Natural Language Understanding

Lecture 7: Introduction to Dependency Parsing

Adam Lopez
Credits: Mirella Lapata, Frank Keller, and Mark Steedman

26 January 2018

School of Informatics
University of Edinburgh
alopez@inf.ed.ac.uk

Dependency Grammar

- Syntax is often described in terms of constituency
- Dependency syntax is closer to semantics
- Dependency syntax is still (usually) tree-like

Dependency Parsing

- Constituent vs. Dependency Parsing
- Graph-based Dependency Parsing
- Transition-based Dependency Parsing

Reading: Kiperwasser and Goldberg (2016).
Background: Jurafsky and Martin, Ch. 12.7. (14 in the new edition)

Dependency Grammar

Constituents vs. Dependencies

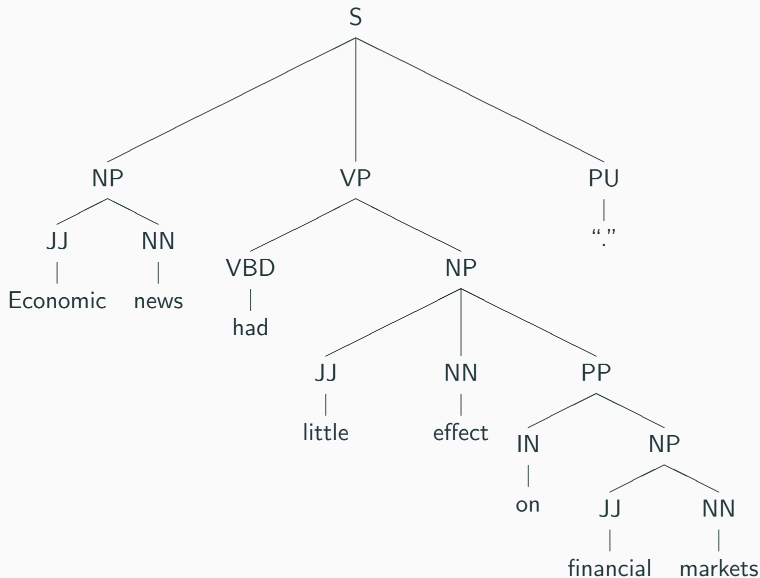
Traditional grammars model *constituent structure:* they capture the configurational patterns of sentences.

For example, verb phrases (VPs) have certain properties in English:

(1)
a. I *like ice cream.* Do you ∅? (*VP ellipsis*)
b. I *like ice cream* and *hate bananas.* (*VP conjunction*)
c. I said I would hit Fred, and *hit Fred* I did. (*VP fronting*)

In other languages (e.g., German), there is little evidence for the existence of a VP constituent.

Constituents form recursive tree structures



Constituents leave out much semantic information

But from a semantic point of view, the important thing about verbs such as *like* is that they license two NPs:

1. an *agent,* found in *subject* position or with *nominative* inflection;
2. a *patient,* found in *object* position or with *accusative* inflection.

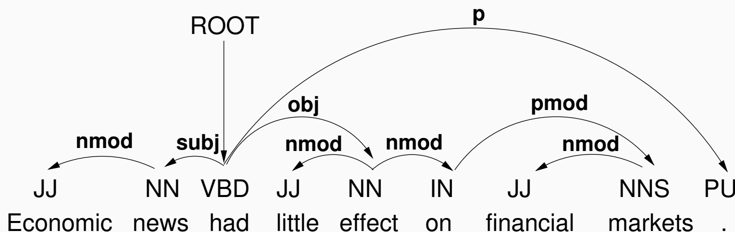
Which arguments are licensed, and which roles they play, depends on the verb (configuration is secondary).

To account for semantic patterns, we focus *dependency.* Dependencies can be identified even in non-configurational languages.

Dependency Structure

A dependency structure consists of *dependency relations,* which are *binary* and *asymmetric*. A relation consists of:

- a *head* (H);
- a *dependent* (D);
- a *label* identifying the relation between H and D.



[From Joakim Nivre, Dependency Grammar and Dependency Parsing.]

Dependency Trees

Formally, the dependency structure of a sentence is a graph with the words of the sentence as its nodes, linked by directed, labeled edges, with the following properties:

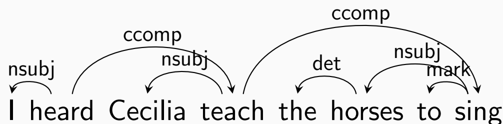
- *connected:* every node is related to at least one other node, and (through transitivity) to ROOT;
- *single headed:* every node (except ROOT) has exactly one incoming edge (from its head);
- *acyclic:* the graph cannot contain cycles of directed edges.

