 indicates the first input token.

Then up to four integers indicating are allowed:

- Relative offset, positive or negative, in the original string.

- Positive offset of the HEAD function. Thus 2 means head of head.
- Offset of the leftmost child function (negative value) or rightmost child function (positive value).
- Offset of the left sibling (negative value) or right sibling function (positive value).

The default value for these parameters is 0 and if they are not used they can be omitted.

This far the parameter set is compatible with the MaltParser 0.4 parameter set.

The primary format for the current parser is another, called navigational format, which uses a node as starting point and then stepwise instructions of how to find a certain node.

This parameter format use the same first three parts, feature type, data structure and data structure index as described above. This could be the complete parameter and if so it indicates a node present in either the stack or input string. The parameter can then contain a list of a number of navigational steps from this node. These steps described how to move from the current node to the next. Possible steps are:

- **h** Move to the *head* of the current node.
- **lc** Move to the *leftmost child* of current node.
- **rc** Move to the *rightmost child* of the current node.
- **ls** Move to the *left sibling* of the current node.
- **rs** Move to the *right sibling* of the current node.
- **pw** Move to the node representing the *previous word* in the original string.
- **fw** Move to the node representing the *following word* in the original string.

If at any step the requested node doesn't exist, the parameter description evaluates to NOTHING. This is the same as for the nodes in the data structures.

An example of a parameter is:

POS STACK 1 lc

This interpreted as: find the second node of the stack. Move to the left child of that node. Get the part-of-speech of that node.

A longer parameter is:

POS STACK 1 ls rc pw h

The easiest way to read this in common English is to read the feature and then the rest of the line right to left, like so:

Get the part-of-speech
of the head
of the previous word
of the right child
of the left sibling
of the second node
of the stack.

In this way any node can be pinpointed if it exists. If the indicated node doesn't exist, the value of the feature of the parameter is NOTHING.

5.3 Investigations of the feature set

The investigation of the feature set for the parsing algorithm was done in two ways. First a study of different combinations of parameters was made and then a systematic search method was constructed and tested.

The corpus used for the first study and part of the second study is the Swedish corpus used in CoNLL-X 2006, "conll06_data_swedish_talbanken05_train_v1.1". It consists of 11431 sentences, with 20057 unique lexical tokens. The annotations uses 41 part-of-speech labels, POS, and 64 dependency labels, DEP labels.

The evaluation of each result was made with a script, eval.pl, which was published as part of CoNLL-X. This script generates three result values and some diagnostic information. For this study the two interesting results are:

Labeled attachment score The proportion of tokens that are assigned both the correct head and the correct dependency label.

Unabeled attachment score The proportion of tokens that are assigned the correct head.

Chapter 6

Matrix study and intuitive search

In the first part of the study tests were made with different combinations of features. For each combination a matrix of up to 8 stack tokens and up to 8 input tokens were tested. Sentences were parsed in both Left-Right and Right-Left directions. Evaluations were made for both labeled and unlabeled graphs.

As an example, a table of the results for the simplest combination, the part-of-speech for each node, is shown. The columns shows the number of input tokens and the rows shows the number of tokens on the stack in each of the 64 tests.

Results for attribute POS, parsed in Left-Right direction are shown in Tables 6.1 and 6.2.

Stack\Input	1	2	3	4	5	6	7	8
1	58.43	61.06	61.20	61.44	61.68	61.86	62.14	62.02
2	63.41	67.02	67.44	67.42	67.74	67.56	67.64	67.78
3	63.31	67.12	67.56	67.66	**67.87**	67.87	67.68	67.68
4	63.35	67.30	67.82	67.60	67.74	67.54	67.72	67.10
5	63.59	67.36	67.68	67.70	67.64	67.62	67.44	67.34
6	63.67	67.22	67.84	67.52	67.54	67.42	67.52	67.18
7	63.57	67.26	67.76	67.36	67.44	67.68	67.48	67.40
8	63.63	67.26	67.58	67.28	67.56	67.48	67.50	67.42

Table 6.1: Attribute POS. Labeled attachment score.

The top score for each matrix is indicated by the value in bold. When values are equal, the combination requiring fewer number of parameters are selected.

For the labeled attachment score the best score is 67.87% correct arcs with correct labeling. This result is achieved for the training using the POS values for the first 5 nodes in the input string and the top 3 nodes on the stack, a total of 8 parameters. As described previously some of these nodes may be missing at some

Stack\Input	1	2	3	4	5	6	7	8
1	68.29	71.20	71.42	71.46	71.70	71.96	72.20	71.98
2	76.24	79.69	80.04	79.84	79.96	79.94	79.98	80.00
3	76.36	80.00	80.14	80.28	80.42	80.40	79.96	79.67
4	76.28	80.18	80.52	80.30	80.36	79.96	79.92	79.37
5	76.42	80.14	80.46	80.28	80.44	80.02	80.00	79.47
6	76.46	80.24	80.54	80.24	80.36	79.94	79.98	79.47
7	76.50	80.20	**80.58**	80.12	80.28	80.04	79.76	79.61
8	76.76	80.06	80.42	79.92	80.32	79.82	79.82	79.67

Table 6.2: Attribute POS. Unabeled attachment score.

stage during parsing, when the stack is too shallow or the remaining input string is shorter than 5 nodes. In this case the missing node will be represented by the pseudo value NOTHING. Apart from being the only value that can be returned in this case it also indirectly reveals some information. For instance if a value for input token 3 is NOTHING we are dealing with the two last two words in a sentence (since the last token is the punctuation).

