Università degli Studi di Padova

Analysis and improvement of a non-projective dependency parsing algorithm

Laureando:
Marco Camillo

Relatore:
Prof. Giorgio Satta

Corso di Laurea Magistrale in Ingegneria Informatica

12/12/2016

A.A. 2015-2016

Abstract

This essay inspects the previous work of [Nivre, 2009] about dependency parsing of non-projective structures using a special SWAP operation, and proposes an improvement of the system in terms of number of transitions and, consequently, execution time. We'll show how a two-step parsing process of non-projective sentences brings, in most cases, to a reduction of the number of SWAP operations, which are responsible of the whole parsing execution time to exceed linear performances.

Ringraziamenti

Voglio ringraziare tutti quelli che mi hanno permesso di completare il mio percorso di studi.

Per primo ringrazio il prof. Giorgio Satta, che mi ha sostenuto durante la redazione di questa Tesi ed aiutato a superare le difficoltà incontrate.

Ringrazio i miei colleghi ingegneri, in particolare Andrea, mio fedele compagno di mille progetti, giornate di studio e vita universitaria in generale, dal primo all'ultimo anno; i nuovi amici della magistrale, con i quali confrontarsi è stato un piacere ed uno stimolo; e tutti i membri della compagnia della triennale, che nonostante le separazioni è sempre un piacere ritrovare.

Ringrazio i miei amici di più lunga data, che da ormai dieci anni mi forniscono il migliore supporto possibile: l'Amicizia, quella vera e sincera, senza retorica. Non me ne vogliano gli altri, sapete quanto tenga a TUTTI, ma cito in particolare Claudia, Silvia, Micheal e Matteo, che in diverse occasioni mi hanno dimostrato quanto preziose possano essere le parole giuste al momento giusto. Grazie in generale a chiunque mi abbia mai voluto un minimo di bene, senza di voi non sarei quello che sono ora.

Infine ringrazio la mia Famiglia: mia sorella, che è stata gentilissima ad aiutarmi con la stesura della Tesi nonostante la maternità, le mie nonne, i miei zii e cugini, che con la loro stima mi hanno spinto a migliorare sempre di più, ed mio padre e mia madre, dei quali ogni azione ed ogni pensiero è sempre stato volto al mio bene, e che non ringrazierò mai e poi mai abbastanza per il sostegno che mi hanno dato.

Contents

Chapter 1

Introduction to Natural Language Processing

The Human Language Technology or, as it is also called, Natural Language Processing, is a field of information technology that is quickly becoming of great relevance. It comprises knowledge and concepts from many different disciplines such as linguistics, computer science, logic and cognitive science, as its main focus is to define an automated system capable of understanding and analyzing the language that humans use naturally speaking to each other. In the last few years many examples of application of these researches have become quite popular, among them the most remarkable are:

- Google Translate, a service running on a website that simultaneously translates text between a great set of human languages;
- Siri/Google Assistant/Cortana, the new personal assistants we can find respectively in iOS (from version 5), Android (full version from version 7.0, previously integrated in Google Now) and Windows Phone (from 8.1 version) smartphones and tablets, that interpret natural language queries and extract information from texts and documents, like schedules and user preferences;
- Wolfram|Alpha, a computational knowledge engine that interprets the query to extract information from the keyword and try to compose an answer looking to the knowledge it has, instead of giving a link to a pre existent document.

The first researches on this field date back to the mid twentieth century: in 1950 the British mathematician Alan Turing proposed the well known *Touring Test*, which was used to determine whether a machine could be considered intelligent: if a natural language conversation with it would make it indistinguishable from an human interlocutor, the machine could be defined intelligent. Between the first prototypes of human-like talker, the ELIZA

project [Weizenbaum, 1966] is one of the most remarkable. The system was capable of carrying a limited conversation, simulating the behaviour of a Rogerian psychotherapist. Many "patients" of this system were induced to believe that they had been talking to a real doctor, and some of them refused to believe that it was, in fact, a machine even after been explained its operation. Years later, the increased computational power made it possible to develop machine learning techniques. This brought even more attention to this field, as the possibility of an automated system to create a set of rules from a dataset of previously solved instances of a problem lead to a completely new approach on natural language interpretation.

The whole analysis of an human language conversation is composed by a series of phases, usually performed one after the other.

- Symbol analysis. The input is formatted to plain text and tokenized following a dictionary.
- Grammar analysis (or *part-of-speech* tagging). Every token is annotated with its grammatical category (like English name, adjective, pronoun, ...) and specific morfologic form (given by gender, number, ...)
- Syntactic analysis (or *parsing*). The single tokens are grouped in graph-like structures that represent the relations between them.
- Semantic analysis. This structures are used to understand what a sentence means, using other knowledge.
- Pragmatic analysis. The whole sentence is evaluated within the conversation context, considering who stated that and what do we know about him.
- Discourse analysis. The whole conversation is scanned in order to find more complex structures that include more than one sentence.

This essay focuses on the parsing process: we will define a modern theory about how the structure of a sentence can be represented and how the words it is composed by are related to each other. The next chapter introduces this theory, then we will see an example of a parsing algorithm (proposed by Joakim Nivre in [Nivre, 2009]) and how it can be improved.

Chapter 2

Dependency grammars and dependency parsing

Dependency grammars are a popular tool used by researchers in Natural Language Processing for automatic syntactic parsing of sentences. They define a set of relations between the words of a sentence, giving information about how they depend on each other, which add more details to the sense of a sentence than the simple word list. For example, if an automatic translation engine could know which is the main verb in a complex phrase or which context a preposition is used into, it could make a better guess about the meaning of the whole sentence and provide a better translation.
These information are given by this set of binary, asymmetric relations between words, called *dependencies*, that can be used to create a representation of the whole sentence as a *dependency tree*. To create this, we use a *dependency parsing model,* which can infer which relation stands between the words of a sentence and eventually construct the whole tree given the simple sequence of the words.

In this chapter we'll present the formal definition of dependency structures and how a parsing system can create a dependency tree from any sentence.

2.1 Definitions

To define a system that operates over sentences, we must formally define what a sentence is:

Definition 1. A *sentence* is a sequence of tokens (a word or a punctuation marker) we denote by:

$$S = w_0 w_1 w_2 \cdots w_n$$

Actually, w_0 is an artificial token, not representing any word, that we add to the beginning of a sentence, as it will serve as root of the dependency

tree. This may seem to make the very basis of syntactic analysis unrealistic, but it will become clear later that this addition provides great algorithmical generalization ability.

These tokens are connected by a set of relations, each of them will have a *dependency relation type*, that will result in an *arc label* when the tree will be constructed. As this essay will be focused on guided parsing process, for which relation type is irrelevant, we won't specify them any further. Anyway, knowing this, we can define dependency graphs.

Definition 2. Given a sentence $S = w_0 w_1 \cdots w_n$ and a relation type set R, a *dependency graph* $G = (V, A)$ is defined as a standard labeled, directed graph with a set of nodes, V, and a set of arcs, A, such as:

- $V \subseteq \{w_0, w_1, \ldots, w_n\}$,
- $A \subseteq V \times R \times V$,
- if $(w_i, r, w_j) \in A$, then $(w_i, r', w_j) \notin A$ for every $r' \neq r$.

We call it a *well-formed* one (aka a *dependency tree*) if it has the form of a directed tree with w_0 as *root* node and $V = \{w_0, w_1, \ldots, w_n\}$

Figure 2.1 shows how a dependency tree looks like:

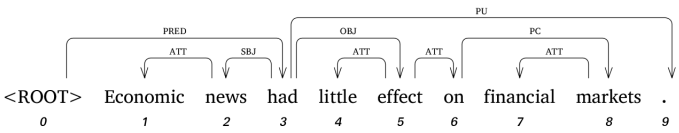


Figure 2.1: Example of a dependency tree

Note that for any sentence S and relation set R there may be many valid trees, corresponding to many syntactic interpretations. This phenomenon is called *syntactic ambiguity* and it's very well known in linguistic research field. An example can be seen in a famous joke by Groucho Marx:

> One morning, I shot an elephant in my pajamas. How he got into my pajamas, I'll never know.

The first sentence may lead to think that the person who shot the elephant is in his pajamas, but the second half explains that this wasn't true. The two interpretations of the ambiguous phrase correspond to two different well-formed dependency trees, as we see in Figure 2.2. The different head of the arc involving the group *"in my pajamas"* is what defines which of the two interpretation is represented.

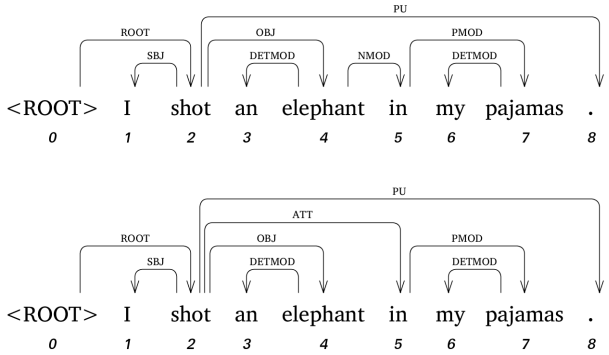


Figure 2.2: Example of syntactically ambiguous sentence

2.2 Properties of dependency trees

First of all, it will be useful to have some simple notational conventions to help us in the analysis of dependency trees (all of the subsequent definitions are valid for a generic well-formed tree $G = (V, A)$)

- $w_i \rightarrow w_j$ denotes that there is a generic *dependency relation* between w_i and w_j, so $(w_i, r, w_j) \in A$ for some $r \in R$. We'll call w_i *head* of the relation, and w_j the *dependent*.

