UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Sede Amministrativa: Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione

SCUOLA DI DOTTORATO DI RICERCA IN: INGEGNERIA DELL'INFORMAZIONE
INDIRIZZO: SCIENZA E TECNOLOGIA DELL'INFORMAZIONE
CICLO: 26°

Improvements in Transition Based Systems for Dependency Parsing

Direttore della Scuola: Ch.mo Prof. Matteo Bertocco
Coordinatore d'indirizzo: Ch.mo Prof. Carlo Ferrari
Supervisore: Ch.mo Prof. Giorgio Satta

Dottorando : Francesco Sartorio

Acknowledgements

First and foremost I thank my advisor Giorgio Satta. His guide and his advices have been invaluable and working with him is the most precious experience of my studies.

I would like to acknowledge my co-authors: Joakim Nivre, Yoav Goldberg and Carlos Gómez-Rodríguez. I consider myself lucky to know them, our discussions are still source of inspiration.

A special thanks to Mark Steedman for hosting me in Edinburgh where I had the opportunity to meet great researchers and students. Academically I thanks for their time Tejaswini Deoskar and Shay Cohen. While for both work discussions and social time I thank Mike (my personal CCG parser), Gregor (the Italian guy that does not know an Italian word), Bharat (that always locks the office) and Eva (the whisky expert).

In addition I thank Alessandra Calore of the administrative faculty staff, she helps me with the forest of administrative procedures required to finish my PhD.

Finally I want to thanks my family. Last years have not been simple for all of us, however I always had their support and everything is slowly going better now.

Abstract

This thesis investigates transition based systems for parsing of natural language using dependency grammars. Dependency parsing provides a good and simple syntactic representation of the grammatical relations in a sentence. In the last years, this basic task has become a fundamental step in many applications that deal with natural language processing.

Specifically, transition based systems have strong practical and psycholinguistic motivations. From a practical point of view, these systems are the only parsing systems that are fast enough to be used in web-scale applications. From a psycholinguistic point of view, they very closely resemble how humans incrementally process the language. However, these systems fall back in accuracy when compared with graph-based parsing, a family of parsing techniques that are based on a more traditional graph theoretic / dynamic programming approach, and that are more demanding on a computational perspective.

Recently, some techniques have been developed in order to improve the accuracy of transition based systems. Most successful techniques are based on beam search or on the combination of the output of different parsing algorithms. However, all these techniques have a negative impact on parsing time.

In this thesis, I will explore an alternative approach for transition based parsing, one that improves the accuracy without sacrificing computational efficiency. I will focus on greedy transition based systems and I will show how it is possible to improve the accuracy by using a dynamic oracle and a flexible parsing strategy. Dynamic oracles allow to reduce the error propagation at parsing time. Dynamic oracles may have some impact on training time, but there is no efficiency loss at parsing time. A flexible parsing strategy allows to reduce constraints over the parsing process and the time impact in both training and parsing time is almost negligible. Finally, these two techniques work really well when combined together,

and they are orthogonal to previously explored proposals such as beam search or system combinations. As far as I know, the obtained experimental results are still state-of-the-art for greedy transition based parsing based on dependency grammars.

Sommario

La tesi riguarda gli algoritmi incrementali per l'analisi del linguaggio (naturale) usando grammatiche alle dipendenze. Queste grammatiche permettono di dare una chiara rappresentazione delle relazioni sintattiche che intercorrono tra le varie parole della frase. Negli ultimi anni tali rappresentazioni hanno rivestito grande interesse, fino a diventare un passaggio fondamentale in moltissime applicazioni che trattano il linguaggio.

I sistemi incrementali trovano forti motivazioni sia pratiche che psicolinguistiche. Da un punto di vista pratico, questi sistemi sono gli unici algoritmi in grado di processare velocemente grandi quantità di dati. Da un punto di vista psicolinguistico sono sistemi che simulano il modo in cui l'uomo elabora e capisce il linguaggio.

Se in termini di velocità i sistemi incrementali sono i migliori, esistono sistemi basati sulla teoria dei grafi che ottengono una migliore precisione. Recentemente si è cercato di migliorare i sistemi incrementali con l'ausilio di tecniche più o meno elaborate di "beam search" o combinando i risultati provenienti da diversi algoritmi. Sebbene queste tecniche migliorino la precisione dei sistemi, hanno un impatto negativo sulla velocità degli algoritmi.

Durante il mio lavoro di ricerca ho elaborato sistemi alternativi che migliorano la precisione senza sacrificare l'efficienza. In particolare nella tesi descriverò come sia possibile migliorare i sistemi incrementali agendo sulle funzioni oracolo e aumentando la flessibilità degli algoritmi. Agendo sulle funzioni oracolo, che guidano l'apprendimento dei modelli statistici usati in fase applicativa, è possibile ridurre la propagazione degli errori che tipicamente affligge gli algoritmi incrementali. Le nuove funzioni riducono leggermente la velocità della fase di apprendimento, ma non hanno alcun impatto sull'efficienza in fase applicativa. Invece, agendo sulla flessibilità degli algoritmi, è possibile creare sistemi incrementali con meno vincoli

con un miglioramento della precisione a scapito di una praticamente trascurabile riduzione dell'efficienza. Concluderò mostrando come queste due nuove idee funzionino bene combinate l'una con l'altra raggiungendo risultati tuttora allo stato dell'arte.

List Of Publications

This thesis is mostly based on the following publications:

- A Tabular Method for Dynamic Oracles in Transition-Based Parsing
  Yoav Goldberg, Francesco Sartorio and Giorgio Satta
  Journal - TACL 2014
  http://anthology.aclweb.org/Q/Q14/Q14-1010.pdf

- A Polynomial-Time Dynamic Oracle for Non-Projective Dependency Parsing
  Carlos Gòmez-Rodrìguez, Francesco Sartorio and Giorgio Satta
  Conference Proceeding - EMNLP 2014
  http://anthology.aclweb.org/D/D14/D14-1099.pdf

- A Transition-Based Dependency Parser Using a Dynamic Parsing Strategy
  Francesco Sartorio, Giorgio Satta and Joakim Nivre
  Nominated for the best paper award
  Conference Proceeding - ACL 2013
  http://anthology.aclweb.org/P/P13/P13-1014.pdf

Contents

Chapter 1

Introduction

I started my PhD years after my master degree. I did not take my master degree in this university so the first couple of months I met many professors to know what they were working on. I had just a vague desire to study something related to artificial intelligence but something not just theoretical, something that can have some practical application.

When I first met my actual advisor I was fascinated by natural language processing. All human knowledge, everything we know is written somewhere and obviously it is written in some (natural) language.

Today most of human knowledge is freely available, Wikipedia has 4 706 409 articles, only considering the ones written in English. A lifetime is not enough to read all of them and Wikipedia is just one possible source of information. We already have automatic systems that can access to all this amount of information but imagine if these systems would be able to understand these information, to organize and elaborate them. These systems would know more history of any history professor, they would know more about economy than every man in Wall-Street and they would know more math than any winner of the Fields medal.

Every day millions of new words are written. Some years ago I tried to read completely a newspaper from the first to the last word. It took me 2 days, but unfortunately the second day the news were already old! A system able to process and understand every day all information from newspapers, blogs, Facebook status would know more about our society condition than any sociologist or any shrink in the world.

Every day we communicate to a computer with some interface and with some software, we write some keyword on Google to search some information on the web, we use some application in order to complete a task, we sometimes teach it to do something by writing a code. We adapt our way to communicate to the interface offered by the system. But imagine an universal interface to which you can simply speak or the possibility to teach some task to an automatic system by simply explaining it by using using our way: the (natural) language.

Now, understanding language is not the only problem that we need to solve in order to obtain systems like the ones imagined before. The information need to be processed, selected and elaborated. Understand something does not imply the ability to relate all these information in order to produce something "intelligent". However understand the language is certainly a crucial first step.

Feet back to Earth

Convinced my advisor to give me a trial, I started to study something. After the enthusiastic beginning I was so upset with myself:

Studying syntax, really???

I hated so much syntax at school, specially when I was constrained to study Latin grammar about 15 years ago. I remember the first paper that I read, it was about part-of-speech tagging [Collins, 2002] and I thought: "why we need to know that this is a determiner and this is a name?" Fortunately part-of-speech tagging is an almost solved problem and my advisor suggested me to read something else...

Finally

I studied something about parsing, particularly I remember one paper about a new parsing algorithm [Goldberg and Elhadad, 2010]. The authors were using an almost trivial idea, that exactly because it was simple was working well and I found it great. Studying something more, I became a little bit more aware about the problems of natural language: the ambiguity that is inside the language, the problem of its representations. I understood that there are so many multidisciplinary aspects involved: linguistic, psycholinguistic, graph theory, algorithms, ...

At the end I am here, happy to write my thesis about parsing, knowing that there are many unsolved problems, many things that I do not know and I want to learn. But at least, now, consciously fascinated by natural language processing.

1.1 Who is interested in Natural Language Processing?

If I read the first paragraph of this thesis three years ago I probably told myself: "Francesco, 2001 is already passed and Hal 9000 is far to be invented !". But today I do not think that this systems are just for science-fiction movies. We already have cool applications that make huge use of natural language processing and they are already in our life:

- Siri from Apple makes a first step toward natural language interface,
- Google Now collects data about us to present useful information,
- Google translator app is a first attempt to realize a speech-to-speech translation system,
- IBM Whatson is a great attempt to create a cognitive model, it won a special edition of the quiz show Jeopardy.

If we look at Figure 1.1 we can see the sponsors of the last main conference of the Association for Computational Linguistics. It is true that most of them are big companies that support many information technology events, but it is also clear that natural language processing raises some interest.

Figure 1.1: Backcover of the handbook of the 52nd Annual Meeting of the Association for Computational Linguistics

1.2 Why parsing?

I focus my work on the syntax representation based on dependency grammar. Not all natural language applications use parsing, there are many information extraction systems that are based on statistical models and they work well. For example most of tools for text classification do not use parsing at all.

Probably I have not enough experience to make a prediction but I think that soon almost all natural language applications will use some parsing pre-processing.

Indeed we will soon reach a point after which we will need to understand how words are related in order to improve actual natural language applications.

Dependency parsing is not the most famous syntax representation formalism. Specially in English linguistic tradition the most used syntax representation is a phrase tree, a representation based on constituencies like noun phrase and verb phrase. However dependency parsing is really interesting because it highlight exactly what is most important: the relation between words. For the same reason I think that it is really interesting the predicate argument relation that can be extracted from a CCG (Combinatory Categorial Grammar) derivation.

1.3 Content of Thesis

In this thesis I will try to describe formally my work, but most of all I will try to give the motivations that are behind each idea. Probably this is a simple thesis written in a simple (and often bad) English, but IF it will ever useful to somebody I think that the most important thing is to transmit the reasons that are behind something without shame if they look trivial. The chapters are organized as follow.

Chapter 2: Dependency Tree

I will give the definition of dependency tree and discuss some properties of such syntactic representation. At the end of the chapter I will analyze some pros and cons of the dependency tree formalism with respect to other syntactic representations.

Chapter 3: Dependency Parsing

I will give a quick overview of different approaches to the dependency parsing task. Today the most common approaches are data-driven, however there is still a lot of interest in hybrid approaches that are grammar-based and data-driven.

Chapter 4: Transition Based Dependency Parsing

Transition Based Dependency Parsing is the approach to dependency parsing that I used in my work. In this chapter I will go into the details of the most used

transition based algorithms and I will give my personal point of view of them. I will introduce the oracle function and explain how we can train a model by using machine learning techniques in order to create automatic systems for dependency parsing.

Chapter 5: Oracles

In this chapter we will see one of the central point of my work. I will describe the details of the oracles functions and we will see how it is possible to improve the performances of almost all transition based dependency parsers by using a non-deterministic oracle and a dynamic oracle.

Chapter 6: LR-Spines

In this chapter we will see another central point of my work. I will present a new transition based algorithm, specifically designed to introduce an high degree of flexibility in the parsing process. It has the property to be highly incremental but it is also able to postpone some critical decisions during the parsing process.

Experimental Results

In this chapter I will try to convince the reader that the new ideas presented in the previous chapters are useful from a practical point of view. Specifically I will compare the accuracy obtained by using a static, non-deterministic and dynamic oracle. I will also present the experimental results obtained by using the LR-Spines algorithm.

Conclusion

This chapter include some final consideration and some idea for future works.

Chapter 2

Dependency Tree

The dependency syntactic representation (or *dependency grammar*) has become increasingly popular in the last decade or so. Above all I think that the raise of the dependency representation is due to its clarity and simplicity that makes it a great interface for downstream applications. Indeed the key idea into the dependency syntactic representation is the predicate argument relation between words. These relations are represented as arcs and the whole set of such relations is a graph that connects the words of the sentence. In dependency grammar it is common to assume that such graph is a tree, although there are some exceptions (for example multi-stratal dependency theories) a tree is expressive enough to represent most of the linguistic relations into a sentence.

In the following chapters we will see that this simple formalism allows to use well known algorithms from graph theory and from formal languages while the increased availability of dependency tree banks allows to use modern machine learning techniques.

In this chapter I will give the formal definition of dependency tree and I will discuss some properties. Then we will see the advantage and disadvantages of this representation along with some practical examples. But let me start first with some historical background.

2.1 Some History

The concept of dependencies is extremely old and can be found in many ancient grammars specially in Europe for Classical and Slavik languages. Anyway the starting point of modern theoretical tradition is considered the work of the French linguist Lucien Tesnière so let me quote his words [Tesnière, 1959]

> Every word in a sentence is not isolated as it is in the dictionary. The mind perceives connections between a word and its neighbors. The totality of these connections forms the scaffold of the sentence. These connections are not indicated by anything, but it is absolutely crucial that they be perceived by the mind; without them the sentence would not be intelligible.
>
> [...] a sentence of the type Alfred spoke is not composed of just the two elements Alfred and spoke, but rather of three elements, the first being Alfred, the second spoke, and the third the connection that unites them, without which there would be no sentence.

Tesnière called *governor* the word "spoke" and *subordinate* the word "Alfred". In modern dependency grammars we represent this idea of dependency relation by means of a graph theoretic arc. In figure 2.1 we can see this simple example from Tesnière in the contemporary representation of dependency tree, the arc that goes from the head, "Alfred", to the dependent "spoke" is exactly the third element that Tesnière is talking about. We can also see that we add an artificial word: -ROOT- and a label to the arcs, we will discuss these elements in the next section.

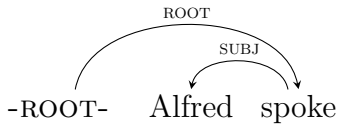


Figure 2.1: Simple example of dependency tree

2.2 Formal Definitions

As we have just seen the *dependency tree* of a *sentence* describes the syntactic structure as words linked by binary relations called *arcs*, at each arc is associated an *arc label*. I will formally define these element and give some property useful as background for the rest of the thesis.

Definition 2.1. A *sentence* S is considered as a sequence of tokens:

$$S = w_0 w_1 w_2 \dots w_n, \; w_0 = \text{-ROOT-} \tag{2.1}$$

Each element of the sentence (word, punctuation, digit or symbol) is a different token. In most natural languages the tokenization is quite straightforward and mostly correspond to the words' separation in a sentence. However, there are cases in which we need a preprocess, for example the word "won't" needs to be split in "will" and "n't". In some languages the token separation is not trivial and the preprocessing step is harder and still introduces errors and noise in the data (e.g., in Chinese). A further description of the problem can be found in [Guo, 1997]; for a discussion of some recent tools for this task we refer the reader to [Dridan and Oepen, 2012]. We always set w_0 as an artificial token -ROOT- . This is a technical assumption which we use here since it is useful to have all dependency trees rooted by the same token and this also simplifies some of the definitions. We remark here that including or not the -ROOT- during the parsing process can have an impact over the performances, as it will be discussed in more details.

Definition 2.2. An *arc label (or dependency label)* l_i identifies the type of a syntactic relation. The set of arc labels is finite and is defined by the dependency grammar.

$$l_i \in L, \text{ where } L = \{l_0, l_1, l_2, \dots, l_m\} \tag{2.2}$$

Unfortunately most of the treebanks use a language specific set of arc labels. In my examples I will use the Stanford Dependency Labels from [de Marneffe and Manning, 2008].

Definition 2.3. An *arc (or dependency relation)* a is a binary and asymmetric link between a syntactically subordinate token (*dependent* or *child*) and another token on which it depends (*head* or *parent*). It is represented by a tuple:

$$a = (w_i, l_k, w_j) \tag{2.3}$$

where the tokens w_i, w_j are respectively the head and the dependent and l_k is the arc label.

Definition 2.4. A *dependency tree* T with respect to a set of dependency labels L and a sentence $S = w_0 w_1 w_2 \dots w_n$ is a directed, ordered tree:

$$T = (V, A) \tag{2.4}$$

where:

1. $V = \{w_0, w_1, w_2, \dots, w_n\}$ is the set of nodes
2. $A \subset V \times L \times V$ is the set of arcs
3. T is rooted by $w_0 =$ -ROOT-

Recall that an ordered tree is a rooted tree for which an ordering is specified for the children of each node. Note that I restrict the directed tree to the case where the arcs are all directed away from the root node.

Sometimes it is useful to consider an *unlabelled* dependency tree, by simply ignoring the arc labels. Indeed for some application (e.g. speech, prosody, language modelling) it is enough to know that the dependency relation between words exists. In the following chapters I will often ignore the arc labels, mostly to simplify the notation by using $w_i \to w_j$ to indicate a generic arc from w_i to w_j.

2.2.1 Example

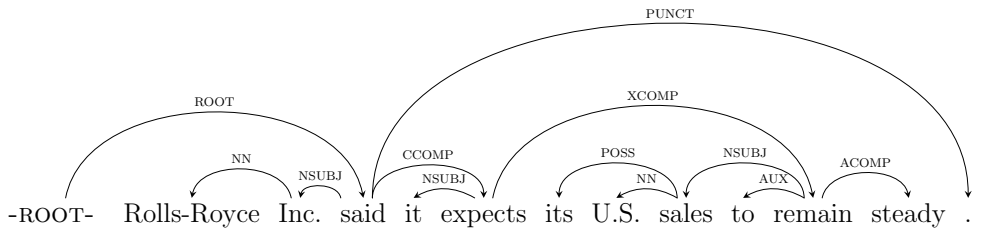


Figure 2.2: Example of dependency tree

In figure 2.2 it is possible to see a well-formed dependency tree. Following the definition 2.4 the dependency tree is in the form $T = (V, A)$ where:

$V = \{$-ROOT- , Rolls-Royce, Inc., said, it, expects, its, U.S., sales,
to, remain, steady, .$\}$
$A = \{$(Inc.,NN,Rolls-Royce),(said,NSUBJ,Inc.),(-ROOT- ,ROOT,said),
(expects,NSUBJ,it),(said,CCOMP,expects),(sales,POSS,its),
(sales,NN,U.S.),(remain,NSUBJ,sales),(remain,AUX,to),
(expects,XCOMP,remain),(remain,ACOMP,steady),(said,PUNCT,.)$\}$

it is easy to see that all the constrains given in the definition 2.4 are respected. Each arc has a direct syntactic meaning, for example the arc (Inc.,NN,Rolls-Royce) implies that "Rolls-Royce" is a noun compound modifier of "Inc.", the arc (said,NSUBJ,Inc.) means that "Inc." is the nominal subject of "said".

2.2.2 Properties of a Dependency Tree

Given the definition 2.4 of a dependency tree we can highlight some properties and give some other definitions.

Property 2.5 (Single head)**.** For all $w_j \in V \setminus \{$-ROOT- $\}$, $\exists!\ a \in A$ of the form $w_i \rightarrow w_j, w_i \in V$. This means that each token in the sentence has one and only one head.

Property 2.6 (Single label)**.** If $(w_i, l_k, w_j) \in A$ then $\nexists\ (w_i, l', w_j) \in A$ s.a. $l' \neq l_k$. This means that can exists only one arc with a specific label that connects two nodes.

In figure 2.2 it is evident that each word except the -ROOT- has one and only one head and each arc has one and only one label.

Property 2.7 (Span of a node)**.** A node w' of a dependency tree is the root of a sub-tree $T' = (V', A')$ (possibly consisting of a single node). The set of nodes V' is a subsequence of the sentence S and is called the *span* of w'.

Definition 2.8 (Gap-degree)**.** If the span of a node w' is a contiguous subsequence (a substring) of S, the gap-degree of w' is 0; if the span is composed by 2 contiguous

subsequence of S, the gap-degree of w' is 1, and so on. The gap-degree of a tree is the maximum gap-degree over all its nodes.

Considering the example in figure 2.3 the span of w_5 is the subsequence $S' = w_5w_6w_7$ and w_5 has gap degree 0, the span of w_2 is the subsequence $S' = w_1w_2w_5w_6w_7$ and w_2 has gap degree 1, the span of w_3 is the subsequence $S' = w_1w_2w_3w_4w_5w_6w_7w_8w_9$ and w_2 has gap degree 0, while the gap degree of the tree is 1.

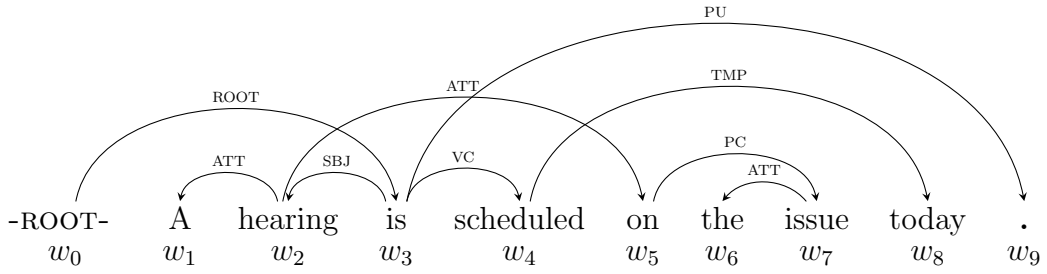


Figure 2.3: Example of dependency tree

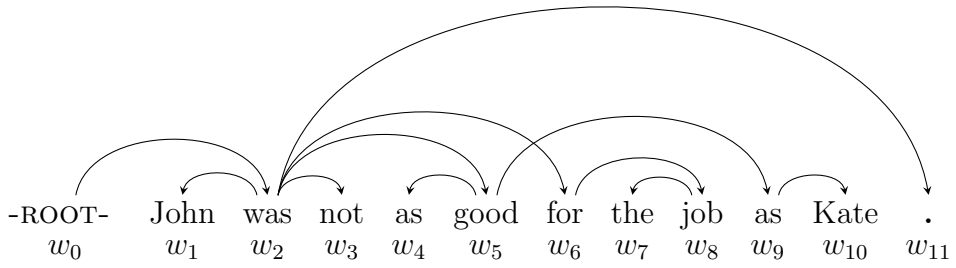
2.3 Projectivity

This is an important characterization of a dependency tree that practically splits the whole world of parsing algorithms in two categories: the algorithms that are able or not to deal with non projective dependency trees. In figure 2.4a we can see the usual representation of a dependency tree, the arcs are drawn in the semi-plane over the ordered sequence of the sentence. The arcs $(w_2 \rightarrow w_6)$ and $(w_5 \rightarrow w_9)$ cross each other. If a dependency tree has crossing arcs is non-projective, formally:

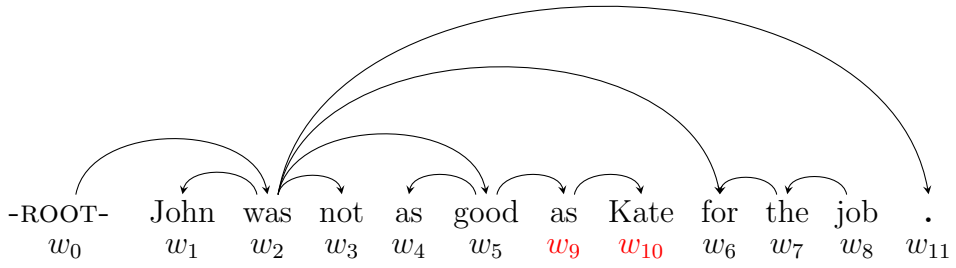
Definition 2.9. A dependency tree is *projective* if and only if all nodes have gap-degree 0. Otherwise a dependency tree is *non projectice*.

Note that the equivalence between non-projective and dependency tree with crossing arcs holds only if we insert the artificial node -ROOT- at the beginning or at the end of the sentence.

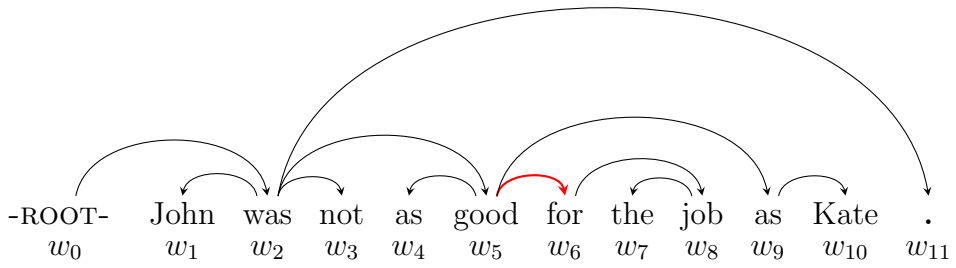
Considering that a dependency tree is a tree, a non projective structure can always became projective by reordering the tokens in the sentence as we can see



(a) Non projective dependecy tree



(b) Dependecy tree projectivize by movig the words $w_9 w_1 0$



(c) Dependency tree projectivize by changing the arc $w_2 \rightarrow w_6$ into the arc $w_5 \rightarrow w_6$

Figure 2.4: Unlabelled non projective dependency tree with different projectivize by using word reordering and arcs modification

in 2.4b. Another way to projectivize a dependency tree is by changing some arcs like in figure 2.4c, this is a useful technique when we want to use a projective parsing algorithm with non projective sentences. An interesting discussion with

experimental results can be found in [Nivre and Nilsson, 2005].

Designing a parsing algorithm that deals only with projective trees can really improve the performance, mostly in terms of computational time (speed) but also in terms of precision. This can be a good compromise for languages like English and Italian where the non-projective structures are infrequent, while it can raise a coverage problem for languages like Czech or Hungarian that have more than 20% of non-projective sentences or for languages like German with free word order. We will see something more about this in the following chapters.

2.4 Pros & Cons of Dependency Tree

The main advantage of the dependency tree representation is that it directly encodes the predicate argument relations. These relations can be extracted also from other representations but it is not so straightforward. For example in the phrase structure in figure 2.5 we need to navigate the tree in order to understand that the subject of "said" is "Inc.".

