Dependency Parsing 2

CMSC 723 / LING 723 / INST 725

Marine Carpuat

Fig credits: Joakim Nivre, Dan Jurafsky & James Martin

Dependency Parsing

- Formalizing dependency trees
- Transition-based dependency parsing
  - Shift-reduce parsing
  - Transition system
  - Oracle
  - Learning/predicting parsing actions

Data-driven dependency parsing

Goal: learn a good predictor of dependency graphs

Input: sentence

Output: dependency graph/tree G = (V,A)

Can be framed as a structured prediction task

- very large output space
- with interdependent labels

2 dominant approaches: transition-based parsing and graph-based parsing

Transition-based dependency parsing

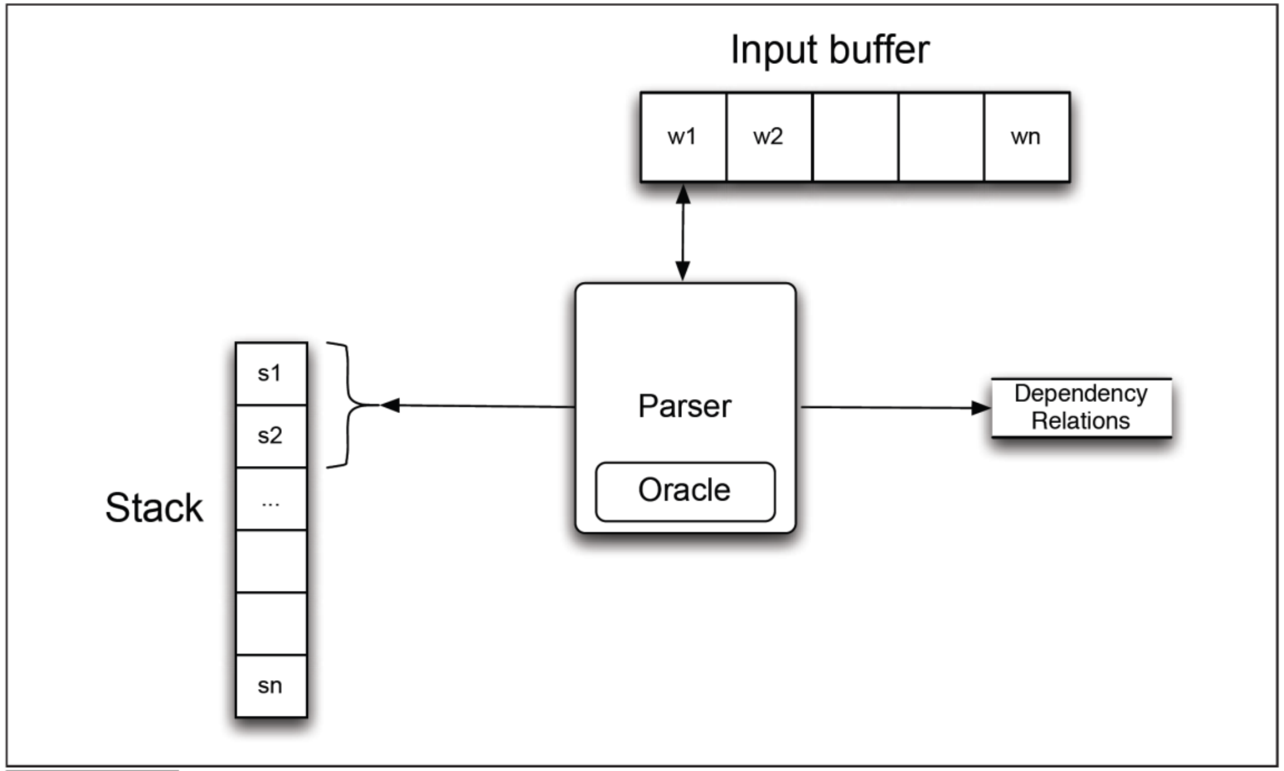


Figure 14.5 Basic transition-based parser. The parser examines the top two elements of the stack and selects an action based on consulting an oracle that examines the current configuration.