These conditions ensure that the dependency structure is a tree.

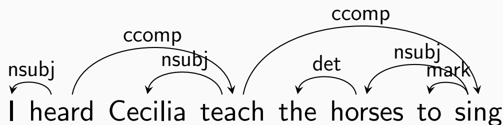
Dependency trees can be projective

We distinguish projective and non-projective dependency trees:

A dependency tree is *projective* wrt. a particular linear order of its nodes if, for all edges $h \to d$ and nodes w, w occurs between h and d in linear order only if w is *dominated* by h.



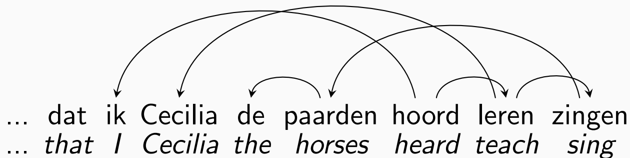
Projective trees can be described with context-free grammars



$$
\begin{aligned}
\text{S} &\rightarrow \textsc{Nsubj}\ \text{heard}\ \textsc{Ccomp} \\
\textsc{Ccomp} &\rightarrow \textsc{Nsubj}\ \text{teach}\ \textsc{Ccomp} \\
\textsc{Nsubj} &\rightarrow \text{I} \\
\textsc{Nsubj} &\rightarrow \text{Cecilia} \\
\textsc{Ccomp} &\rightarrow \textsc{Nsubj}\ \textsc{Mark}\ \text{sing} \\
\textsc{Mark} &\rightarrow \text{to} \\
\textsc{Nsubj} &\rightarrow \textsc{Det}\ \text{horses} \\
\textsc{Det} &\rightarrow \text{the}
\end{aligned}
$$

Dependency Trees can be non-projective

A dependency tree is *non-projective* if *w* can occur between *h* and *d* in linear order without being dominated by *h*.



A non-projective dependency grammar is not context-free. It's still possible to write non-projective grammars in *linear context-free rewriting systems.* (These are very interesting! But well beyond the scope of the course.)

Dependency Parsing

Dependency parsing is different from constituent parsing

In ANLP and FNLP, we've already seen various parsing algorithms for context-free languages (shift-reduce, CKY, active chart).

Why consider *dependency parsing* as a distinct topic?

- context-free parsing algorithms base their decisions on *adjacency;*
- in a dependency structure, a dependent need not be adjacent to its head (even if the structure is projective);
- we need new parsing algorithms to deal with non-adjacency (and with non-projectivity if present).

There are many ways to parse dependencies

We will consider two types of dependency parsers:

1. graph-based dependency parsing, based on *maximum spanning trees* (MST parser, **?**);
2. transition-based dependency parsing, an extension of *shift-reduce parsing* (MALT parser, **?**).

Alternative 3: map dependency trees to phrase structure trees and do standard CFG parsing (for projective trees) or LCFRS variants (for non-projective trees). We will not cover this here.

Note that each of these approach arises from different views of syntactic structure: as a set of constraints (MST), as the actions of an automaton (transition-based), or as the derivations of a grammar (CFG parsing). It is often possible to translate between these views, with some effort.

Graph-based dependency parsing as tagging

Goal: find the highest scoring dependency tree in the space of all possible trees for a sentence.

Let $\mathbf{x} = x_1 \cdots x_n$ be the input sentence, and $\mathbf{y}$ a dependency tree for $\mathbf{x}$. Here, $\mathbf{y}$ is a set of dependency edges, with $(i, j) \in \mathbf{y}$ if there is an edge from x_i to x_j.

Intuition: since each word has exactly one parent, this is like a tagging problem, where the possible tags are *the other words in the sentence* (or a dummy node called *root*). If we *edge factorize* the score of a tree so that it is simply the product of its edge scores, then we can simply select the best incoming edge for each word... subject to the *constraint* that the result must be a tree.

Formalizing graph-based dependency parsing

The score of a dependency edge (i, j) is a function $s(i, j)$. We'll discuss the form of this function a little bit later.

Then the score of dependency tree $\mathbf{y}$ for sentence $\mathbf{x}$ is:

$$s(\mathbf{x}, \mathbf{y}) = \sum_{(i,j) \in \mathbf{y}} s(i, j)$$

Dependency parsing is the task of finding the tree $\mathbf{y}$ with highest score for a given sentence $\mathbf{x}$.