- $w_i \rightarrow^* w_j$ denotes the *reflexive transitive closure* of the relation operator. That is, it tells that there's an arbitrary number of relations that connects w_i to w_j, so it is either $w_i = w_j$ or both $w_i \rightarrow^* w_{i'}$ and $w_{i'} \rightarrow w_j$ are true for some $w_{i'} \in V$.
  We say that w_j belongs to the *yield* of w_i.

- $w_i \leftrightarrow w_j$ denotes that there is a relation between the two tokens, but does not specify which is the head and which the dependant. So, either $w_i \rightarrow w_j$ or $w_j \rightarrow w_i$ is true.

- $w_i \leftrightarrow^* w_j$ denotes the *reflexive transitive closure* of the undirected relation operator. That is, either $w_i = w_j$ or both $w_i \leftrightarrow^* w_{i'}$ and $w_{i'} \leftrightarrow w_j$ are true.

Now we can state some properties of dependency trees

Property 1. A dependency tree $G = (V, A)$ always satisfies the *root property*, which states that there is no node $w_i \in V$ such that $w_i \rightarrow w_0$

This property comes straightforward from the artificial addition of w_0 to the sentence's tokens as the root of the dependency tree.

Property 2. A dependency tree $G = (V, A)$ always satisfies the *connectedness property*, which states that for every two nodes $w_i, w_j \in V$, $w_i \leftrightarrow^* w_j$ will always be true.

This is also pretty obvious, as all nodes will always be connected if they belong to the yield of the same root. As we will see, it may happen that a parsing process leads to the creation of a non-connected graph. If this is the case, we add a fictitious arc from w_0 to the roots of the single disjoint connected components of the graph.

Property 3. A dependency tree $G = (V, A)$ always satisfies the *single head property*, which states that, for every node w_j, if $w_i \rightarrow w_j$ there is no other node w_k such that $w_k \rightarrow w_j$

Complying with this last property has the consequence to force the graph to be acyclic, as a tree should be. These three properties are all uncorrelated with each other (i.e., none of them can be deduced from any combination of the others), and together they set enough constraints to define a dependency graph as a directed tree rooted at w_0.
There is also a fourth property that, differently from the others, does not stand for all trees:

Property 4. A dependency tree $G = (V, A)$ satisfies the *projectivity property* if, and only if, each yield of all its nodes forms a convex set with respect to precedence (that is, all the node belonging to the yield form a contiguous subsequence of the sentence).

As an example, see the sentence in Figure 2.3 and its associated dependency tree.

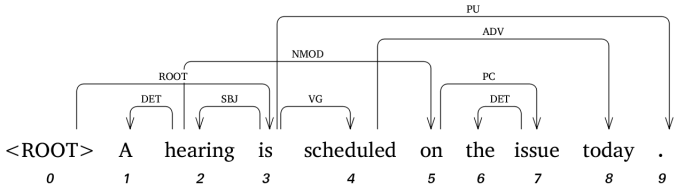


Figure 2.3: Example of a non-projective dependency tree

As you can see, considering w_2, we can see that its yield (the set $\{w_1, w_2, w_5, w_6, w_7\}$) is not a convex set, as w_3 and w_4 are missing. We call the arc closing that gap (in this case the arc (w_2, NMOD, w_5)) a *non-projective* arc. A direct consequence of non-projectivity is that if we draw all the arcs on the same semiplane respect to the sentence, non-projective arcs will always cross the path of one or more of the others.

Non-projective structures are quite rare in English language, but they are noticeably frequent in Nordic languages like Danish and Swedish. Note that this property is greatly affected by the order of the tokens that compose the sentence. In fact, any dependency tree can always be made projective if we change this order. This will come more clear in the next chapter, when we'll show how we can parse non-projective structures.

2.3 Dependency parsing models

There is a great variety of approaches that can be used to solve the dependency parsing problem (i.e., the task of automatically building the dependency structure of a given input sentence). Broadly speaking, we can divide them in two classes: *grammar-based* approaches, which use formal grammars and define formal languages to categorize sentences; and *data-driven* approaches, which use machine learning techniques over annotated linguistic data. In this essay we will focus on *transition-based parsers*, an example of the latter category which make use of *supervised* machine learning.

Namely, the data-driven approach divides the whole problem in two phases: the *learning phase*, where a *parsing model* is inferred through the analysis of annotated sentence sets called *treebanks*; and the *parsing phase*, where the parsing model is used to build brand new dependency trees from plain sentences.

2.3.1 Transition based parsers

Transition-based parsers consist in two components: a transition system, which represents the parsing model that must be induced in the learning phase; and a parsing algorithm that, given the sentence, uses the transition system to build the dependency tree.

Transition system

First of all, we must define what the transitions are made over.

Definition 3. A *configuration* is a triple $c = (\Sigma, B, A)$ such that Σ and B are two disjoint sublists of V_x, the ordered list of all tokens of a sentence x, and A is a set of arc over the elements of V_x.

We will call Σ and B the *stack* and the *buffer* of the transition system, respectively. As an example, given the sentence seen in Figure 2.1, a valid configuration could be $c_k = ([w_0, w_2, w_3, w_4], [w_5, w_6, w_7, w_8, w_9], \{(w_2, \text{ATT}, w_1)\})$.

With this definition, we can now say what a transition system is.

Definition 4. A *transition system* for dependency parsing can be defined as a quadruple $S = (C, T, c_s, C_t)$, where

- C is a set of *configurations,*
- T is a set of *transitions*, which are functions in the form $t : C \mapsto C$ used to modify the configurations,
- c_s is an *initialization function*, which maps a sentence x to an initial configuration $c \in C$,
- $C_t \subseteq C$ is a set of *terminal configurations.*

In this essay, we will use a dummy initialization function that creates a standard starting configuration, $c_s(x) = ([w_0], [w_1, \ldots, w_n], \{\})$, for any sentence $x = w_0 \ldots w_n$. Also, we will define the set of terminal configurations C_t as composed by all configurations of the form $c = ([w_0], [], A)$, with a generic set of arcs A. The parsing algorithm, then, will use transitions contained into T to switch step by step from the initial configuration to one of those contained in C_t, generating a transition sequence.

Definition 5. A *transition sequence* for a sentence x is a sequence of configurations $C_{0,m} = (c_0, c_1, \ldots, c_m)$ where:

- $c_0 = c_s(x)$, i.e., the first configuration is the one generated with the initialization function.
- $c_m \in C_t$, i.e., the last configuration must be contained in the terminal configurations set.
- $\forall i$ with $1 < i \leq m$, $c_i = t(c_{i-1})$ for some $t \in T$, i.e., every configuration, apart from the first, is the result of a transition of the previous one.

Eventually, the result of the parsing process will be a dependency graph $G_{c_m} = (V_x, A_{c_m})$, with A_{c_m} as the arc set in configuration c_m. To make sure that the parsed graph is also a tree, the transition set will ensure that the three necessary properties defined in section 2.2 are achieved.

Parsing algorithm

The last statement of definition 5 is crucial to define what a deterministic parsing algorithm must know: for every configuration of the sequence (except the terminal ones) it must choose which transition function to use, in order to create the dependency tree. This choice is made by the *oracle* function, that gives us the optimal transition for every configuration.
Defining an oracle function is the central problem in dependency parsing, and it's the goal of the classification phase of the whole parsing process, which lies outside the focus of this essay.

Given this function, though, the parsing algorithm is straightforward:

```
1: function PARSE(o, x)
2:     c ← c_s(x)
3:     while c ∉ C_t do
4:         t ← o(c)
5:         c ← t(c)
6:     end while
7:     return G_c
8: end function
```

The only component still not specified is the transition function set. We will use a quite common approach for generic parsing, the *shift-reduce* technique. Basically, we need a way to transfer tokens from the buffer to the stack (a *shift* operation), and a way to remove them from the stack, replacing them with a subtree they belong to, while adding some arc to the arc set (a *reduce* operation). A minimal example of transition set (for projective sentences' parsing only) is equipped with the following transition functions:

- the SHIFT transition is identical to the one used in common shift/reduce parsers: it removes the first token from the buffer and pushes it to the top of the stack. It can always be done, given that the buffer is non-empty. Formally,

$$\text{SHIFT}((\sigma|w_i, w_j|\beta, A_c)) = (\sigma|w_i|w_j, \beta, A_c)$$

- the LEFT-ARC$_r$ transition, given any label $r \in R$, looks at the top two elements of the stack, say w_i and w_j, and adds the arc (w_j, r, w_i) to the arc set. Also, it removes w_i from the stack. This is permitted only if $i \neq 0$, otherwise it violates the root property. Formally,

$$\text{LEFT-ARC}_r((\sigma|w_i|w_j, B, A_c)) = (\sigma|w_j, B, A_c \cup \{(w_j, r, w_i)\})$$

- the RIGHT-ARC$_r$ transition, for any label $r \in R$, works similarly to LEFT-ARC$_r$. It takes the first two elements of the stack, again we'll call them w_i and w_j, and adds the arc (w_i, r, w_j) to the arc set. Additionally, it pops w_j from the stack. Formally,

$$\text{RIGHT-ARC}_r((\sigma|w_i|w_j, B, A_c)) = (\sigma|w_i, B, A_c \cup \{(w_i, r, w_j)\})$$

These transitions define the *Arc-Standard* parsing system, which was firstly hinted in [Abney and Johnson, 1991] and is the most simple example of a dependency parser for projective sentences only.