The unlabeled attachment score has a maximum value, 80.58% for the first 3 nodes in the input string and the top 7 nodes on the stack.

6.1 Parser hints

In the section about the parser implementation it was described that when the stack is empty, the parser will override any other action than Shift, and perform a Shift. This is simply because that is the only action that can be performed. Other actions are not inhibited even if they result in the graph being not connected. To reduce this effect the parameters supplied during training data can be extended with boolean flags indicating whether the actions can be legally performed. It turns out that usually it is sufficient to extend the parameter set with a single flag to achieve this. The condition is that the parameter set contains a parameter which indicates if the stack is empty or not. Such a parameter is either of POS or LEX of the top of the stack, with the pseudo token NOTHING indicating an empty stack. The overview in Table 6.3 indicates this. The stack has three modes, top node has head, top node has not head and stack is empty. It is assumed that there is always an input node. The possibility to perform any of the four actions is directly related to the three stack modes.

To the right is a column indicating the Boolean value for “stack is empty”. This value in combination with either of canLeftArc or canReduce defines the

stack	canLA	canRA	canS	canR	stackEmpty
head	F	T	T	T	F
no head	T	T	T	F	F
empty	F	F	T	F	T

Table 6.3: Overview of how the three modes of the stack affects the possibility to perform the four transitions.

other three values.

canLA = not stackEmpty and not canR

canRA = not stackEmpty

canS = true

A new set of test was made with the same parameters as before, the part-of-speech for each node, this time extended with the boolean parameter canReduce. Table 6.4 contains the result for the unlabeled attachment score (UAS). For labeled attachment score the results are similar.

Stack\Input	1	2	3	4	5	6	7	8
1	71.74	79.03	79.43	80.00	80.22	80.50	80.32	80.44
2	74.21	80.96	81.52	82.02	82.31	82.21	82.27	82.15
3	74.39	81.52	82.15	82.53	82.63	82.41	82.37	82.14
4	74.37	81.72	82.27	82.51	82.79	82.10	82.19	81.96
5	74.45	81.66	82.23	82.53	82.81	82.45	82.19	82.00
6	74.45	81.70	82.06	82.49	82.63	82.49	82.33	82.15
7	74.59	81.68	82.06	82.47	82.51	82.49	82.29	82.51
8	74.57	81.60	82.17	82.35	82.85	**82.89**	82.33	82.49

Table 6.4: Attribute POS with CanReduce flag. UAS.

There is an overall improvement for every combination. The score for the previous best combination has increased from 80.58% to 82.06% and there is a new best score of 82.89% for the combination 6 input and 8 stack nodes. Previously the score for this combination was 79.82%.

This parameter obviously makes a big difference. Even though the impact is expected to be less dramatic when the information is overlapped by other parameters, for instance using any parameter referring to the head of STACK 0, it was decided to keep this parameter in all future parameter sets.

6.2 Right to left parsing

While a sentence is usually parsed in the reading direction of the language, it is interesting to compare the parser performance if the sentence is scanned in the opposite direction. Table 6.5 shows the result for of POS when parsed right to left, and the difference between right-left and left-right parsing is shown in Table 6.6.

Stack\Input	1	2	3	4	5	6	7	8
1	68.67	76.66	80.32	80.52	80.62	80.78	81.10	80.90
2	68.27	74.89	81.44	82.37	82.87	82.23	82.27	82.57
3	68.47	75.16	81.66	82.23	**82.75**	82.08	82.02	82.53
4	68.51	75.18	81.52	82.49	82.12	82.15	82.33	81.94
5	68.45	75.16	81.62	82.49	82.49	82.06	82.23	82.00
6	68.41	74.97	81.46	82.47	82.10	82.19	82.14	81.78
7	68.75	75.22	81.44	82.45	82.25	82.15	82.04	81.82
8	68.51	75.26	81.36	82.35	82.12	82.00	82.04	81.70

Table 6.5: POS Right-Left. UAS

Stack\Input	1	2	3	4	5	6	7	8
1	-3.07	-2.37	0.89	0.52	0.40	0.28	0.78	0.46
2	-5.94	-6.07	-0.08	0.35	0.56	0.02	0.00	0.42
3	-5.92	-6.36	-0.49	-0.30	0.12	-0.33	-0.35	0.39
4	-5.86	-6.54	-0.75	-0.02	-0.67	0.05	0.14	-0.02
5	-6.00	-6.50	-0.61	-0.04	-0.32	-0.39	0.04	0.00
6	-6.04	-6.73	-0.60	-0.02	-0.53	-0.30	-0.19	-0.37
7	-5.84	-6.46	-0.62	-0.02	-0.26	-0.34	-0.25	-0.69
8	-6.06	-6.34	-0.81	0.00	-0.73	-0.89	-0.29	-0.79

Table 6.6: Attribute POS. Difference between Right-Left and Left-Right parsing. UAS

It can be seen that, at least for this parameter setting, the performance is slightly worse in most cases and quite a bit worse when large stack depths and 1-2 input tokens are considered. The performance for labeled attachment score is similar.