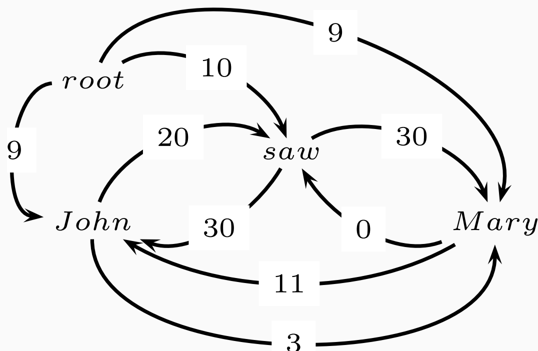
The best dependency parse is the maximum spanning tree

This task can be achieved using the following approach (**?**):

- start with a *totally connected graph* G, i.e., assume a directed edge between every pair of words;
- assume you have a scoring function that assigns a score $s(i, j)$ to every edge (i, j);
- find the *maximum spanning tree* (MST) of G, i.e., the directed tree with the highest overall score that includes all nodes of G;
- this is possible in $O(n^2)$ time using the Chu-Liu-Edmonds algorithm; it finds a MST which is not guaranteed to be projective;
- the highest-scoring parse is the MST of G.

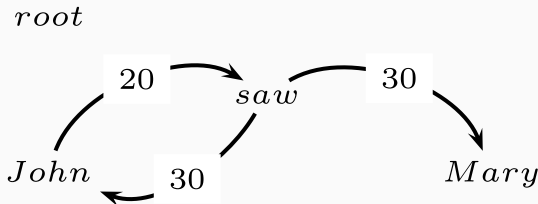
Chu-Liu-Edmonds (CLE) Algorithm

Example: $\mathbf{x} =$ *John saw Mary*, with graph $G_{\mathbf{x}}$. Start with the fully connected graph, with scores:



Chu-Liu-Edmonds (CLE) Algorithm

Each node j in the graph greedily selects the incoming edge with the highest score $s(i, j)$:



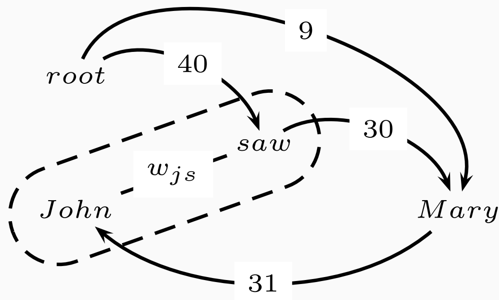
If a tree results, it is the maximum spanning tree. If not, there must be a cycle.

Intuition: We can break the cycle if we replace a single incoming edge to one of the nodes in the cycle. Which one? Decide recursively.

CLE Algorithm: Recursion

Identify the cycle and contract it into a single node and recalculate scores of incoming and outgoing edges.

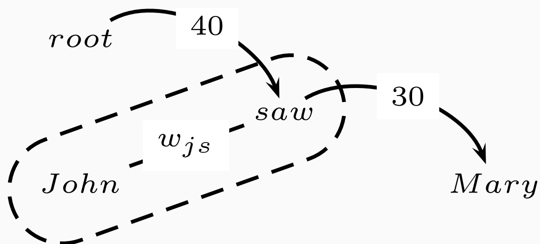
Intuition: edges into the cycle are the weight of the cycle with only the dependency of the target word changed.



Now call CLE recursively on this contracted graph. MST on the contracted graph is equivalent to MST on the original graph.

CLE Algorithm: Recursion

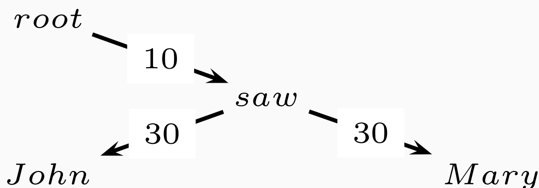
Again, greedily collect incoming edges to all nodes:



This is a tree, hence it must be the MST of the graph.

CLE Algorithm: Reconstruction

Now reconstruct the uncontracted graph: the edge from w_{js} to *Mary* was from *saw*. The edge from ROOT to w_{js} was a tree from ROOT to *saw* to *John*, so we include these edges too:



Where do we get edge scores $s(i, j)$ from?

$$s(\mathbf{x}, \mathbf{y}) = \sum_{(i,j) \in \mathbf{y}} s(i, j)$$

Where do we get edge scores $s(i, j)$ from?

$$s(\mathbf{x}, \mathbf{y}) = \sum_{(i,j) \in \mathbf{y}} s(i, j)$$

For the decade after 2005: linear model trained with clever variants of SVMs, MIRA, etc.