As we already said, the parsing system will build a dependency graph over a sentence. We would like to know, though, if, given these transition

functions, the graph built is also a dependency tree. To find this out, we look at the three properties defined in section 2.2, and check if they are fullfilled:

- *Root property*: the only way an arc with w_0 as a tail could be added would be if a LEFT-ARC transition is used on a configuration with $\Sigma = [w_0, w_1]$. However, we know that it is not permitted in this case, so the root property will be always complied.
- *Connectedness property*: the terminal configurations we defined for this system must all have an empty buffer and a stack that contains w_0 only. As the only way to remove a token from the stack-buffer combined list is by adding an arc with the token to remove as the tail, we know for sure that there won't be any token not connected to the others.
- *Single head property*: the two transitions that augment the arc set also remove from the stack the tail of the dependency they create. For this reason, it's impossible for a token to be tail of more than one dependency.

So, since all properties are valid for this system, it will always produce a well-formed dependency tree.

2.3.2 Application example

Having defined all the components of a dependency parsing system, we hereby provide an example of execution of the parsing algorithm over the sentence provided in Figure 2.1. We will use the transitions we just defined, and a simple, deterministic oracle function that can access the arc set A of a valid dependency tree associated to the sentence (this is obviously an ideal oracle, in practice it is approximated by the classifier built from the treebank). Basically, given a generic configuration $c = (\sigma|w_i|w_j, \beta, A_c)$, it checks these conditions:

- if $(w_j, r, w_i) \in A$, for any $r \in R$, it predicts $t = \text{LEFT-ARC}_r$;
- if $(w_i, r, w_j) \in A$, for any $r \in R$, and if there is no other arc $(w_j, r', w_l) \in A$ but not belonging to A_c, it predicts $t = \text{RIGTH-ARC}_r$;
- if both the two previous conditions aren't met, it predicts $t = \text{SHIFT}$.

The additional condition imposed for the choice of $t = \text{RIGTH-ARC}_r$ is crucial for the possibility of the parsing algorithm to reach a terminal configuration, that is, to empty the stack. In fact, if a node of the tree that is head of some relation is removed before all of his dependants, these latter tokens will never be put beside their "father" in the stack, and thus never

popped out of it, causing the algorithm to fail.
This system, with slightly different variants, is widely used in dependency parsing literature. Table 2.1 shows the stack and buffer content in the configuration sequence, the transition the oracle chooses for all of them, and the arc the two reduce-like transitions add to the arc set (for the sake of clarity, in such tables we will use the real word with its position n in the sentence to represent the node w_n).

Stack	Buffer	Transition	Arc added (if any)
[ROOT$_0$]	[*Economic*$_1, \ldots, .9$]	SHIFT	-
[ROOT$_0$, *Economic*$_1$]	[*news*$_2, \ldots, .9$]	SHIFT	-
[ROOT$_0$, *Economic*$_1$, *news*$_2$]	[*had*$_3, \ldots, .9$]	LEFT-ARC$_{\text{ATT}}$	(w_2, ATT, w_1)
[ROOT$_0$, *news*$_2$]	[*had*$_3, \ldots, .9$]	SHIFT	-
[ROOT$_0$, *news*$_2$, *had*$_3$]	[*little*$_4, \ldots, .9$]	LEFT-ARC$_{\text{SBJ}}$	(w_3, SBJ, w_2)
[ROOT$_0$, *had*$_3$]	[*little*$_4, \ldots, .9$]	SHIFT	-
[ROOT$_0$, *had*$_3$, *little*$_4$]	[*effect*$_5$,. . . ,.9]	SHIFT	-
[ROOT$_0, \ldots,$ *little*$_4$, *effect*$_5$,]	[*on*$_6, \ldots, .9$]	LEFT-ARC$_{\text{ATT}}$	(w_5, ATT, w_4)
[ROOT$_0$, *had*$_3$, *effect*$_5$]	[*on*$_6, \ldots, .9$]	SHIFT	-
[ROOT$_0, \ldots,$ *effect*$_5$, *on*$_6$]	[*financial*$_7, \ldots, .9$]	SHIFT	-
[ROOT$_0, \ldots,$ *on*$_6$, *financial*$_7$]	[*markets*$_8, .9$]	SHIFT	-
[ROOT$_0, \ldots,$ *financial*$_7$, *markets*$_8$]	[.9]	LEFT-ARC$_{\text{ATT}}$	(w_8, ATT, w_7)
[ROOT$_0, \ldots,$ *on*$_6$, *markets*$_8$]	[.9]	RIGHT-ARC$_{\text{PC}}$	(w_6, PC, w_8)
[ROOT$_0, \ldots,$ *effect*$_5$, *on*$_6$]	[.9]	RIGHT-ARC$_{\text{ATT}}$	(w_5, ATT, w_6)
[ROOT$_0$, *had*$_3$, *effect*$_5$]	[.9]	RIGHT-ARC$_{\text{OBJ}}$	(w_3, OBJ, w_5)
[ROOT$_0$, *had*$_3$]	[.9]	SHIFT	-
[ROOT$_0$, *had*$_3$, .9]	[]	RIGHT-ARC$_{\text{PU}}$	(w_3, PU, w_9)
[ROOT$_0$, *had*$_3$]	[]	RIGHT-ARC$_{\text{PRED}}$	(w_0, PRED, w_3)
[ROOT$_0$]	[]	STOP	-

Table 2.1: Execution of the parsing algorithm on the sentence in Figure 2.1

As we can see, in the sixth row we had w_0 and w_3 at the top of the stack but the oracle correctly suggested to use the SHIFT transition, even if the arc $(w_0, \text{PRED}, w_3) \in A$.

Chapter 3

Non-projective dependency parser

3.1 Definition of projective order

As it has already been noted, we can't say that a dependency tree is intrinsically projective or not: in fact, this property is related with the order of the tokens of the sentence the tree is associated with. This has the consequence of making projectivity a crucial property when executing parsing algorithms that, as we have seen in the previous chapter, are greatly affected by this order. As an example, if we try to parse the sentence in Figure 2.3 with the Arc-standard parsing system, we get the result shown in Table 3.1.

Stack	Buffer	Transition	Arc added (if any)
$[\text{ROOT}_0]$	$[A_1, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, A_1]$	$[hearing_2, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, A_1, hearing_2]$	$[is_3, \ldots, .9]$	LEFT-ARC$_{\text{DET}}$	(w_2, DET, w_1)
$[\text{ROOT}_0, hearing_2]$	$[is_3, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, hearing_2, is_3]$	$[scheduled_4, \ldots, .9]$	LEFT-ARC$_{\text{SBJ}}$	(w_3, SBJ, w_2)
$[\text{ROOT}_0, is_3]$	$[scheduled_4, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, is_3, scheduled_4]$	$[on_5, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, \ldots, scheduled_4, on_5]$	$[the_6, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, \ldots, on_5, the_6]$	$[issue_7, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, \ldots, the_6, issue_7]$	$[today_8, .9]$	LEFT-ARC$_{\text{DET}}$	(w_7, DET, w_6)
$[\text{ROOT}_0, \ldots, on_5, issue_7]$	$[today_8, .9]$	RIGHT-ARC$_{\text{PC}}$	(w_5, PC, w_7)
$[\text{ROOT}_0, \ldots, scheduled_4, on_5]$	$[today_8, .9]$	SHIFT	-
$[\text{ROOT}_0, \ldots, on_5, today_8]$	$[.9]$	SHIFT	-
$[\text{ROOT}_0, \ldots, today_8, .9]$	$[]$	ERROR	-

Table 3.1: Attempt of execution of the parsing algorithm of the previous chapter on the sentence in Figure 2.3

As expected the algorithm failed, as it can't reach any valid terminal configurations. Analysing the parsing sequence, we note that we start to go off track on the fifth step, when we add (w_2, DET, w_1) to the arc set even if we couldn't link w_2 with its right subtree. That's because we can't extract w_3 and w_4, which separate w_2 from w_5, because they can't be connected to their dependent w_8. If we try to rearrange the list of the tokens, though, we can obtain a projective tree, as we can see in Figure 3.1.

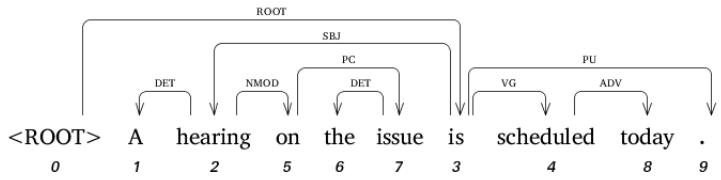


Figure 3.1: Example of a different order for the sentence in Figure 2.3

Any order of the tokens of a sentence that can be associated to a projective tree is called a *projective order*. As we already mentioned before, for every tree it is possible to define a projective order. This will be proved later on in this chapter, as will be the very basis of the definition of a new parsing system able to directly build non-projective trees.