6.3 Combinations

A series of combinations of parameters were tested. In the following the results for unlabeled assignment score, left to right parsing is presented. If not indicated otherwise the results for the labeled assignment score has a similar pattern but lower overall score. Furthermore the right to left parsing typically has results 2-3% lower in both cases.

The result for parsing based only on lexical tokens, parameter LEX, is shown in Table 6.7. As expected the performance is not as good as when parsing only with part-of-speech. Since there is a great possibility for unknown words to show up in the test corpus the parser will not have been trained for those and will simply have to deal with them as UNKNOWN. However, a good training corpus should contain most of the basic building stones of a language, common prepositions, determiners, pronouns etc, so the parser will have a fair chance to recognize common language constructs.

Stack\Input	1	2	3	4	5	6	7	8
1	54.57	61.86	66.52	66.92	68.43	68.87	69.55	69.33
2	55.85	64.89	68.51	69.77	70.30	69.79	70.17	69.83
3	56.50	66.42	69.85	70.94	70.66	70.42	70.64	70.23
4	57.48	67.58	70.74	71.36	71.04	70.76	70.66	70.60
5	58.39	68.59	71.74	71.62	71.42	70.82	70.72	70.46
6	59.47	69.51	71.92	72.08	71.28	70.86	70.68	70.15
7	60.49	69.93	72.04	72.08	71.14	70.94	70.50	69.99
8	61.44	70.19	**72.44**	72.20	71.30	71.28	70.74	70.11

Table 6.7: LEX Left-Right. UAS

A combination of POS and LEX parameters was tested. It was expected that the performance of the POS parameter should be improved with the guidance from lexical values, while the unknown lexical values would not have a negative influence. The results are shown in Table 6.8.

The combination of POS and DEP is not shown. The impact is very small and often negative. The reason may be that with this parameter setting the only nodes that have a dependency arc are some of the nodes in the stack and in that case the head of the dependency is the node one level down. This means that the POS value for that node usually is among the parameters. Possibly there is a partial overlap between the dependency label and the part-of-speech for the participants in a dependency relation.

We will now continue by investigating the influence of parameters for the child of a node. Table 6.9 contains the results for tests using parameters POS + POS

Stack\Input	1	2	3	4	5	6	7	8
1	76.50	83.07	83.53	84.15	83.49	83.23	83.29	82.81
2	78.65	84.94	84.90	84.76	84.62	84.07	83.93	83.95
3	78.55	84.68	**85.04**	84.94	84.98	84.13	84.25	83.85
4	78.25	84.54	84.78	84.90	84.60	84.23	84.09	83.91
5	78.11	84.35	84.47	84.78	84.31	84.05	83.85	83.91
6	78.39	84.19	84.74	84.66	84.05	83.77	83.83	83.85
7	78.81	84.01	84.72	84.37	83.71	83.71	83.83	83.73
8	78.85	83.97	84.39	84.13	83.45	83.63	83.61	83.67

Table 6.8: POS + LEX Left-Right. UAS

of the leftmost child and in Table 6.10 the parameters used are POS + DEP of the leftmost child. This time a smaller matrix is chosen.

Stack\Input	1	2	3	4	5	6
1	74.57	83.03	84.05	84.11	84.15	84.60
2	76.92	85.18	85.70	86.26	**86.44**	86.40
3	77.24	85.32	85.82	86.60	85.88	86.24
4	77.10	84.88	85.74	85.58	85.64	85.78
5	77.45	84.80	85.36	85.80	86.34	86.00

Table 6.9: POS + POS lc. UAS

Stack\Input	1	2	3	4	5	6
1	75.08	83.37	84.29	84.84	84.72	84.68
2	77.18	85.26	86.32	86.36	**86.70**	86.58
3	77.85	84.94	86.12	86.28	86.42	85.88
4	78.19	85.24	85.86	85.94	86.56	86.06
5	77.91	85.16	86.06	85.84	85.98	85.70

Table 6.10: POS + DEP lc. UAS

In both cases the additional parameter from the child has a positive impact. The values are increased by on average 3.58 for POS of the leftmost child and 3.86 for DEP of the leftmost child. The corresponding values for the labeled assignment score are 5.04 and 5.25, respectively.

We combine the three values and investigate the parameter set POS + POS of leftmost child + DEP of leftmost child.

Stack\Input	1	2	3	4	5	6
1	75.30	83.43	84.17	84.74	84.35	84.62
2	77.75	85.02	86.70	**86.91**	86.91	86.50
3	78.79	85.58	86.34	86.32	85.96	85.94
4	78.15	85.36	85.74	86.36	85.74	85.86
5	78.15	84.86	85.60	85.82	85.66	85.66

Table 6.11: POS + POS lc + DEP lc. UAS

The additional value of using both POS and DEP of leftmost child is very low. Compared to only POS the average increase is almost the same as for either of the child values, 3.89 for unlabeled and 5.26 for labeled assignment score. There is obviously an overlap between POS and DEP in this case.