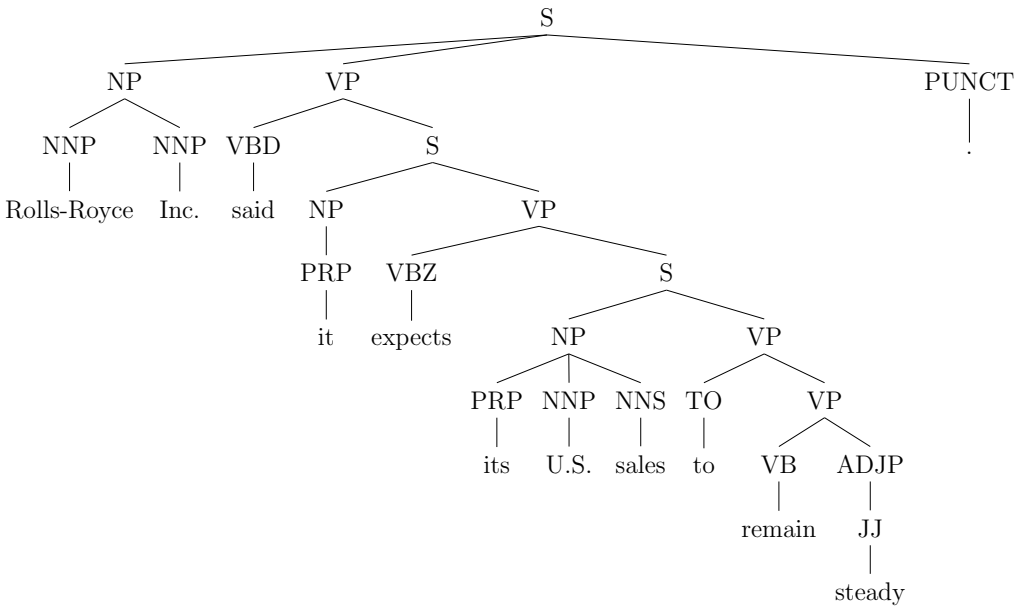


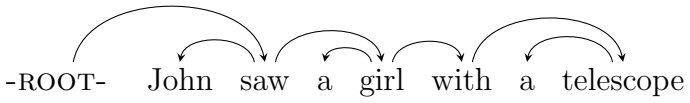
Figure 2.5: Example of Phrase Structure representation

The simple representation of the predicate argument relations has other ad-

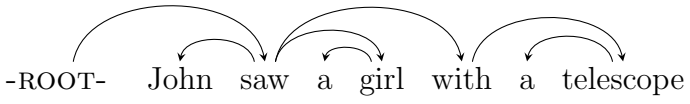
vantages, however it has also some limits. In the following sections I will analyze some pros and cons.

2.4.1 Pros: PP-attachment

The prepositional phrase attachment is a typical problem related to the ambiguity of natural language. The same sentence with a different pp-attachment can have a completely different meaning. The sentences in figure 2.6 are both syntactically correct but for the first sentence we can imagine that John saw a clever girl that was carrying a telescope while in the second John is a voyeur that is spying a girl with a telescope. It is worth noting that while the pp-attachment is represented at the syntactic level, its disambiguation needs to resort to some kind of semantic interpretation that can deal with phenomena ranging from lexical semantics, to pragmatics and general world knowledge. In a dependency tree we have a clear representation of the pp-attachment problem, it is easy to identify the words involved into the relation and it is possible to choose above the possible interpretations by simply changing an arc.



(a) The girl was carrying a telescope



(b) John was using a telescope

Figure 2.6: Unlabelled dependency trees that represent two different interpretation of the same sentence

2.4.2 Pros: free word order languages

Languages with (relatively) free word order can be easily dealt with dependency trees because changing the word order does not change the relation between words. For example in figure 2.7 we have two sentences with identical meaning but with different word order. For both sentences the dependency tree is the same $T = (V, A)$ with:

$V = \{$-ROOT- , der, Hund, beißt, die,Frau$\}$
$A = \{$(-ROOT- ,ROOT,beißt), (beißt,SUBJ,Hund), (beißt,OBJ,Frau),
(Hund,DET,der),(Frau,DET,dir)$\}$

Otherwise a phrase structure grammar would need separate rules to handle different positions of the subject/object.

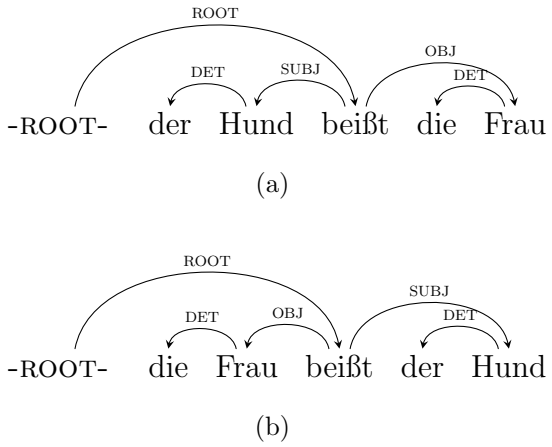
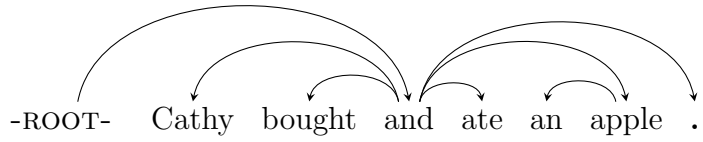


Figure 2.7: Example of a dependency tree for a sentence in a free word order language (German, the dog bites the woman)

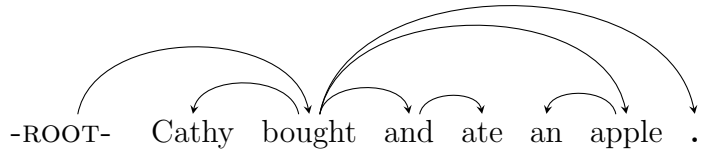
2.4.3 Cons: coordination

The single head constrain may be a limitation in case of coordination. For example in the sentence “Cathy bought and ate an apple”, it is clear that “Cathy” is the subject of both verbs as well as the word “apple” is the object of both. There are

different ways to treat coordinations, the most popular are in Figure 2.8, however in all cases the dependency relations are not all evident.



(a) The coordination problem solved by using the conjunction as head



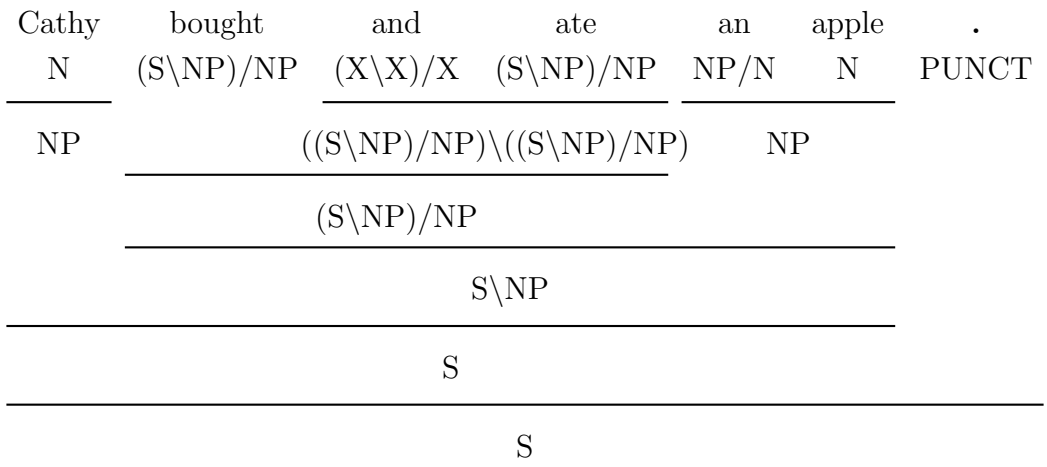
(b) The coordination problem solved by using the first conjunct as head the conjunction as a dependent

Figure 2.8: Example of coordination problem in dependency parsing

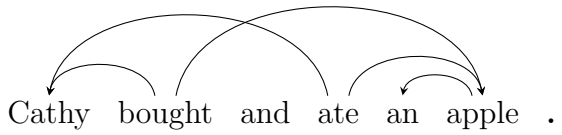
The Combinatory Categorial Grammar performs better from this point of view because the predicate argument relation extracted from the derivation represents all the dependencies. As we can see in Figure 2.9 the CCG uses a special category $(X\backslash X/X)$ that allow to merge and combine the verbs.

2.4.4 Cons: compound names

In compound names we have to choose an head above all noun. Usually the right most or the left most noun is chosen, however often the chosen node is not the most significant. For example in figure 2.10 "Inc." is chosen above all other nouns in "Rolls-Royce motor cars Inc.". In phrase structure all noun are represented as sibling of the same node with a more clear representation.



(a) CCG derivation in case of coordination



(b) Predicate-argument relation for the derivation in (a)

Figure 2.9: Example of coordination solved with CCG

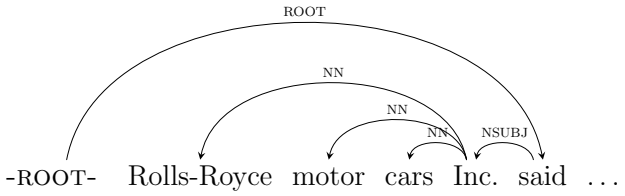


Figure 2.10: Example of dependency tree

Chapter 3

Dependency Parsing

Dependency parsing is the task of automatically mapping a sentence into the dependency tree that represents the correct syntactic relations. There are two main approaches to the problem:

1. data-driven, an approach that uses machine learning techniques over a data set of syntactically correct dependency trees.

2. grammar-based, an approach that uses formal grammars in order to define a formal language that eventually recognise an input sentence

I worked exclusively on data driven approaches, however in this chapter I will try to give a quick overview of all systems.

3.1 Data Driven approach

Data driven is the most popular parsing approach this days. These methods consider the parsing task a supervised machine learning problem and they obtain really good performances in terms of accuracy. They rely over a statistical model that is learned from a dataset. The dataset is a list of couples:

$$D = \{S_i, T_i\}_{d=0}^{|D|}$$

where S_i is a sentence and T_i is the syntactically correct dependency tree for S_i.

Many datasets are available for many languages, unfortunately most of them make different assumptions in treating some dependency relations, for example in

case of coordinations or -ROOT- dependents. Different datasets have also different label-sets with a different detail level.

However a data driven model can be trained almost independently from the datasets' peculiarities so we can use the same technique with different datasets. This is important because it allows us to study the parsing problem from a multi-language point of view. Indeed we can train a language specific model by simply using a dataset with sentences in such language.

In the following sections we will see the two most popular data driven approaches:

1. Graph Based parsing

2. Transition Based Parsing

3.1.1 Graph Based Parsing

In Graph Based Parsing we use a traditional graph theory approach. We consider the parsing problem as a maximization problem where the objective of the parsing algorithm is to find a dependency tree that maximizes a score function. In the training phase we learn the parameters of the score function. In this section I do not pretend to give a complete description of graph based parsing systems but a general description in order to compare them with the Transition Based Parsing approach. A certainly better introduction to this technique can be found in [McDonald and Pereira, 2006], and for a further analysis the Ryan MacDonald PhD thesis is a must reading.

Score function

The score of a dependency tree T_S should represent how likely the structure of the tree represents the syntactically correct relations into the sentence S. The basic assumption of graph based techniques is that the score of a dependency tree T_S is the factor of the scores of the subtrees of T_S.

The smallest (non-complete) subtree into a dependency tree $T_S = (V_S, A_S)$ is

an arc. So we can write:

$$score(T_S) = \sum_{\substack{w_i,w_j \in V_S \text{ s.a.} \\ (w_i \rightarrow w_j) \in A_S}} score'(w_i, w_j)$$

Specifically this is the score function for an arc-factored parsing algorithm that independently score each arc in A_S. The Independence assumption is strong and allows to use efficient algorithms, but it is generally wrong to assume that the relation between two words is independent from other words into the sentence. However a scores function that consider couple of arcs is enough to reach state of the art results.

$$score(T_S) = \sum_{\substack{w_i,w_j,w_k \in V_S \text{ s.a.} \\ (w_i \rightarrow w_j),(w_j \rightarrow w_k) \in A_S}} score'(w_i, w_j, w_k)$$

Parsing Algorithm

The parsing algorithm is a maximum spanning tree algorithm that search the tree T_S that maximize the score function over all possible dependency trees of a given sentence S.

$$T_M = \arg\max_{T_i \in D(S)} score(T_i)$$

Where $D(S)$ is the set of all possible dependency trees of S and T_M is the dependency tree retrieved by the parsing algorithm.

Given a sentence of length n the number of possible dependency trees is exponential. However we can use use dynamic programming techniques to compute all possible trees in $D(S)$ in polynomial time and space. After that a Viterbi search find the tree that maximize the score function.

Training Algorithm

Typically the score function is a perceptron algorithm trained by using an on-line learning technique. In order to stabilize the model parameters we normally use the MIRA update technique from [Taskar, Klein, Collins, Koller, and Manning, 2004] or the averaged perceptron from [Freund and Schapire, 1999].

We will see more details when we will use the perceptron algorithm for the transition based algorithms in chapter 4.

Computational Complexity

The computational complexity depends on two factors:

1. the search space (projective or non-projective dependency trees)
2. the scope (size of the subtree) considered by the score function

Using the naive arc-factored parsing algorithm the parsing process has complexity $\mathcal{O}(n^3)$ in case of projective or non projective dependency trees as in [McDonald, Pereira, Ribarov, and Hajič, 2005]. However if we consider three nodes in the tree the complexity grows to $\mathcal{O}(n^4)$ in the projective case and $\mathcal{O}(n^5)$ in the non-projective case (with some constraints). Note that this analysis is far to be complete and precise because a further computation analysis depends on the detail of the algorithm. However it is useful to understand that the complexity is too high to use the graph based systems in web-scale applications.

These systems reach state of the art results in dependency parsing reaching accuracy[1] of 92-94%. So they are good systems if the data to be processed are limited.

3.1.2 Transition Based Parsing

Transition Based Parsing is the core subject of this thesis. These systems use an incremental non-deterministic algorithm that analyzes the sentence left to right. At each step the parsing algorithm can choose an action (called transition) to proceed with the parsing process. The choice is supported by a model trained by using machine learning techniques. I will dedicate the whole chapter 4 to these algorithms and the following chapters to see how it is possible improve them. For now let me underline some general characteristics.

These algorithms reach an accuracy[1] of about 88/90% depending on the details of the algorithm. Although they fall back in terms of accuracy compared to graph based algorithms, transition based algorithms rise a lot of interest for their efficiency. They are able to process an input sentence in linear time reaching a throughput of several thousands of tokens for second. This is extremely important if we want to use a parser into a web-scale application.

[1] Accuracy over English Penn Tree Bank [Marcus, Marcinkiewicz, and Santorini, 1993] converted into a dependency tree bank

3.2 Grammar Based Approach

The Grammar Based Approaches rely on an explicitly defined formal grammar. I never used these approaches but they deserve a mention into a thesis about dependency parsing, for a further introduction see chapter 5 in [Kübler, McDonald, and Nivre, 2009]. Specifically there are two main approaches:

1. context-free grammar
2. constraint satisfaction

3.2.1 Context-Free Grammar

The *context-free grammar* approach for dependency parsing is similar to the context-free grammar approach used in graph-based parsing. The problem is restricted to projective (usually unlabelled) dependency trees.

Definition 3.1. A context free grammar Γ is a tuple (N, Σ, Π, S) where:

1. N is a finite set of non-terminal symbols
2. Σ is a finite set of terminal symbols
3. Π is a set of production rules of type $x \rightarrow \{N \cup \Sigma\}^*, x \in N$ (* is the Kleene star operator)
4. S is the start symbol

In dependency parsing the starting symbol S is the root node -ROOT- , the set of terminal symbols Σ is the set of all possible words and the set of non-terminal symbols N is a subset of the set of all possible words $N \subseteq \Sigma$.

The advantage of this technique is that we can used well known algorithms used in phrase structure parsing like CKY [Kasami, 1965] or Earley [Earley, 1970] algorithms. Usually a probability is associated to each production rule obtaining a probabilistic context-free grammars. These probabilities can be learned by using a data driven approach, for example by using a maximum entropy model, obtaining an hybrid approach: grammar based and data driven.

The complexity is $\mathcal{O}(n^3)$ in case of a context free grammar approach and $\mathcal{O}(n^5)$ in case of bi-lexical context free grammars.

3.2.2 Constraint Satisfaction

The constrain satisfaction approach defines a set of constraints that has to be satisfied by the dependency tree. The interesting aspect is that constraints can be syntactic, for example the verb need a subject, or semantic, for example the object of the verb "play" must be a game or a musical instrument.

Definition 3.2. A constraint dependency grammar Γ is a tuple (Σ, L, C) where:

1. Σ is a finite set of terminal symbols
2. L is the label set
3. C is the set of boolean constraints

There can be hard constraints, constraints that must be satisfied, for example in English a verb need a subject. Or weighted constraints in which the weight represent a cost in case of the constrain is not satisfied. Given the high degree of exceptions in natural language hard constraints are usually avoided.

The weights can be manually defined or learned by using a data driven approach. An example of manually written constraint dependency grammar for German is [Foth, Daum, and Menzel, 2004] where there are 700 constraints.

There are two big problems with this approach. First of all the set of constraints is strictly language dependent, second the constraints satisfaction problem is in general NP-hard so we usually need to use an heuristic to treat the problem.

Chapter 4

Transition Based Dependency Parsing

There are few things in which psycholinguistics, formal linguistics and engineers agree upon, one of this is the left to right incremental parsing. The incremental strategy is a largely accepted hypothesis from psycholinguistics about how humans process and understand the language. In spoken, but also in written language is clear the intuition that the comprehension of a sentence proceed and grow as soon as a word is encountered [Altmann and Steedman, 1988]. It is interesting the experiment in [Marslen-Wilson, 1973] where the authors realize that most of the errors in speech shadowing [1] were syntactically and semantically correct with respect to the previous part of the sentence; this suggests that the previous grammatical structure was already built and that the subjects were unconsciously following such structure. In the perspective of formal language theory, it is easy to see similarities with well studied algorithms for formal grammars. Indeed most of these algorithms are taken and adapted from the wide literature on context-free and context-sensitive grammars. From an engineering point of view the transition based systems have good performances in terms of precision and speed. The precision is close to the state of the art reached from graph based parsers, while the complexity with respect to the length of the input sentence is linear (or almost linear).

[1] Speech shadowing is an experimental technique in which subjects repeat speech immediately after hearing it (usually through earphones) with a latency of about 500-1500 ms

As we have breefly seen in chapter 3 the transition based dependency parsers process the input sentence left to right by incrementally building the dependency tree. It relies on two components:

1. a parsing algorithm
2. a trained model

The parsing algorithm is a non deterministic algorithm that step by step builds a dependency tree. At each step it has to take a decision about how to proceed and the choice is supported by a statistical model. The model is trained from a data set of syntactically correct dependency trees.

The transition based approach for dependency parsing was pioneered by Taku Kudo and Yuji Matsumoto. In [Kudo and Matsumoto, 2002] they applied this approach on Japanese dependency parsing and they claim the independence of the parsing algorithm from the machine learning system used for the model. The following years the same approach was used in [Yamada and Matsumoto, 2003], [Nivre, 2003] and [Attardi, 2006]. Probably the best description of this systems is [Nivre, 2008] where one can find both simple and formal descriptions of the algorithms along with the relatives proofs.

In this chapter I will firstly describe the parsing algorithms in the most possible general way to show that all this kind of systems share the same basic idea. After that I will describe the four most used transition based systems: Arc-Standard, Arc-Eager, Attardi's algorithm and Swapping Algorithm. At the end I will describe some possible ways to train a model for such algorithms.

4.1 Parsing Algorithms

I like to view a transition based parsing algorithm as a push down automaton with:

- a stack σ,
- a buffer β,
- an alphabet (in this case equal for the the stack and the buffer), that is the set of all possible tokens in a sentence

- a state called *configuration*
- a set of *transitions* that can change the state of the automaton from a *start configuration* to a *final configuration*

While it is reasonable to consider the set of possible tokens limited, like the alphabet of a push down automaton, the comparison is not formally correct because in a (transition based) parsing algorithm we have a more complex definition of configuration with respect to the state of a push down automaton. However I think that the analogy gives a good idea about what we are dealing with.

Definition 4.1. A *configuration* c, given a sentence $s = w_0, w_1, \ldots, w_n$ with the set of nodes V_s is a tuple:

$$c = (\sigma, \beta, A) \tag{4.1}$$

where:

- σ is the stack and it is a possibly non contiguous subsequence of s
- β is the buffer and it is a contiguous subsequence of s
- A is the set of already built arcs, with $A \subset V_s \times V_s$

The *initial configuration* is:

$$c_0 = ([\,], [w_0 w_1 \ldots w_n], \varnothing) \tag{4.2}$$

and the *final configuration* is:

$$c_f = ([w_0], [\,], A_f), \text{ where } |A_f| = n \tag{4.3}$$

Definition 4.2. A *transition* τ is an operator that maps a configuration c into another c'. The notation:

$$c \vdash_\tau c'$$

or

$$c' = \tau(c)$$

means that by applying the transition τ to the configuration $c = (\sigma, \beta, A)$ we obtain a new configuration $c' = (\sigma', \beta', A')$. The functional notation $\tau(c)$ is is useful to denote the obtained configuration.

In practice a transition modifies the configuration by moving/removing the tokens into the stack and the buffer and/or by creating a new arc. As we will see in the next sections a transition can create at most one arc, giving the typical incremental behaviour of the transition based dependency parsing algorithms.

Usually the transitions involve the topmost elements in the stack and the first elements in the buffer. To highlight such elements is useful the notation $\sigma|w_i|w_j$ and $w_k|\beta$ where w_i and w_j are the two topmost elements of a generic stack and w_k is the first element of a generic buffer.

Transitions are not applicable to all configurations. To easily identify which transitions are applicable to a configuration we use a *precondition* for each transition.

Definition 4.3. The *precondition of a transition* is a logical condition that defines the applicability of a transition over a configuration.

Example 4.4. In the Arc-Standard algorithm the LEFT-ARC transition (briefly LA) creates an arc from the first topmost to the second topmost element of the stack and removes the dependent of the new arc from the stack. To represent the behaviour of the transition we highlight the differences in the configurations c and c' respectively before and after the transition:

$$c \vdash_{\text{LA}} c' \tag{4.4}$$

$$c = (\sigma|w_i|w_j, \beta, A) \tag{4.5}$$

$$c' = (\sigma|w_j, \beta, A \cup \{w_j \rightarrow w_i\}) \tag{4.6}$$

In the Arc-Standard algorithm the transition LEFT-ARC has two preconditions:

1. the stack must contain at least two tokens
2. w_i is not the token -ROOT-

The first precondition is implicit in the representation $c = (\sigma|w_i|w_j, \beta, A)$ and will be omitted when I will describe the parsing algorithms. The second needs to be explicitly added in the preconditions with the condition $w_i \neq$ -ROOT- . The first entry in table 4.1 represents the behaviour of the LEFT-ARC in a synthetic and readable way.

Sometimes I will use the notation $\text{LA}(i, j)$ to highlight the nodes directly involved into the transition, in this case w_i, w_j.

Algorithm 4.1 General Parsing Algorithm

Input: sentence $s = w_0 w_1 \cdots w_n$
Output: dependency tree T_s

$c = (\sigma, \beta, A) \leftarrow ([\,], [w_0, \ldots, w_n], \varnothing)$ ▷ initialize starting configuration
while $|\sigma| > 1 \vee |\beta| > 0$ **do** ▷ while c is not final do
  $\mathcal{T}' \leftarrow \varnothing$
  for each τ in $\mathcal{T}$ **do** ▷ select the applicable transitions
    if applicable(τ,c) **then**
      $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{\tau\}$
  $\tau \leftarrow$ model.giveBestTransition($\mathcal{T}', c$)
  $c' = (\sigma', \beta', A') \leftarrow \text{apply}(\tau, c)$ ▷ apply(τ, c) returns c' s.t. $c \vdash_\tau c'$
  $c = (\sigma, \beta, A) \leftarrow c'$ ▷ update the current configuration
$V_s \leftarrow \{w_0, w_1, \cdots, w_n\}$
$A_s \leftarrow A$
return $T_s = (V_s, A_s)$

In algorithm 4.1 we can see the general parsing algorithm. Given a sentence the algorithm initializes a starting configuration and enters in a loop. At each iteration it selects the applicable transitions and asks to the trained model which transition is the best one among the applicable transitions in the current configuration. Following the suggestion of the model, the algorithm modifies the current configuration and iterates until a final configuration is reached. The set of arcs A in the final configuration is the set of arcs A_s of the dependency tree returned by the parser.

The sequence of transitions applied to obtain the tree T_s is called *derivation* of T_s. More generally we can speak about *derivation of a configuration* and it is defined as follows.

Definition 4.5. A *derivation (or computation) for a configuration c_j* given a configuration c_i is a sequence of transitions $d = \tau_0 \tau_1 \cdots \tau_k$ s.t.:

$$c_i \vdash_{\tau_0} c_{i+1} \vdash_{\tau_1} \cdots \vdash_{\tau_k} c_j \tag{4.7}$$

Sometimes I will speak about reachability of a configuration c_j from another c_i if exists a derivation that reaches the configuration c_j. If the reachable configuration

is final and represent the dependency tree T_s we can say that the dependency tree T_s is reachable from c_i

All parsing algorithms described in this chapter follow exactly the algorithm 4.1 and the different behaviour of each algorithm derives only from the different set of possible transitions $\mathcal{T}$. In order to give a comparable description I will discuss all algorithms in terms of:

- coverage
- incremental strategy
- spurious ambiguity

In the next sections I will describe these three important properties and I will go into the details of the most used transition based algorithms. To maintain the notation simple I will formalize the algorithms in case of unlabelled dependency parsing, at the end I will show how to extend this techniques to the labelled case.

4.1.1 Coverage

The coverage of a parsing algorithm depends on which types of dependency trees are *reachable* from the system.

Definition 4.6. Given a sentence $s = w_0 w_1 \dots w_n$ the dependency tree $T_s = (V_s, A_s)$ is *reachable* if there exists a derivation $d = \tau_0 \tau_1 \dots \tau_k$ such that:

$$c_0 \vdash_{\tau_0} c_1 \vdash_{\tau_1} \dots \vdash_{\tau_k} c_f \tag{4.8}$$

where the final configuration $c_f = ([w_0], [\,], A_f)$ has the set of arcs equal to the one in T_s: $A_f = A_s$

Usually we distinguish the parsing algorithm in:

- *projective*, parsing algorithms that can reach only projective dependency trees
- *non-projective*, parsing algorithms that can reach all dependency trees

Clearly the coverage of a parsing algorithm has direct impact over the search space of the algorithm. For languages with most of the dependency structures projective it is convenient to limit the search space by using a projective parsing algorithm, otherwise it's better to use a non-projective algorithm.

Also if the search space change it does not mean that a non-projective algorithm is slower than a projective one, at least in terms of asymptotic complexity with respect to the length of the sentence.