- Builds on shift-reduce parsing
  [Aho & Ullman, 1927]
- **Configuration**
  - **Stack**
  - **Input buffer** of words
  - Set of dependency relations
- Goal of parsing
  - find a final configuration where
  - all words accounted for
  - Relations form dependency tree

Transition operators

- Transitions: produce a new configuration given current configuration

- Parsing is the task of
  - Finding a sequence of transitions
  - That leads from start state to desired goal state

- Start state
  - Stack initialized with ROOT node
  - Input buffer initialized with words in sentence
  - Dependency relation set = empty

- End state
  - Stack and word lists are empty
  - Set of dependency relations = final parse

Arc Standard Transition System

- Defines 3 transition operators [Covington, 2001; Nivre 2003]
- LEFT-ARC:
    - create head-dependent rel. between word at top of stack and 2nd word (under top)
    - remove 2nd word from stack
- RIGHT-ARC:
    - Create head-dependent rel. between word on 2nd word on stack and word on top
    - Remove word at top of stack
- SHIFT
    - Remove word at head of input buffer
    - Push it on the stack

Arc standard transition systems

- Preconditions
    - ROOT cannot have incoming arcs
    - LEFT-ARC cannot be applied when ROOT is the 2^{nd} element in stack
    - LEFT-ARC and RIGHT-ARC require 2 elements in stack to be applied

Transition-based Dependency Parser

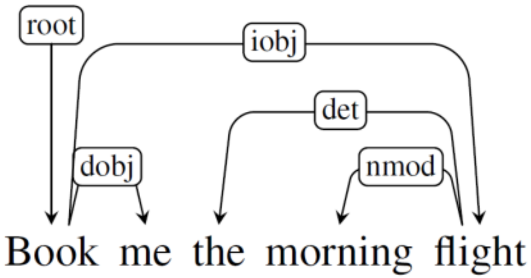
```
function DEPENDENCYPARSE(words) returns dependency tree

  state ← {[root], [words], [] }  ; initial configuration
  while state not final
    t ← ORACLE(state)          ; choose a transition operator to apply
    state ← APPLY(t, state)  ; apply it, creating a new state
  return state
```

Figure 14.6 A generic transition-based dependency parser

- Assume an oracle
- Parsing complexity
  - Linear in sentence length!
- Greedy algorithm
  - Unlike Viterbi for POS tagging

Transition-Based Parsing Illustrated



Step	Stack	Word List	Action	Relation Added
0	[root]	[book, me, the, morning, flight]	SHIFT	
1	[root, book]	[me, the, morning, flight]	SHIFT	
2	[root, book, me]	[the, morning, flight]	RIGHTARC	(book → me)
3	[root, book]	[the, morning, flight]	SHIFT	
4	[root, book, the]	[morning, flight]	SHIFT	
5	[root, book, the, morning]	[flight]	SHIFT	
6	[root, book, the, morning, flight]	[]	LEFTARC	(morning ← flight)
7	[root, book, the, flight]	[]	LEFTARC	(the ← flight)
8	[root, book, flight]	[]	RIGHTARC	(book → flight)
9	[root, book]	[]	RIGHTARC	(root → book)
10	[root]	[]	Done	

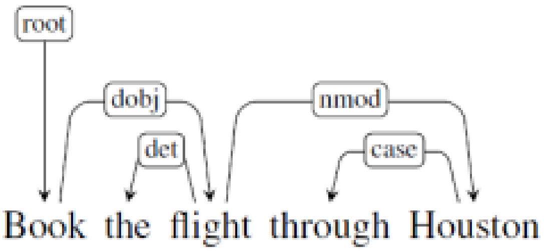
Figure 14.7 Trace of a transition-based parse.

Where to we get an oracle?

- Multiclass classification problem
  - Input: current parsing state (e.g., current and previous configurations)
  - Output: one transition among all possible transitions
  - Q: size of output space?
- Supervised classifiers can be used
  - E.g., perceptron
- Open questions
  - What are good features for this task?
  - Where do we get training examples?

Generating Training Examples

- What we have in a treebank



- What we need to train an oracle
  - Pairs of configurations and predicted parsing action

Step	Stack	Word List	Predicted Action
0	[root]	[book, the, flight, through, houston]	SHIFT
1	[root, book]	[the, flight, through, houston]	SHIFT
2	[root, book, the]	[flight, through, houston]	SHIFT
3	[root, book, the, flight]	[through, houston]	LEFTARC
4	[root, book, flight]	[through, houston]	SHIFT
5	[root, book, flight, through]	[houston]	SHIFT
6	[root, book, flight, through, houston]	[]	LEFTARC
7	[root, book, flight, houston]	[]	RIGHTARC
8	[root, book, flight]	[]	RIGHTARC
9	[root, book]	[]	RIGHTARC
10	[root]	[]	Done

Figure 14.8 Generating training items consisting of configuration/predicted action pairs by simulating a parse with a given reference parse.