Well over 70 different parameter combinations were tested and the best score for both labeled and unlabeled attachment was attained with parameter combination in Figure 6.1.

POS STACK *s*
LEX STACK *s*
POS STACK *s* lc
POS STACK *s* rc
POS INPUT *i*
LEX INPUT *i*
POS INPUT *i* lc
POS INPUT *i* rc

Figure 6.1: Parameters for best score. Parameters are for each stack node *s* in [0,1] and each input node *i* in [0,1,2]. Note that parameters for child nodes for input > 0 are omitted.

The results are shown in Table 6.12. The combination of parameters in Figure 6.1 applied to the 2 top stack nodes and the 3 first input nodes has the score 89.38% for unlabeled and 86.58% for labeled attachment score.

Stack\Input	1	2	3	4	5	6
1	80.88	88.29	87.93	87.63	87.65	87.09
2	82.77	88.75	**89.38**	88.77	88.41	88.01
3	82.83	88.05	88.53	88.49	88.75	88.37
4	83.59	87.51	88.13	87.85	88.19	87.63
5	82.81	87.07	87.79	87.79	87.77	87.73

Table 6.12: Score for parameters in Fig. 6.1. UAS.

Chapter 7

Systematic exploration of the feature space

It is often the case that, when results for dependency parsing is reported, the parameters are published but not how they were found. The previous chapter presented an intuitive search method guided to some degree by an attempt to analyze the results with the hope to find indications on how to find a better parameter set. It is a time consuming and tedious method and it's hard to know if one is on the right track.

Admittedly we found useful results, but can we do better?

This chapter introduces and evaluates a simple method to iteratively search systematically and automatically for the best parameter to add to a parameter set.

7.1 Axis search

The search method is based on the assumption that if a feature makes an important contribution, then one or more neighboring features could also be important. The neighbors considered are both in a feature and node sense. e.g. POS STACK 0 means the part-of-speech feature of the node on top of the stack. Neighboring features are found in different directions from this node-feature. In this report these directions are called axes and 4 of them are used.

The data structure axis. These are the nodes in the data structure, stack or input stream, where the current node possibly resides. The immediate neighbors are the nodes above or below in the stack or before and after in the input stream. The top node on the stack and the next input node have a special connection since they are the ones used when building the graph. Therefore these two are considered immediate neighbors to each other.

For a node this axis may be missing. It may have been popped from the stack.

The horizontal and vertical graph axes. These two axes traverse the partially constructed graph. The horizontal axis is the direction between sibling nodes, connected by a common parent. The immediate neighbors are the nearest sibling to the left and the nearest sibling to the right.

The vertical axis is the direction between parent and child nodes. The immediate neighbors are the parent node, the leftmost child node and the rightmost child node.

For a node this axis may be missing. It may not yet be part of the graph.

The sentence axis. This direction traverses the nodes in the order they appear in the original sentence. The input data structure overlaps by definition all or the latter part of this axis. The immediate neighbors are the previous and following node in the sentence order.

In addition to this a particular feature has neighboring features consisting of the set of remaining features belonging to the same node. This is not an axis since all features are considered neighbors of each other.

7.2 Finding neighbors

For a particular feature, the neighbors are found using this method:

(i) Find the current node, which is the node that the feature belongs to.

(ii) Find all immediate neighbor nodes, when present, to the current node along the axes.

(iii) Neighboring features are all features of the current node and neighboring nodes.

7.3 First evaluation on Swedish corpus

An evaluation of this method was made. The corpus used is the same as in the previous investigation, conll06_data_swedish_talbanken05_train, but in this case it was split into a train set and a development set. The split was made simply by picking out every 10:th sentence from the train set and put them into the development set. Training was then done on the train set and each result was evaluated on the development set. The test set conll06_data_swedish_talbanken05_test was used for reference measurement of how well the training generalized. The search should be for the best unlabeled attachment score.

It was decided that one node should be the starting point. This node should be evaluated as a single parameter and then this value should be compared with an evaluation of this parameter in combination with each of its immediate neighbors. The combinations generating the best results should be picked and then these should be evaluated in combination with each one in the collection of immediate neighbors of them, and so on.

In this report each iteration is called a *generation* where generation 1 consists of a single parameter. For each generation a new parameter is added. These parameters are called fixed parameters for the generation. The fixed parameters are the collection which is tested with each one of the neighboring parameters.

As node for the starting point the first node of the input string was selected. This is the only node which is certain to exist during parsing, simply because parsing is defined to continue as long as the input is not empty. It is also very probable that this node is important. Two tests were initiated, each one starting off from one of the two static features of the input node, POS and LEX.