The coverage is an important characteristic also for graph based parsing where, differently from transition based parsing, it has a huge impact over the performances of the system in terms of speed. For example the second order maximum spanning tree algorithm has complexity $\mathcal{O}(n^3)$ in the projective case while it is NP-hard if we consider non-projective dependency trees [McDonald and Pereira, 2006]. This has pushed researchers to consider other constraints in non-projective parsing in order to improve the coverage of these algorithms maintaining the complexity polynomial. Most of recent works limit the coverage to non-projective trees imposing a fixed maximum gap-degree, allowing to design maximum spanning tree algorithms that work in polynomial time (from $\mathcal{O}(n^5)$ to $\mathcal{O}(n^7)$) [Pitler, Kannan, and Marcus, 2012]. Other interesting works that consider similar constraints are [Satta and Kuhlmann, 2013] and [Pitler, 2014]

4.1.2 Incremental Strategy

We have seen that a parsing algorithm builds incrementally the dependency tree by creating at most an arc at each step. The incremental strategy can be:

- bottom-up
- top-down

A parsing algorithm builds arcs following a bottom-up strategy if all arcs that involves a token w_i as head ($w_i \rightarrow w_k$) are built before the arc in which w_i is a dependent ($w_j \rightarrow w_i$). Otherwise we have a top-down strategy when the arc $w_j \rightarrow w_i$ is created before w_i collects any dependents.

4.1.3 Spurious Ambiguity

In parsing, spurious ambiguity refers to the ambiguity that occurs when different derivations are *equivalent* in term of produced syntactic structure.

Definition 4.7. In a transition based parsing algorithm two derivations (or computations) d and d' are *equivalent* if, when applied to the same configuration c_i, they derive the same configuration c_j

In practice a parsing algorithm with spurious ambiguity can have many equivalent derivations that reach the same dependency tree T_s. It is possible to compare two algorithms in term of *degree of spurious ambiguity*.

Definition 4.8. An algorithm A has a greater *degree of spurious ambiguity* than an algorithm B when the algorithm A has more possible derivations to reach T_s than the algorithm B, for all possible dependency trees T_s reachable from both algorithms.

Sometimes it is clear that an algorithm has an higher degree of spurious ambiguity than another (for example when the set of transitions $\mathcal{T}_A$ is a super-set of the set $\mathcal{T}_B$). Other times it is less clear because an algorithm A can offer more possible derivations than an algorithm B for a sentence s_1 while the opposite for a different sentence s_2. However I will compare the degree of spurious ambiguity by statistically comparing the number of derivations from real samples.

Traditionally, spurious ambiguity has been considered a problem in dependency parsing and in parsing in general. The main reason is that when a grammar is enriched with probabilities the statistical model is defined over the derivation of a structure. And if we have many possible computations the probability of the final structure becomes the marginalized probability over all possible computations. Usually in parsing we get rid of the spurious ambiguity by electing a *canonical derivation* above all possible computations. The main approach in transition based algorithms is to choose the derivation that maintains the stack shorter [Cohen, Gómez-Rodríguez, and Satta, 2012]. This can be easily obtained by introducing a bias over different transitions that start equivalent derivations. In practice the choice of the first transition implies the choice of a derivation over another and follows these principles:

1. prefer a transition that creates an arc
2. if many transitions can create an arc prefer the one that has minimal distance between the head and the dependent
3. prefer a transition that reduces the stack

For most of the algorithms it is easy to extract the canonical derivation. It is equivalent to collect the transitions suggested by a *static oracle* as in algorithm 4.2. I will dedicate the whole chapter 5 to the oracle functions. Until that chapter it is enough to consider the following definition of *static oracle*.

Definition 4.9. Given a dependency tree $T_s = (V_s, A_s)$ and a configuration c_i from which it exists a derivation s.t.:

$$c_i \vdash_{\tau_i} c_{i+i} \vdash_{\tau_{i+1}} \ldots \vdash_{\tau_k} c_f = ([w_0], [\,], A_s)$$

the *static oracle* for an algorithm is a function that retrieve an transition:

$$\text{staticOracle}(T_s, c_i) = \tau_i$$

where τ_i follow the principles of a canonical derivation.

Note that in many works that does not consider different types of oracles the *static oracle* is simply called *oracle*. In chapter 5 we will see how it is possible to take advantage from the spurious ambiguity by carefully designing a non deterministic oracle and how design a dynamic oracle in order to avoid error propagation in parsing algorithms.

4.1.4 Arc-Standard

The Arc-Standard algorithm is one of the simplest and widely used parsing algorithms. It was firstly used in dependency parsing in [Yamada and Matsumoto, 2003]. In table 4.1 we can see that there are two transitions that can create an arc and one that moves elements from the buffer into the stack. A new arc can be created only when the head and the dependent are adjacent and at the top of the stack. Given a sentence of length n the arc-standard algorithm requires exactly $2n-1$ transitions to retrieve a dependency tree, because all nodes need to be pushed into the stack and for each node, except the root, an arc needs to be built.

Algorithm 4.2 Extract the canonical derivation by using a static oracle

Input:

sentence $s = w_0 w_1 \cdots w_n$

syntactically correct dependency tree T_s

staticOracle$_A$ ▷ static oracle for algorithm A

Output:

canonical derivation d

1: $d \leftarrow [\,]$ ▷ initialize the derivation as an empty sequence
2: $c = (\sigma, \beta, A) \leftarrow ([\,], [w_0, \ldots, w_n], \varnothing)$
3: **while** $|\sigma| > 1 \vee |\beta| > 0$ **do** ▷ while c is not final do
4: $\tau_o \leftarrow \text{staticOracle}(T_s, c)$
5: $d \leftarrow d + [\tau_o]$ ▷ update the derivation
6: $c' = (\sigma', \beta', A') \leftarrow \text{apply}(\tau_o, c)$
7: $c = (\sigma, \beta, A) \leftarrow c'$ ▷ update the current configuration
8: **return** d

Transition	$c = (\sigma, \beta, A)$	$c' = (\sigma', \beta', A')$	Preconditions
LEFT-ARC	$(\sigma\|w_i\|w_j, \beta, A)$	$(\sigma\|w_j, \beta, A \cup \{w_j, l, w_i\})$	$w_i \neq$ -ROOT-
RIGHT-ARC	$(\sigma\|w_i\|w_j, \beta, A)$	$(\sigma\|w_i, \beta, A \cup \{w_i, l, w_j\})$	
SHIFT	$(\sigma, w_i\|\beta, A)$	$(\sigma\|w_i, \beta, A)$	

Table 4.1: Transitions in the Arc-Standard parsing algorithm

Coverage: given a sentence s the algorithm can reach all possible projective dependency trees but no non-projective tree.

Incremental Strategy: the algorithm implements a pure bottom-up strategy. When a new arc is created, the dependent is removed from the stack so it can't be used to create other arcs.

Spurious ambiguity: a node can collect independently left and right dependents rising a spurious ambiguity that follows the pattern $[\text{LA}, \text{SH}, \ldots, \text{RA}]$ or $[\text{SH}, \ldots, \text{RA}, \text{LA}]$.

Example 4.10. There are 7 possible derivations to build the dependency tree in

figure 4.1; two of them are:

$$d_1 = \textsc{sh}(0),\ \textsc{sh}(1),\ \textsc{sh}(2),\ \textsc{sh}(3),\ \textsc{la}(2,3),\ \textsc{la}(1,3),\ \textsc{sh}(4),\ \textsc{sh}(5),\ \textsc{sh}(6),\ \textsc{la}(5,6),\ \textsc{ra}(4,6),\ \textsc{ra}(3,4),\ \textsc{sh}(7),\ \textsc{ra}(3,7),\ \textsc{ra}(0,3)$$

$$d_2 = \textsc{sh}(0),\ \textsc{sh}(1),\ \textsc{sh}(2),\ \textsc{sh}(3),\ \textsc{la}(2,3),\ \textsc{sh}(4),\ \textsc{sh}(5),\ \textsc{sh}(6),\ \textsc{la}(5,6),\ \textsc{ra}(4,6),\ \textsc{ra}(3,4),\ \textsc{la}(1,3),\ \textsc{sh}(7),\ \textsc{ra}(3,7),\ \textsc{ra}(0,3)$$

d_1 is the canonical derivation. d_2 is not the canonical derivation because considering the 4-th transition (the first difference between d_1 and d_2) for d_1 we have a transition that creates an arc, while for d_2 we have a transition that does not create an arc.

We can see the different pattern: $[\textsc{la}(1,3), \textsc{sh}(4), \ldots, \textsc{ra}(3,4)]$ for the derivation d_1 and $[\textsc{sh}(4), \ldots, \textsc{ra}(3,4), \textsc{la}(1,3)]$ for d_2.

Both derivations build the dependency tree in a bottom-up fashion, for example w_3 has to collect all its dependents before the creation of the arc $w_0 \rightarrow w_3$. Note also that the left dependents of a node are always collected in the same order: first w_2 and later w_1. The same for the right dependents w_4 and w_7. Otherwise the order between the attachment of left and right dependents can be mixed: d_1 connects w_1 before w_4 while d_2 works the other way around.

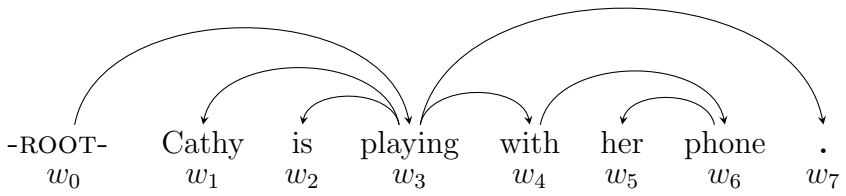


Figure 4.1: Example of dependency tree

4.1.5 Arc-Eager

The arc-Eager algorithm was firstly used in dependency parsing in [Nivre, 2004]. In table 4.2 we can see that, differently from the Arc-Standard algorithm, the transitions that create a new arc involve the topmost node of the stack and the first node in the buffer. Note also that the RIGHT-ARC transition does not remove

the dependent but pushes it into the stack. This implies the need of an extra extra transition REDUCE that simply remove the topmost element into the stack.

The precondition over the LEFT-ARC and REDUCE transitions check the existence of an arc that has the topmost element of the stack as dependent: for LEFT-ARC it is necessary to avoid that a token has two heads, while for REDUCE it is necessary to avoid that a token is removed without a head. In order to obtain a well formed dependency tree we have also to guarantee that the last token of a sentence will be pushed into the stack after all other tokens have received a head. Some implementations of the Arc-Eager algorithm do not take care of the last token or about the precondition for REDUCE, the main reason is to limit the error propagation during parsing but the obtained T_s can be a forest.

The exact number of transitions can slightly change from an implementation to another but is bounded by $2n - 1$ like in the Arc-Standard algorithm.

Transition	$c = (\sigma, \beta, A)$	$c' = (\sigma', \beta', A')$	Preconditions
LEFT-ARC	$(\sigma\|w_i, w_j\|\beta, A)$	$(\sigma, w_j\|\beta, A \cup \{w_j, l, w_i\})$	$w_i \neq$ -ROOT- $(w_k, l', w_i) \notin A$
RIGHT-ARC	$(\sigma\|w_i, w_j\|\beta, A)$	$(\sigma\|w_i\|w_j, \beta, A \cup \{w_i, l, w_j\})$	
REDUCE	$(\sigma\|w_i, \beta, A)$	(σ, β, A)	$(w_j, l, w_i) \in A$
SHIFT	$(\sigma, w_i\|\beta, A)$	$(\sigma\|w_i, \beta, A)$	

Table 4.2: Transitions in the Arc-Eager parsing algorithm

Coverage: given a sentence s the algorithm can reach all possible projective dependency trees but no non-projective tree.

Incremental Strategy: the algorithm implements a bottom-up strategy for the left dependents and a top-down strategy for the right ones. This produces derivations that statistically maintains the stack shorter than the Arc-Standard. The top-down strategy for right dependents gives to the algorithm an interesting incremental behaviour in the creation of arcs $w_i \rightarrow w_j$ with $i < j$, but it requires a

REDUCE transition that can force early decisions in parsing.

Spurious ambiguity: when a node is in the buffer it cannot collect any right dependents, otherwise when it is pushed into the stack it cannot collect left dependents. This implies that to reach a dependency tree T_s the algorithm has to respect a strict order for arcs' creation. However the Arc-Eager algorithm has spurious ambiguity that follows the pattern $[\text{RE}, \text{SH}, \ldots]$ or $[\text{SH}, \ldots, \text{RE}]$.

Example 4.11. There are 3 possible derivations to build the dependency tree in figure 4.2; two of them are:

$$\begin{aligned} d_1 = {} & \text{SH}(0),\ \text{SH}(1),\ \text{SH}(2),\ \text{LA}(2,3),\ \text{LA}(1,3),\ \text{RA}(0,3),\ \text{RA}(3,4),\ \text{RA}(4,5), \\ & \text{RE}(5),\ \text{RE}(4),\ \text{SH}(6),\ \text{LA}(6,7),\ \text{RA}(3,7),\ \text{RE}(7),\ \text{RA}(3,8),\ \text{RE}(8),\ \text{RE}(3) \\ d_1 = {} & \text{SH}(0),\ \text{SH}(1),\ \text{SH}(2),\ \text{LA}(2,3),\ \text{LA}(1,3),\ \text{RA}(0,3),\ \text{RA}(3,4),\ \text{RA}(4,5), \\ & \text{RE}(5),\ \text{SH}(6),\ \text{LA}(6,7),\ \text{RE}(4),\ \text{RA}(3,7),\ \text{RE}(7),\ \text{RA}(3,8),\ \text{RE}(8),\ \text{RE}(3) \end{aligned}$$

d_1 is the canonical derivation, d_2 cannot be the canonical derivation because the 10-th transition (the first difference between d_1 and d_2) is a SHIFT while for d_1 it is a REDUCE. The different pattern between these two derivations is: $[\text{RE}(4), \text{SH}(6), \ldots]$ for d_1 and $[\text{SH}(6), \ldots, \text{RE}(4)]$ for d_2. Note that the LEFT-ARC and RIGHT-ARC transitions are exactly in the same order in both cases.

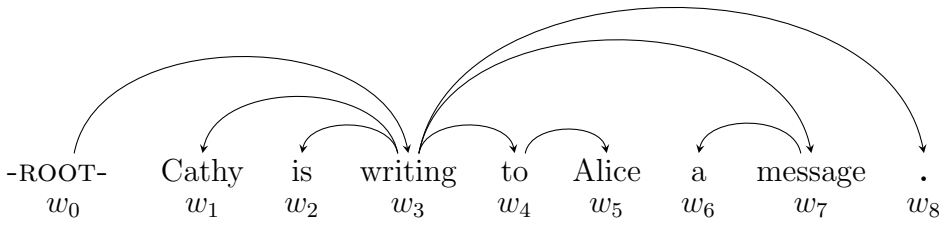


Figure 4.2: Example of dependency tree

4.1.6 Attardi's algorithm (simplified)

The Attardi's algorithm is the first transition based algorithm designed to deal with non-projective dependency trees. Here I describe a simplified version of the one in [Attardi, 2006]. In table 4.3 there are the five transitions that characterize the algorithm: SHIFT, LEFT-ARC$_1$ and RIGHT-ARC$_1$ are the same as for the Arc-Standard algorithm while LEFT-ARC$_2$ and RIGHT-ARC$_2$ are the ones that allow to build non-projective arcs. These new transitions create arcs that involve the topmost and the 3rd topmost token into the stack without the need that the two elements are adjacent in the stack.

Transition	$c = (\sigma, \beta, A)$	$c' = (\sigma', \beta', A')$	Preconditions
LEFT-ARC$_1$	$(\sigma\|w_i\|w_j, \beta, A)$	$(\sigma\|w_j, \beta, A \cup \{w_j, l, w_i\})$	$w_i \neq$ -ROOT-
RIGHT-ARC$_1$	$(\sigma\|w_i\|w_j, \beta, A)$	$(\sigma\|w_i, \beta, A \cup \{w_i, l, w_j\})$	
LEFT-ARC$_2$	$(\sigma\|w_i\|w_j\|w_k, \beta, A)$	$(\sigma\|w_j\|w_k, \beta, A \cup \{w_k, l, w_i\})$	$w_i \neq$ -ROOT-
RIGHT-ARC$_2$	$(\sigma\|w_i\|w_j\|w_k, \beta, A)$	$(\sigma\|w_i\|w_j, \beta, A \cup \{w_i, l, w_k\})$	
SHIFT	$(\sigma, w_i\|\beta, A)$	$(\sigma\|w_i, \beta, A)$	

Table 4.3: Transitions in a simplified version of the Attardi's parsing algorithm

Coverage: This simplified version of Attardi's algorithm can reach many non-projective dependency trees but not all of them. Unfortunately it's hard to define a property that identifies which type of dependency trees are reachable or not. In figure 4.4 I provide an example of a dependency tree that cannot be reached. The extended version of the algorithm uses an auxiliary stack that guaranties the coverage of all possible non-projective dependency trees but complicates a lot the model that needs to be trained. However this version already reaches a good coverage for many languages, maintaining simple the algorithm.

Incremental Strategy: like the Arc-Standard, the Attardi's algorithm implements a pure bottom-up strategy. When a new arc is created it is removed from

the stack so it cannot be used to create other arcs.

Spurious ambiguity: in terms of spurious ambiguity it has a behaviour similar to the Arc-Standard plus the ambiguity that derives from two LEFT-ARC$_1$ and LEFT-ARC$_2$ transitions where the topmost element into the stack can take 2 different dependents.

Example 4.12. The canonical derivation to build the dependency tree in figure 4.3 is:

$$\begin{aligned} d_1 = \ & \textsc{sh}(0),\ \textsc{sh}(1),\ \textsc{sh}(2),\ \textsc{la}_1(1,2),\ \textsc{sh}(3),\ \textsc{ra}_1(2,3),\ \textsc{sh}(4),\ \textsc{sh}(5), \\ & \textsc{la}_1(4,5),\ \textsc{sh}(6),\ \textsc{sh}(7),\ \textsc{sh}(8),\ \textsc{la}_1(7,8),\ \textsc{ra}_1(6,8),\ \textsc{ra}_2(2,6), \\ & \textsc{sh}(9),\ \textsc{sh}(10),\ \textsc{ra}_1(9,10),\ \textsc{ra}_1(5,9),\ \textsc{la}_1(2,5),\ \textsc{sh}(11), \\ & \textsc{ra}_1(2,11),\ \textsc{ra}_1(0,2) \end{aligned}$$

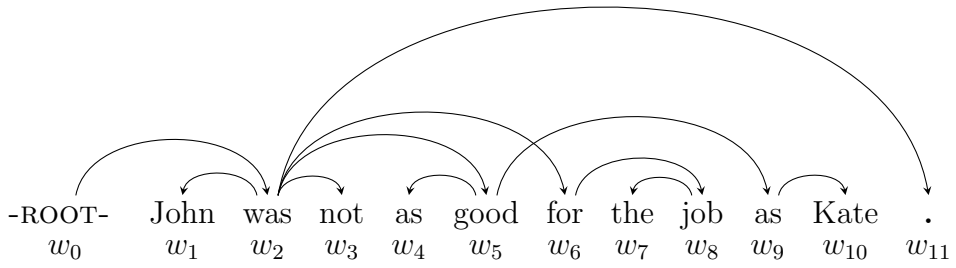


Figure 4.3: Example of dependency tree

4.1.7 Swapping Arc-Standard algorithm

This algorithm is described in [Nivre, 2009]. The basic idea is that by reordering the tokens of a non-projective dependency tree it is always possible to obtain a projective dependency tree. This is straightforward if we consider that a tree is always a planar graph. The algorithm has the same transitions of the Arc-Standard plus a SWAP transition. At parsing time the new transition works like a bubble sort

algorithm by reordering the tokens into the sentence. The precondition of SWAP is necessary to avoid infinite loops into the algorithm's iterations. The complexity of the algorithm is $\mathcal{O}(n^2)$ but given that the reordering necessity is limited in a real sentences the algorithm works in expected linear time.

Transition	$c = (\sigma, \beta, A)$	$c' = (\sigma', \beta', A')$	Preconditions
LEFT-ARC	$(\sigma\|w_i\|w_j, \beta, A)$	$(\sigma\|w_j, \beta, A \cup \{w_j, l, w_i\})$	$w_i \neq$ -ROOT-
RIGHT-ARC	$(\sigma\|w_i\|w_j, \beta, A)$	$(\sigma\|w_i, \beta, A \cup \{w_i, l, w_j\})$	
SHIFT	$(\sigma, w_i\|\beta, A)$	$(\sigma\|w_i, \beta, A)$	
SWAP	$(\sigma\|w_i\|w_j, \beta, A)$	$(\sigma\|w_j, w_i\|\beta, A)$	$i < j$

Table 4.4: Transitions in the Swapping Arc-Standard parsing algorithm

Coverage: The Swapping algorithm can reach all possible dependency trees, projective or not.

Incremental Strategy: like the Arc-Standard, this algorithm implements a pure bottom-up strategy. When a new arc is created it is removed from the stack so it cannot be used to create other arcs.

Spurious ambiguity: the left/right dependents of a node can be reorder by the SWAP transition given. In the Arc-Standard algorithm we have seen that the left dependents has to be collected in order

This gives an higher degree of spurious ambiguity respect to the Arc-Standard algorithm. The canonical derivation prefers the swap transition if the two topmost nodes into the stack are not in projective order. The projective order is obtained by reordering the sentence to obtain a projective dependency tree without changing the relative order of the dependents of each node.

Example 4.13. The canonical derivation to build the dependency tree in figure 4.3

is:

$$
\begin{aligned}
d_1 = \ & \textsc{sh}(0),\ \textsc{sh}(1),\ \textsc{sh}(2),\ \textsc{la}(1,2),\ \textsc{sh}(3),\ \textsc{sh}(4),\ \textsc{sh}(5),\ \textsc{swap}(4,5), \\
& \textsc{swap}(3,5),\ \textsc{sh}(3),\ \textsc{sh}(4),\ \textsc{sh}(6),\ \textsc{sh}(4,6),\ \textsc{sh}(3,6),\ \textsc{sh}(3),\ \textsc{sh}(4), \\
& \textsc{sh}(7),\ \textsc{swap}(4,7),\ \textsc{swap}(3,7),\ \textsc{la}(6,7),\ \textsc{ra}(5,7),\ \textsc{ra}(2,5),\ \textsc{sh}(3), \\
& \textsc{la}(2,3),\ \textsc{sh}(4),\ \textsc{sh}(8),\ \textsc{ra}(4,8),\ \textsc{ra}(3,4),\ \textsc{sh}(9),\ \textsc{ra}(3,9),\ \textsc{ra}(0,3)
\end{aligned}
$$

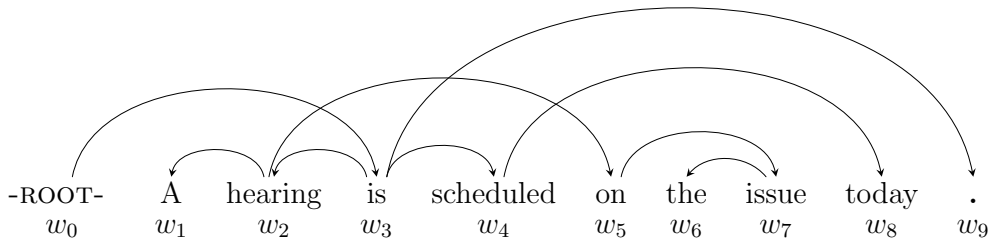


Figure 4.4: Example of dependency tree

4.1.8 From Unlabelled to Labelled

I described all algorithms in case of unlabelled dependency parsing. However extend them to the labelled case is simple. We have seen that each parsing algorithm has some transition that creates an arc. In order to obtain a labelled parsing algorithm we extend each transition that creates an arc with as many transitions as the number of possible labels. In this way each transition that creates an arc has associated a specific label.

Example 4.14. Consider the set of possible labels $L = \{l_1, l_2, l_3\}$ and the arc-standard algorithm. The set of possible transitions in the unlabelled case is $\mathcal{T} = \{\textsc{la}, \textsc{ra}, \textsc{sh}\}$. Otherwise the set of possible transitions in the labelled case is $\mathcal{T}_L = \{\textsc{la}_{l_1}, \textsc{la}_{l_2}, \textsc{la}_{l_3}, \textsc{ra}_{l_1}, \textsc{ra}_{l_2}, \textsc{ra}_{l_3}, \textsc{sh}\}$. Where given the same configuration all transitions $\textsc{la}_{l_1}, \textsc{la}_{l_2}, \textsc{la}_{l_3}$ create an arc with the same head and the same dependent but with different label.

4.2 Train a Model

As we have seen in section 4.1, the parsing algorithm is non deterministic and the model chooses which transition the parsing algorithm will apply to proceed.

Consider a dependency tree $T_s = (V_s, A_s)$ for the sentence $s = w_0 w_1 \dots w_n$ and a parsing algorithm with a set of transitions $\mathcal{T}$ in which T_s is reachable. The algorithm incrementally builds the tree through a derivation where each transition is applied to a configuration (definition 4.6):

$$c_0 \vdash_{\tau_0} c_1 \vdash_{\tau_1} c_2 \vdash_{\tau_2} \cdots \vdash_{\tau_k - 1} c_k \vdash_{\tau_k} c_f$$
$$c_0 = ([\,], [w_0 \dots w_n], \varnothing)$$
$$c_f = ([w_0], [\,], A_f), \text{where } A_f = A_s$$

In practice the task of the *model* is to choose for each configuration the transition that allow the parsing algorithm to reach the final configuration $c_f = ([w_0], [\,], A_f)$ that represents the syntactically correct dependency tree ($A_f = A_s$). From this point of view the model is a classifier that maps a configuration c into a transition $\tau \in \mathcal{T}$.

The classification is a standard problem in machine learning where, in a supervised setting, we need a training data set (instances labelled with their correct class). In this case the instances should be a configuration and the class one possible transition $\tau_i \in \mathcal{T}$. However the data sets available are treebanks: a list of sentences with the respective syntactically correct dependency trees. We can treat the problem in 2 ways:

- standard learning
- on-line learning

Another important aspect is the feature representation of our instances (the configurations). The details of the chosen feature representation depends on the specific machine learning approach, but it follows similar principles described in section 4.2.3

4.2.1 Standard Learning

In standard (off-line) learning we need a data-set over which we train a model. Using the algorithm 4.2, we extract the canonical derivation $d = \tau_0 \tau_1 \dots \tau_k$ that

reaches the corresponding syntactically correct dependency tree $T_s = (V_s, A_s)$:

$$c_0 \vdash_{\tau_0} c_1 \vdash_{\tau_1} \ldots c_k \vdash_{\tau_k} c_f = ([w_0], [\,], A_s)$$

The couples constituted by a configuration and the relative transition are the training samples extracted from the sentence:

$$(c_0, \tau_0), (c_1, \tau_1), \ldots, (c_k, \tau_k)$$

Doing that for each sentence of the treebank we covert the syntactically correct dependency trees into a set D' of pairs:

$$D' = \{c_i, \tau_i\}_{d=0}^{|D'|}$$

Example 4.15. Considering the Arc-Standard algorithm we have three possible transitions (LEFT-ARC, RIGHT-ARC, SHIFT). A derivation d for a sentence $s = w_0 w_1 \ldots w_n$ is composed by $2n$ transitions. If we consider the English data set the average length of a sentence (excluding the -ROOT- node) is 23.85 and the number of sentences usually used for training a model for English[2] are 39 831. So the training samples are about 950000. The number of sentences into the available treebanks change with the language but this number gives a good idea about the size of the data set in terms of pairs (c, τ).