3.2 Nivre's non-projective dependency parser definition

The Swedish researcher Joakim Nivre, in his already cited 2009 paper, presented a simple extension of the Arc-standard system with the aim to expand the set of feasible trees to all dependency trees. We will call it *Nivre parsing system.*

3.2.1 Transition system

The transition system is very similar to the one used in the Arc-standard. It uses equally structured configurations, the same starting and terminal functions but a different transition set: in addition to the SHIFT, LEFT-ARC$_r$ and RIGHT-ARC$_r$ transitions, it has a fourth one, defined as follows

- the SWAP transition, given a configuration like $c = (\sigma|w_i|w_j, \beta, A_c)$, takes w_i and pushes it back in the buffer, changing its relative order with w_j. This is only permitted if $0 < i < j$. Formally,

$$\text{SWAP}(\sigma|w_i|w_j, \beta, A_c) = (\sigma|w_j, w_i|\beta, A_c)$$

The condition imposed to the permissibility of the operation prevents two nodes to be swapped more than once, avoiding deadlocks. It is clear that, if we now can change the order of the nodes, we can also rearrange them so as to push them in the stack in the projective order. Also, note that the transition does not swap two elements keeping them in the stack, but it pops an element out and pushes it back in the buffer, allowing the element that was previously on the top to be linked to the lower elements of the stack.

3.2.2 Parsing algorithm

In Nivre system it is used the same parsing algorithm we used in the previous chapter, but obviously the oracle function must be modified to include the case when a SWAP is needed. Therefore, we need to find a way to define a projective order for any possible dependency tree. An example can be an inorder traversal of the tree itself.

Definition 6. The *inorder* traversal of a tree processes all its nodes visiting recursively the left subtree, the root, and the right subtree, respectively.

For example, given the tree in Figure 3.2, if we print the label of each node while visiting them, the result will be "D B E A F C G".

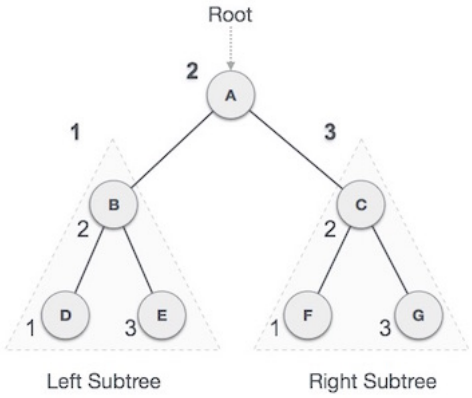


Figure 3.2: Example of a tree

If we execute an inorder traversal of a projective tree, the result will always follow the original token order of the sentence. If we take any non-projective tree, instead, it won't follow it, but we are sure that the arcs connecting the nodes of the tree won't cross each other, so none of them will be non-projective with respect to the new order. So, the inorder visit will always define a sequence of token in some projective order. We will use the symbol $<_P$ to say that a node comes before another one in this projective order. Figure 3.1 shows precisely the inorder sequence of that tree: when we reach node w_2, we first visit its right subtree before returning to its father w_3.

Now it is possible to define the oracle function used in the Nivre system, which chooses the optimal transition t for a generic configuration $c = (\sigma|w_i|w_j, w_k|\beta, A_c)$ (that is, the one that will let to build a given graph $G = (V, A)$):

- if $(w_j, r, w_i) \in A$, for any relation type r, and if there is no other arc $(w_i, r', w_l) \in A$ but not belonging to A_c, it predicts $t = \text{LEFT-ARC}_r$;
- if $(w_i, r, w_j) \in A$, for any relation type r, and if there is no other arc $(w_j, r', w_l) \in A$ but not belonging to A_c, it predicts $t = \text{RIGHT-ARC}_r$;
- if $w_j <_P w_i$, it predicts $t = \text{SWAP}$;
- otherwise, it predicts $t = \text{SHIFT}$.

Note that the double condition we have previously imposed only to the second choice must now be applied also to the first one, as a previous SWAP operation may have moved an incomplete node deeper into the stack. Applying the whole algorithm to the same non-projective sentence we tried to parse earlier, we get the transition sequence shown in Table 3.2.

As we can see, the SWAP transition is used to move w_3 and w_4 after the group w_5, w_6, w_7, in order to be able to link w_2 to w_5.
It is important to note that if the target tree associated to the input sentence is projective, the transition sequence will be identical to the one produced by the Arc-Standard system, as no SWAP operations would be necessary.

3.3 Time complexity

The time needed for the complete parsing of a sentence is obviously directly related to its length n. If we use a deterministic oracle function, such as those we defined until now, we can ignore the real classification and decision process and just consider that every prediction and transition will always need constant time to complete. Therefore, we can use the number of transitions needed to convert the initial configuration into a terminal one as a direct measure of the total execution time of the parsing algorithm.

If we consider the Arc-Standard system, as we must move all tokens from the buffer to the stack and then remove them leaving only the root, we will always need n SHIFT operations + n LEFT/RIGHT-ARC transitions to parse a sentence of n words. So the time complexity will be $T_{as}(n) = c \cdot 2n = \mathcal{O}(n)$, with c as the constant execution time of $o(c)$ and $t(c)$.

Regarding Nivre system, if the target is a projective structure the execution time will obviously be identical. If not, though, the transition sequence will surely contain a SWAP operation. Consequently, as this transition pushes back to the buffer an element from the stack, it will be necessary an additional SHIFT to restore the cardinality of the buffer. For this reason,

Stack	Buffer	Transition	Arc added (if any)
[ROOT$_0$]	[$A_1, \ldots, .9$]	SHIFT	-
[ROOT$_0$, A_1]	[$hearing_2, \ldots, .9$]	SHIFT	-
[ROOT$_0$, A_1, $hearing_2$]	[$is_3, \ldots, .9$]	LEFT-ARC$_{\text{DET}}$	(w_2, DET, w_1)
[ROOT$_0$, $hearing_2$]	[$is_3, \ldots, .9$]	SHIFT	-
[ROOT$_0$, $hearing_2$, is_3]	[$scheduled_4, \ldots, .9$]	SHIFT	-
[ROOT$_0$, $\ldots$, is_3, $scheduled_4$]	[$on_5, \ldots, .9$]	SHIFT	-
[ROOT$_0$, $\ldots$, $scheduled_4$, on_5]	[$the_6, \ldots, .9$]	SWAP	-
[ROOT$_0$, $\ldots$, is_3, on_5]	[$scheduled_4, \ldots, .9$]	SWAP	-
[ROOT$_0$, $hearing_2$, on_5]	[$is_3, \ldots, .9$]	SHIFT	-
[ROOT$_0$, $\ldots$, on_5, is_3]	[$scheduled_4, \ldots, .9$]	SHIFT	-
[ROOT$_0$, $\ldots$, is_3, $scheduled_4$]	[$the_6, \ldots, .9$]	SHIFT	-
[ROOT$_0$, $\ldots$, $scheduled_4$, the_6]	[$issue_7, \ldots, .9$]	SWAP	-
[ROOT$_0$, $\ldots$, is_3, the_6]	[$scheduled_4, \ldots, .9$]	SWAP	-
[ROOT$_0$, $\ldots$, on_5, the_6]	[$is_3, \ldots, .9$]	SHIFT x 3	-
[ROOT$_0$, $\ldots$, $scheduled_4$, $issue_7$]	[$today_8, .9$]	SWAP	-
[ROOT$_0$, $\ldots$, is_3, $issue_7$]	[$scheduled_4, \ldots, .9$]	SWAP	-
[ROOT$_0$, $\ldots$, the_6, $issue_7$]	[$is_3, \ldots, .9$]	LEFT-ARC$_{\text{DET}}$	(w_7, DET, w_6)
[ROOT$_0$, $\ldots$, on_5, $issue_7$]	[$is_3, \ldots, .9$]	RIGHT-ARC$_{\text{PC}}$	(w_5, PC, w_7)
[ROOT$_0$, $hearing_2$, on_5]	[$is_3, \ldots, .9$]	RIGHT-ARC$_{\text{NMOD}}$	(w_2, NMOD, w_5)
[ROOT$_0$, $hearing_2$]	[$is_3, \ldots, .9$]	SHIFT	-
[ROOT$_0$, $hearing_2$, is_3]	[$scheduled_4, \ldots, .9$]	LEFT-ARC$_{\text{SBJ}}$	(w_3, SBJ, w_2)
[ROOT$_0$, is_3]	[$scheduled_4, \ldots, .9$]	SHIFT	-
[ROOT$_0$, is_3, $scheduled_4$]	[$today_8, .9$]	SHIFT	-
[ROOT$_0$, $\ldots$, $scheduled_4$, $today_8$]	[$.9$]	RIGHT-ARC$_{\text{ADV}}$	(w_4, ADV, w_8)
[ROOT$_0$, is_3, $scheduled_4$]	[$.9$]	RIGHT-ARC$_{\text{VG}}$	(w_3, VG, w_4)
[ROOT$_0$, is_3]	[$.9$]	SHIFT	-
[ROOT$_0$, is_3, $.9$]	[]	RIGHT-ARC$_{\text{PU}}$	(w_3, PU, w_9)
[ROOT$_0$, is_3]	[]	RIGHT-ARC$_{\text{ROOT}}$	(w_9, ROOT, w_3)
[ROOT$_0$]	[]	STOP	-

Table 3.2: Nivre system's parsing sequence of the sentence in Figure 2.3. Projective order used: $[w_0, w_1, w_2, w_5, w_6, w_7, w_3, w_4, w_8, w_9]$

every SWAP needed adds two steps to the total count, so $T_N(n) = c(2n+2k)$ with k SWAPs done. To find the asymptotic complexity, we must know how k is related to n. As we said, this transition can only be executed when the two nodes at the top of the stack follow the original token order. Hence, the number of SWAPs is capped by $\frac{n(n-1)}{2}$, which means that in the worst case $T_N = c(2n + 2 \cdot \frac{n^2 - n}{2}) = c(n^2 + n) = \mathcal{O}(n^2)$. This is a pretty high cap; in fact, experimental studies show that the real time complexity is more close to linear than to quadratic performances. In the next chapter we'll show

some more specific observations done on the matter, as we'll try to understand how the projective order used influences the length of the transition sequences.