7.3.1 Rules for parameter selection

A set of rules derived from the axis search was compiled on how to select the set of new parameters to add to the next generation. The rules are presented in Table 7.1. The purpose is to avoid making recursive navigational paths. One example is to notice that if we move to the left sibling of the right sibling of a node we return to the original node. Depending of the kind of node to start from some of the directions could be omitted since they were redundant. For instance if a node has been brought in to the fixed parameter set as being a leftmost child of another node in the fixed set, the head of that leftmost child-node is already in the fixed set. Similarly if the node STACK 2 is in the fixed set it has been brought in as being neighbor in the stack to STACK 1, so nodes above the current node in the stack are omitted. In this way the search for neighbors along the axes is generally reduced to only one direction depending on in what respect the node is defined in the parameter.

7.3.2 The first generation

The parameters for the first test generation were found by applying the rules as follows. The selected starting node is INPUT 0 and two tests were prepared, one for POS and one for LEX. Beginning with the feature POS INPUT 0 as the fixed

				PLD	PLD	PLD	PLD	PLD	PLD
STACK n = 0	**STACK n > 0**	**INPUT n = 0**	**INPUT n > 0**	**h**	**lc, rc**	**ls**	**rs**	**pw**	**fw**
PLD	PLD	PL	PL	h				h	h
PLD h	PLD h				lc	lc	lc	lc	lc
PLD lc	PLD lc	PLD lc			rc	rc	rc	rc	rc
PLD rc	PLD rc	PLD rc		ls	ls	ls		ls	ls
PLD ls	PLD ls			rs	rs		rs	rs	rs
PLD rs	PLD rs			pw	pw	pw	pw	pw	
PLD pw	PLD pw	PLD pw		fw	fw	fw	fw		fw
PLD fw	PLD fw								
PLD STACK n+1	PLD STACK n+1	PL INPUT n+1	PL INPUT n+1						
PL INPUT 0		PLD STACK 0							

Table 7.1: Rules for adding parameters for a new generation. For each node of type T, in bold on row 2, add the features in the column below. PLD = abbreviation for features POS LEX DEP.

parameter, we read the third row in the rules table and see that the neighboring nodes we should add are

- POS and LEX of the node. We already have POS so we add LEX INPUT 0.
- POS, LEX and DEP of the leftmost and rightmost child of the node.
- POS and LEX of the previous word in the original string. Since the input string is the same as the rest of the original string, the 'following word in the original string' is the same as the next input node, which is added in the next step. For that reason "following word" is not added.
- POS and LEX for the node INPUT n+1, in this case INPUT 1.
- POS, LEX and DEP for the STACK 0. This is the special rule that connects the stack and input nodes.

As long as a node is still in the input string it can not have an arc to a head. Therefore we don't include the dependency for input nodes as a parameter.

The parameters found using this methods are listed in Figure 7.1. Note that PLD is an abbreviation for the tree parameters POS, LEX and DEP of a particular node.

POS	INPUT 0
LEX	INPUT 0
PLD	INPUT 0 lc
PLD	INPUT 0 rc
PLD	INPUT 0 pw
PL	INPUT 1
PLD	STACK 0

Figure 7.1: Parameters for the first generation, originating from POS INPUT 0.

The parameters for the other test case LEX INPUT 0 are almost the same. We only have to switch place between LEX INPUT 0 and POS INPUT 0 in the list.

The results of the two tests are shown in Table 7.2. At the head of each table is the fixed parameter and then follows rows of the results of evaluation of unlabeled and labeled attachment score for each of the neighboring parameters. The row with a blank parameter represents the values for the fixed parameters only, and is included for comparison. We can see that for unlabelled test the best result was the combination of POS INPUT 0 and POS STACK 0, which was evaluated to 74.02% correct. A great improvement compared to the result 43.49% for only

POS INPUT 0. We can also see that for the two parameters at the end of the list the result evaluated to a decreased result compared to only POS INPUT 0.

The second test, for LEX INPUT 0, have similar improvements. In this case also, the most contributing parameter is POS STACK 0.

POS INPUT 0			LEX INPUT 0		
UAS	LAS		UAS	LAS	
74.02	59.67	POS STACK 0	**65.86**	52.18	POS STACK 0
67.77	54.50	LEX STACK 0	**58.59**	45.51	LEX STACK 0
58.37	41.83	POS INPUT 0 pw	**51.98**	37.70	POS INPUT 0 pw
55.28	38.49	LEX INPUT 0 pw	50.44	29.71	POS INPUT 1
51.53	30.43	POS INPUT 1	50.38	35.24	LEX INPUT 0 pw
51.05	32.66	LEX INPUT 0 lc	49.37	32.27	POS INPUT 0
49.71	31.54	POS INPUT 0 lc	48.91	27.77	LEX INPUT 1
49.49	29.18	LEX INPUT 1	48.66	29.91	LEX INPUT 0 lc
49.37	32.27	LEX INPUT 0	47.25	28.92	LEX INPUT 0 rc
48.68	29.34	DEP STACK 0	47.09	28.65	POS INPUT 0 lc
48.47	30.84	LEX INPUT 0 rc	46.68	27.08	DEP INPUT 0 lc
46.77	26.86	DEP INPUT 0 lc	45.69	27.83	POS INPUT 0 rc
46.40	29.95	POS INPUT 0 rc	44.77	26.17	DEP STACK 0
43.49	26.45		44.43	26.47	DEP INPUT 0 rc
42.27	25.21	DEP INPUT 0 pw	42.76	23.56	
41.04	26.56	DEP INPUT 0 rc	41.87	23.04	DEP INPUT 0 pw

Table 7.2: Results for the first generation of each of the two initial tests. The eight best combinations (in bold) are selected for the next generation.