Note that the training set obtained in this way is specific for a transition based algorithm, indeed the configurations and the transitions are meaningless for another algorithm.

Using a standard machine learning approach we can now consider each pair (c, τ) an independent training sample and use whichever classifier, where the set of possible classes is the set of possible transitions of the parsing algorithm. The most used techniques are linear classifiers (like the perceptron algorithm), Maximum Entropy Models and Neural Networks. There are some systems that use Memory Based Learning systems but I found these techniques in contrast with the parsing efficiency that in general we want to reach by using a transition based algorithm. Great results have been reached also by using Support Vector Machines, however the training can be computationally intensive for large training sets.

[2]The values are taken considering the sections of the Penn-Treebank generally used for training (from 2 to 21)

Advantages of Standard Learning Converting the treebank into a static set of samples allows to easily test many different types of learning algorithms. Representing the data in couples (instance,class) is standard in machine learning and we can choose above many different classifier algorithms. There are also many available tools for classifications tasks, so we can easily train and test different learning algorithms in few hours. As we will see a great impact on the precision of a transition based algorithm is given by the used feature representation. With a standard approach it is simple to test different features and eventually try new representations like in [Chen and Manning, 2014]. We can also easily preprocess the sentences to add new information to the training/test data like in [Ambati, Deoskar, and Steedman, 2013].

Disavantage of Standard Learning The main problem is that we represent the data as a flat set of configurations and relative transitions. In this way we loose information about the sequentiality that they have at parsing time. What we are really training by using this approach is a model that, given a configuration c_i, retrieves the most probable transition to obtain the following configuration c_{i+1} into the canonical derivation for a tree T_s. But at parsing time the configuration c_i is reached only if the previous transitions are well predicted:

$$\mathbf{c_0} \vdash_{\tau_0} \mathbf{c_1} \vdash_{\tau_1} \ldots \mathbf{c_{i-1}} \vdash_{\tau_{i-1}} c_i \vdash_{\tau_i} \ldots \vdash_{\tau_k} c_f$$

Informally we can say that the previous transitions are more important because without them we can loose the path of the canonical derivation.

4.2.2 On-line Learning

On-line learning is used when the data becomes available in a sequential fashion, in order to determine a mapping from the sample to the corresponding class. The model is updated after the arrival of every new training sample. We use the general parsing algorithm 4.1 to provide the data instances (configurations) in a sequential order and the *oracle* function to get the correct classes (transitions). In algorithm 4.3 there is an on-line training algorithm that updates the model only when the model's prediction is wrong.

The model is initialized with the possible classes (the set of possible transitions $\mathcal{T}$ of a parsing algorithm). Then the algorithm process all trees into the treebank.

Each tree is built by using the parsing algorithm but each transition predicted by the model is compared with the one retrieved by the oracle. If the predicted transition is not the one retrieved by the static oracle we update the model.

In the on-line setting, the most used machine learning technique is the averaged perceptron [Freund and Schapire, 1999]. Personally I use the averaged perceptron algorithm for structure prediction as described in [Daumé III, 2006].

After a model update there are three way to proceed:

1. early update
2. aggressive update
3. correct and go on

Early update is used in algorithm 4.3 where at line 13 the algorithm return to line 2. Practically after an update the algorithm proceed with another sentence. The basic idea is that if the parsing algorithm fails to reach a configuration it is not significant to proceed. In this way the algorithm gives more importance to the prediction of the early transitions of a derivation.

Aggressive update substitutes line 13 with *goto line 4*. In this configuration, instead of skip the sentence, we continually update the model until it does not retrieve the correct prediction for the current configuration. The advantage of this technique is that we use the whole sentence during training. Otherwise this aggressive strategy implies many updates with possible undesirable oscillations into the model parameters.

Correct and go on is show in algorithm 4.4 where, after the model update the algorithm correct the predicted transition with the oracle's suggestion and proceed. This is my favourite technique because we can use the whole sentence during training without the model oscillations of the aggressive update. As far as I know the first works in dependency parsing that uses this way to update the model were [Sartorio, Satta, and Nivre, 2013] and [Goldberg and Nivre, 2012]. The reason that motivates me and the other authors to use this technique is that this way to proceed is useful in case of non-deterministic oracle as in [Sartorio, Satta,

Algorithm 4.3 On-line Training Algorithm (early update)

Input:

treeBank= $[(s_1, T_{s_1}), (s_2, T_{s_2}), \ldots, (s_m, T_{s_m})]$

Output:

model ▷ return a trainined model

1: model $\leftarrow$ newModel($\mathcal{T}$) ▷ initialization of the model
▷ one class for each $\tau \in \mathcal{T}$

2: **for each** s, T_s **in** treeBank **do**

3: $c = (\sigma, \beta, A) \leftarrow ([\,], s, \varnothing)$ ▷ initialize starting configuration

4: **while** $|\sigma| > 1 \vee |\beta| > 0$ **do** ▷ while c is not final do

5: $\mathcal{T}' \leftarrow \varnothing$

6: **for each** τ in $\mathcal{T}$ **do** ▷ select the applicable transitions

7: **if** applicable(τ,c) **then**

8: $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{\tau\}$

9: $\tau \leftarrow$ model.giveBestTransition($\mathcal{T}', c$)

10: $\tau_o \leftarrow$ staticOracle(T_s, c)

11: **if** $\tau \neq \tau_o$ **then** ▷ model retrieve a bad prediction w.r.t. oracle

12: model.update($\mathcal{T}', c, \tau_o$) ▷ update the model

13: **goto** line 2

14: $c' = (\sigma', \beta', A') \leftarrow$ apply(τ, c) ▷ apply(τ, c) returns c' s.t. $c \vdash_\tau c'$

15: $c = (\sigma, \beta, A) \leftarrow c'$ ▷ update the current configuration

16: **return** model

and Nivre, 2013] and a similar approach is fundamental in case of dynamic oracle like in [Goldberg and Nivre, 2012]

Advantages of on-line Learning The advantage of using a on-line learning system is that during training we use the exact strategy that we use at parsing time. In other words, differently from the standard learning approach we do not loose the sequentiality of a derivation.

In an on-line setting it is easy to limit the choice of the model above the only applicable transitions. In this way the update involve only applicable transitions without learning useless constraints about impossible transitions. In a standard learning setting it is possible to do something like that, specially if we use binary classifiers in an one-versus-all strategy. Anyway it is not straightforward as in an

Algorithm 4.4 On-line Training Algorithm (correct and go on)

Input:

treeBank= $[(s_1, T_{s_1}), (s_2, T_{s_2}), \ldots, (s_m, T_{s_m})]$

Output:

model ▷ return a trainined model

1: model ← newModel($\mathcal{T}$) ▷ initialization of the model
▷ one class for each $\tau \in \mathcal{T}$
2: **for each** s, T_s **in** treeBank **do**
3: $c = (\sigma, \beta, A) \leftarrow ([\,], s, \varnothing)$ ▷ initialize starting configuration
4: **while** $|\sigma| > 1 \vee |\beta| > 0$ **do** ▷ while c is not final do
5: $\mathcal{T}' \leftarrow \varnothing$
6: **for each** τ in $\mathcal{T}$ **do** ▷ select the applicable transitions
7: **if** applicable(τ,c) **then**
8: $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{\tau\}$
9: $\tau \leftarrow$ model.giveBestTransition($\mathcal{T}', c$)
10: $\tau_o \leftarrow$ staticOracle(T_s, c)
11: **if** $\tau \neq \tau_o$ **then** ▷ model retrieve a bad prediction w.r.t. oracle
12: model.update($\mathcal{T}', c, \tau_o$) ▷ update the model
13: $\tau \leftarrow \tau_o$
14: $c' = (\sigma', \beta', A') \leftarrow$ apply(τ, c) ▷ apply(τ, c) returns c' s.t. $c \vdash_\tau c'$
15: $c = (\sigma, \beta, A) \leftarrow c'$ ▷ update the current configuration
16: **return** model

on-line setting and as far as I know it is not used in practice.

Disavantage of on-line learning Obviously we have to use a machine learning approach that allows an on-line strategy and this limits the possible choices. In on line learning the model's parameters are not globally optimized over all training samples, so we usually need many iterations over the training samples in order to obtain a model that converges to stable parameters. The order in which the sentences are processed during training may have impact over the model parameters and consequently to the performances of the model. To obtain consistent results we should try different random reordering by creating different models and testing them separately.

4.2.3 Feature Representation

We have seen that both approaches use as training sample couples of configurations and correct transition: (c, τ). The configuration has a complex form: a stack, a buffer and a set of already built arcs, all of them bounden only by the length of the sentence. In order to treat the learning problem with standard machine learning techniques we need to introduce a level of abstraction over the practically infinite set of possible configurations. We introduce a feature function f that maps a configuration c into a n-dimensional vector:

$$f(c) = v$$

In general the feature function can use arbitrary attributes of the configuration. There are complex feature functions that consider extra information obtained by preprocessing the data set or that use global information about the sentence. However many transition based parsers obtain good results by using a feature function that consider simple features of few tokens into a configuration. These features are properties of tokens[3] in particular positions of the configuration such as the topmost tokens into the stack and the first elements of the buffer. Given the incremental behaviour of the parsing algorithm it is possible to include information about the already built structure (the set of arcs into a configuration). Usually the left/right most dependents of the top most tokens into the stack are important to discriminate the class of a configuration, so we include features like the POS and the arc label of such dependents.

Note that the number of usable feature depends on the learning system adopted. Indeed in parsing we already have a huge number of samples, using a big number of features in learning algorithms like support vector machines or memory based classifiers can be impracticable.

Some machine learning algorithms automatically combine the features extracted by the configuration. For example the support vector machines usually use a kernel or a neural networks use (in some sense) the hidden layers. Otherwise learning algorithms like the averaged perceptron needs a manually designed feature template to eventually combine the simple features extracted from a configuration.

[3]In parsing we usually consider the sentence preprocessed by a POS (part-of-speech) tagger so for each token of the sentence we know the form, the POS, and eventually the lemma.

Chapter 5

Oracles

In the previous chapter we briefly introduced the static oracle and we saw that it is strictly related to the canonical derivation. The oracle is useful in a standard learning approach to convert the treebank into training samples. In an on-line learning setting the oracle function is crucial to decide when the model predicts a wrong transition and needs to be updated. In this chapter I will focus on the on-line learning approach and I will give a more general interpretation of the oracle function.

Specifically I will consider three types of oracle function:

1. static oracle
2. non-deterministic oracle
3. dynamic oracle

Given a sentence of length n a careful implementation of the static and non-deterministic oracles leads to a constant time complexity of the oracle function $\mathcal{O}(1)$ but it requires a preprocessing of the sentence with time complexity $\mathcal{O}(n)$. For each token of the sentence the preprocessing needs to extract and store some information from the gold (syntactically correct) dependency tree: the parent, the left-most-child and the right-most-child tokens into the gold dependency tree. Some implementations, instead of the left-most-child and right-most-child, stores the number of dependents of each token. In case of dynamic oracle the computational complexity depends on the specific parsing algorithm and we will see it case by case.

It is important to remark that the complexity of the oracle has impact only on training time. Unless we have an exponential or high degree polynomial complexity, which implies extremely long training time, the oracle complexity is not critical from a practical point of view.

I will use the mostly standard notation:

- $s_0, s_1, s_2, \ldots$ elements of the stack starting from the topmost token into the stack (s_0);
- $b_0, b_1, \ldots$, the first tokens into the buffer,
- $T_G = (V, A_G)$, the gold (syntactically correct) dependency tree for the considered sentence where the A_G is the set of correct arcs.
- $c = (\sigma, \beta, A)$, the usual configuration where A is the set of already built arcs
- $\text{P}(w_i)$, to indicate the parent node of the node w_i

Considering that the position of a token into the sentence uniquely identifies it, I will often treat the tokens like numbers that identify the token position into the stack. So I will use i to identify the token w_i.

In this chapter I will consider the Arc-Standard, Arc-Eager and Attardi's algorithm. Firstly I will describe the static oracles and the non-deterministic oracles for such algorithms, then I will describe the dynamic oracle. At the moment I don't know if it is possible to implement a dynamic oracle for the swap algorithm.

5.1 Static Oracle

Until a couple of years ago the static oracle was the only defined oracle function. So it was simply called oracle. In the previous chapter we saw the following definition of static oracle.

Definition 5.1. Given a configuration c_i from which it is possible to reach the gold dependency tree $T_G = (V, A_G)$ the *static oracle* is a function that retrieves a transition:

$$\text{staticOracle}(T_G, c_i) = \tau_o$$

where τ_o begins a derivation that reaches T_G and follows the principles of a canonical derivation in 4.1.3.

The static oracle is simple and easy to implement. Given a parsing algorithm we can define a set of conditions for each possible transition. In algorithm 5.1 given a configuration and the correct dependency tree, the static oracle takes in consideration the transitions of a parsing algorithm in a fixed order and returns the first transition in which the conditions are satisfied. The order in which the transitions are examined is based on the canonical derivation principles: first we consider the transitions that create an arc and last the SHIFT transition that increases the length of the stack.

Algorithm 5.1 Static Oracle Algorithm

Input:
configuration c
gold dependency tree T_g
Output:
transition τ_o ▷ return a transition
1: $\mathcal{T}_a \leftarrow [\tau_0, \tau_1, \ldots, \text{SHIFT}]$ ▷ ordered sequence of transitions for the algorithm
▷ that follows the canonical derivation principles
2: **for each** τ_i **in** $\mathcal{T}_a$ **do**
3: **if** conditions(T_g, c, τ_i) **then**
4: $\tau_o \leftarrow \tau_i$
5: **break**
6: **return** τ_o

For each algorithm I will present a table with the conditions checked by the oracle for all transitions. The rows in the tables follow the order in which the relative transitions are considered by the static oracle.

5.1.1 Arc-Standard Static Oracle

In table 5.1 I report the conditions checked by the static oracle for the Arc-Standard algorithm. We can switch LEFT-ARC and RIGHT-ARC in the sequence $\mathcal{T}_a$ but it's important that the shift transition will be returned by the oracle if and only if the other conditions are not satisfied. Considering that the arc-standard algorithm can

create an arc only between the two topmost elements into the stack and considering the projectivity constrain, for the transition LEFT-ARC we do not need to check if the node has already taken all its dependents.

Transition	Oracle's Condition
LEFT-ARC	$(s_0 \rightarrow s_1) \in A_g$
RIGHT-ARC	$(s_1 \rightarrow s_0) \in A_g$ and $\forall w_i \in V, \nexists\, (s_0 \rightarrow w_i) \in A_g \setminus A$
SHIFT	no conditions

Table 5.1: Static oracle conditions for Arc-Standard algorithm

5.1.2 Arc-Eager Static Oracle

In table 5.2 I report the conditions checked by the static oracle for the Arc-Eager algorithm. Similarly to the Arc-Standard algorithm the transitions RIGHT-ARC, LEFT-ARC and REDUCE can be exchanged. The LEFT-ARC and RIGHT-ARC transitions only check that the arc created by the transition exists into the gold dependency tree. Indeed, considering the bottom-up strategy for left dependents and the top-down strategy for right dependents, there is no need for other conditions.

5.1.3 Attardi's algorithm Static Oracle

For the simplified version of the Attardi's algorithm the conditions checked by the static oracle are in table 5.3. Following the most used definition of the canonical computation the transitions LEFT-ARC$_2$ must be evaluated after LEFT-ARC$_1$.

Transition	Oracle's Conditions
LEFT-ARC	$(b_0 \rightarrow s_0) \in A_g$
RIGHT-ARC	$(s_0 \rightarrow b_0) \in A_g$
REDUCE	$\exists\, w_i \in \sigma \mid (w_i \rightarrow s_0) \in A$ and $\forall\, w_j \in \beta, \nexists\, (s_0 \rightarrow w_i) \in A_g$
SHIFT	no conditions

Table 5.2: Static oracle conditions for Arc-Eager algorithm

Transition	Oracle's Condition
LEFT-ARC$_1$	$(s_0 \rightarrow s_1) \in A_g$ and $\forall w_i \in V, \nexists\, (s_1 \rightarrow w_i) \in A_g \setminus A$
RIGHT-ARC$_1$	$(s_1 \rightarrow s_0) \in A_g$ and $\forall w_i \in V, \nexists\, (s_0 \rightarrow w_i) \in A_g \setminus A$
LEFT-ARC$_2$	$(s_0 \rightarrow s_2) \in A_g$ and $\forall w_i \in V, \nexists\, (s_2 \rightarrow w_i) \in A_g \setminus A$
RIGHT-ARC$_1$	$(s_2 \rightarrow s_0) \in A_g$ and $\forall w_i \in V, \nexists\, (s_0 \rightarrow w_i) \in A_g \setminus A$
SHIFT	no conditions

Table 5.3: Static oracle conditions for the Attardi's algorithm

5.2 Non-Deterministic Oracle

The non-deterministic oracle was firstly introduced in [Goldberg and Nivre, 2012] and in [Sartorio, Satta, and Nivre, 2013]. Despite the fact that the second paper was published some months after the first paper, the authors of the two papers have independently reached the idea of a non-deterministic oracle following two different

motivations. The authors of the first paper were looking for a way to reduce error propagation and as intermediate step they developed a non-deterministic oracle for the Arc-Eager algorithm. On the other hand, I was trying to take advantage from the spurious ambiguity of a parsing algorithm and I developed a non-deterministic oracle for the arc-standard algorithm and for LR-Spine algorithm that we will see in chapter 6. Note that in [Sartorio, Satta, and Nivre, 2013] I called this type of oracle *easy-first strategy* because the objective of such training is to choose the easier computation above all possible computations: An idea that shares some principles with the Easy-First algorithm in [Goldberg and Elhadad, 2010].

The idea that pushed me to explore this kind of technique is simple. During training, especially in an on-line setting, we try to learn a model that approximates the oracle function. But the static oracle strictly follows the canonical derivation among many possible derivations (due to algorithm's spurious ambiguity). So we train a model that tries to reproduce such behaviour adding a non necessary constrain: our objective is to reach the gold dependency tree and not to follow the canonical computation! I thought it reasonable that training a model without useless constraints would be simpler to learn. Indeed we can avoid to update the model when it is not strictly necessary.

It came out that for many algorithms it is easy to design an oracle that during training takes in consideration all possible derivations that reach the correct dependency tree. As we will see in the experimental results the performance improvements by using a non-deterministic oracle is more effective on algorithms with a high degree of spurious ambiguity. Formally we can define a *non-deterministic* oracle as follow.

Definition 5.2. Given a configuration c_i from which the gold dependency tree $T_G = (V, A_G)$ can be reached, the *non-deterministic oracle* is a function that retrieves a set of transitions:

$$\text{nondetOracle}(T_s, c_i) = \mathcal{T}_o = \{\tau_1, \tau_2, \ldots, \tau_k\}$$

where all transitions $\tau \in \mathcal{T}_o$ start a different derivation that reaches the same T_G.

Similarly to what we did for the static oracle, we can design conditions for each transition of a parsing algorithm that satisfy the property of producing a new configuration that is in the set $\mathcal{T}_o$. In this case the conditions define whether a transition is *correct*.

Definition 5.3. Given a parsing algorithm and given a configuration c_i, from which the gold dependency tree T_G is reachable, the transition τ is *correct* if $c_i \vdash_\tau c_{i+1}$ and from c_{i+1} T_G is still reachable.

In algorithm 5.2 we can see that, differently from the static oracle, the non-deterministic oracle checks the conditions for all transitions and it returns a set containing all correct transitions. For LEFT-ARC and RIGHT-ARC the conditions are identical to the static oracle; for SHIFT new conditions are needed because this transition can no longer be chosen by exclusion.

Algorithm 5.2 Non-Deterministic Oracle Algorithm

Input:
configuration c
gold dependency tree T_g ▷ correct dependency tree for sentence s
Output:
transition $\mathcal{T}_o$ ▷ return a set of transitions
1: $\mathcal{T}_a \leftarrow \{\tau_0, \tau_1, \ldots, \text{SHIFT}]\}$ ▷ set of transition for a specific algorithm
2: **for each** τ_i **in** $\mathcal{T}_a$ **do**
3: **if** conditions(T_g, c, τ_i) **then**
4: $\mathcal{T}_o \leftarrow \mathcal{T}_o \cup \{\tau_i\}$
5: **break**
6: **return** $\mathcal{T}_o$

In order to use such oracle the learning algorithm needs to be slightly modified from the one in section 4.2.2 because the non-deterministic oracle retrieves a set and not a single transition. In Algorithm 5.3 we can see that the model is updated only if the predicted transition is not in the set retrieved by the oracle (line 11). If we choose a training strategy correct-and-go-on we have to choose a transition over the ones retrieved by the oracle. We can do it randomly but using a model that retrieves a score for each transition I prefer to choose the transition $\tau_i \in \mathcal{T}_o$ that maximize the model's score.

5.2.1 Arc-Standard Non-Deterministic Oracle

In table 5.4 I present the conditions checked by the non-deterministic oracle for the Arc-Standard algorithm. LEFT-ARC and RIGHT-ARC have the same conditions of

Algorithm 5.3 On-line Training Algorithm (correct-and-go-on) using a non deterministic oracle

Input:

treeBank= $[(s_1, T_{s_1}), (s_2, T_{s_2}), \ldots, (s_m, T_{s_m})]$

Output:

model ▷ return a trainined model

1: model $\leftarrow$ newModel($\mathcal{T}$)
2: **for each** s, T_s **in** treeBank **do**
3: $c = (\sigma, \beta, A) \leftarrow ([\,], s, \varnothing)$
4: **while** $|\sigma| > 1 \vee |\beta| > 0$ **do** ▷ while c is not final do
5: $\mathcal{T}' \leftarrow \varnothing$
6: **for each** τ in $\mathcal{T}$ **do** ▷ select the applicable transitions
7: **if** applicable(τ,c) **then**
8: $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{\tau\}$
9: $\tau \leftarrow$ model.giveBestTransition($\mathcal{T}', c$)
10: $\mathcal{T}_o \leftarrow$ nonDeterministicOracle(T_s, c)
11: **if** $\tau \notin \mathcal{T}_o$ **then** ▷ model retrieve a bad prediction w.r.t. oracle
12: model.update($\mathcal{T}', c, \mathcal{T}_o$) ▷ update the model
13: $\tau \leftarrow \tau_i$, $\tau_i \in \mathcal{T}_o$ ▷ $\mathcal{T}_o$ contains one or more transitions
14: $c' = (\sigma', \beta', A') \leftarrow$ apply(τ, c) ▷ apply(τ, c) returns c' s.t. $c \vdash_\tau c'$
15: $c = (\sigma, \beta, A) \leftarrow c'$
16: **return** model

Transition	Oracle's Condition
LEFT-ARC	$(s_0 \rightarrow s_1) \in A_g$
RIGHT-ARC	$(s_1 \rightarrow s_0) \in A_g$ and $\forall w_i \in V, \nexists\, (s_0 \rightarrow w_i) \in A_g \setminus A$
SHIFT	$(s_0 \rightarrow s_1) \notin A_g$ and $(s_1 \rightarrow s_0) \notin A_g$ or $\exists\, (s_0 \rightarrow w_i) \in A_g \mid w_i \in \beta$

Table 5.4: Non-deterministic oracle conditions for Arc-Standard algorithm

the static oracle. SHIFT is retrieved if there are no LEFT-ARC and RIGHT-ARC (as for the static oracle) or if there exists a right dependent of s_0 into the buffer. This new condition takes care of the spurious ambiguity pattern described in section 4.1.4: $[\text{LA}, \text{SH}, \ldots, \text{RA}]$ or $[\text{SH}, \ldots, \text{RA}, \text{LA}]$.

The SHIFT conditions in table 5.4 are equivalent to:

Lemma 5.4. given a configuration c from which it is possible to reach the gold dependency tree T_G, the transition SHIFT is incorrect if and only if the following conditions are both satisfied:

1. $(s_0 \rightarrow s_1) \in A_G$ or $(s_1 \rightarrow s_0) \in A_G$
2. $\nexists\, (s_0 \rightarrow w_i) \in A_G \mid w_i \in \beta$

Proof. Let $c = (\sigma|s_1|s_0, \beta, A)$ and $c' = \text{SHIFT}(c)$
If statement.
Assuming 1 and 2 satisfied I argue that from c' it is not possible to reach T_G. The only way to create the arc $(s_0 \rightarrow s_1) \in A_G$ or the arc $(s_1 \rightarrow s_0) \in A_G$ is to reach a configuration $c'' = (\sigma|s_1|s_0, \beta'', A'')$ with the same stack of c. From c', the only way to reach the configuration c'' is to reduce the stack by doing a RIGHT-ARC that creates an arc $(s_0 \rightarrow b_i)$ but this contradicts the condition 2.
Only if statement.
If condition 1 is not satisfied, then LEFT-ARC and RIGHT-ARC are not correct and the only possible transition is SHIFT (as in the static oracle). If condition 2 is not satisfied, then there exists a complete (except the root) subtree of T_G rooted by s_0 that spans over the substring of $s : [s_0, \ldots, w_j], w_j \geq w_i$. Such subtree is the complete subtree of a reachable tree T_G, so there exists a derivation that reduces the subtree to its root s_0. In such way we can reach a configuration $c'' = (\sigma|s_1|s_0, \beta'', A'')$ where the stack is the same of c and β'' has been reduced by optimal transitions. Clearly from c'' the parsing algorithm can reach T_G.

□

5.2.2 Arc-Eager Non-Deterministic Oracle

In table 5.5 I present the conditions checked by the non-deterministic oracle for the Arc-Eager algorithm. Note that the transitions REDUCE and SHIFT are correct

Transition	Oracle's Conditions
LEFT-ARC	$(b_0 \rightarrow s_0) \in A_g$
RIGHT-ARC	$(s_0 \rightarrow b_0) \in A_g$
REDUCE	$\exists\, w_i \in \sigma \mid (w_i \rightarrow s_0) \in A$ and $\forall\, w_j \in \beta, \nexists\, (s_0 \rightarrow w_j) \in A_g$
SHIFT	$\exists\, w_i \in \beta \mid (w_i \rightarrow b_0) \in A_g$ and $\nexists\, w_j \in \sigma \mid (b_0 \rightarrow w_j) \in A_g$

Table 5.5: Non deterministic oracle conditions for Arc-Eager algorithm

only if LEFT-ARC and RIGHT-ARC are both wrong. This is due to the lack of spurious ambiguity over the creation of an arc. Otherwise SHIFT and REDUCE can be both satisfied.

The SHIFT conditions in table 5.5 are simple to prove.