Chapter 4

An improvement to Nivre parsing system

When we use the Nivre system's parsing algorithm, it is clear that the main factor that affects the execution time is the number of Swaps needed, and so, being its cause, how "far" from their original positions are the tokens in the projective order. If we could find a way to relate this distance to the execution time of the parsing process, we could then imagine how an optimal projective order can look like, and use that instead of the inorder traversal sequence. For this reason, first of all we have to find a way to measure how much scrambled any sort is with respect to the original token order.

4.1 Definition of displacement score

As it has been observed in [Havelka, 2007] and [Kuhlmann and Nivre, 2006], the vast majority of non-projective structures are quite simple, with a single group of tokens placed in the middle of a connected sequence. Therefore, the projective order will differ from the original one only by the position of these tokens. We will talk about this difference as the displacement score of a group of tokens. Basically, it is defined as follows:

Definition 7. For every token, we define its *displacement score* as the difference from the position it has in the original order (that is, its subscript) and the one it has in a given sort. We will call every consecutive subsequence of tokens with the same score a *displaced group*.

For example, if a valid projective order of a sentence of nine words is $[w_0, w_1, w_2, w_3, w_4, w_8, w_5, w_6, w_7, w_9]$, we will have two displaced groups: one composed by $[w_5, w_6, w_7]$, with displacement score $+1$, and one composed by $[w_8]$ only, which has displacement score -3.

A simple property of this quantity is that the displacement scores of every token in a sentence will always sum to 0, as the change of position of

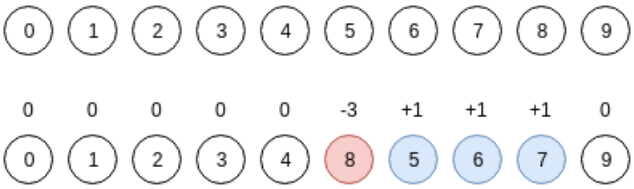


Figure 4.1: Visualization of the sentence used in section 4.1

a group can be always seen as the consequence of another shift.

4.2 Finding a better order

We will now take a closer look to the Nivre parsing algorithm, and in particular to which conditions lead to the necessity of a SWAP transition. We know that, without any non-projective structures, the projective order is equal to the original word order, so the tokens are naturally pushed from the buffer to the stack following the correct sequence. If the projective order is different, though, the oracle will predict the need of a SWAP as soon as the two tokens on the top of the stack are reciprocally unordered, moving back by one position the top element in the combined sequence of stack and buffer. This will continue until the token that is being "overtaken" by the SWAPs will be in the desired position, and the just mentioned combined sequence will follow the projective order.

For example, let's imagine a parsing system which only uses the SWAP and SHIFT transitions, whose oracle predicts the first one in the same cases in which the oracle in Nivre system does (it is obvious that this system will only rearrange the tokens until the projective order is followed). If we try to use this system on the hypothetical sentence of section 4.1, the transition sequence would be the one showed in Table 4.1.

Stack	Buffer	Transition	Arc added (if any)
$[w_0]$	$[w_1, \ldots, w_9]$	SHIFT	-
$[w_0, w_1]$	$[w_2, \ldots, w_9]$	SHIFT	-
$[w_0, w_1, w_2]$	$[w_3, \ldots, w_9]$	SHIFT x 6	-
$[w_0, \ldots, w_5, w_6, w_7, w_8]$	$[w_9]$	SWAP	-
$[w_0, \ldots, w_5, w_6, w_8]$	$[w_7, w_9]$	SWAP	-
$[w_0, \ldots, w_5, w_8]$	$[w_6, w_7, w_9]$	SWAP	-
$[w_0, \ldots, w_4, w_8]$	$[w_5, w_6, w_7, w_9]$	SHIFT x 4	-
$[w_0, \ldots, w_8, w_5, w_6, w_7, w_9]$	[]	STOP	-

Table 4.1: Execution of rearranging algorithm defined in section 4.2 over the hypothetical sentence in figure 4.1. Red coloured tokens have negative displacement score, blue coloured have positive score

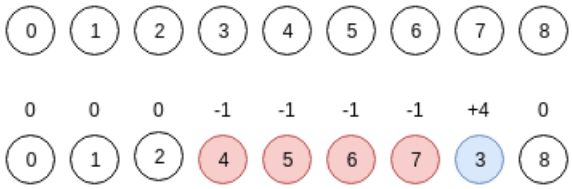


Figure 4.2: Visualization of the sentence of eight words used in the second example in section 4.2

We can see that, to obtain the desired order, we must swap the member of the displaced group with *positive* displacement score over all the tokens of the other group. It is evident, then, that the more tokens we have to rearrange in the sequence, the more transitions are needed. Still, it is also true that in this example we omitted any possibly existent arc between the positive scored tokens $[w_5, w_6, w_7]$. If, for example, this group could be reduced to only one token, then only one SWAP transition would have been needed.

Let's take, as another example, a sentence of eight words with $[w_0, w_1, w_2, w_4, w_5, w_6, w_7, w_3, w_8]$ as projective ordered sequence. As we can see in Figure 4.2, in this case we have a displaced group composed by $[w_4, w_5, w_6, w_7]$ with score -1 and another composed by $[w_3]$ with score $+4$. When we try to parse this sentence, the first time we need a SWAP transition will be when w_3 and w_4 are at the top of the stack. This will push w_3 back in the buffer, so the necessity of a rearrangement will recur as we reach w_5: again w_3 will be pushed back, until it will reach its correct position in the projective ordered sequence, after w_7, as shown in Table 4.2.

This time we had only one token with positive displacement score, and a group with negative score. As the tokens are pushed in the stack following the original order, we will always "meet" nodes with positive score before nodes with negative score. For this reason, it's impossible to add any arc between them before the SWAP transitions, so these possible reductions would be useless in term of reduction of execution time.

So, if we could find a way to compute a projective order that produces more positive scoring tokens than negative scoring ones, we would probably see an improvement in term of length of the transition sequence and, by consequence, of overall time performances. An attempt has been made with a sketchy algorithm we designed, that tries to obtain a valid projective order operating directly on the trees. The basic idea is to locate, for every non-projective arc, the set of tokens that do not belong to the yield of the head of this arc, but still are between its extremes. To make it projective we move these tokens to the left, before the leftmost element between the head and the tail of the examined arc. This way the selected group will get negative displacement score, and all the tokens between its original and final

Stack	Buffer	Transition	Arc added (if any)
$[w_0]$	$[w_1, \ldots, w_8]$	SHIFT	-
$[w_0, w_1]$	$[w_2, \ldots, w_8]$	SHIFT x 3	-
$[w_0, \ldots, w_3, w_4]$	$[w_5, w_6, w_7, w_8]$	SWAP	-
$[w_0, w_1, w_2, w_4]$	$[w_3, w_5, w_6, w_7, w_8]$	SHIFT	-
$[w_0, \ldots, w_4, w_3]$	$[w_5, w_6, w_7, w_8]$	SHIFT	-
$[w_0, \ldots, w_3, w_5]$	$[w_6, w_7, w_8]$	SWAP	-
$[w_0, \ldots, w_4, w_5]$	$[w_3, w_6, w_7, w_8]$	SHIFT	-
$[w_0, \ldots, w_4, w_5, w_3]$	$[w_6, w_7, w_8]$	SHIFT	-
$[w_0, \ldots, w_4, w_5, w_3, w_6]$	$[w_7, w_8]$	SWAP	-
$[w_0, \ldots, w_4, w_5, w_6]$	$[w_3, w_7, w_9]$	SHIFT x 2	-
$[w_0, \ldots, w_4, w_5, w_6, w_3, w_7]$	$[w_8]$	SWAP	-
$[w_0, \ldots, w_4, w_5, w_6, w_7]$	$[w_3, w_8]$	SHIFT x 2	-
$[w_0, \ldots, w_4, w_5, w_6, w_7, w_3, w_8]$	[]	STOP	-

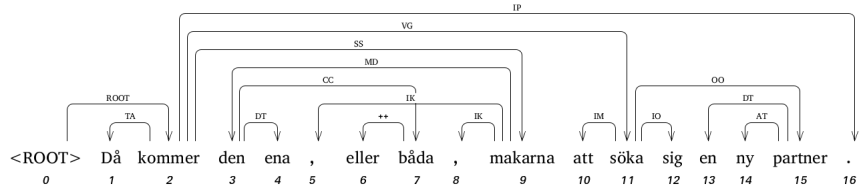
Table 4.2: Execution of rearranging algorithm defined in section 4.2 over a the hypothetical sentence in figure 4.2.

positions will get positive scores. As the moved groups are usually composed by few tokens (as we stated at the beginning of section 4.2), this will usually produce more positive scored tokens than negative scored ones.