The training results are sorted by unlabeled attachment score. For each generation, except the first, the eight best results are selected as starting points for a next test generation.

7.3.3 The second generation

For the first generation the combination of POS INPUT 0 and POS STACK 0 gets the best score, 74.02%. This combination is selected as the first of eight fixed parameter sets for eight new tests in the next generation and the neighboring nodes are added.

Looking at the table of rules we see that for INPUT 0 the nodes to add are precisely the same as before, except that now POS STACK 0 is part of the fixed parameters, so it should not be added. Next we look at STACK 0 and see in the first column that we should add:

- LEX and DEP for that node. These were added in the previous generation so we don't do that again.
- POS, LEX and DEP for the head of the node.
- In the same way POS, LEX and DEP for the leftmost and rightmost child, the left and right sibling and the previous and following word In the original sentence for that node.
- POS, LEX and DEP for the node on step down in the stack, STACK 1.
- By the special rule POS and LEX for INPUT 0 should be added. Since we decided to start the tests with that node this is already in the collection.

The other seven tests were constructed in a similar way.

Table 7.3 contains the result for the combination of POS INPUT 0 and POS STACK 0 in generation 2. This time all the best eight results belong to this collection, and the best score is achieved when the fixed parameters are combined with POS INPUT 1.

In the results table is also shown for comparison not only the result of evaluation on the development set but also for the train set.

7.3.4 Results of first evaluation

A total of 16 generations were tested using the method described above. Table 7.4 shows a list of the best results for each generation.

We can see that the test set closely follows the development set with values about 1% lower. The peak is at generation 12 after which the results for the test set decreases and the development set continues to increase. This is the typical indication of the point where overfitting occurs. Beyond this point the generalization is degraded as shown by the continuously falling results for the test set.

The parameters generating the result for generation 12 is shown in Figure 7.5, in order of being included in the set.

7.4 Second evaluation on Swedish corpus

The same method was then applied to the same corpus, but the search was for the optimal labeled attachment sccore. In the first evaluation each the eight best results for each generation were selected and tested in the next generation. The

POS INPUT 0
POS STACK 0

Dev set		Test set		
UAS	LAS	UAS	LAS	
79.50	65.34	79.07	65.86	POS INPUT 1
78.73	66.98	76.04	64.51	LEX STACK 0 fw
77.42	63.08	74.63	61.86	LEX INPUT 1
77.06	64.54	75.28	62.90	LEX INPUT 0 pw
76.83	66.01	73.61	63.77	LEX INPUT 0
76.63	63.62	74.75	63.17	POS STACK 0 fw
76.44	64.24	74.09	62.02	LEX STACK 0
76.39	63.12	73.99	61.16	LEX INPUT 0 lc
76.25	62.51	73.87	61.08	LEX INPUT 0 rc
75.97	62.47	73.55	60.78	POS INPUT 0 lc
75.97	62.20	74.45	61.40	DEP STACK 0 fw
75.88	62.42	74.03	61.30	DEP INPUT 0 lc
75.75	62.11	73.53	60.76	POS INPUT 0 rc
75.74	62.08	73.59	61.06	DEP INPUT 0 rc
75.71	62.15	74.67	62.88	DEP INPUT 0 pw
75.55	61.37	73.81	60.96	POS STACK 1
75.51	62.22	74.85	62.24	POS INPUT 0 pw
75.22	61.08	73.83	60.98	POS STACK 0 pw
74.94	60.85	73.59	60.47	LEX STACK 0 pw
74.88	60.59	72.93	59.57	LEX STACK 1
74.81	60.56	72.77	60.03	DEP STACK 0 pw
74.58	60.44	72.50	59.89	POS STACK 0 h
74.56	60.12	72.42	58.81	DEP STACK 1
74.44	60.36	72.14	59.05	LEX STACK 0 rs
74.37	60.05	72.67	60.05	DEP STACK 0 rs
74.36	60.62	72.24	59.63	LEX STACK 0 lc
74.30	60.16	72.67	60.11	DEP STACK 0
74.24	59.87	71.78	58.65	POS STACK 0 ls
74.23	59.84	71.82	58.75	LEX STACK 0 ls
74.21	59.88	71.88	58.71	DEP STACK 0 ls
74.16	60.14	72.48	59.81	DEP STACK 0 lc
74.16	59.93	71.86	58.69	LEX STACK 0 h
74.15	59.88	71.84	58.61	POS STACK 0 rs
74.12	60.38	72.06	59.61	POS STACK 0 lc
74.09	60.53	72.38	60.11	POS STACK 0 rc
74.08	59.75	71.70	58.39	DEP STACK 0 h
74.02	59.97	72.10	59.15	DEP STACK 0 rc
74.02	59.67	71.60	58.37	
74.00	60.71	72.38	59.77	LEX STACK 0 rc

Table 7.3: Results for the second generation with the fixed parameters POS INPUT 0 and POS STACK 0.