Lemma 5.5. given a configuration c from which it is possible to reach the gold dependency tree T_G, the transition SHIFT is correct if and only if the following conditions are both satisfied:

1. $\exists\, w_i \in \beta \mid (w_i \rightarrow b_0) \in A_g$
2. $\nexists\, w_j \in \sigma \mid (b_0 \rightarrow w_j) \in A_g$

Proof. If statement. Considering the projectivity, if both conditions are satisfied it means that there exists a complete (except the root) subtree of T_G rooted by $\text{P}(b_0)$ that spans over the substring $s' = [b_0, \ldots, \text{P}(b_0)]$. The nodes of such subtree can be reduced to the root $\text{P}(b_0)$.
Only if statement.
If the first condition is not satisfied it means that $\text{P}(b_0) \in \sigma$ and in the Arc-Eager algorithm it is possible to create an arc only between a node in the stack and a node in the buffer. Similarly, if the second condition is not satisfied it means that there is a dependent of b_0 in the stack, and it will be impossible to create the associated arc after shifting b_0 into the stack. So in both cases SHIFT is wrong. □

5.2.3 Attardi's Non-Deterministic Oracle

Transition	Oracle's Condition
LEFT-ARC$_1$	$(s_0 \rightarrow s_1) \in A_g$ and $\forall w_i \in V, \nexists\, (s_1 \rightarrow w_i) \in A_g \setminus A$
RIGHT-ARC$_1$	$(s_1 \rightarrow s_0) \in A_g$ and $\forall w_i \in V, \nexists\, (s_0 \rightarrow w_i) \in A_g \setminus A$
LEFT-ARC$_2$	$(s_0 \rightarrow s_2) \in A_g$ and $\forall w_i \in V, \nexists\, (s_2 \rightarrow w_i) \in A_g \setminus A$
RIGHT-ARC$_1$	$(s_2 \rightarrow s_0) \in A_g$ and $\forall w_i \in V, \nexists\, (s_0 \rightarrow w_i) \in A_g \setminus A$
SHIFT	reachable$(T_G, \text{SHIFT}(c))$

Table 5.6: Non deterministic oracle conditions for Attardi's algorithm, the reachable function checks if the gold dependency tree is reachable from the configuration obtained by applying a SHIFT to the current configuration

Unfortunately for Attardi's algorithm it is hard to define whether the gold dependency tree for a sentence is reachable or not, unless we try to parse it. Similarly we have the same problem for subsequences of the original sentence and we cannot apply the same trick that we use for the SHIFT transition in the Arc-Standard algorithm.

We can approximate a non-deterministic oracle by retrieving at the same time LEFT-ARC$_1$ and LEFT-ARC$_2$ when both are correct. However this approach does not capture most of the possible derivations, indeed most of the spurious ambiguity relies over the choosing of the SHIFT transition instead of a LEFT-ARC. A real non-deterministic oracle can be obtained by testing if after a SHIFT the resulting configuration can reach the gold dependency tree by applying a static oracle.. This requires the parsing of the whole sentence each time that there is a configuration in which we can make a reduction (LEFT-ARC, RIGHT-ARC, LEFT-ARC$_2$, RIGHT-ARC$_2$). In table 5.6 the function *parsable* checks the reachability of the gold dependency

tree after a SHIFT. Given a sentence of length n, the time complexity of this test is $\mathcal{O}(n)$ and it should be done at each reduction. To parse a sentence we need exactly $n-1$ reductions leading to a time complexity of $\mathcal{O}(n^2)$ at training time.

5.3 Dynamic Oracle

One of the main problem in greedy transition based parsers is the error propagation. When a parser commits an error at test time it reaches configurations that are unlikely to have significant features. The model is trained only over configurations from which it is reachable the correct dependency tree and it learns to discriminate the right transition in such context. Otherwise if at test time the parser fails the model is constrained to classify configurations it has never seen before. In [Goldberg and Nivre, 2012] and [Goldberg and Nivre, 2013] the authors had a simple idea that achieves really good results: let the parser fail also at training time.

For the non-deterministic oracle we define the correctness of a transition by looking if the following configuration can reach the gold dependency tree. In a configuration that cannot reach the gold dependency tree we need a different way to recognize the best transitions: the *cost* function.

5.3.1 Loss and Cost function

In dependency parsing the objective is to retrieve the syntactically correct dependency tree. However if we have a configuration in which the gold dependency tree is not reachable we can still reach a tree that has few errors. If we consider all arcs with the same importance we can say that the best tree is the one that contains less errors. We can define a *loss function* that compares any complete dependency tree with the gold dependency tree.

Definition 5.6. The *loss* $\mathcal{L}$ of a dependency tree $T = (V, A)$ with respect to the gold dependency tree $T_G = (V, A_G)$ is the cardinality of the set difference of set A from set A_G:

$$\mathcal{L}(T, T_G) = |A \setminus A_G|$$

Note that considering all arcs with the same importance can be arguable. An arc that represents the relation between subject and verb can be considered more important than the relation between a name and an adjective. However let me consider all arcs with equal importance.

For a configuration we can consider all reachable dependency trees and we can define a loss function as follows.

Definition 5.7. The *loss* $\mathcal{L}$ of a configuration c with respect to the gold dependency tree $T_G = (V, A_G)$ is the minimum loss above all reachable dependency trees:

$$loss(c, T_G) = \min_{T \in \mathcal{D}(c)} \mathcal{L}(T, T_G)$$

where $\mathcal{D}(c)$ is the set of all reachable dependency trees from c.

Having defined the loss of a configuration we are able to compare different configurations (obviously with respect to the same gold dependency tree). This is particularly interesting if we look to transitions that belongs to the same derivation:

$$c_0 \vdash_{\tau_0} c_1 \vdash_{\tau_1} c_2 \vdash_{\tau_2} c_3 \vdash_{\tau_3} \ldots \vdash_{\tau_k} c_f$$

If we consider the incremental behaviour of the transition based algorithms it is clear that:

$$\mathcal{L}(c_0) = 0 \leq \mathcal{L}(c_1) \leq \mathcal{L}(c_2) \leq \mathcal{L}(c_3) \leq \ldots \leq \mathcal{L}(c_f) = \mathcal{L}(T)$$

where T is the tree resulting from the derivation. For example, if we have $\mathcal{L}(c_2) = 3$ the gold dependency tree is not reachable, so we have done some mistake in the previous transitions. However if $\mathcal{L}(c_3) = 10$ it is clear that the transition τ_2 is not the best one.

By using this principle we can define the *cost of a transition*.

Definition 5.8. Given a gold dependency tree and a configuration c_i the *cost* $\mathcal{C}$ of a transition τ such as $c_i \vdash_\tau c_{i+1}$ is the difference over the loss of c_{i+1} and c_i:

$$\mathcal{C}(\tau, c_i, T_G) = \mathcal{L}(c_{i+1}, T_G) - \mathcal{L}(c_i, T_G)$$

General Dynamic Oracles

Using the cost function above, we are able to compare transitions of a parsing algorithm when they are applied to configurations in which the gold dependency tree is not reachable. Following the same principles of the non-deterministic oracle, we define a dynamic oracle that retrieves a set of 0-cost transitions.

Definition 5.9. Given a configuration c and a gold dependency tree $T_G = (V, A_G)$ the *dynamic oracle* is a function that retrieves a set of transitions:

$$\text{dynamicOracle}(T_G, c) = \mathcal{T}_o = \{\tau_i \mid \mathcal{C}(\tau, c, T_G) = 0\}$$

where all transitions $\tau_i \in \mathcal{T}_o$ start different derivations that can reach different trees T with the same loss.

To train a model that is able to reduce the error propagation we need to explore wrong configurations. As we can see in algorithm 5.4 we update the model as usual in case of wrong prediction but we let the parser proceed with the predicted transition independently if it is a 0-cost transition. If we do not remove line 13 in the training algorithm we constrain the system to follow 0-cost transitions, obtaining exactly the same behaviour of the non-deterministic oracle.

In [Goldberg and Nivre, 2013] the authors use two parameters to limit the error-exploring behaviour. The first one constrains the algorithm to follow the 0-cost transitions for the first k iterations. The second parameter defines a probability p: with probability p the training chooses the 0-cost transition, and with probability $1 - p$ it follows the wrong prediction. The parameter k is interesting if we assign to it a small value, for example it is reasonable to start the error-exploring after the first iteration when we already have a model that avoid to explore really wrong transitions. Otherwise I don't like too much the random behaviour given by parameter p. Another interesting parameter can be a loss bound b, maybe parametrized with the length of the sentence $b(\text{length}(s))$: if the loss of a configuration is greater than the loss bound we follow only the 0-cost transitions. It can be interesting to use this kind of parameter with a beam search technique and a dynamic oracle; for example we can choose to update the model if into the beam there are no configurations with loss less than the bound value. However I prefer to avoid parameters that are hard to set and can lead to inconsistent experimental results so in my tests I do not use them.

Algorithm 5.4 On-line Training Algorithm (error-exploring) using dynamic oracle

Input:

treeBank= $[(s_1, T_{s_1}), (s_2, T_{s_2}), \ldots, (s_m, T_{s_m})]$

Output:

model ▷ return a trainined model

1: model $\leftarrow$ newModel($\mathcal{T}$)
2: **for each** s, T_s **in** treeBank **do**
3: $c = (\sigma, \beta, A) \leftarrow ([\,], s, \varnothing)$
4: **while** $|\sigma| > 1 \vee |\beta| > 0$ **do** ▷ while c is not final do
5: $\mathcal{T}' \leftarrow \varnothing$
6: **for each** τ in $\mathcal{T}$ **do** ▷ select the applicable transitions
7: **if** applicable(τ,c) **then**
8: $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{\tau\}$
9: $\tau \leftarrow$ model.giveBestTransition($\mathcal{T}', c$)
10: $\mathcal{T}_o \leftarrow$ dinamicOracle(T_s, c)
11: **if** $\tau \notin \mathcal{T}_o$ **then** ▷ model retrieve a bad prediction w.r.t. oracle
12: model.update($\mathcal{T}', c, \mathcal{T}_o$) ▷ update the model
13: ~~$\tau \leftarrow \tau_i$, $\tau_i \in \mathcal{T}_o$~~
14: $c' = (\sigma', \beta', A') \leftarrow$ apply(τ, c) ▷ apply(τ, c) returns c' s.t. $c \vdash_\tau c'$
15: $c = (\sigma, \beta, A) \leftarrow c'$
16: **return** model

Unfortunately given a generic configuration it is not easy to compute the loss. In [Goldberg and Nivre, 2013] the authors identify a property that holds for some parsing algorithm. This property allows to simplify the dynamic oracle for some parsing algorithms, we will see it in the case of the arc-eager algorithm. For other algorithms, I will present a more general approach based on a dynamic programming technique. In [Goldberg, Sartorio, and Satta, 2014] we use this technique for the arc-standard algorithm while in [Gómez-Rodríguez, Sartorio, and Satta, 2014] we use an even more general approach for the Attardi's Algorithm.

5.3.2 Arc-Eager Dynamic Oracle

In [Goldberg and Nivre, 2013] the authors define the arc-decomposition property and they show that the Arc-Eager algorithm has this property. They start by defining the *reachability* of an arc.

Definition 5.10. Given a configuration $c_i = (\sigma_i, \beta_i, A_i)$, an arc $a \notin A$ is *reachable* if there exists a derivation $d = \tau_0\tau_1 \dots \tau_k$ such as:

$$c_i \vdash_{\tau_0} c_{i+1} \vdash_{\tau_1} \cdots \vdash_{\tau_k} c_{i+k} = (\sigma_{i+k}, \beta_{i+k}, A_{i+k})$$
$$a \in A_{i+k}$$

In practice an arc is reachable from a configuration if there exists a derivation that builds the arc starting from the configuration.

Obviously, if we consider a gold dependency tree $T_G = (V, A_G)$ that is reachable by using a parsing algorithm, we have that the whole set of arcs A_G is reachable from the initial configuration c_0:

$$A_G = \{a \mid a \in A_G \wedge a \text{ is reachable from } c_0\}$$

Otherwise if we have a configuration $c_i = (\sigma_i, \beta_i, A_i)$ from which it is not possible to reach the gold dependency tree we have:

$$A_G \neq A_i \cup \{a \mid a \in A_G \wedge a \text{ is reachable from } c_i\}$$

The *arc-decomposition* property of a parser considers the subset of arcs of a dependency tree that are reachable from a configuration.

Definition 5.11. A transition based parsing algorithm has the arc-decomposition property if for every reachable dependency tree $T = (V, A)$ and for all possible configurations c_i there exists a derivation $d = \tau_0\tau_1 \dots \tau_k$ such that:

$$c_i \vdash_{\tau_0} c_{i+1} \vdash_{\tau_1} \cdots \vdash_{\tau_k} c_{i+k} = (\sigma_{i+k}, \beta_{i+k}, A_{i+k}),$$
$$\{a \mid a \in A \wedge a \text{ is reachable from } c_i\} \subseteq A_{i+k}$$

The arc-decomposition property holds for all possible dependency trees but in practice what interests us is that in a arc-decomposable algorithm, given a gold dependency tree, every gold arc reachable from a configuration is mutually reachable.

This is a powerful property because it allows to consider all arcs independently. The authors of [Goldberg and Nivre, 2013] show that this property holds for the arc-eager algorithm but unfortunately not for the arc-standard algorithm.

Considering the modifications of the configuration, when we apply a transition, we can see that each transition may prevent the reachability of some arcs. Given the arc-decomposition property each arc can be considered independently from the others, so the number of prevented arcs is the cost of the transition.

In algorithm 5.5 we can see the algorithm to compute the cost function of a configuration. The algorithm considers that if a token i is removed from the stack there is no derivation that can create arcs of type: $(i \rightarrow j), j \in \beta$. Otherwise if a token is moved from the buffer into the stack it is impossible to reach any arc $(i \rightarrow j), j \in \sigma$. Note that the cost is 0 if the new arc $(i \rightarrow j) \in A_G$ but also if $(i \rightarrow j) \notin A_G$ and there are no reachable arcs prevented by the new arc. Note also that the dynamic oracle does not guaranties that the retrieved graph is a dependency tree because REDUCE may be a 0-cost transition if s_0 has no head and the head is not available (by means $\text{P}(s_0) \notin \beta$).

Computational Analysis

Clearly the algorithm 5.5 works with complexity $\mathcal{O}(n)$ where n is the length of the input sentence. Indeed for each transition we need to check the prevented arcs by analyzing the gold parent for each token into the buffer or into the stack, both bounded by n. However, by using a careful implementation that incrementally saves the reachable arcs of a configuration (with respect to the gold dependency tree) the increased training time compared with a static oracle is barely notable.

5.3.3 Arc-Standard Dynamic Oracle

In [Goldberg, Sartorio, and Satta, 2014] we solve the dynamic oracle for the Arc-Standard algorithm by using a polynomial tabular method to compute the loss of a configuration.

The algorithm consists of two steps. Informally, in the first step we compute the largest subtrees, of the gold tree T_G that have their span entirely included in the buffer β. The root nodes of these tree fragments are then arranged into a list, according to the order in which they appear in β. We call this structure the

Algorithm 5.5 Computation of the cost function for the Arc-Eager algorithm

```
1: cost ← 0
2: if τ = LEFT-ARC then
3:     if P(s_0) ≠ b_0 and P(s_0) ∈ β then
4:         cost ← cost +1
5:     for b_i ∈ β do
6:         if P(b_i) = s_0 then
7:             cost ← cost +1
8: if τ = RIGHT-ARC then
9:     if P(b_0) ≠ s_0 and P(b_0) ∈ σ then
10:        cost ← cost +1
11:    for s_i ∈ σ do
12:        if P(s_i) = b_0 then
13:            cost ← cost +1
14: if τ = REDUCE then
15:    if P(s_0) ∈ β then
16:        cost ← cost +1
17:    for b_i ∈ β do
18:        if P(b_i) = s_0 then
19:            cost ← cost +1
20: if τ = SHIFT then
21:    if P(b_0) ∈ σ then
22:        cost ← cost +1
23:    for s_i ∈ σ do
24:        if P(s_i) = b_0 then
25:            cost ← cost +1
26: return cost
```

reduced buffer β_R. Intuitively, β_R can be viewed as the result of pre-computing β by applying all sequences of transitions that match T_G and that can be performed independently of the stack in the input configuration c.

In the second step of the algorithm we use dynamic programming techniques to simulate all computations of the Arc-Standard algorithm starting in a configuration with stack σ and with a buffer now represented by β_R. The search space defined by these computations includes at least one the dependency tree for the

sentence s that is reachable from the input configuration c and that have minimum loss. We then perform a Viterbi search to pick up the loss value.

The second step is very similar to standard implementations of the CKY parser for context-free grammars [Hopcroft and Ullman, 1979], running on an input string obtained as the concatenation of σ and β_R. The main difference is that we restrict ourselves to parse only those constituents in $\sigma\beta_R$ that dominate the topmost element of σ (the rightmost element, if σ is viewed as a string). In this way, we account for the additional constraint that we visit only those configurations of the Arc-Standard parser that can be reached from the input configuration c. For instance, this excludes the reduction of two nodes in σ that are not at the two topmost positions. This would also exclude the reduction of two nodes in β_R: this is correct, since the associated subtrees have been chosen as the largest such fragments in β.

Reduction of the Buffer

In the first step we process β and construct β_R, which we call the *reduced buffer*

Definition 5.12. Given a configuration $c = (\sigma, \beta, A)$ and a gold dependency tree $T_G = (V, A_G)$ the *reduced buffer* (for the Arc-Standard Algorithm) is a subsequence of β in which each token is the root of a tree T that satisfies the following properties:

1. T is a subtree of the gold tree T_G having span entirely included in the buffer β;

2. T is *bottom-up complete* for T_G, meaning that for each node w_i of T different from the root of T, the dependents of w_i in T_G cannot be in σ;

3. t is *maximal* for T_G, meaning that every supertree of T in T_G violates the above conditions.

The stack β_R is incrementally constructed by processing β from left to right. Each node i is copied into β_R if it satisfies any of the following conditions

1. the parent node of i in T_G is not in β;

2. some dependent of i in T_G is in σ or has already been inserted in β_R.

It is not difficult to see that the nodes in β_R are the roots of tree fragments of T_G that satisfy the condition of bottom-up completeness and the condition of maximality defined above. Another way to reduce the buffer is to run the arc-standard algorithm over the buffer with a static oracle, this is not the most efficient way to do it but considering that it is clear that the operation can be done in linear time.

In order to simplify the specification of the loss computation algorithm, we assume below that first element in β_R is the topmost element in σ, so β_R and σ has the same topmost element. Therefore the other elements of β_R are shifted of one position.

Algorithm 5.6 Computation of the loss function for the Arc-Standard algorithm

$\mathcal{A}[1,1](\sigma[1]) \leftarrow \sum_{i \in [1,|\sigma|\,]} \mathcal{L}(T(\sigma[i]), T_G)$ ▷ in $\mathcal{A}[1,1]$ the loss of all already computed subtrees
for $d \leftarrow 1$ **to** $|\sigma| + |\beta_R| - 1$ **do** ▷ d is the index of a sub-anti-diagonal
  for $j \leftarrow \max\{1, d - |\sigma| + 1\}$ **to** $\min\{d, |\beta_R|\}$ **do** ▷ j is the column index
    $i \leftarrow d - j + 1$ ▷ i is the row index
    if $i < |\sigma|$ **then** ▷ expand to the left
      for each $h \in \Delta_{i,j}$ **do**
        $\mathcal{A}[i+1,j](h) \leftarrow \min\{\mathcal{A}[i+1,j](h),\ \mathcal{A}[i,j](h) + \delta_G(h \rightarrow \sigma[i+1])\}$
        $\mathcal{A}[i+1,j](\sigma[i+1]) \leftarrow \min\{\mathcal{A}[i+1,j](\sigma[i+1]),\ \mathcal{A}[i,j](h) + \delta_G(\sigma[i+1] \rightarrow h)\}$
    if $j < |\beta_R|$ **then** ▷ expand to the right
      for each $h \in \Delta_{i,j}$ **do**
        $\mathcal{A}[i,j+1](h) \leftarrow \min\{\mathcal{A}[i,j+1](h),\ \mathcal{A}[i,j](h) + \delta_G(h \rightarrow |\beta_R|[j+1])\}$
        $\mathcal{A}[i,j+1](\beta_R[j+1]) \leftarrow \min\{\mathcal{A}[i,j+1](\beta_R[j+1]), \mathcal{A}[i,j](h) + \delta_G(\beta_R[j+1] \rightarrow h)\}$
return $\mathcal{A}[\,|\sigma|, |\beta_R|\,](0)$

Computation of Configuration Loss

Let me introduce some notation:

- $|\sigma|$ and $|\beta_R|$ to denote the length of the left stack and of the reduced buffer
- $\sigma[i]$ and $\beta_R[i]$ to denote the i-th element of σ and of β_R, $\sigma[1]$ the topmost element into σ and $\beta_R[1]$ the first element into β_R
- $T(\sigma[i])$ and $T(\beta_R[i])$ to denote the corresponding subtree respectively rooted

by $\sigma[i]$) and $\beta_R[i]$), where $T(\sigma[i])$ is a subtree already built by the parser and $T(\beta_R[i])$ is a subtree of T_G

Algorithm 5.6 uses a two-dimensional array $\mathcal{A}$ (a table) of size $|\sigma| \times |\beta_R|$, where each entry $\mathcal{A}[i, j]$ is an association list from integers to integers. An entry $\mathcal{A}[i, j](h)$ stores the minimum loss among dependency trees rooted at h that can be obtained by running the parser on the first i elements of stack σ and the first j elements of buffer β_R. More precisely, let

$$\Delta_{i,j} = \{\sigma[k] \mid k \in [1, i]\} \cup \{\beta_R[k] \mid k \in [1, j]\}$$

For each $h \in \Delta_{i,j}$, the entry $\mathcal{A}[i, j](h)$ is the minimum loss among all dependency trees defined as above and with root h. We also assume that $\mathcal{A}[i, j](h)$ is initialized to $+\infty$ for all possible h (not reported in the algorithm).

Algorithm 5.6 starts at the top-left corner of $\mathcal{A}$, visiting each individual sub-anti-diagonal of $\mathcal{A}$ in ascending order, reaching the bottom-right corner of the table. The entry $\mathcal{A}[1, 1](\sigma[1])$ is initialized with the loss of all already built subtrees with roots in σ. For each entry $\mathcal{A}[i, j]$, the left expansion is considered (lines 5 to 8) by combining with tree fragment $\sigma[i + 1]$, through a left or a right arc reduction. This results in the update of $\mathcal{A}[i+1, j](h)$, for each $h \in \Delta_{i+1,j}$, whenever a smaller value of the loss is achieved for a tree with root h. The Kronecker-like function used at line 8 provides the contribution of each single arc to the loss of the current tree. Denoting with A_G the set of arcs of T_G, such a function is defined as

$$\delta_G(i \to j) \quad = \quad \begin{cases} 0, & \text{if } (i \to j) \in A_G; \\ 1, & \text{otherwise.} \end{cases} \tag{5.1}$$

A symmetrical process is implemented for the right expansion of $\mathcal{A}[i, j]$ through subtrees $\beta_R[j + 1]$ (lines 9 to 12).

The quantity $\mathcal{A}[|\sigma|, |\beta_R|](0)$ is the minimal loss above all reachable trees. Note that contribute to the loss of all subtrees $T(\beta_R[j])$ is zero because they are optimal subtrees of T_G. Otherwise the contribute to the loss of all $T(\sigma[i])$ is constant during the computation and it is assign to $\mathcal{A}[1, 1](\sigma(1))$

Computational Analysis

The reduction of the buffer is an important step because it allows to completely reduced subtrees that span only into the buffer. This allow the oracle step to

always combine an inferred entry in the table with either a node from the stack or from the reduced buffer. In such way the dynamic programming technique can avoid to combine inferred entries toghether.

The reduced buffer β_R can be easily constructed in time $\mathcal{O}(n)$, n the length of the input string. In the loss computation, for each entry $\mathcal{A}[i,j]$ and for each $h \in \Delta_{i,j}$, we update $\mathcal{A}[i,j](h)$ a number of times bounded by a constant which does not depend on the input. Each updating can be computed in constant time as well. We thus conclude that Algorithm 5.6 runs in time $\mathcal{O}|\sigma| \cdot |\beta_R| \cdot (|\sigma| + |\beta_R|)$. All quantities are bounded by n so the complexity is $\mathcal{O}(n^3)$. However, in practice, the former is significantly smaller: when measured over the sentences in the Penn Treebank, the average value of $\frac{|\sigma|+|\beta_R|}{n}$ is 0.29. In terms of runtime, training is 2.3 times slower when using our oracle instead of a static oracle.

5.3.4 Attardi's algorithm Dynamic Oracle

When I firstly try to make a dynamic oracle for the Attardi's algorithm I thought that I could use the same procedure of the Arc-Standard parser. Unfortunately some helpful properties that hold with projective trees are no longer satisfied in the non-projective case. In the projective case, as we have seen in the previous section, subtrees that are in the buffer can be completely reduced. As a consequence, each loss computation step always combines an inferred entry in the table with either a node from the left stack or a node from the reduced buffer. Otherwise, in the non-projective case, subtrees in the buffer can not always be completely reduced. As a consequence, the oracle needs to make cell updates in a more general way, which includes linking pairs of elements in the reduced buffer or pairs of inferred entries in the table.

Consider the dependency tree in figure 5.1 and assume a configuration $c = (\sigma, \beta, A)$ where $\sigma = [0, 1, 2, 3, 4]$, $\beta = [5, \ldots, 11]$, and $A = \varnothing$. It is easy to see that the loss of c is greater than zero, since the gold tree is not reachable from c: parsing the subtree rooted at node 5 requires shifting 6 into the stack, and this makes it impossible to build the arcs $2 \rightarrow 5$ and $2 \rightarrow 6$. However, if we reduced the subtree in the buffer with root 5, we would incorrectly obtain a loss of 0, as the resulting tree is parsable if we start with SHIFT followed by LEFT-ARC and RIGHT-ARC$_2$. Note that there is no way of knowing whether it is safe to reduce

Example 5.13.

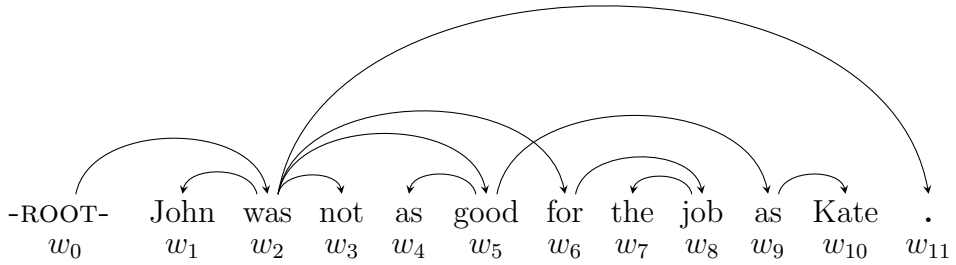


Figure 5.1: Example of a gold tree such that not all the subtrees in the buffer can be reduced in configuration $c = (\sigma, \beta, A)$ where $\sigma = [0, 1, 2, 3, 4]$, $\beta = [5, \ldots, 11]$, and $A = \varnothing$.

the subtree rooted at 5 without using non-local information. For example, the arc $2 \rightarrow 6$ is crucial here: if 6 depended on 5 or 4 instead, the loss would be zero. These complications are not found in the projective case.