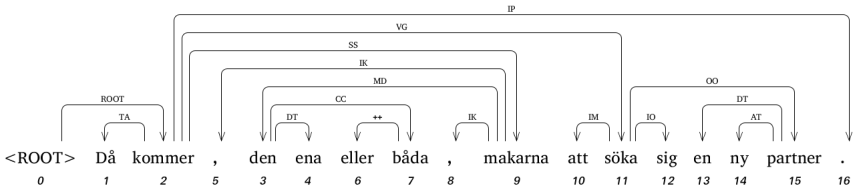
```
1: function ProjSort(x, G(V, A))
2:     for every w_i ∈ V, i ≠ 0 do
3:         w_j ∈ V | a = (w_j, r, w_i) ∈ A
4:         Y ← yield(w_j)
5:         if Y forms a contiguous sequence then
6:             a is projective
7:         else
8:             W ⊂ V | (W ∩ Y = ∅) ∧ (w_k ∈ W ⟺ (i < k < j ∨ j < k < i))
   ▷ W is the set of nodes that makes the yield unconnected
9:             move W before w_i
10:        end if
11:    end for
12: end function
```

For example, if we take the Swedish sentence in Figure 4.3a and apply this simple piece of pseudocode, we obtain the sentence in Figure 4.3b

This approach has a major flaw: the relocation of the token belonging to W is uncontrolled, and can make other arcs non-projective. For example, if we use this approach to the sentence we used in chapter 2 the result would be what can be seen in Figure 4.4. In addition to the sentence being unreadable, the relocation of w_3, which was the responsible of the non-

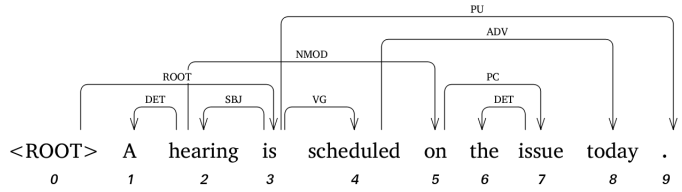


(a)

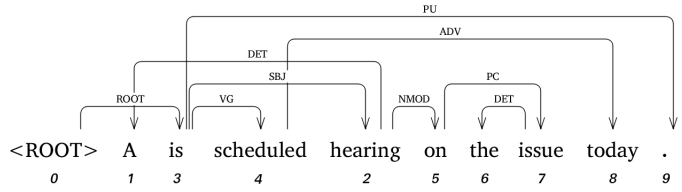


(b)

Figure 4.3: Example of working execution of ProjSort algorithm. Considering (w_3, OC, w_7), the W set contains only w_5, which made the yield of w_3 discontinuous.



(a)



(b)

Figure 4.4: Example of a not working application of ProjSort algorithm.

projectivity of $(w_2, \mathrm{NMOD}, w_5)$, created another non-projective structure, so it was completely useless.

Many tries were made to develop a deterministic way to find a suitable

place to move the W set to without making other yields unconnected. For example, we could place it next to the nearest ancestor of its tokens which is outside the unconnected yield. This may work, as we are sure it won't interpose other yields, but we can't know for sure whether this node is left to the original position or not, so we could end up making a relocation that will be useless in term of parsing performances. Also, if one of the tokens in W set has other dependents, the arc connecting to them can become not-projective.

Due to not being able to determine a generally working algorithm, we switched to another approach: a two-step algorithm who parses a sentence with two different transition sets.

4.3 TwoStep parsing system

While looking more deeply at the Nivre parsing algorithm, we understood that if we could reduce a sentence to as few tokens as possible, the number of SWAP transitions needed would be reduced. We know that, if we have a token with positive displacement score, the oracle will force us to move it to its correct position in the projective ordered sequence before checking if there is any arc between the negative scored tokens, as we have seen in Table 4.2.

Therefore, any reduction will be made after it would have been useful. Instead of looking for a different order, we tried to modify the whole parsing system in order to overcome this flaw. We kept the same basis of the two transition based systems we have defined, changing only the transition sets and oracle functions used by the parsing algorithm. The latter has been divided in two different phases, each of them generating a different transition sequence.

4.3.1 First step: "Arc-standard"-like reduction

The first phase aims to add every possible arc we can *without the use of* SWAP *transitions*. To do this we use a transition set with the same LEFT-ARC, RIGHT-ARC and SHIFT transitions we already defined, plus a SAVE operation defined as follows:

- the SAVE transition saves the number of elements contained in the stack Σ_c of the configuration c is used onto (we will call it s_p), then it operates a SHIFT transition.

The oracle function for this phase is similar to the one in Nivre system: we use LEFT-ARC and RIGHT-ARC when it exists an arc between the two elements at the top of the stack, checking if the tail has no other dependents still in the stack or the buffer. We will use the SAVE transition only once:

when we *first* meet a couple of tokens not following the used projective order, that is, the first time the oracle we used with Nivre system would have predicted a SWAP transition. Instead of swapping, we store the point in the sentence we reached and we continue to analyze what remains of it, adding any existent arc. This way, when we meet the positively displaced tokens, we ignore them and keep shifting, reaching the negative scored group and reducing it as much as possible.

The result of this phase is a transition sequence leading to a non-terminal configuration, as it happened when we tried to parse a non-projective sentence with the Arc-standard system (see Table 3.1). The last configuration of this sequence c_f will have an empty buffer and a stack containing all the tokens that was not possible to reduce. This will be used, together with s_p, to define the starting configuration of the second step.

4.3.2 Second step: reset and complete

In the second phase of the algorithm, we complete the parsing of the sentence using the same transition set and oracle function as in the Nivre system. Instead of considering the whole sentence, though, we use a custom initial configuration:

- the stack will contain the first s_p elements, starting from the bottom, of the stack in the final configuration of the first phase c_f.
- The buffer will contain all the remaining elements of Σc_f not inserted in the new one.
- The arc set will be the same of the one in c_f.

In this way we'll start with the stack containing the same elements contained by the one that leads to the SAVE transition in the first phase, while the buffer will hold only the tokens not reduced in the first phase. We are sure that this will not affect the correctness of the parsing algorithm, because even if we parse all the tokens in Σc_f we will eventually reach this configuration with repeated SHIFT transitions, as all other cases have been checked in the first step.
Making the algorithm analyze even a small subset of the same sentence twice may seem to worsen its performances: in fact, a considerable overhead is introduced by this approach. In the next chapter we will see how the advantages brought by the additional reductions we manage to operate in this way still lead to an overall reduction of the transitions needed.

Table 4.3 shows the two transition sequences generated by this algorithm over the same sentence parsed with Nivre system in chapter 3. In Figure 4.5 we recall the projective order produced by the inorder traversal and compute its displacement scores.

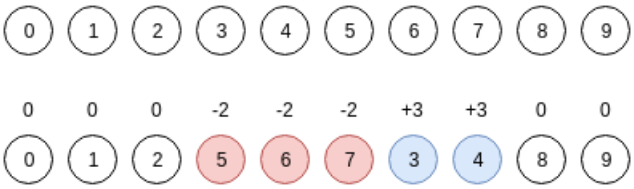


Figure 4.5: Displacement scores for the inorder traversal of the dependency tree in Figure 2.3.

Stack	Buffer	Transition	Arc added (if any)
$[\text{ROOT}_0]$	$[A_1, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, A_1]$	$[hearing_2, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, A_1, hearing_2]$	$[is_3, \ldots, .9]$	LEFT-ARC$_{\text{DET}}$	(w_2, DET, w_1)
$[\text{ROOT}_0, hearing_2]$	$[is_3, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, hearing_2, is_3]$	$[scheduled_4, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, \ldots, is_3, scheduled_4]$	$[on_5, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, \ldots, scheduled_4, on_5]$	$[the_6, \ldots, .9]$	SAVE	$*s_p = 5*$
$[\text{ROOT}_0, \ldots, on_5, the_6]$	$[issue_7, today_8, .9]$	SHIFT	-
$[\text{ROOT}_0, \ldots, the_6, issue_7]$	$[today_8, .9]$	LEFT-ARC$_{\text{DET}}$	(w_7, DET, w_6)
$[\text{ROOT}_0, \ldots, on_5, issue_7]$	$[today_8, .9]$	RIGHT-ARC$_{\text{PC}}$	(w_5, PC, w_7)
$[\text{ROOT}_0, \ldots, scheduled_4, on_5]$	$[today_8, .9]$	SHIFT	-
$[\text{ROOT}_0, \ldots, on_5, today_8]$	$[.9]$	SHIFT	-
$[\text{ROOT}_0, \ldots, today_8, .9]$	$[]$	STOP	-
$[\text{ROOT}_0, hearing_2, is_3, scheduled_4, on_5]$	$[today_8, .9]$	SWAP	-
$[\text{ROOT}_0, hearing_2, is_3, on_5]$	$[scheduled_4, today_8, .9]$	SWAP	-
$[\text{ROOT}_0, hearing_2, on_5]$	$[is_3, \ldots, .9]$	RIGHT-ARC$_{\text{NMOD}}$	(w_2, NMOD, w_5)
$[\text{ROOT}_0, hearing_2]$	$[is_3, \ldots, .9]$	SHIFT	-
$[\text{ROOT}_0, hearing_2, is_3]$	$[scheduled_4, today_8, .9]$	LEFT-ARC$_{\text{SBJ}}$	(w_3, SBJ, w_2)
$[\text{ROOT}_0, is_3]$	$[scheduled_4, today_8, .9]$	SHIFT	-
$[\text{ROOT}_0, is_3, scheduled_4]$	$[today_8, .9]$	SHIFT	-
$[\text{ROOT}_0, is_3, scheduled_4, today_8]$	$[.9]$	RIGHT-ARC$_{\text{ADV}}$	(w_4, ADV, w_8)
$[\text{ROOT}_0, is_3, scheduled_4]$	$[.9]$	RIGHT-ARC$_{\text{VG}}$	(w_3, VG, w_4)
$[\text{ROOT}_0, is_3]$	$[.9]$	SHIFT	-
$[\text{ROOT}_0, is_3, .9]$	$[]$	RIGHT-ARC$_{\text{PV}}$	(w_3, PV, w_9)
$[\text{ROOT}_0, is_3]$	$[]$	RIGHT-ARC$_{\text{ROOT}}$	(w_0, ROOT, w_3)
$[\text{ROOT}_0]$	$[]$	STOP	

Table 4.3: Execution of the two-step algorithm over the non-projective sentence of Figure 2.3 using the inorder traversal sequence as projective order.