	Dev set		Test set	
Generation	UAS	LAS	UAS	LAS
1	**74.02**	59.67	**71.60**	58.37
2	**79.50**	65.34	**79.07**	65.86
3	**83.58**	71.76	**82.75**	70.98
4	**85.96**	76.03	**84.82**	74.75
5	**87.23**	77.32	**86.34**	76.52
6	**88.23**	79.28	**87.21**	78.29
7	**88.42**	80.00	**87.67**	78.99
8	**89.43**	81.56	**88.09**	80.26
9	**89.84**	83.20	**88.69**	82.33
10	**90.23**	83.89	**89.17**	83.31
11	**90.49**	84.31	**89.58**	83.85
12	**90.73**	84.47	**89.66**	83.83
13	**90.81**	84.60	**89.52**	83.75
14	**90.81**	84.70	**89.32**	83.73
15	**90.85**	84.67	**89.13**	83.21
16	**90.84**	84.68	**88.65**	82.75

Table 7.4: Best results for each generation. Swedish corpus, optimized for UAS.

POS INPUT	0
POS STACK	0
POS INPUT	1
LEX STACK	0 fw
LEX STACK	0
LEX INPUT	0 lc
POS STACK	1
LEX INPUT	1
LEX INPUT	0
DEP STACK	0 lc
POS STACK	0 fw
LEX STACK	0 fw ls

Table 7.5: Parameters used in generation 12 in Table 7.4.

results showed that only the four best of these contributed to the best results in the next generation. Therefore, in this evaluation, only the four best were selected.

The results of the 16 first generations are shown in Table 7.6.

	Dev set		Test set	
Generation	UAS	LAS	UAS	LAS
1	74.02	**59.67**	71.60	**58.37**
2	78.73	**66.98**	76.04	**64.51**
3	83.58	**71.76**	82.75	**70.98**
4	85.92	**76.28**	84.39	**74.57**
5	86.61	**78.71**	85.22	**77.02**
6	87.62	**80.86**	86.26	**79.21**
7	88.81	**82.22**	88.23	**81.40**
8	89.23	**83.24**	88.61	**82.63**
9	89.97	**83.93**	89.07	**83.41**
10	90.30	**84.44**	89.34	**83.97**
11	90.55	**84.83**	89.58	**83.85**
12	90.62	**84.98**	89.62	**84.13**
13	90.78	**85.17**	89.68	**84.19**
14	90.88	**85.31**	89.54	**84.21**
15	90.99	**85.41**	89.29	**83.91**
16	90.95	**85.45**	89.56	**84.09**

Table 7.6: Best results for each generation. Swedish corpus, optimized for LAS.

The behavior is similar to that of the first evaluation. The train set follows the development set with increasing values for each generation but 1-2 % lower. The optimal value seems to be in generation 14 with 84.21% for the test set. After that, the performance for the test set decreases. The parameters for generation 14 are shown in Table 7.7.

7.5 Evaluation on English corpus.

Finally, as a comparison, the method was applied to another corpus, the English corpus from CoNLL 2008. This corpus contains sections from the Penn Treebank. For training and development set the sections contains text from Wall Street Journal. A test set with text from Wall Street Journal was also supplied. In addition, there was also a second test set from the Brown corpus which contains texts from other fields. The intent was to investigate how well the parsers adapt to other domains. As before, the four best results for each generation was selected to be

POS INPUT	0
POS STACK	0
LEX STACK	0 fw
POS INPUT	1
LEX STACK	0
LEX INPUT	0
LEX INPUT	0 lc
LEX INPUT	1
LEX STACK	0 lc
POS STACK	1
LEX INPUT	0 pw
POS STACK	0 fw
POS INPUT	2
DEP INPUT	0 lc

Table 7.7: Parameters used in generation 14 in Table 7.6.

the starting point for the next. The tests were optimized for unlabeled attachment score.

The results after 12 generations are shown in Table 7.8. It is interesting to notice that this time the results for the in-domain test corpus, WSJ, exceeds the results for the train corpus. This indicates that the train corpus actually is harder to classify than the test corpus and after training the parser is well equipped to handle the train corpus. As expected, the results for the out-of-domain corpus are quite a bit lower. After 12 generations there is still no sign of overfitting, so the best result is the last one 90.64% for in-domain test and 87.35% for out-of-domain test. The parameters used for this score is shown in Table 7.9.