We can still reduce the buffer using principles similar to the Arc-Standard case but we have to use a more general dynamic programming technique to the loss computation.

Reduction of the Buffer

We use the same first two principles defined in 5.3.3 for the preprocessing of the buffer but instead of the maximality we consider subtrees with Zero gap-degree. This is an important requirement for the construction of $t(\beta_R[i])$ from β, since a tree fragment having a discontinuous span over β might not be constructable independently of σ. More specifically, parsing such fragment implies dealing with the nodes in the discontinuities, and this might require transitions involving nodes from σ.

Definition 5.14. Given a configuration $c = (\sigma, \beta, A)$ and a gold dependency tree $T_G = (V, A_G)$ the *reduced buffer* (for the Arc-Standard Algorithm) is a subsequence of β in which each token is the root of a tree T that satisfies the following properties:

1. T is a subtree of the gold tree T_G having span entirely included in the buffer β;

2. T is *bottom-up complete* for T_G, meaning that for each node w_i of T different from T's root, the dependents of w_i in T_G cannot be in σ;

3. T has *Zero gap-degree*, meaning that all the nodes of T form a contiguous substring of s.

If T satisfies the above conditions, then we can safely reduce the nodes of T appearing in β, creating a right stack β_R replacing them with the node root node h. This is clearly true because the Zero gap-degree condition guarantees that the span of T over the nodes of β is not interleaved by nodes that do not belong to T. The bottom-up complete condition guarantee that all nodes, except the roots, of the subtrees have no arcs with other elements of the buffer or of the stack. A subtree of a tree reachable from a parsing algorithm is reachable too. And if a tree is reachable from a parsing algorithm means that exists a derivation that can optimally reduce the span of a tree into one element (the root).

The sufficient condition above allow to compute β_R. We process the buffer β from left to right and for each node k we test the Bottom-up completeness condition and the Zero gap-degree condition for the complete subtree T of T_G rooted at k. We substitute the span of the subtree with k if the conditions are satisfied. Note that in this process a node k resulting root of a reduced subtree T might be removed from β if, at some later point, we reduce a supertree of T.

Computation of the Loss

The loss computation is based on the dynamic programming technique in [Kuhlmann, Gómez-Rodríguez, and Satta, 2011] and more specifically over the tabular parsing algorithm in [Cohen, Gómez-Rodríguez, and Satta, 2011]. Given an input string their algorithm produces a compact representation of the set of all possible computations of a transition based algorithm.

In our case the input string γ is the concatenation of the stack and the buffer:

$$\gamma = \sigma \; \beta_R$$

where the elements are ordered by following the order into the input sentence s, so $\gamma[0]$ is the last (left-most) element into the stack σ. $\gamma[i]$ is the $(i+1)$-th node in γ for $0 \leq i \leq |\gamma| - 1$. Let $\ell = |\sigma|$ be the boundary between the stack and the buffer in γ. So $\gamma[i] \in \sigma$ if $i < \ell$, otherwise $\gamma[i] \in \beta_R$ if $i \geq \ell$.

Algorithm 5.7 Computation of the loss function for Attardi's algorithm (simplified)

$\mathcal{A}[0,1]([\$,\$0]) \leftarrow 0$ ▷ shift node 0 on top of empty stack symbol \$
for $i \leftarrow 1$ **to** $\ell - 1$ **do**
  $\mathcal{A}[i,i+1]([\gamma[i-1],\gamma[i-1]\gamma[i]]) \leftarrow 0$ ▷ shift node $\gamma[i]$ with $\gamma[i-1]$ on top of the stack
for $i \leftarrow \ell$ **to** $|\gamma|$ **do**
  for $h \leftarrow 0$ **to** $i-1$ **do**
    $\mathcal{A}[i,i+1]([\gamma[h],\gamma[h]\gamma[i]]) \leftarrow 0$ ▷ shift node $\gamma[i]$ with $\gamma[h]$ on top of the stack
for $d \leftarrow 2$ **to** $|\gamma|$ **do** ▷ consider substrings of length d
  for $i \leftarrow \max\{0, \ell - d\}$ **to** $|\gamma| - d$ **do** ▷ i = beginning of substring
    $j \leftarrow i + d$ ▷ $j - 1$ = end of substring
    ProcessCell$(\mathcal{A}, i, i+1, j)$ ▷ range $k = i+2$ to $\max\{i+2, \ell\} - 1$ omitted
    for $k \leftarrow \max\{i+2, \ell\}$ **to** j **do** ▷ factorization of substring at k
      ProcessCell$(\mathcal{A}, i, k, j)$
return $\mathcal{A}[0,|\gamma|]([\$,\$0]) + \sum_{i \in [0,\ell-1]} \mathcal{L}(\sigma[i], T_G)$

procedure ProcessCell$(\mathcal{A}, i, k, j)$
  for each key $[h_1, h_2h_3]$ **defined in** $\mathcal{A}[i,k]$ **do**
    for each key $[h_3, h_4h_5]$ **defined in** $\mathcal{A}[k,j]$ **do** ▷ h_3 must match
      $loss_{\text{LA}} \leftarrow \mathcal{A}[i,k]([h_1,h_2h_3]) + \mathcal{A}[k,j]([h_3,h_4h_5]) + \delta_G(h_5 \rightarrow h_4)$
      if $(i < \ell) \vee \delta_G(h_5 \rightarrow h_4) = 0 \vee (h_5 \notin \gamma)$ **then**
        $\mathcal{A}[i,j]([h_1,h_2h_5]) \leftarrow \min\{loss_{\text{LA}}, \mathcal{A}[i,j]([h_1,h_2h_5])\}$ ▷ cell update LA
      $loss_{\text{RA}} \leftarrow \mathcal{A}[i,k]([h_1,h_2h_3]) + \mathcal{A}[k,j]([h_3,h_4h_5]) + \delta_G(h_4 \rightarrow h_5)$
      if $(i < \ell) \vee \delta_G(h_4 \rightarrow h_5) = 0 \vee (h_4 \notin \gamma)$ **then**
        $\mathcal{A}[i,j]([h_1,h_2h_4]) \leftarrow \min\{loss_{\text{RA}}, \mathcal{A}[i,j]([h_1,h_2h_4])\}$ ▷ cell update RA
      $loss_{\text{LA}_2} \leftarrow \mathcal{A}[i,k]([h_1,h_2h_3]) + \mathcal{A}[k,j]([h_3,h_4h_5]) + \delta_G(h_5 \rightarrow h_2)$
      if $(i < \ell) \vee \delta_G(h_5 \rightarrow h_2) = 0 \vee (h_5 \notin \gamma)$ **then**
        $\mathcal{A}[i,j]([h_1,h_4h_5]) \leftarrow \min\{loss_{\text{LA}_2}, \mathcal{A}[i,j]([h_1,h_4h_5])\}$ ▷ cell update LA_2
      $loss_{\text{RA}_2} \leftarrow \mathcal{A}[i,k]([h_1,h_2h_3]) + \mathcal{A}[k,j]([h_3,h_4h_5]) + \delta_G(h_2 \rightarrow h_5)$
      if $(i < \ell) \vee \delta_G(h_2 \rightarrow h_5) = 0 \vee (h_2 \notin \gamma)$ **then**
        $\mathcal{A}[i,j]([h_1,h_2h_4]) \leftarrow \min\{loss_{\text{RA}_2}, \mathcal{A}[i,j]([h_1,h_2h_4])\}$ ▷ cell update RA_2

As for the Arc-Standard algorithm 5.7 uses a two-dimensional array $\mathcal{A}$. The dimension is $(|\gamma| - 1) \times (|\gamma| - 1)$, with row indexes range from 0 to $|\gamma| - 1$ while the column indexes from 1 to $|\gamma|$, and only the cells $\mathcal{A}[i,j]$ with $i < j$ are filled.

Each entry $\mathcal{A}[i,j]$ is an association list whose keys are items $[h_1, h_2h_3]$, where h_1, h_2, h_3 are nodes in γ. The value stored at $\mathcal{A}[i,j]([h_1,h_2h_3])$ is the minimum loss contribution due to the computations represented by $[h_1, h_2h_3]$.

Specifically the variables into the notation $\mathcal{A}[i,j]([h_1,h_2h_3]) = v$ denote a set of computations such as:

- i,j the computations involve the tokens from the $\gamma[i]$ to $\gamma[j]$
- h_1 was the topmost element into the stack when the computations began
- h_2h_3 are the topmost elements when the computations end
- v is the minimal loss contribution above all the computations where the previous holds

To correctly represent the inference model, we assume that our parser starts with a symbol $\$ \notin V_w$ in the stack, denoting the bottom of the stack.

We initialize the table by populating the cells of the form $\mathcal{A}[i,i+1]$ with information about the trivial computations consisting of a single SHIFT transition that shifts the node $\gamma[i]$ into the stack. These computations are known to have zero loss contribution, because a SHIFT transition does not create any arcs. In the case where the node $\gamma[i]$ belongs to σ, i.e., $i < \ell$, we assign loss contribution 0 to the entry $\mathcal{A}[i,i+1]([\gamma[i-1],\gamma[i-1]\gamma[i]])$ (line 3 of Algorithm 5.7), because $\gamma[i]$ is shifted with $\gamma[i-1]$ at the top of the stack. On the other hand, if $\gamma[i]$ is in β, i.e., $i \geq \ell$, we assign loss contribution 0 to several entries in $\mathcal{A}[i,i+1]$ (line 6) because, at the time $\gamma[i]$ is shifted, the content of the stack depends on the transitions executed before that point.

After the above initialization, we consider pairs of contiguous substrings $\gamma[i]\cdots\gamma[k-1]$ and $\gamma[k]\cdots\gamma[j-1]$ of γ. At each inner iteration of the nested loops of lines 7-11 we update cell $\mathcal{A}[i,j]$ based on the content of the cells $\mathcal{A}[i,k]$ and $\mathcal{A}[k,j]$. We do this through the procedure ProcessCell$(\mathcal{A},i,k,j)$, which considers all pairs of keys $[h_1,h_2h_3]$ in $\mathcal{A}[i,k]$ and $[h_3,h_4h_5]$ in $\mathcal{A}[k,j]$. Note that we require the index h_3 to match between both items, meaning that their computations can be concatenated. In this way, for each reduce transition τ in our parser, we compute the loss contribution for a new piece of computation defined by concatenating a computation with minimum loss contribution in the first item and a computation with minimum loss contribution in the second item, followed by the transition τ.

Two pieces of computation are combined by following the inference rules in

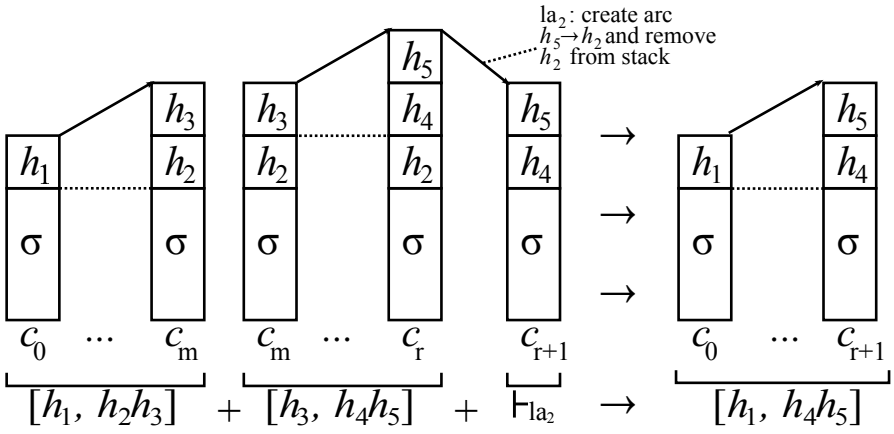


Figure 5.2: Concatenation of two computations/items and transition LEFT-ARC$_2$, resulting in a new computation/item.

[Cohen, Gómez-Rodríguez, and Satta, 2011]:

$$\text{LEFT-ARC}_1 : \frac{[i,k,h_1,h_2h_3]\ [k,j,h_3,h_4h_5]}{[i,j,h_1,h_2h_5]}$$

$$\text{RIGHT-ARC}_1 : \frac{[i,k,h_1,h_2h_3]\ [k,j,h_3,h_4h_5]}{[i,j,h_1,h_2h_4]}$$

$$\text{LEFT-ARC}_2 : \frac{[i,k,h_1,h_2h_3]\ [k,j,h_3,h_4h_5]}{[i,j,h_1,h_4h_5]}$$

$$\text{RIGHT-ARC}_2 : \frac{[i,k,h_1,h_2h_3]\ [k,j,h_3,h_4h_5]}{[i,j,h_1,h_2h_4]}$$

In Figure 5.2 its represented the concatenation of two pieces of computation in case of LEFT-ARC$_2$.

The computed loss contribution is used to update the entry in $\mathcal{A}[i,j]$ corresponding to the item associated with the new computation. The loss contribution provided by the arc created by τ is computed as into the arc-standard case by the δ_G function (lines 17, 20, 23 and 26) which is defined as:

$$\delta_G(i \to j) \quad = \quad \begin{cases} 0, & \text{if } i \to j \text{ is in } T_G; \\ 1, & \text{otherwise.} \end{cases} \tag{5.2}$$

It is important to remark that the nature of the problem allows to apply several shortcuts and optimizations that would not be possible in a setting where we

actually needed to parse the string γ. First, the range of variable i in the loop in line 8 starts at $\max\{0, \ell - d\}$, rather than at 0, because we do not need to combine pairs of items originating from nodes below the topmost node into the stack σ, as the items resulting from such combinations correspond to computations that do not contain our input configuration c. Second, when we have set values for i such that $i + 2 < \ell$, we can omit calling ProcessCell for values of the parameter k ranging from $i+2$ to $\ell - 1$, as those calls would use as their input one of the items described above, which are not of interest. Finally, when processing substrings that are entirely in β_R ($i \geq \ell$) we can restrict the transitions that we explore to those that generate arcs that either are in the gold tree T_G, or have a parent node which is not present in γ (see conditions in lines 18, 21, 24, 27), because we know that incorrectly attaching a buffer node as a dependent of another buffer node, when the correct head is available, can never be an optimal decision in terms of loss, and if the head is not available we can always attach it into the following steps.

Once we have filled the table $\mathcal{A}$, the loss for the input configuration c can be obtained from the value of the entry $\mathcal{A}[0, |\gamma|]([\$, \$0])$, representing the minimum loss contribution among computations that reach the input configuration c and parse the whole input string. To obtain the total loss, we add to this value the loss contribution accumulated by the dependency trees with root in the stack σ of c. This is represented in Algorithm 5.7 as $\sum_{i \in [0, \ell-1]} \mathcal{L}(\sigma[i], T_G)$, where $\mathcal{L}(\sigma[i], T_G)$ is the count of the descendants of $\sigma[i]$ (the $(i+1)$-th element of σ) that had been assigned the wrong head by the parser with respect to T_G.

Computational Analysis

The first stage of our algorithm can be implemented in time $\mathcal{O}(|\beta||T_G|)$, where $|T_G|$ is the number of nodes in T_G, which is equal to the length n of the input sentence.

For the worst-case complexity of the second stage (Algorithm 5.7), note that the number of cell updates made by calling ProcessCell$(\mathcal{A}, i, k, j)$ with $k < \ell$ is $\mathcal{O}|\sigma|^3|\gamma|^2|\beta_R|$. This is because these updates can only be caused by procedure calls on line 10 (as those on line 12 always set $k \geq \ell$) and therefore the index k always equals $i + 1$, while h_2 must equal h_1 because the item $[h_1, h_2 h_3]$ is one of

the initial items created on line 3. The variables i, h_1 and h_3 must index nodes on the stack σ as they are bounded by k, while j ranges over β_R and h_4 and h_5 can refer to nodes either on σ or on β_R.

On the other hand, the number of cell updates triggered by calls to ProcessCell such that $k \geq \ell$ is $\mathcal{O}(|\gamma|^4|\beta_R|^4)$, as they happen for four indices referring to nodes of β_R (k, j, h_4, h_5) and four indices that can range over σ or β_R (i, h_1, h_2, h_3).

Putting everything together, we conclude that the overall complexity of our algorithm is $\mathcal{O}(|\beta||T_G| + |\sigma|^3|\gamma|^2|\beta_R| + |\gamma|^4|\beta_R|^4)$. When expressed as a function of n, our dynamic oracle has a worst-case time complexity of $\mathcal{O}(n^8)$. This is also the time complexity of the dynamic programming algorithm of [Cohen, Gómez-Rodríguez, and Satta, 2011] we started with, simulating all computations of our parser.

In practice, quantities $|\sigma|$, $|\beta_R|$ and $|\gamma|$ are significantly smaller than n. For instance, when measured on the Czech treebank, the average value of $|\sigma|$ is 7.2, with a maximum of 87. Even more interesting, the average value of $|\beta_R|$ is 2.6, with a maximum of 23. Comparing this to the average and maximum values of $|\beta|$, 11 and 192, respectively, we see that the buffer reduction is crucial in reducing training time. These considerations are coherent with the reasonable training time obtained by testing the dynamic oracle. The extra processing due to the dynamic oracle made training about 4 times slower, on average, than using a static oracle.

5.3.5 Optimizations

There are few things that I want to point out about implementation details of the loss functions.

First, we do not need to compute the loss for all transitions, for example in the case of the Attardi's algorithm we do not need to compute the loss of any of the 5 transition at each iteration. We just need to start from the predicted one and eventually check the others, following the model's score, in case of bad prediction.

Second, it is possible to remove all nodes from σ and β_R that have already collected all their dependents and that have no head into $\sigma\beta_R$. These nodes will be certainly linked by a wrong arc so we can directly add them to the final loss count.

Third, during training we are interested to know if the loss of the configuration $\tau(c)$ is greater or equal to the loss of c. So when we are looking for the Zero cost transition we can add a branch and bound search strategy into the loss-function: when the algorithm is filling the entries into the table $\mathcal{A}$ is useless to explore computations which loss is already greater than the loss of c.

These optimizations do not reduce the worst case complexity but drastically reduce the training time.

Chapter 6

LR-Spines

In the previous chapter we have seen how a non-deterministic Oracle can take advantage from spurious ambiguity by learning the easiest way to build the tree, instead of learning how to reproduce the canonical computation. However the parsing algorithms described have many constrains:

1. bottom-up / top-down strategy
2. arcs can be created only at certain conditions
3. low or no flexibility to postpone decisions that require more information

All these constraints are due to the specific transitions set that characterize a parsing algorithm.

In the Arc-Standard and Attardi's algorithms the tree is built in a bottom-up fashion and a node needs to collect all its dependents before being attached to the parent. In case of right dependents this constrains the algorithm to SHIFT a node also if the arc created by a RIGHT-ARC is in the gold dependency tree. In my opinion, this is a huge problem because the learning system has to discriminate not only if the transition creates a link that is in T_G but also if it is the right moment to create it. Consider also that postponing the arc creation can depend on information that involve nodes far from the topmost nodes into the stack, so information that are far from the feature extraction scope.

The arc-eager algorithm gets partially rid of the previous problem by using a top-down strategy for right dependents. However it introduce a REDUCE transition

to remove nodes into the stack with the irreversible result that an eliminated node cannot take other dependents. The arc-eager lack of flexibility is evident if we consider that it has no spurious ambiguity over the arc creation, since an arc must be created, by using a LEFT-ARC or a RIGHT-ARC, as soon as this action is possible, because it will be impossible to create it later.

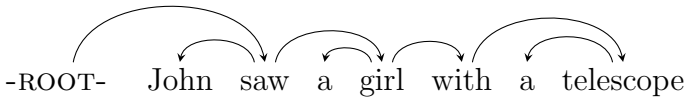
Both Arc-Standard and Arc-Eager algorithms evaluate if it is possible to create an arc between two nodes (by using LEFT-ARC, RIGHT-ARC) but they do not directly compare different arc options. The choice of other arcs are not simultaneously available but different options require different computations using the combination of at least a SHIFT and a LEFT-ARC / RIGHT-ARC transitions. Attardi's algorithm, as a side effect of the transitions used to build non-projective arcs, has the capability to compare two heads for the same token with RIGHT-ARC$_1$ and RIGHT-ARC$_2$, but the other structural constraints are identical to the Arc-Standard algorithm.

Example 6.1. The lack of flexibility problem is evident if we consider the well know PP-attachment issue whose schema is illustrated in figure 6.1c. Here we have to choose whether to attach node P as a dependent of V (arc α_2) or else as a dependent of N1 (arc α_3).

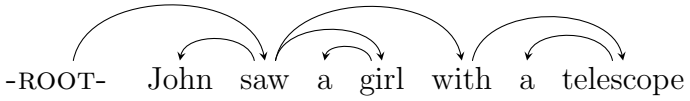
The purely bottom-up arc-standard model has to take a decision as soon as N1 is placed into the stack (to be precise if we consider the sentences in figure 6.1a and 6.1b after the LA(girl,a)). This is so because the construction of α_1 excludes α_3 from the search space, while the alternative decision of shifting P into the stack excludes α_2. This is bad, because the information about the correct attachment could generally come from the lexical content of node P.

The arc-eager model performs slightly better, since it can delay the decision up to the point in which α_1 has been constructed and P is read from the buffer. However, at this point it must make a commitment and either construct α_3 or pop N1 from the stack (implicitly committing to α_2) before N2 is read from the buffer.

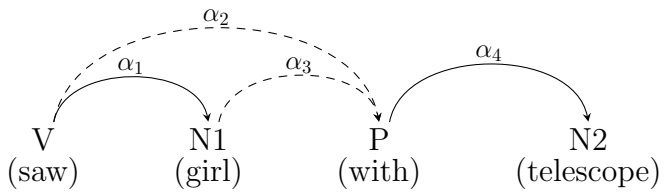
Attardi's algorithm performs slightly better than the arc-standard algorithm because it can reach a configuration in which V, N1, P are the topmost nodes into the stack and both α_2 and α_3 can be constructed by using a RIGHT-ARC$_1$ and RIGHT-ARC$_2$. However this is possible if and only if the arc α_1 has not been created, N2 has already collected all its dependents and the choice regards only two



(a) The girl was carrying a telescope



(b) John was using a telescope



(c) PP-attachment schema, with dashed arcs identifying two alternatives

Figure 6.1: PP-attachment example

possibilities (consider for example a sentence as:"John saw a girl with a telescope with his friend").

In [Sartorio, Satta, and Nivre, 2013] we propose an algorithm where a flexible strategy allows a transition system to decide between the attachments α_2 and α_3 after it has seen all of the four nodes V, N1, P and N2. A system where the correctness of the creation of an arc depends only from the existence in T_G and that can, in many cases, postpone critical decisions. In [Sartorio, Satta, and Nivre, 2013] we called it dynamic strategy, but let me call it flexible strategy to avoid confusion with the oracles (yes ... the word "dynamic" is sometimes abused in NLP)

6.1 LR-Spines Algorithm

Consider the arc-standard algorithm and let me give a different interpretation of the stack data structure. In a configuration $c = (\sigma, \beta, A)$, each stack element is a token. But considering the set of already built arcs each stack element is the root node of a tree spanning some (contiguous) substring of the input sentence s. Specifically, in the arc-standard algorithm each token in the stack is the root of a (complete) subtree of every dependency tree reachable from c.

Using the same notation of chapter 5 the parser can combine two trees $t(s_i)$ and $t(s_j)$ through attachment operations, called left-arc or right-arc, under the condition that s_i and s_j appear at the two topmost positions in the stack. Crucially, only the roots of $t(s_i)$ and $t(s_j)$ are available for attachment; see Figure 6.2(a).

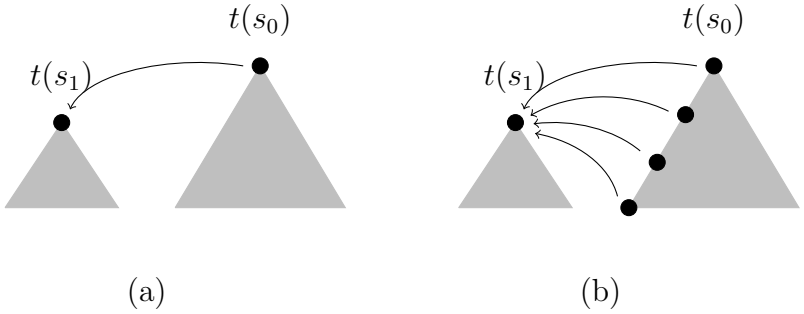


Figure 6.2: Left-arc attachment of $t(s_1)$ to $t(s_0)$ in case of (a) standard transition-based parsers and (b) our parser.

In contrast, into the new transition based algorithm LR-Spines, a stack element records the entire *left spine* and *right spine* of the associated tree.

Definition 6.2. The *left spine* of a dependency tree T is an ordered sequence $\langle ls[1], \ldots, ls[p] \rangle$ with $p \geq 1$ and $ls[i] \in V_w$ for $i \in [1, p]$, consisting of all nodes in a descending path from the root of T taking the leftmost child node at each step.

And symmetrically:

Definition 6.3. The *right spine* of a dependency tree $T = (V, A)$ is an ordered sequence $\langle rs[1], \ldots, rs[q] \rangle$ with $q \geq 1$ and $rs[i] \in V$ for $i \in [1, q]$, consisting of all nodes in a descending path from the root of T taking the rightmost child node at each step.

Note that the left and the right spines share the root node and no other node, see figure 6.3 for an example.

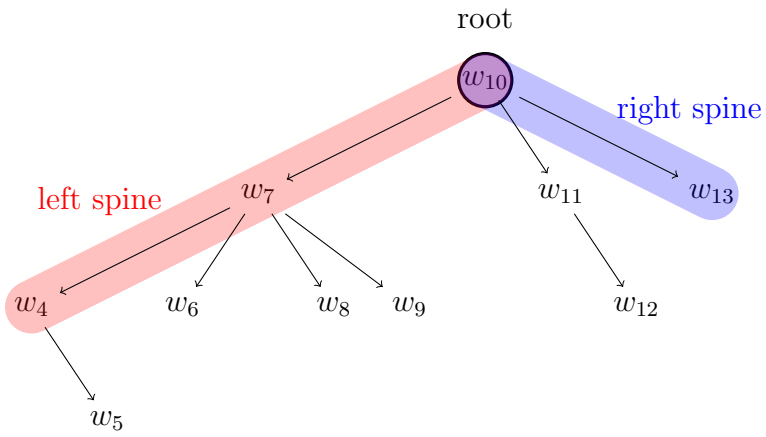


Figure 6.3: Root, Left Spine and Right Spine of a dependency tree, note that the root node belongs to both spines.

This allows to extend the inventory of the attachment operations of the parser by including the attachment of tree $t(s_1)$ as a dependent of any node in the left spine of $t(s_0)$ and symmetrically it allows to attach $t(s_0)$ to each node of the right spine of $t(s_1)$. See Figure 6.2(b) for an example.