Where do we get edge scores $s(i, j)$ from?

$$s(\mathbf{x}, \mathbf{y}) = \sum_{(i,j) \in \mathbf{y}} s(i, j)$$

For the decade after 2005: linear model trained with clever variants of SVMs, MIRA, etc.

More recently: neural networks, of course.

Scoring edges with a neural network

There are a few different formulations of this. An effective one from Zhang and Lapata (2016):

$$s(i, j) = P_{head}(w_j | w_i, \mathbf{x}) = \frac{\exp(g(\mathbf{a}_j, \mathbf{a}_i))}{\sum_{k=0}^{|\mathbf{x}|} \exp(g(\mathbf{a}_k, \mathbf{a}_i))}$$

We get $\mathbf{a}_i$ by concatenating the hidden states of a forward and backward RNN at position i.

The function $g(\mathbf{a}_j, \mathbf{a}_i)$ computes an *association score* telling us how much word w_i prefers word w_j as its head. A simple option from among many:

$$g(\mathbf{a}_j, \mathbf{a}_i) = \mathbf{v}_a^\top \cdot \tanh(\mathbf{U}_a \cdot \mathbf{a}_j + \mathbf{W}_a \cdot \mathbf{a}_i)$$

Association scores are a useful way to select from a dynamic group of candidates, and underly the idea of *attention* used in MT.

Transition-based Dependency Parsing

An MST parser builds a dependency tree though graph surgery. An alternative is transition-based parsing:

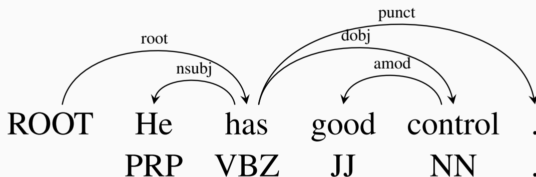
- for a given parse state, the transition system defines a set of actions $\mathcal{T}$ which the parser can take;
- if more than one action is applicable, a classifier (e.g., an SVM) is used to decide which action to take;
- just like in the MST model, this requires a mechanism to compute scores over a set of (possibly dynamic) candidates.

Transition-based Dependency Parsing

The *arc-standard transition* system:

- configuration $c = (s, b, A)$ with stack s, buffer b, set of dependency arcs A;
- initial configuration for sentence $w_1, \ldots, w_n$ is $s = [\texttt{ROOT}], b = [w_1, \ldots, w_n], A = \emptyset$;
- c is terminal if buffer is empty, stack contains only ROOT, and parse tree is given by A_c;
- if s_i is the ith top element on stack, and b_i the ith element on buffer, then we have the following transitions:
- LEFT-ARC(l): adds arc $s_1 \to s_2$ with label l and removes s_2 from stack; precondition: $|s| \geq 2$;
- RIGHT-ARC(l): adds arc $s_2 \to s_1$ with label l and removes s_1 from stack; precondition: $|s| \geq 2$;
- SHIFT: moves b_1 from buffer to stack; recondition: $|b| \geq 1$.

Transition-based Dependency Parsing



Transition	Stack	Buffer	A
	[ROOT]	[He has good control .]	$\emptyset$
`SHIFT`	[ROOT He]	[has good control .]	
`SHIFT`	[ROOT He has]	[good control .]	
`LEFT-ARC(nsubj)`	[ROOT has]	[good control .]	$A\cup$ nsubj(has,He)
`SHIFT`	[ROOT has good]	[control .]	
`SHIFT`	[ROOT has good control]	[.]	
`LEFT-ARC(amod)`	[ROOT has control]	[.]	$A\cup$amod(control,good)
`RIGHT-ARC(dobj)`	[ROOT has]	[.]	$A\cup$ dobj(has,control)
...	...	...	...
`RIGHT-ARC(root)`	[ROOT]	[]	$A\cup$ root(ROOT,has)

Summary

Comparing MST and transition-based parsers:

- the MST parser selects the globally optimal tree, given a set of edges with scores;
- it can naturally handle projective and non-projective trees;
- a transition-based parser makes a sequence of local decisions about the best parse action;
- it can be extended to projective dependency trees by changing the transition set;
- accuracies are similar, but transition-based is faster;
- both require dynamic classifiers, and these can be implemented using neural networks, conditioned on bidirectional RNN encodings of the sentence.