	Dev set		Test set WSJ		Test set Brown	
Generation	UAS	LAS	UAS	LAS	UAS	LAS
1	**64.42**	55.64	**64.71**	56.44	**71.29**	62.41
2	**78.62**	68.77	**78.99**	70.30	**78.67**	65.17
3	**81.83**	76.67	**82.46**	77.82	**80.57**	72.95
4	**84.43**	79.78	**84.89**	80.88	**84.03**	76.99
5	**85.95**	81.60	**86.61**	82.93	**84.55**	77.80
6	**86.95**	82.73	**87.73**	84.09	**85.26**	78.48
7	**88.03**	83.62	**88.52**	84.74	**85.66**	78.73
8	**88.61**	84.97	**89.15**	86.20	**86.29**	79.86
9	**89.09**	85.43	**89.47**	86.60	**86.43**	80.02
10	**89.54**	85.87	**90.25**	87.40	**87.00**	80.75
11	**89.95**	86.21	**90.63**	87.77	**86.87**	80.46
12	**90.26**	86.56	**90.64**	87.80	**87.35**	80.86

Table 7.8: Best results for each generation. English corpus, optimized for UAS.

POS INPUT	0
LEX STACK	0
POS INPUT	1
LEX STACK	0 fw
POS STACK	0
DEP INPUT	0 lc
LEX STACK	1
LEX INPUT	1
LEX INPUT	0
POS INPUT	2
POS STACK	0 pw
POS INPUT	3

Table 7.9: Parameters used in generation 12 in Table 7.8.

Chapter 8

Conclusions

This report contains an investigation of the feature set for Nivre's Parser and describes two methods of searching for an optimal combination used in training. The method for intuitive matrix search uses a combination of comparing different parameter settings and intuition. The method of systematic exploration uses a set of rules founded on an initial assumption about neighboring features. In both cases useful parameter combinations were found, but clearly the first method required more work and was less successful. In the cases described in this report the advantage of a systematic search is clear. It can be fully automated, and no linguistic knowledge of the language being trained for is required. The level of greediness for the search can be adjusted by how many fixed parameter sets will be used for each generation. In the evaluations reported it was found that eight was unnecessary many, and four was a good value.

The results attained with this method compares well with other results. For unlabeled attachment score it is 89.66%. In CoNLL-X the two best results were 89.54% and 89.50%. Our best results for labeled attachment score were 84.21%, and the two best results in CoNLL-X were 84.58% and 82.55%. When comparing, it is important to remember that although the training set was the same in these cases, the results are based on different test sets. Our results for the English corpus from CoNLL 2008 are based on unlabeled attachment score, 90.64% for in-domain test and 87.35% for out-of-domain. There was no such category in the competition. It was labeled attachment score only. In CoNLL 2007, though, the two best scores were 90.63% and 90.13%. In this case it is even more important to stress that the conditions are different. In our test it was found that the train set was easier to classify than the train set, which resulted in very good scores. This is usually not the case. A different corpus was used in ConNLL 2007.

Although the systematic search is automated and requires no intuitive guessing, it is still time consuming. Depending on the size of the corpora a test can take between 1 - 130 hours. This is to a large extent due to the learning algorithm used, SVM. The training takes a long time, but the results are usually superior to other algorithms. Nevertheless there is a need to reduce the number of tests required. The first generation contains ca 15 tests per fixed feature set and since parameters are only added and not removed for every generation the number of tests can grow to 100 at generation 10. With the method described in this report the starting node is INPUT 0. This is an obvious choice, since it is the only node that is guaranteed to always have a value other than NOTHING during the parsing process. The two values POS and LEX were selected as starting features, and even if this was for a reason of objectivity, we already knew from the very first tests made that LEX can usually not outperform POS as the single parameter, which all our evaluations confirmed.

An improvement to the method would be to analyze the behavior of parameters and investigate if any can be removed. One example of this is parameters referring to the head of a node on the stack. If a stack node has a head, it is the node below on the stack, and ether the parameter for the **STACK n+1** or **STACK n h** should be removed. This could be extended to further investigate if there are other patterns on features that always outperform others.

Another example of finding candidates for removal would be to investigate if the age of a parameter has any influence. In this context, age refers to how many generations the parameter has been in the parameter set. The first generation a parameter is brought in its age is 0. The next generation the parameter is 1, and so on. A brief study of this was made and no obvious behavior could be discerned, so a more thorough investigation would have to be done.

Bibliography

Chang, C.-C. and Lin, C.-J. (2001). *LIBSVM: a library for support vector machines*. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Charniak, E. (1993). *Statistical Language Learning*. MIT Press, Cambridge, Massachusetts.

Chomsky, N. (1957). *Syntactic structures*. Mouton, The Hague.

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

Keith, A. (2007). *The Western Classical Tradition in Linguistics*. Equinox Publishing Ltd, London.

Lin, D. (1998). Dependency-based evaluation of minipar. In *Workshop on the Evaluation of Parsing Systems*.

McDonald, R. (2006). *Discriminative Training and Spanning Tree Algorithms for Dependency Parsing*. PhD thesis, University of Pennsylvania.

Mel'čuk, I. A. (1988). *Dependency Syntax: Theory and Practice*. State University Press of New York, Albany.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill Higher Education.

Nivre, J. (2003). An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, pages 149–160, Nancy.

Nivre, J. (2005). *Inductive Dependency Parsing of Natural Language Text*. PhD thesis, School of Mathematics and System Engineering, Växjö University.

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