As you can see, we managed to complete the parsing using only two SWAP transitions instead of six. In fact, ignoring the reciprocal disorder of w_4 and w_5 and all the subsequent ones, we managed to reduce the three negatively displaced group $[w_5, w_6, w_7]$ into only one token. Globally, the number of

transitions needed is significantly lower than before: 24 transitions against 30, an improvement of 20%, so we can say that the overhead has been well compensated.

4.4 Related work

A similar approach has been used in [Nivre et al., 2009]. In that work, in order to reduce the number of swaps the oracle function has been modified, so that it delays the prediction of this transition until it is strictly necessary. To know when this point is, it uses the concept of *maximal projective components (MPC)*. It conjectures a previous parse with the same oracle used in [Nivre, 2009] deprived of the SWAP transition case; as it has been already shown, if the input is a non-projective sentence some tokens will remain in the stack, and not all the arcs forming the tree will be present in the last configuration. For each of these nodes, we define a maximal projective component as the set of tokens that can be reached from them using only the added arcs. Thus, the new oracle will predict that a SWAP transition is needed only if the two tokens at the top of the stack don't follow the projective order *and* the next token in the buffer belongs to a different MPC. Formally, given a configuration like $(\sigma|w_i|w_j, w_k|\beta, A_c)$, we'll have to swap only if $j <_p i \wedge MPC(j) \neq MPC(k)$.

In this essay we used a different approach: we considered the computation of the maximal projective components as the first phase of a two-stepped algorithm, instead of taking it for granted, and then operate the needed swaps to complete the parsing process in the second phase. In fact, our main goal is to show that the overall execution time, in term of total number of transitions made (comprehensive of the first phase), is still lower, unlike the one of the cited paper which focuses on the accuracy of the decisions a classifier based on that system would take. This is the main goal of the next chapter, as we will describe the implementation of the two systems we introduced and discuss the results of their execution over some treebanks.

Chapter 5

Statistical comparison of the approaches

To have a more realistic idea on whether the TwoStep approach is to be preferred over Nivre system, we created a Java executable which performs the parsing of a set of sentences knowing the dependency tree that must be built. We used a set of treebanks in different languages to avoid any possible correlation to some typical structure that can be found in a single language. The datasets are stored in plain text files written using the CoNLL-X format, a standard way to represent a dependency tree developed during the *Tenth Conference on Computational Natural Language Learning*, that took place at New York City in 2006. They contains all the tags and annotation needed to define a dependency tree and to train a classificator able to emulate an oracle function.

5.1 Java implementation of the systems

To try to evaluate and compare the two approaches we wrote two Java classes that implements the parsing systems we described. The most important source files used can be found in Appendix A.

We used an abstract class NonProjectiveParsingSystem, based on a set of support classes which represent arcs, nodes and whole trees, to stub for a generic transition based system capable of parsing non-projective structures. It carries two abstract methods, predictAction and execute, that stand for the oracle function and the parsing algorithm respectively, and a set of methods any implementations of these can use. Namely, the findArc subroutine checks, looking in the gold standard tree acquired from the treebank, if there is any addable arc between two given nodes, and the createInorderTraversal function builds the projective ordered sequence the oracle must follow to decide when a SWAP is necessary.

The two implementations are contained in two classes, NivreParsingSys-

tem and TwoStepParsingSystem. The parsing algorithms use a switch-case structure to operate the transitions following the result of the predictAction method. In the TwoStep system we unified the oracles used in the two phases to avoid repeated code and implemented the differences between a SAVE and a SWAP transition directly in the parsing algorithm.

5.2 Simulation results

As we already said, we will use the number of transitions needed to reach a terminal configuration as a direct metric of the execution time of the parsing algorithm. We hereby present the results we got parsing all the sentences of some treebanks in different languages[1]. We reported the composition of the corpora, the number of sentences for which one system needed less transition than the other and the total number of SWAP transitions needed for all sentences. We also presented the total number of transitions used to parse only non-projective sentences, as for projective ones the two systems need the same one.

		Swedish	Danish	Portuguese	English	Dutch
# sentences	Total	11042	5190	9071	12543	13349
	Non-projective	1079	811	1718	655	4865
# sentences parsed with fewer transitions	Nivre	424	209	294	306	3447
	TwoStep	616	565	977	318	1312
# SWAP transitions	Nivre	9444	8296	18144	4787	28154
	TwoStep	2620	1497	4072	1307	16000
% of reduction	-	72.3%	82.0%	77.6%	72.7%	43.2%
# of transitions in non-projective sents.	Nivre	74836	58716	147602	45646	256810
	TwoStep	65100	48004	127372	40953	252860
% of improvement	-	13.0%	18.2%	13.7%	10.3%	2.5%

Table 5.1: Report of the execution of the two parsing algorithms over five treebanks in different languages

We can see that in all cases the TwoStep system outperforms the old Nivre system, reducing the overall time needed to parse the whole corpus. The exact improvement, though, is strongly different from language to language. For example, we observe that in four out of five languages the new system produces shorter transitions sequences for the majority of the sentences, but when it comes to Dutch we have the opposite situation, even if the total number of transitions is still significantly lower. We can also note

[1] All the dataset used are available online. Swedish, Danish, Portuguese and Dutch corpora can all be found on the official site of the *CoNLL-X* shared task (`http://ilk.uvt.nl/conll/free_data.html`, last checked on 11/11/2016), while the English one can be found on `https://github.com/UniversalDependencies/UD_English` (last checked 7/11/2016).

that some languages use non-projective structures more frequently than others: in the English corpus, for example, only 5% of its sentences have such structure. This obviously causes a lower overall improvement percentage, but we can see that the reduction of the number of swaps is similar to the one in other languages.

Looking at the single sentences, we can note some differences between the ones in which one of the two approaches is clearly faster. We can confirm what we discussed in chapter 4: the more left-displaced tokens we have, the more we can gain reducing them before doing the necessary swaps, which will consequently reduce in number. Conversely, we favour Nivre system when the projective order is very similar to the original sequence, so the SWAP transition it uses are all indispensable; when the SAVE transition in TwoStep system is called nearly at the beginning of the sentence, so the second phase introduces a lot of overhead; or when we have a very small negatively displaced group, so Nivre system also reduces what it is possible before doing the swap.

Chapter 6

Conclusion

In this essay we have analyzed the parsing system presented in [Nivre, 2009], discussing the core idea is based on and its major flaws that brings its time complexity to quadratic time in the worst case. We presented a new parsing system that, as we showed, it considerably improves this performance. The new system does not reduce the parsing time for all single sentences, but considering a simulation over a sizeable dataset the overall time is decreased.

We can prove that this system is able to parse any non-projective sentences. That's because we know that the tokens left after the first phase form a connected tree, as we know that the sentence we start from is associated with such a tree and every time we remove a node from the stack we check if all its children has been linked to it. Since the second phase is operated by a system which is proofed to be able to parse every non projective sentence, the new system altogether can do the same.

We may wonder if the number of SWAP transitions we still must execute after the first phase is minimal, or instead there is a way to reduce them further. If this would be proofed to be false, it would lead to an even more performing parsing system that could reach linear execution time. Also, we did not discuss how the accuracy of the resulting system in non-guided parsing. The results in [Nivre et al., 2009] shows that an oracle equipped with the information we extract in our first phase has better scores about prediction of the right arcs, but our separation of the two steps of the algorithm probably affects these results, whether in a positive o negative way it is unknown.

Bibliography

[Abney and Johnson, 1991] Abney, S. P. and Johnson, M. (1991). Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research*, 20(3):233–250.

[Havelka, 2007] Havelka, J. (2007). Beyond projectivity: Multilingual evaluation of constraints and measures on non-projective structures.

[Kubler et al., 2009] Kubler, S., McDonald, R., Nivre, J., and Hirst, G. (2009). *Dependency Parsing*. Morgan and Claypool Publishers.

[Kuhlmann, 2010] Kuhlmann, M. (2010). *Dependency Structures and Lexicalized Grammars: An Algebraic Approach*. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

[Kuhlmann and Nivre, 2006] Kuhlmann, M. and Nivre, J. (2006). Mildly non-projective dependency structures. In *Proceedings of the COLING/ACL on Main conference poster sessions*, pages 507–514. Association for Computational Linguistics.

[Nivre, 2007] Nivre, J. (2007). Incremental non-projective dependency parsing. In *HLT-NAACL*, pages 396–403.

[Nivre, 2009] Nivre, J. (2009). 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: Volume 1-Volume 1*, pages 351–359. Association for Computational Linguistics.

[Nivre, 2013] Nivre, J. (2013). Transition-based parsing.