Differently from Arc-Standard and Attardi's algorithm, the LR-Spines algorithm implements a mix of bottom-up and top-down strategies, since after any of the attachments in Figure 6.2(b) is performed, additional dependencies can still be created for all element in the new spines.

The new strategy is more powerful than the strategy of the arc-eager model, since we can use top-down parsing at left arcs, which is not allowed in arc-eager parsing, and we do not have the restrictions of parsing right arcs ($h \rightarrow d$) before the attachment of right dependents at node d, without the need of the REDUCE transition as for the Arc-Eager algorithm.

6.1.1 Formal Definition

Configuration

LR-Spine is a transition based algorithm for projective dependency parsing. As usual the state of the algorithm is defined by a *configuration*:

$$c = (\sigma, \beta, A)$$

The *stack* σ is an ordered sequence of stack elements:

$$\sigma = [\sigma_d, \ldots, \sigma_1]$$

and we can use the same notation introduced in chapter 4 and we write $\sigma = \sigma'|\sigma_1$ to indicate that σ_1 is the topmost element of σ.

Differently from other algorithms each *stack element* is a pair:

$$\sigma_k = (ls_k, rs_k)$$

where:

$$ls_k = \langle ls_k[1], \ldots, ls_k[p] \rangle$$
$$rs_k = \langle rs_k[1], \ldots, rs_k[q] \rangle$$

ls_k and rs_k are the left and the right spines, respectively, of the tree associated with σ_k. Recall that $ls_k[1] = rs_k[1]$, since the root node of the associated tree is shared by the two spines.

Similarly to the Arc-Standard parser, the buffer β stores the portion of the input string still to be processed.

Transitions

The set of transitions has three types of transitions, defined in what follows:

- SHIFT. This transition removes the first node from the buffer and pushes into the stack a new element. The nodes into the stack are considered trees with an associated left and right spine, so formally:

$$(\sigma, w_i|\beta, A) \vdash_{\text{SH}} (\sigma|\sigma_{\text{SH}}, \beta, A)$$

where:

$$\sigma_{\text{SH}} = (ls, rs)$$
$$ls_{\text{SH}} = \langle w_i \rangle$$
$$rs_{\text{SH}} = \langle w_i \rangle$$

When pushed into the stack w_i is considered the root of an unitary subtree so it is the first and only element $ls_{\text{SH}}[1]$ and $rs_{\text{SH}}[1]$ in the left and right spine of the new stack element σ_{SH}.

- LEFT-ARC$_k$, $k \geq 1$. Let h be the k-th node in the left spine of the topmost tree in the stack, and let d be the root node of the second topmost tree in the stack. This transition creates a new arc $(h \rightarrow d)$. Furthermore, the two topmost stack elements are replaced by a new element associated with the tree resulting from the $(h \rightarrow d)$ attachment. The transition does not advance with the reading of the buffer. More formally:

$$(\sigma|\sigma_2|\sigma_1, \beta, A) \vdash_{\text{LA}_k} (\sigma|\sigma_{\text{LA}_k}, \beta, A \cup \{h \rightarrow d\})$$

where:

$$\sigma_1 = (ls_1, rs_1)$$
$$\sigma_2 = (ls_2, rs_2)$$
$$h = ls_1[k]$$
$$d = ls_2[1]$$
$$\sigma_{\text{LA}_k} = (\langle ls_1[1], \ldots, ls_1[k] \rangle \oplus ls_2 \ , \ rs_1)$$

Note that the right spine rs_2 of σ_2 disappears from the stack because its nodes became internal into the tree $t(\sigma_{\text{LA}_k})$. The left spine of σ_{LA_k} is the concatenation, denoted by the symbol $\oplus$, of the first k elements into the left spine of σ_1 and the left spine of σ_2. The missing nodes $\langle ls_1[k+1], \ldots ls_1[p] \rangle$ with $p = |ls_1|$ are removed because they do not belongs to the spines of $t(\sigma_{\text{LA}_k})$

- RIGHT-ARC$_k$, $k \geq 1$. This transition is defined symmetrically with respect to LEFT-ARC$_k$:

$$(\sigma|\sigma_2|\sigma_1, \beta, A) \vdash_{\text{RA}_k} (\sigma|\sigma_{\text{RA}_k}, \beta, A \cup \{h \rightarrow d\})$$

where:

$$\begin{aligned}
\sigma_1 &= (ls_1, rs_1) \\
\sigma_2 &= (ls_2, rs_2) \\
h &= rs_2[k] \\
d &= rs_1[1] \\
\sigma_{\text{RA}_k} &= (ls_2\ ,\ \langle rs_2[1], \ldots, rs_2[k]\rangle \oplus rs_2,)
\end{aligned}$$

For the same reason as the LEFT-ARC transition, the left spine of σ_1 disappears while the right spine of σ_{RA_k} is the concatenation of the of the first k elements into the right spine of σ_2 and the right spine of σ_1

Transitions LEFT-ARC$_k$ and RIGHT-ARC$_k$ are parametric in k, where k is bounded by the length of the input sentence and not by a fixed constant. Thus the system uses an unbounded number of transition relations, which has an apparent disadvantage for learning algorithms that have to disambiguate many possible transitions. We will see how it is possible to solve this problem in the following section.

Considering that this new algorithm can always simulate the behaviour of the Arc-Standard parser, it is not difficult to see that the LR-Spines algorithm is complete, meaning that every (projective) dependency tree for w is constructed by some complete computation on w. It is also sound, meaning that the set of arcs constructed in any complete computation on the input sentence s is always a dependency tree for s. It is easy to see that all transitions respect the projectivity constraint and they guaranty that the reached dependency tree is well-formed.

6.2 The Context

We have seen in chapter 4 that a set of atomic features is statically defined and extracted from each configuration. These features are then combined together into complex features, according to some feature template, and joined with the available transition types. This is not possible in our system, since the number of transitions LEFT-ARC$_k$ and RIGHT-ARC$_k$ is not bounded by a constant. Furthermore, it is not meaningful to associate transitions LEFT-ARC$_k$ and RIGHT-ARC$_k$, for any $k \geq 1$, always with the same features, since the constructed arcs impinge on nodes at

different depths in the involved spines. It seems indeed more significant to extract information that is local to the arc $h \rightarrow d$ being constructed by each transition, such as for instance the grandparent and the great grandparent nodes of h. This is possible if we introduce a higher level of abstraction than in existing transition-based parsers. This kind of abstraction makes the feature representation more similar to the ones typically found in graph-based parsers, which are centered on arcs or subgraphs of the dependency tree.

We index the nodes in the stack σ relative to the head node of the arc being constructed, in case of the transitions LEFT-ARC$_k$ or RIGHT-ARC$_k$, or else relative to the root node of σ_1, in case of the transition SHIFT. More precisely,

Definition 6.4. let $c = (\sigma, \beta, A)$ be a configuration and let τ be a transition. We define the *context* of c and τ as the tuple $C(c, t) = (s_3, s_2, s_1, q_1, q_2, gp, gg)$, whose components are placeholders for word tokens in σ or in β.

All these placeholders are specified in Table 6.1, for each transition type τ and for k values $1, 2$ and greater than 2. We do not need to specify greater values of k because we consider a feature template that at most considers 2 elements before and after the evaluated arc $h \rightarrow d$. Note that in Table 6.1 placeholders are dynamically assigned in such a way that s_1 and s_2 refer to the nodes in the constructed arc $h \rightarrow d$, and gp, gg refer to the grandparent and the great grandparent nodes, respectively, of d. Furthermore, the node assigned to s_3 is the parent node of s_2, if such a node is defined; otherwise, the node assigned to s_3 is the root of the tree fragment in the stack underneath σ_2. Symmetrically, placeholders q_1 and q_2 refer to the parent and grandparent nodes of s_1, respectively, when these nodes are defined; otherwise, these placeholders get assigned tokens from the buffer.

The placeholders in $C(c, \tau)$ is the set of atomic features and they are combined together following a feature template. To be consistent with all other experiments I use the feature template of [Zhang and Nivre, 2011], originally developed for the arc-eager model. To be precise the feature template is slightly extended because the grandparent gp and great-grandparent gg features are considered also for transition of type LEFT-ARC$_k$. I also add the right child features for the dependent d in case of RIGHT-ARC$_k$. However I think that the extended feature template guaranties comparable results because they are excluded into the arc-eager feature template only because such information are never available.

context placeholder	SHIFT	LEFT-ARC$_k$			RIGHT-ARC$_k$		
		$k=1$	$k=2$	$k>2$	$k=1$	$k=2$	$k>2$
s_1	$ls_1[1] = rs_1[1]$	$ls_1[k]$			$ls_1[1] = rs_1[1]$		
s_2	$ls_2[1] = rs_2[1]$	$ls_2[1] = rs_2[1]$			$rs_2[k]$		
s_3	$ls_3[1] = rs_3[1]$	$ls_3[1] = rs_3[1]$			$ls_3[1] = rs_3[1]$	$rs_2[k-1]$	
q_1	b_1	b_1	$ls_1[k-1]$		b_1		
q_2	b_2	b_2	b_1	$ls_1[k-2]$	b_2		
gp	NONE	NONE	$ls_1[k-1]$		NONE	$rs_1[k-1]$	
gg	NONE	NONE	NONE	$ls_1[k-2]$	NONE	NONE	$rs_1[k-2]$

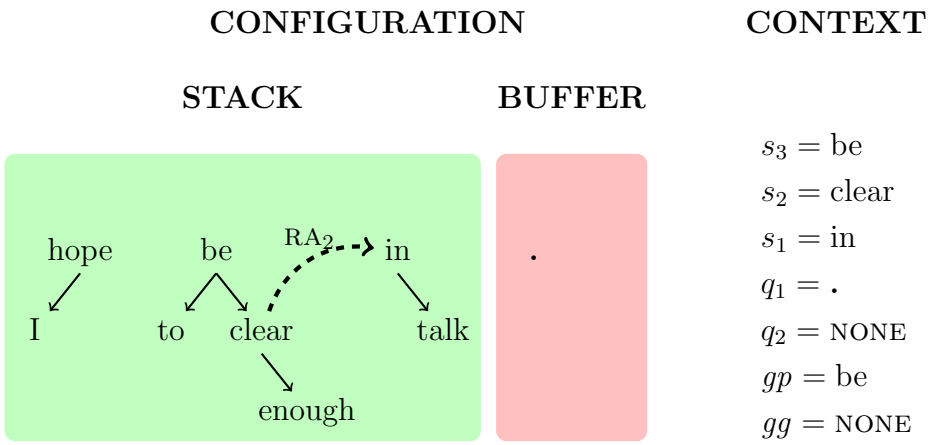
Table 6.1: Definition of context $C(c, \tau) = (s_3, s_2, s_1, q_1, q_2, gp, gg)$, for a configuration $c = (\sigma'|\sigma_3|\sigma_2|\sigma_1, b_1|b_2|\beta, A)$ and a transition τ of type SHIFT or LEFT-ARC$_k$, RIGHT-ARC$_k$, $k \geq 1$. Symbols $ls_j[k]$ and $rs_j[k]$ are the k-th nodes in the left and right spines, respectively, of stack element σ_j, with $ls_j[1] = rs_j[1]$ being the shared root of σ_j; NONE is an artificial element used when some context's placeholder is not available.

Example 6.5. Figure 6.4 shows some examples of context extraction for the RIGHT-ARC$_k$ transtions. The input sentence (taken from my first talk) is: "I hope to be clear enough in this talk . ". The green area represent the stack elements which elements are spines, the red area represent the buffer elements that into the example contain only the token "." (dot).

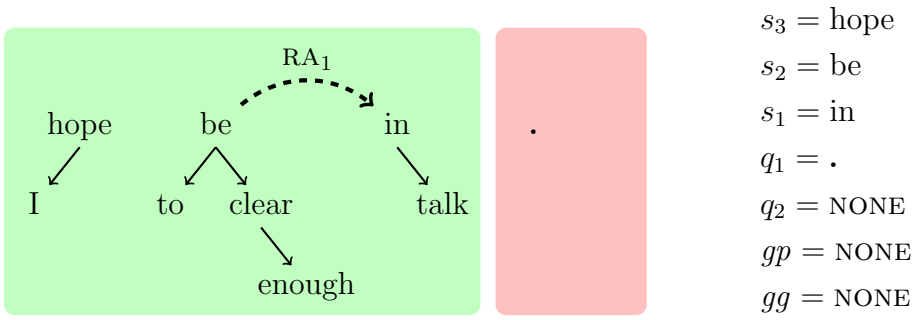
Figure 6.4a and figure 6.4b show the different contexts retrieved from the same configuration by the same transition type RIGHT-ARC$_k$ with $k = 2$ in figure 6.4a and $k = 1$ in figure 6.4b.

In figure 6.4b and figure 6.4c we have two different configurations, the same transition type RIGHT-ARC$_k$ with k values 1 and 2. However we can observe that the obtained context is really similar because the transitions are considering the same arc (be $\rightarrow$ in). The only difference is that in figure 6.4b gp = NONE while

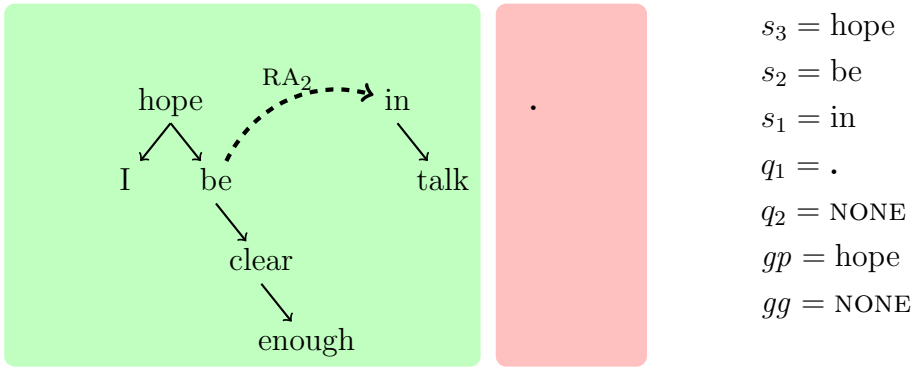
in figure 6.4c gp = hope. The difference is due to the more information available in 6.4c because the arc (hope → be) has already been created. This is important because implies that most of the features are shared if we evaluate the same arc in different configurations. Statistically it means that if the arc is simple to recognize by the model the scores will be similar. Otherwise if the arc is difficult the features with none value can help to postpone the decision by downgrading the score in order to let the model prefer a SHIFT.



(a) Context extracted for RIGHT-ARC$_2$ given a configuration c



(b) Context extracted for RIGHT-ARC$_1$ given a configuration c



(c) Context extracted for RIGHT-ARC$_2$ given a configuration c' different from the configuration c in (a) and (b)

Figure 6.4: Example of context extraction for configurations that are processing the sentence $s = \langle$ I, hope, to, be, clear, enough, in, this, talk, .$\rangle$. In (a) and (b) we have the same configuration c but the context is extracted for a different transitions, RIGHT-ARC$_2$ and RIGHT-ARC$_1$. In (c) we have a different configuration c' where the algorithm has already created the arc (hope $\rightarrow$ be). Note that the transition and the configuration in (b) and (c) are different but the context extracted is similar because the transitions RIGHT-ARC$_1$ in (b) and RIGHT-ARC$_2$ in (c) will create the same arc (be $\rightarrow$ in).

6.3 Oracles for LR-Spines

As in chapter 5 we can define a static, a non-deterministic and a dynamic oracle for the LR-Spines algorithm in order to train the model. The static and non-deterministic oracles are described in [Sartorio, Satta, and Nivre, 2013] while the dynamic oracle is presented in [Goldberg, Sartorio, and Satta, 2014]. As usual for the static and non-deterministic oracle we consider a configuration c from which the gold dependency tree T_G is reachable. Otherwise for the dynamic oracle we can consider any configuration derived from the initial configuration c_0.

We have seen that in a configuration there are p LEFT-ARC$_k$ transitions available, p the length of the left spine of σ_1, and s RIGHT-ARC$_k$ transitions available, s the length of the right spine of σ_2. Obviously the conditions for such transitions take in consideration different nodes of the spine depending on the k value.

6.4 Static Oracle

As for other oracles in chapter 5 the transitions that create an arc can be considered in any order, given that only one of them can be correct into a configuration. As usual for a static oracle the SHIFT transition is chosen by exclusion, if no other transition is correct. As we can see in table 6.2, the static oracle simply checks the existence of the created arc into the gold dependency tree T_G. Indeed given the flexible bottom-up/top-down strategy of the algorithm, a node attached to its correct parent can still take its dependents.

Transition	Oracle's Condition
LEFT-ARC$_k$	$(ls_1[k] \rightarrow ls_2[1]) \in A_g$
RIGHT-ARC$_k$	$(rs_2[k] \rightarrow rs_1[1]) \in A_g$,
SHIFT	no conditions

Table 6.2: Static oracle conditions for LR-Spines algorithm

Lemma 6.6. given a configuration c from which it is possible to reach the gold dependency tree T_G, the transitions LEFT-ARC$_k$ and RIGHT-ARC$_k$ are incorrect if and only if they create a new arc $(h \rightarrow d) \notin A_G$.

Proof. If statement is self-evident.
Only if statement. Assuming that transition RIGHT-ARC$_k$ creates a new arc $(h \rightarrow d) \in A_G$, we argue that from configuration c' with $c \vdash_{\text{RA}_k} c'$ we can still reach the final configuration associated with A_G. We have $h = rs_2[k]$ and $d = rs_1[1]$. The tree fragments in σ with roots $rs_2[k+1]$ and $rs_1[1]$ must be adjacent siblings in the tree associated with A_G, since c is a correct configuration for A_G and $(rs_2[k] \rightarrow rs_1[1]) \in A_G$. This means that each of the nodes $rs_2[i]$, $i > k$, in the right spine of σ_2 must have already acquired all of its right dependents, since the tree is projective, therefore it is safe for transition RIGHT-ARC$_k$ to eliminate such nodes. For the same reason it is safe to remove all nodes $ls_1[j], \forall j > 1$, because they have already acquired all its left dependents. □

6.4.1 Non deterministic Oracle

As for dynamic oracles saw in chapter 5 we focus over the SHIFT transition, that now can be retrieved by the oracle with other transitions. The SHIFT conditions in table 6.3 are equivalent to the following lemma.

Transition	Oracle's Condition
LEFT-ARC$_k$	$(ls_1[k] \rightarrow ls_2[1]) \in A_g$
RIGHT-ARC$_k$	$(rs_2[k] \rightarrow rs_1[1]) \in A_g$,
SHIFT	$\exists\ (rs_1[k] \rightarrow w_i) \in A_g, k \in [1, \lvert rs_1 \rvert] \mid w_i \in \beta$ or $\exists\ (w_i \rightarrow rs_1[k]) \in A_g \mid w_i \in \beta$

Table 6.3: Non deterministic oracle conditions for LR-Spines algorithm

Lemma 6.7. given a configuration c from which it is possible to reach the gold dependency tree T_G, the transition SHIFT is incorrect if and only if the following conditions are both satisfied:

1. there exists an arc $(h \rightarrow d)$ in A_G such that p is in σ and $c = rs_1[1]$;

2. there is no arc $(h \rightarrow d)$ in A_G with $h = rs_1[k]$ for all k and for all $d \in \beta$.

Proof. Let $c = (\sigma|s_1|s_0, \beta, A)$ and $c' = \textsc{shift}(c) = (\sigma', \beta', A)$
If statement Assuming conditions 1 and 2 are verified, we argue that c' is incorrect. Node c is the head of σ'_2. Arc $(h \rightarrow d)$ is not in A, and the only way we could create $(h \rightarrow d)$ from c' is by reaching a new configuration with c in the topmost stack symbol, which amounts to say that σ'_1 can be reduced by a correct transition. Node p is in some σ'_i, $i > 2$, by condition 1. Then reduction of σ'_1 implies that the root of σ'_1 is reachable from the root of σ'_2, which contradicts condition 2.
Only if statement. Assuming 1 is not satisfied, we argue that SHIFT is correct for c and T_G. There must be an arc $(h \rightarrow d)$ not in A with $d = v_{1,1}$ and p is some token w_i in β. From stack $\sigma' = \sigma''|\sigma'_2|\sigma'_1$ it is always possible to construct $(h \rightarrow d)$ consuming the substring of β up to w_i and ending up with stack $\sigma''|\sigma_{red}$, where σ_{red} is a stack element with root w_i. From there, the parser can move on to the final configuration c_f with $A_f = A_G$. A similar argument applies if we assume that condition 2 is not satisfied. □

6.4.2 Dynamic Oracle

The dynamic oracle for LR-Spines algorithm is very similar to the one for the Arc-Standard algorithm in chapter 5. The algorithm computes the loss of a configurations in order to find the Zero cost transitions by using the already seen steps: buffer reduction and loss computation.

Buffer Reduction

The buffer reduction is practically identical to the procedure described in section 5.1.1. The only difference is that in the reduced buffer β_R each element $\beta_R[j]$ is now a pair of spines $(ls_{R,j}, rs_{R,j})$. However considering that the buffer reduction requires that the tree fragment $t(\beta_R[j])$ is bottom-up complete, we now restrict the search space in such a way that only the root node $root(\beta_R[j])$ can take dependents. This is done by setting $ls_{R,j} = rs_{R,j} = \langle root(\beta_R[j]) \rangle$ for each $j \in [1, |\beta_R|]$. In order to simplify the presentation we also assume $\beta_R[1] = \sigma[1]$, as we have done for the Arc-Standard dynamic oracle.

Loss Computation

In the second phase we compute the loss of an input configuration using a two-dimensional array $\mathcal{A}$, defined as in section 5.1.1. However, because of the way transitions are defined in the LR-Spine parser, we now need to distinguish tree fragments not only on the basis of their roots, but also on the basis of their left and right spines. Accordingly, we define each entry $\mathcal{A}[i,j]$ as an association list with keys of the form (ls, rs). More specifically, $\mathcal{A}[i,j](ls, rs)$ is the minimum loss of a tree with left and right spines ls and rs, respectively, that can be obtained by running the parser on the first i elements of stack σ and the first j elements of buffer β_R.

The Algorithm follow the main idea of Algorithm 5.6 and expand each tree in $\mathcal{A}[i,j]$ at its left side, by combining with tree fragment $T(\sigma[i+1])$, and at its right side, by combining with tree fragment $T(\beta_R[j+1])$.

Differently from the Arc-Standard where trees can be combined only through the roots, in the LR-Spines algorithm the new tree can be created in many ways. Specifically we consider the combination of a tree T_a from $\mathcal{A}[i,j]$ and tree $T(\sigma[i+1])$ by means of a LEFT-ARC$_k$ transition. All other cases are treated symmetrically. Let (ls_a, rs_a) be the spine pair of T_a, so that the loss of T_a is stored in $\mathcal{A}[i,j](ls_a, rs_a)$. Let also (ls_b, rs_b) be the spine pair of $T(\sigma[i+1])$. In case there exists a gold arc in T_G connecting a node from ls_a to $r(\sigma[i+1])$, we choose the transition LEFT-ARC$_k$, $k \in [1, |ls_a|]$, that creates such arc. In case such gold arc does not exists, we choose the transition LEFT-ARC$_k$ with the maximum possible value of k, that is, $k = |ls_a|$. We therefore explore only one of the several possible ways of combining these two trees by means of a LEFT-ARC$_k$ transition.

Note that the above strategy is safe, in fact, in case the gold arc exists, no other gold arc can ever involve the nodes of ls_a eliminated by LEFT-ARC$_k$, because arcs can not cross each other in a projective dependency tree. In case the gold arc does not exist, our choice of $k = |ls_a|$ guarantees that we do not eliminate any element from ls_a.

Once a transition LEFT-ARC$_k$ is chosen, as described above, the reduction is performed and the spine pair (ls, rs) for the resulting tree is computed from (ls_a, rs_a) and (ls_b, rs_b), as defined in section 6.1.1. At the same time, the loss of the resulting tree is computed, on the basis of the loss $\mathcal{A}[i,j](ls_a, rs_a)$, the loss of tree

$T(\sigma[i+1])$, and a Kronecker-like function defined below. This loss is then used to update $\mathcal{A}[i+1,j](ls,rs)$.

Each time we combine two trees we have to update the loss with the contribution of the new arc. Let T_a and T_b be two trees that must be combined in such a way that T_b becomes the dependent of some node in one of the two spines of T_a. Let also $p_a = (ls_a, rs_a)$ and $p_b = (ls_b, rs_b)$ be spine pairs for T_a and T_b, respectively. Recall that A_G is the set of arcs of T_G. The new Kronecker-like function for the computation of the loss is defined as

$$\delta_G(p_a,p_b) = \begin{cases} 0, & \text{if } rs_a[1] < rs_b[1] \ \wedge \ \exists k[(rs_a[k] \rightarrow rs_b[1] \in A_G]; \\ 0, & \text{if } ls_a[1] > ls_b[1] \ \wedge \ \exists k[(ls_a[k] \rightarrow ls_b[1] \in A_G]; \\ 1, & \text{otherwise.} \end{cases}$$

Efficiency Improvement

The loss computation in this case has an exponential behaviour. To see why, consider trees in $\mathcal{A}[i,j]$. These trees are produced by the combination of trees in $\mathcal{A}[i-1,j]$ with tree $T(\sigma[i])$, or by the combination of trees in $\mathcal{A}[i,j-1]$ with tree $T(\beta_R[j])$. Since each combination involves either a left"-arc or a right"-arc transition, we obtain a recursive relation that resolves into a number of trees in $\mathcal{A}[i,j]$ bounded by 4^{i+j-2}.

We introduce now two restrictions to the search space of that result in a huge computational saving. For a spine s, we write $\mathcal{N}(s)$ to denote the set of all nodes in s. We also let $\Delta_{i,j}$ be the set of all pairs (ls,rs) such that $\mathcal{A}[i,j](ls,rs) \neq +\infty$.

- Every time a new pair (ls,rs) is created in $\Delta[i,j]$, we remove from ls all nodes different from the root that do not have gold dependents in $\{root(\sigma[k])\ k < i\}$, and we remove from rs all nodes different from the root that do not have gold dependents in $\{root(\beta_R[k])\ k > j\}$.

- A pair $p_a = (ls_a, rs_a)$ is removed from $\Delta[i,j]$ if there exists a pair $p_b = (ls_b, rs_b)$ in $\Delta[i,j]$ with the same root node as p_a and with $(ls_a, rs_a) \neq (ls_b, rs_b)$, such that $\mathcal{N}(ls_a) \subseteq \mathcal{N}(ls_b)$, $\mathcal{N}(rs_a) \subseteq \mathcal{N}(rs_b)$, and $\mathcal{A}[i,j](p_a) \geq \mathcal{A}[i,j](p_b)$.