[Nivre et al., 2009] Nivre, J., Kuhlmann, M., and Hall, J. (2009). An improved oracle for dependency parsing with online reordering. In *Proceedings of the 11th international conference on parsing technologies*, pages 73–76. Association for Computational Linguistics.

[Silveira et al., 2014] Silveira, N., Dozat, T., de Marneffe, M.-C., Bowman, S., Connor, M., Bauer, J., and Manning, C. D. (2014). A gold standard

dependency corpus for English. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC-2014)*.

[Weizenbaum, 1966] Weizenbaum, J. (1966). Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45.

Appendix A

Java code used for simulations

We present a part of the code used to implement the two parsing systems and measure the performances of the two approaches, described in chapter 5.

A.1 NonProjectiveParsingSystem.java

```
import java.util.ArrayList;

public abstract class NonProjectiveParsingSystem {

    final int  LEFT_ARC=0, RIGHT_ARC=1, SWAP=2, SHIFT=3;
    DependencyTree gold = null;
    protected int n_shift, n_op, n_swap, sent_length;

    protected final ArrayList<Integer> projective_order =
       new ArrayList<>();
    abstract int predictAction(ArrayList<Integer> sentence,
        int top);
    abstract DependencyTree execute();

    public NonProjectveParsingSystem(DependencyTree target)
        {
        setTargetTree(target);
    }

    /**
     * Sets the Dependency Tree that we are building
     * @param target the target Dependency tree
     */

    public void setTargetTree(DependencyTree target) {
```

```
        this.gold = target;
        Node root = gold.getNodes().get(0);
        if(root == null)
            return;
        createInorderTraversal(root);
    }

    /**
     * Initializes an ArrayList with the sequence generated
         from the inorder traversal
     * @param n the starting Node
     */
    private void createInorderTraversal(Node n) {
        if(n == null)
            return;

        for(Arc left : n.getLeft_children())
            createInorderTraversal(left.getTail());

        projective_order.add(n.getId());

        for(Arc right : n.getRight_children())
            createInorderTraversal(right.getTail());

    }

    /**
     * Checks if an arc between any two nodes exists
     * @param head_id Id of the head of the wanted arc
     * @param tail_id Id of the tail of the wanted arc
     * @return whether an Arc has been found or not
     */
    protected boolean findArc(int head_id, int tail_id) {
        Arc possible_arc = null;
        if (head_id > tail_id) {
            for (Arc arc : gold.getNodes().get(head_id).
                getLeft_children())
                if (arc.getTail().getId() == tail_id)
                    possible_arc = arc;
        }
        else
            for (Arc arc : gold.getNodes().get(head_id).
                getRight_children()) {
                if (arc.getTail().getId() == tail_id)
                    possible_arc = arc;
            }

        //Check if the found arc has a complete tail (we
            have already found all his tail's children)
```

```
            boolean complete = true;
            if (possible_arc != null) {
                Node possible_tail = possible_arc.getTail();
                for (Arc child : possible_tail.getLeft_children
                    ())
                    if (!child.isAdded()) {
                        complete = false;
                        break;
                    }
                for (Arc child : possible_tail.
                    getRight_children())
                    if (!child.isAdded()) {
                        complete = false;
                        break;
                    }
                if (complete) {
                    possible_arc.setAdded(true);
                    return true;
                }
            }
            return false;
        }

        protected void printExecutionStats() {
            System.out.println("Sentence length : " + (
                sent_length - 1));
            System.out.println("# SHIFT: " + n_shift);
            System.out.println("# SWAP: " + n_swap);
            System.out.println("# total operations: " + n_op);

        }

        public int getN_op() {
            return n_op;
        }
    }
```

A.2 NivreParsingSystem.java

```
import java.util.ArrayList;

public class NivreParsingSystem extends
    NonProjectiveParsingSystem {

    public NivreParsingSystem(DependencyTree sent) {
        super(sent);
    }

    /**
     * Oracle function of the parser, tells which
         transition should be chosen given the stack
         situation
     * @param sentence The stack and buffer content
     * @param top Index to the top element of the stack
     * @return Constant coding the action
     */
    public int predictAction(ArrayList<Integer> sentence,
        int top) {

        if (top == 0)
            return SHIFT;
        //Case LEFT ARC
        boolean found = findArc(sentence.get(top), sentence
            .get(top - 1));
        if (found)
            return LEFT_ARC;

        //Case RIGHT ARC
        found = findArc(sentence.get(top - 1), sentence.get
            (top));
        if (found) {
            return RIGHT_ARC;
        }

        //Case SWAP
        if(projective_order.indexOf(sentence.get(top)) <
            projective_order.indexOf(sentence.get(top-1)))
            return SWAP;

        return SHIFT;
    }

    /**
     * Executes the parsing algorithm
     * @return The DependencyTree generated from parsing
```

```
     process
 */
public DependencyTree execute() {

    sent_length = gold.getNodes().entrySet().size();

    ArrayList<Integer> sentence = new ArrayList<>();
    for (int i = 0; i < sent_length; i++)
            sentence.add(i);

    int top_index = 1;
    n_shift = 1; n_op = 1;            //We initialize
       the stack to two elements, as the first choice
       is always SHIFT
    n_swap = 0;

    DependencyTree builded = new DependencyTree(gold.
       getSent_number());

    Node head_node, tail_node;
    while(sentence.size() > 1) {
        switch (predictAction(sentence, top_index)) {
            case LEFT_ARC:

                head_node = builded.addNode(sentence.
                   get(top_index));
                tail_node = builded.addNode(sentence.
                   get(top_index - 1));

                head_node.addLeftSon(tail_node);
                sentence.remove(top_index - 1);
                top_index--;
                break;

            case RIGHT_ARC:

                head_node = builded.addNode(sentence.
                   get(top_index - 1));
                tail_node = builded.addNode(sentence.
                   get(top_index));

                head_node.addRightSon(tail_node);
                sentence.remove(top_index);
                top_index--;
                break;

            case SWAP:

                int swap_id = sentence.get(top_index -
```

```
                        1);
                    sentence.set(top_index - 1, sentence.
                        get(top_index));
                    sentence.set(top_index, swap_id);
                    top_index--;
                    n_swap++;
                    break;

                case SHIFT:

                    top_index++;
                    n_shift++;
                    break;
            }
            n_op++;
        }
        if (n_swap > 0)
            printExecutionStats();

        return builded;
    }

}
```

A.3 TwoStepParsingSystem.java

```
import java.util.ArrayList;

public class TwoStepParsingSystem extends
    NonProjectiveParsingSystem {

    public TwoStepParsingSystem(DependencyTree sent) {
        super(sent);
    }

    /**
     * Oracle function of the parser, tells which
        transition should be chosen given the stack
        situation
     * @param sentence The stack and buffer content
     * @param top Index to the top element of the stack
     * @return Constant coding the action
     */
    public int predictAction(ArrayList<Integer> sentence,
        int top) {

        if (top == 0)
            return SHIFT;
        //Case LEFT ARC
        boolean found = findArc(sentence.get(top), sentence
            .get(top - 1));
        if (found)
            return LEFT_ARC;

        //Case RIGHT ARC
        found = findArc(sentence.get(top - 1), sentence.get
            (top));
        if (found) {
            return RIGHT_ARC;
        }

        //Case SWAP
        if(projective_order.indexOf(sentence.get(top)) <
            projective_order.indexOf(sentence.get(top-1)))
            return SWAP;

        return SHIFT;
    }

    /**
     * Executes the parsing algorithm
     * @return The DependencyTree generated from parsing
        process
```

```
 */
public DependencyTree execute() {

    sent_length = gold.getNodes().entrySet().size();
    DependencyTree builded = new DependencyTree(gold.
       getSent_number());

    ArrayList<Integer> sentence = new ArrayList<>();
    for (int i = 0; i < sent_length; i++)
        sentence.add(i);

    Node head_node,tail_node;
    int top_index = 1;
    n_shift = 1; n_op = 1;

    boolean first_swap = true, first_step = true;
    int swap_point = 0;

    while (sentence.size() > 1) {
        switch (predictAction(sentence, top_index)) {
            case LEFT_ARC:
                head_node = builded.addNode(sentence.
                   get(top_index));
                tail_node = builded.addNode(sentence.
                   get(top_index - 1));

                head_node.addLeftSon(tail_node);
                sentence.remove(top_index - 1);
                top_index--;
                break;

            case RIGHT_ARC:
                head_node = builded.addNode(sentence.
                   get(top_index - 1));
                tail_node = builded.addNode(sentence.
                   get(top_index));

                head_node.addRightSon(tail_node);
                sentence.remove(top_index);
                top_index--;
                break;

            case SWAP:

                if (first_step) {     \\SAVE transition
                    if (first_swap) {
                        swap_point = top_index;
                        first_swap = false;
                    }
```

```
                        top_index++;
                        n_shift++;
                        break;
                    } else {            \\SWAP transition
                        int swap_id = sentence.get(
                            top_index - 1);
                        sentence.set(top_index - 1,
                            sentence.get(top_index));
                        sentence.set(top_index, swap_id);

                        top_index--;
                        n_swap++;
                        break;
                    }

                case SHIFT:
                    top_index++;
                    n_shift++;
                    break;
            } //end switch
            n_op++;
            if (first_step && top_index == sentence.size())
                {
                top_index = swap_point;
                first_step = false;
            }
        }
        if (n_swap > 0)
            printExecutionStats();
        return builded;
    }
}
```