The first restriction above reduces the size of a spine by eliminating a node if it is irrelevant for the computation of the loss of the associated tree. The second

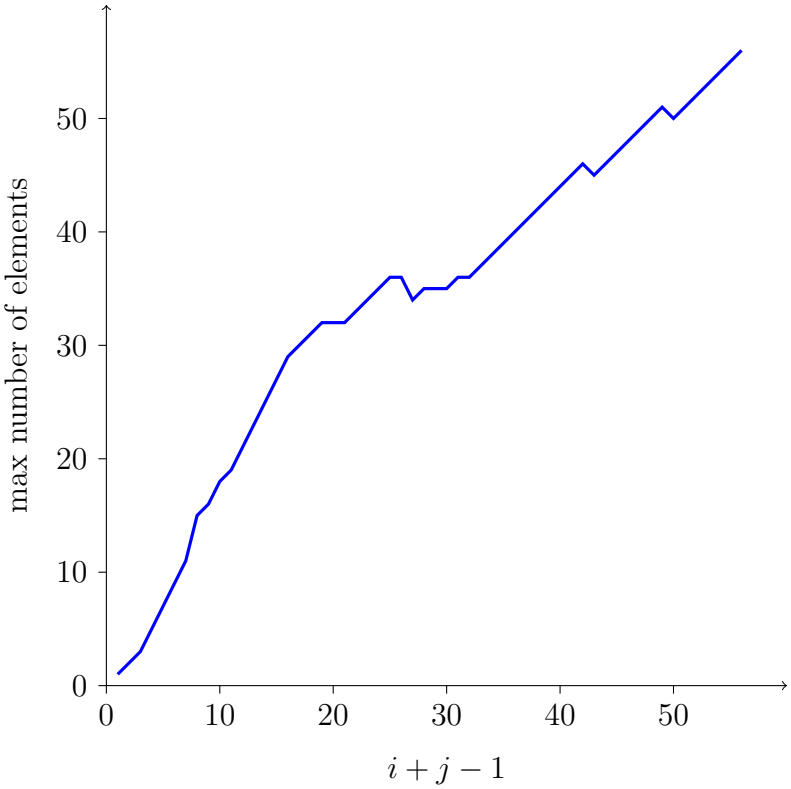


Figure 6.5: Empirical worst case size of $\mathcal{A}[i, j]$ for each value of $i+j-1$ as measured on the Penn Treebank corpus.

restriction eliminates a tree T_a if there is a tree T_b with smaller loss than T_a, such that in the computations of the parser T_b provides exactly the same context as T_a. It is not difficult to see that the above restrictions do not affect the correctness of the algorithm, since they always leave in our search space some tree that has optimal loss.

In order to give an idea about the practical complexity after the above restrictions we can plot the worst case size of $\mathcal{A}[i, j]$, for each value of $j + i - 1$, occurred while training a by using a dynamic oracle.

In Figure 6.5, we can see that $|\mathcal{A}[i, j]|$ grows linearly with $j+i-1$, leading to the same space requirements of algorithm for loss computation in the Arc-Standard case. Empirically, training with the dynamic oracle is only about 8 times slower than training with a static or a non-deterministic oracle.

Chapter 7

Experimental Results

In chapter 5 we saw how we can design an non-deterministic oracle that reduce the learned constraints. Then we analyzed a general approach to explore configurations that cannot reach the gold dependency tree by using a dynamic oracle.

In chapter 6 we saw a new parsing algorithm that maximizes the incremental behaviour and uses a more flexible strategy than traditional algorithms.

In both previous chapters I tried to give the motivations that are behind such ideas. In this chapter I will present my experimental results and I will try to convince you that by applying these new techniques we can improve the accuracy of transition based parsing algorithms. Let me start with some preliminary considerations.

7.1 Some considerations

In order to obtain comparable results it is important that all algorithms are tested under the same conditions. I personally do not like papers in which the authors compare their results with the ones taken from previous works, specially if they take them as they are. As in many fields, in parsing there are many surrounding conditions that have huge impact over the results.

We should retest all systems at the same condition, with exactly the same datasets in order to eliminate all possible causes of noise. If necessary we should re-implement a system or at least check the code of other authors in order to eliminate different assumptions. Indeed simple assumptions that seem irrelevant

can have significant impact.

Let me give some practical examples, where for each case I will indicate the degree of difference in accuracy (based on my experience):

1. Data preprocessing. Apply different preprocessing techniques implies use different training/testing data. For example different taggers or different tree bank conversions (like the one used to convert the Penn Tree Bank into a dependency tree bank) can easily give differences of about 0.5 percentage point.

2. Root position. Simple considerations over the -ROOT- node, such as considering the -ROOT- node at the beginning or at the end of the sentence, can lead to differences up to 3-4 percentage points in accuracy [Ballesteros and Nivre, 2013].

3. Different features. A couple of different features, or the simple choice to include or not into the model features that have null values, can lead to differences of about 0.3-0.7 percentage points.

4. Randomization functions. If our systems use some randomization function (quite common in machine learning approaches) we should try to train our models with different seeds. I found differences of 0.3-0.5 percentage points by simply reordering the training samples.

Such differences are shocking if we consider that many works claim improvements when the accuracy difference is about 0.2 in just one language.

7.2 Experimental Assessments

In this section I will give the details of my experimental assessments.

Data Sets

For performance evaluation I use the Penn Tree Bank [Marcus, Marcinkiewicz, and Santorini, 1993] and the multi-lingual dataset used into the shared task in CoNLL 2007. The Penn Tree Bank uses a phrase structure representation so it needs to be converted in dependency trees. I use the Stanford toolkit with the constraint

to produce projective dependency trees [de Marneffe, MacCartney, and Manning, 2006]. For the Penn Tree Bank I use automatically assigned part of speech tags (97.1% of accuracy) in both training and test sets. Sections 2-21 of Penn Tree Bank are used as training set while section 23 is used as test set. Otherwise I use the CoNLL dataset with the given train/test set splits and the given (correct) part-of-speech tags. In all results of this chapter the accuracies regarding the Penn Tree Bank are reported by excluding punctuation while I include it for CoNLL datasets.

These are the most common settings that I found in literature, however some papers use different set-up regarding punctuation and part of speech tags.

For the Attardi's algorithm I use also the CoNLL 2006 dataset in order to include some languages that are important in non-projective evaluation (such as German and Dutch).

There is a useful tool-kit provided by CoNLL 2006 and CoNLL 2007 committee that analyze the test results. However pay attention that it exclude/include the punctuation by looking at the Unicode type of the tokens and the Unicode type is a little confused between symbols and punctuation marks.

The CoNLL datasets represent a huge resource for dependency parsing tasks but it has small test sets (about 100-200 sentences), specially for some languages. I think that we need to look to the results on their complexity instead of focusing on a particular language. Otherwise we risk to make considerations based on one or two wrong sentences.

Root Position

I usually prefer to not include the -ROOT- into the parsing process. For many datasets the -ROOT- node has only one dependent, in this case I simply attach the -ROOT- dependent after parsing a sentence to the only token that has not yet an head. However in some datasets (for example Czech) the -ROOT- node can have more dependents, for such datasets I consider the -ROOT- the last token into the sentence.

Learning algorithm

I use the averaged perceptron algorithm in an on-line learning configuration. The model is trained up to different iterations depending on the specific parsing algo-

rithm because different algorithms require different number of iterations in order to obtain a stable model. This is reasonable if we consider that an algorithm with high degree of spurious ambiguity can reach more possible configurations than an algorithms with low degree of spurious ambiguity. Specifically the models for the arc-standard algorithms are trained up to 15 iterations, the models for the Attardi's algorithm up to 20 and the models for LR-Spines up to 30.

Accuracy

In the following tables I report the accuracy results for labelled (LAS) and unlabelled (UAS) attachment scores. A labelled arc is correct if head, label and dependent represent an arc that is in the gold dependency tree: $(h, l, d) \in A_G$. Otherwise a unlabelled arc is correct also if the assigned label is not correct. Note that the trained model is the same for labelled and unlabelled scores. The different accuracy is only due to the evaluation function that consider or not the labels.

I report also the results for Unlabelled Exact Match (UEM) that represent the percentage of sentences in which all (unlabelled) arcs are correct. This measure is less common in literature than UAS and LAS, however it is useful to understand that most of dependency trees contains at least one error.

Different Seed

Each number in the following tables is an average of 5 runs with different randomization seeds. The random function is used only to shuffle the training samples at each iteration of training.

7.3 Oracles Comparison

In [Goldberg and Nivre, 2012] the authors analyze the performances of the arc-eager algorithm with different oracles. We will see now that similar results can be obtained with the arc-standard and the Attardi's algorithm. These results were not obvious because the dynamic oracles for such algorithms are clearly more complex than the dynamic oracle for the arc-eager algorithm.

Arc-Standard Algorithm

In table 7.1 we can see that the move from a static to a non-deterministic oracle during training improve the accuracy for most of languages. Making use of the completeness of the dynamic oracle and exploring non-correct configurations during training further improve the results in UAS and LAS.

Otherwise the best results in term of UEM are obtained by using a non-deterministic oracle. This makes sense because the objective of a dynamic oracle is to limit the error propagation at parsing time despite updating the model parameters for non-correct configurations. Otherwise the objective of the non-deterministic oracle is to follow the easy-way to build the correct dependency tree.

The only significant exceptions are Basque, that has a small dataset with more than 20% of non projective sentences, Arabic and Chinese. For Arabic and Chinese we observe a reduction of accuracy in the non-deterministic oracle setup but an increase in the dynamic oracle setup. However, as I told before, I consider more correct to analyze the results in their complexity given the small size of the test sets in the CoNLL datasets.

	static			non-deterministic			dynamic		
	UAS	LAS	UEM	UAS	LAS	UEM	UAS	LAS	UEM
Arabic	81.19	71.44	**14.35**	80.59	70.48	12.98	**82.24**	**72.47**	12.37
Basque	**75.46**	**65.66**	**21.68**	74.71	65.05	21.26	74.69	65.45	19.88
Catalan	90.59	85.27	**27.78**	**90.72**	85.38	**27.78**	90.60	**85.57**	25.27
Chinese	85.34	80.50	**60.70**	84.52	79.78	60.38	**85.98**	**81.61**	60.00
Czech	78.86	71.37	28.46	79.83	71.24	**32.45**	**80.91**	**72.67**	30.07
English	85.91	84.80	28.41	86.86	85.84	**29.07**	**87.66**	**86.76**	28.13
Greek	79.77	72.18	17.26	80.64	72.96	20.30	**81.34**	**73.66**	**20.61**
Hungarian	77.74	67.86	28.72	77.66	67.69	**29.28**	**78.55**	**69.50**	27.79
Italian	82.60	78.55	28.75	83.41	79.46	**31.33**	**84.07**	**80.00**	29.88
Turkish	77.02	66.08	11.40	77.08	66.23	11.60	**77.26**	**66.95**	**13.67**
PTB	89.89	87.59	38.29	90.53	88.26	**39.83**	**90.90**	**88.69**	38.49

Table 7.1: Unlabelled Attachment Score (UAS), Labelled Attachment Score (LAS) and Unlabelled Exact Match using the **arc-standard** algorithm with a static, a non-deterministic and a dynamic oracle. Evaluation on CoNLL 2007 datasets (first block) and on Penn Tree Bank (PTB)

Attardi's Algorithm

In table 7.2 we can see the accuracies by using a static and a dynamic oracle. Unfortunately I realize how to design a non-deterministic oracle when I was reviewing my thesis so I add a section in chapter 5 but I had not time to run a new set of experiments.

The benchmark languages for non-projective parsing algorithms are Dutch, Czech and German. For all of them we observe a good improvement by using a dynamic oracle. Observing all languages in table 7.2, the general trend shows an improvement by using a dynamic oracle. However the results are less consistent than the arc-standard case and we have many exceptions. For Swedish and Bulgarian the accuracy differences are negligible. For Basque, Catalan and Hungarian the performance actually decreases. In order to further understand such behaviour we use a 10-fold cross-validation instead of testing on the standard test sets. The average of the resulting accuracies show improvements for Swedish, Bulgarian and Catalan but not for Basque and Hungarian. More specifically, measured (UAS, LAS) pairs for Swedish are (86.85, 82.17) with dynamic oracle against (86.6, 81.93) with static oracle; for Bulgarian (88.42, 83.91) against (88.20, 83.55); and for Catalan (88.33, 83.64) against (88.06, 83.13). This suggests that the negligible or unfavourable results in table 7.2 for these languages are due to statistical variability given the small size of the test sets. As for Basque, we measure (75.54, 67.58) against (76.77, 68.20); similarly, for Hungarian we measure (75.66, 67.66) against (77.22, 68.42). For Basque we observe a similar exception in the arc-standard case and in the arc-eager results in [Goldberg and Nivre, 2012]. One possible motivation can be that both training and test sets are small. Otherwise I have no explanation for the significant difference in the Hungarian dataset.

	static			dynamic		
	UAS	LAS	UEM	UAS	LAS	UEM
Arabic	80.90	71.56	**16.03**	**82.23**	**72.63**	12.21
Basque	**75.96**	**66.74**	**22.76**	74.32	65.59	19.76
Catalan	**90.55**	**85.20**	**26.83**	89.94	84.96	24.43
Chinese	84.72	79.93	**60.35**	**85.34**	**81.00**	58.29
Czech	79.83	72.69	29.72	**82.08**	**74.44**	**31.82**
English	85.52	84.46	25.14	**87.38**	**86.40**	**28.41**
Greek	79.84	72.26	18.07	**81.55**	**74.14**	**21.73**
Hungarian	**78.13**	**68.90**	**29.90**	76.27	68.14	26.31
Italian	83.08	78.94	**29.00**	**84.43**	**80.45**	28.11
Turkish	**79.57**	69.44	16.40	79.41	**70.32**	**17.93**
Bulgarian	**89.46**	**85.99**	**49.70**	89.32	85.92	45.88
Danish	85.58	81.25	**33.29**	**86.03**	**81.59**	31.99
Dutch	79.05	75.69	26.63	**80.13**	**77.22**	**27.25**
German	88.34	86.48	**47.56**	**88.86**	**86.94**	46.33
Japanese	93.06	91.64	75.29	**93.56**	**92.18**	**77.07**
Portuguese	84.80	81.38	**34.31**	**85.36**	**82.10**	30.14
Slovene	76.33	68.43	**31.64**	**78.20**	**70.22**	**31.64**
Spanish	79.88	76.84	20.68	**80.25**	**77.45**	**21.17**
Swedish	**87.26**	**82.77**	**46.53**	87.24	82.49	44.11
PTB	89.55	87.18	**38.13**	**90.47**	**88.18**	37.48

Table 7.2: Unlabelled Attachment Score (UAS), Labelled Attachment Score (LAS) and Unlabelled Exact Match using the **Attardi** algorithm with a static and a dynamic oracle. Evaluation on CoNLL 2007 datasets (first block), CoNLL 2006 datasets (second block) and on Penn Tree Bank (PTB)

7.4 LR-Spines

Comparing table 7.3 with table 7.1 we can see that the LR-Spines outperform the results obtained by the arc-standard algorithm with the same setup. Only in Hungarian we observe a decrease in accuracy. Otherwise if we consider the results for the Attardi's algorithm in table 7.2 we can see that the LR-Spines is still behind for highly non-projective languages as Czech.

I speculate that the good results obtained by the LR-Spine algorithm can be ascribed to three different factors:

1. the left and right spines allow to directly compare different attachment at the same time,

2. the mixed bottom-up/top-down strategy implies that a transition is always correct if creates a syntactically correct arc (note that this holds also in case of non-correct configurations),

3. the training with a non-deterministic or a dynamic oracle combined with the flexibility of the algorithm allows the model to learn to postpone critical decisions.

Considering the results for different oracles, in table 7.3 we can see a practically uniform improvement in UAS and LAS by using a non-deterministic and a dynamic oracle. The only relevant exception is Basque, but the isolate language seems a constant exception in my experiments. It is interesting to note that as in the arc-standard case the UEM reach top accuracies in case of non-deterministic oracle.

The most widely used transition based algorithms are the arc-standard and the arc-eager algorithms trained with a static oracle. My implementation of such systems reach accuracies (UAS,LAS) over the Penn Tree Bank of (89.89,87.59) for the arc-standard and (89.92, 87.66) for the arc-eager. If we compare such results with the LR-Spines algorithm trained with a dynamic oracle (91.77,89.53) we observe an error reduction of about 18 % for UAS and 15 % for LAS.

	static			non-deterministic			dynamic		
	UAS	LAS	UEM	UAS	LAS	UEM	UAS	LAS	UEM
Arabic	81.67	72.24	**15.27**	83.14	72.94	13.74	**84.54**	**74.54**	14.96
Basque	**76.07**	66.21	**21.98**	75.53	65.66	19.82	75.82	**66.91**	19.64
Catalan	91.47	86.02	28.26	91.31	86.03	**28.62**	**91.92**	**86.83**	27.66
Chinese	84.24	79.36	60.06	84.98	80.47	**62.67**	**86.72**	**82.38**	61.13
Czech	77.93	70.48	27.34	80.03	71.32	**31.40**	**81.19**	**72.72**	30.21
English	86.36	85.38	27.48	88.38	87.45	**31.03**	**89.37**	**88.44**	30.19
Greek	79.43	72.36	17.46	81.12	73.09	19.90	**81.78**	**74.04**	**20.81**
Hungarian	76.56	66.79	**29.54**	76.98	67.70	26.05	**77.48**	**68.76**	23.54
Italian	84.64	80.38	30.20	85.29	81.32	**32.13**	**85.38**	**81.50**	31.16
Turkish	77.00	66.02	11.00	77.63	67.02	**12.13**	**78.61**	**68.06**	11.33
PTB	90.33	88.07	40.52	91.18	88.96	41.74	**91.77**	**89.53**	**41.94**

Table 7.3: Unlabelled Attachment Score (UAS), Labelled Attachment Score (LAS) and Unlabelled Exact Match using the **LR-Spines** algorithm with a static, a non-deterministic and a dynamic oracle. Evaluation on CoNLL 2007 datasets (first block) and on Penn Tree Bank (PTB)

Chapter 8

Conclusions

In this thesis we focus on greedy transition based dependency parsing. We saw how it is possible to improve such systems by using new oracles functions or increasing the flexibility of the algorithms. Specifically the original contribution relies on:

1. the idea of non-deterministic oracles, that take advantage from the spurious ambiguity of a parsing algorithm and avoid the useless constraints of the canonical derivation,

2. dynamic oracles for the Arc-Standard algorithm and for Attardi's algorithm that are able to explore non-correct configurations during training in order to reduce the error propagation that typically affect the transition based algorithms at parsing time,

3. LR-Spines algorithm along with its non-deterministic and dynamic oracles, a new transition based algorithm that using a mixed bottom-up/top-down strategy allows to increase the flexibility of the parsing process.

For such ideas we saw the formal definitions and the experimental improvements obtained. Most of all, I hope to have transmitted the informal motivations behind any idea.

8.1 Future Work

For future work, I would like to pursue the following directions:

1. The general dynamic oracle for the Attardi's algorithm can be used to improve the performances of projective algorithms over non-projective sentences. I have only early results but it seems possible to almost eliminate the gap in accuracy between non-projective and projective algorithms when they parse non-projective sentences.

2. I am not completely satisfied of the features used for the LR-Spines algorithm in case of a SHIFT transition. I think that it is possible to use specifically designed features that try to capture the error condition of SHIFT.

3. The techniques explored in this thesis are orthogonal with a beam search approach. I think that combining a dynamic oracle with a flexible strategy and a beam search could be really effective even with a small beam. However a beam search in a system that has high degree of spurious ambiguity should work well only with a beam search technique that can merge many possible derivations like the one in [Huang and Sagae, 2010].

4. We saw how it is possible to reduce error propagation during parsing by using a dynamic oracle. But what about error-recovery? A first good idea is in [Honnibal, Goldberg, and Johnson, 2013] but there is a lot of other possibilities that I would like to explore.

5. Recently I work a little in CCG parsing. The analogies between the predicate argument relation in CCG and dependency grammars are evident. The logical form that can be extracted from a CCG derivation is a powerful semantic representation. But one of the problem of a CCG parser is the early commitment in choosing the categories. Otherwise in dependency parsing we do not have such problem until we attach a node. There must be a way to define a formalism that take the best of both worlds.

Bibliography

Gerry Altmann and Mark Steedman. Interaction with context during human sentence processing. *Cognition*, 30(3):191–238, 1988.

Bharat Ram Ambati, Tejaswini Deoskar, and Mark Steedman. Using ccg categories to improve hindi dependency parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology-new/P/P13/P13-2107.pdf`.

Giuseppe Attardi. Experiments with a multilanguage non-projective dependency parser. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pages 166–170, New York City, USA, June 2006. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W06/W06-1322`.

Miguel Ballesteros and Joakim Nivre. Going to the roots of dependency parsing. *Computational Linguistics*, 39(1):5–13, 2013. URL `http://www.aclweb.org/anthology/J/J13/J13-1002.pdf`.

Danqi Chen and Christopher D Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750, 2014.

Shay B. Cohen, Carlos Gómez-Rodríguez, and Giorgio Satta. Exact inference for generative probabilistic non-projective dependency parsing. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1234–1245, Edinburgh, Scotland, UK., July 2011. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D11-1114`.

Shay B. Cohen, Carlos Gómez-Rodríguez, and Giorgio Satta. Elimination of spurious ambiguity in transition-based dependency parsing. *arXiv preprint arXiv:1206.6735*, 2012.

Michael Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 1–8. Association for Computational Linguistics, July 2002. doi: 10.3115/1118693.1118694. URL `http://www.aclweb.org/anthology/W02-1001`.

Hal Daumé III. *Practical Structured Learning Techniques for Natural Language Processing*. PhD thesis, University of Southern California, Los Angeles, CA, August 2006. URL `http://hal3.name/docs/#daume06thesis`.

Marie-Catherine de Marneffe and Christopher D. Manning. The stanford typed dependencies representation. In *COLING Workshop on Cross-framework and Cross-domain Parser Evaluation*, 2008. URL `pubs/dependencies-coling08.pdf`.

Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC-06*, 2006. URL `pubs/LREC06_dependencies.pdf`.

Rebecca Dridan and Stephan Oepen. Tokenization: Returning to a long solved problem: A survey, contrastive experiment, recommendations, and toolkit. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 378–382, Jeju Island, Korea, July 2012. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P12-2074`.

Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2): 94–102, February 1970. ISSN 0001-0782. doi: 10.1145/362007.362035. URL `http://doi.acm.org/10.1145/362007.362035`.

Kilian A Foth, Michael Daum, and Wolfgang Menzel. A broad-coverage parser for german based on defeasible constraints. *Constraint Solving and Language Processing*, page 88, 2004.

Yoav Freund and Robert E. Schapire. Large margin classification using the perceptron algorithm. *Machine learning*, 37(3):277–296, 1999.

Yoav Goldberg and Michael Elhadad. An efficient algorithm for easy-first non-directional dependency parsing. In *Proceedings of the 11th Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 742–750, Los Angeles, California, USA, June 2010. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N10-1115`.

Yoav Goldberg and Joakim Nivre. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING)*, pages 959–976, Mumbai, India, December 2012. The COLING 2012 Organizing Committee. URL `http://www.aclweb.org/anthology/C12-1059`.

Yoav Goldberg and Joakim Nivre. Training deterministic parsers with non-deterministic oracles. *Transactions of the Association for Computational Linguistics*, 1:403–414, 2013.

Yoav Goldberg, Francesco Sartorio, and Giorgio Satta. A tabular method for dynamic oracles in transition-based parsing. *Transactions of the Association for Computational Linguistics*, 2:119–130, 2014.

Carlos Gómez-Rodríguez, Francesco Sartorio, and Giorgio Satta. A polynomial-time dynamic oracle for non-projective dependency parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 917–927, Doha, Qatar, October 2014. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D14-1099`.

Jin Guo. Critical tokenization and its properties. *Computational Linguistics*, 23 (4):569–596, 1997.

Matthew Honnibal, Yoav Goldberg, and Mark Johnson. A non-monotonic arc-eager transition system for dependency parsing. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 163–172, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W13-3518`.

John E Hopcroft and Jeffrey D Ullman. Introduction to automata theory, languages, and computation, 1979.

Liang Huang and Kenji Sagae. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1077–1086, Uppsala, Sweden, July 2010. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P10-1110`.

Tadao Kasami. An efficient recognition and syntaxanalysis algorithm for context-free languages. Technical report, DTIC Document, 1965.

Sandra Kübler, Ryan McDonald, and Joakim Nivre. *Dependency Parsing*. Morgan and Claypool, 2009.

Taku Kudo and Yuji Matsumoto. Japanese dependency analysis using cascaded chunking. In *proceedings of the 6th conference on Natural language learning-Volume 20*, pages 1–7. Association for Computational Linguistics, 2002.

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 673–682, Portland, OR, USA, 2011.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

William Marslen-Wilson. Linguistic structure and speech shadowing at very short latencies. *Nature*, 1973.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 523–530. Association for Computational Linguistics, 2005.

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

Joakim Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT*, pages 149–160, Nancy, France, April 2003. Association for Computational Linguistics. URL `http://stp.lingfil.uu.se/~nivre/docs/iwpt03.pdf`.

Joakim Nivre. Incrementality in deterministic dependency parsing. In Frank Keller, Stephen Clark, Matthew Crocker, and Mark Steedman, editors, *Proceedings of the ACL Workshop Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57, Barcelona, Spain, July 2004. Association for Computational Linguistics.

Joakim Nivre. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553, 2008.

Joakim Nivre. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 351–359, 2009. URL `http://www.aclweb.org/anthology/P/P09/P09-1040.pdf`.

Joakim Nivre and Jens Nilsson. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 99–106. Association for Computational Linguistics, 2005.

Emily Pitler. A crossing-sensitive third-order factorization for dependency parsing. *Transactions of the Association for Computational Linguistics*, 2:41–54, 2014. URL `http://www.transacl.org/wp-content/uploads/2014/02/39.pdf`.

Emily Pitler, Sampath Kannan, and Mitchell Marcus. Dynamic programming for higher order parsing of gap-minding trees. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 478–488, Jeju Island, Korea, July 2012. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D12-1044`.

Francesco Sartorio, Giorgio Satta, and Joakim Nivre. A transition-based dependency parser using a dynamic parsing strategy. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 135–144, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P13-1014`.

Giorgio Satta and Marco Kuhlmann. Efficient parsing for head-split dependency trees. *Transactions of the Association for Computational Linguistics*, 1:267–278, 2013.

Ben Taskar, Dan Klein, Michael Collins, Daphne Koller, and Christopher D Manning. Max-margin parsing. In *Proceedings of the conference on Empirical Methods in Natural Language Processing*, volume 1, page 3, 2004.

Lucien Tesnière. *Elements de syntaxe structurale*. Editions Klincksieck, 1959.

Hiroyasu Yamada and Yuji Matsumoto. Statistical dependency analysis with support vector machines. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT*, pages 195–206, Nancy, France, April 2003. Association for Computational Linguistics. URL `http://www.jaist.jp/~h-yamada/pdf/iwpt2003.pdf`.

Yue Zhang and Joakim Nivre. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P11-2033`.