Generating training examples

- Approach: simulate parsing to generate reference tree
- Given
  - A current config with stack S, dependency relations Rc
  - A reference parse (V,Rp)
- Do

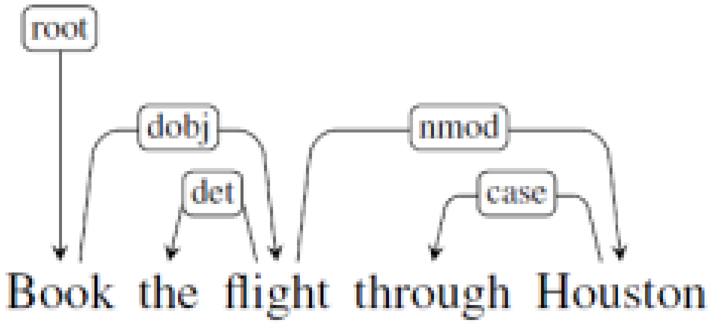
LEFTARC(r): **if** $(S_1\ r\ S_2) \in R_p$

RIGHTARC(r): **if** $(S_2\ r\ S_1) \in R_p$ **and** $\forall r', w\ s.t. (S_1\ r'\ w) \in R_p$ **then** $(S_1\ r'\ w) \in R_c$

SHIFT: **otherwise**

Let’s try it out

LEFTARC(r): **if** $(S_1\ r\ S_2) \in R_p$
RIGHTARC(r): **if** $(S_2\ r\ S_1) \in R_p$ **and** $\forall r', w\ s.t.(S_1\ r'\ w) \in R_p$ **then** $(S_1\ r'\ w) \in R_c$
SHIFT: **otherwise**



Features

- Configuration consist of stack, buffer, current set of relations

- Typical features
  - Features focus on top level of stack
  - Use word forms, POS, and their location in stack and buffer

Features example

- Given configuration

Stack	Word buffer	Relations
[root, canceled, flights]	[to Houston]	(canceled $\rightarrow$ United)
		(flights $\rightarrow$ morning)
		(flights $\rightarrow$ the)

- Example of useful features

$$\langle s_1.w = \mathit{flights}, op = \mathit{shift} \rangle$$
$$\langle s_2.w = \mathit{canceled}, op = \mathit{shift} \rangle$$
$$\langle s_1.t = \mathit{NNS}, op = \mathit{shift} \rangle$$
$$\langle s_2.t = \mathit{VBD}, op = \mathit{shift} \rangle$$
$$\langle b_1.w = \mathit{to}, op = \mathit{shift} \rangle$$
$$\langle b_1.t = \mathit{TO}, op = \mathit{shift} \rangle$$
$$\langle s_1.wt = \mathit{flightsNNS}, op = \mathit{shift} \rangle$$

$$\langle s_1t.s_2t = \mathit{NNSVBD}, op = \mathit{shift} \rangle$$

Features example

Source	Feature templates		
One word	$s_1.w$	$s_1.t$	$s_1.wt$
	$s_2.w$	$s_2.t$	$s_2.wt$
	$b_1.w$	$b_1.w$	$b_0.wt$
Two word	$s_1.w \circ s_2.w$	$s_1.t \circ s_2.t$	$s_1.t \circ b_1.w$
	$s_1.t \circ s_2.wt$	$s_1.w \circ s_2.w \circ s_2.t$	$s_1.w \circ s_1.t \circ s_2.t$
	$s_1.w \circ s_1.t \circ s_2.t$	$s_1.w \circ s_1.t$	

Figure 14.9 Standard feature templates for training transition-based dependency parsers. In the template specifications s_n refers to a location on the stack, b_n refers to a location in the word buffer, w refers to the wordform of the input, and t refers to the part of speech of the input.

Research highlight:
Dependency parsing with stack-LSTMs

- From Dyer et al. 2015: http://www.aclweb.org/anthology/P15-1033

- Idea
  - Instead of hand-crafted feature
  - Predict next transition using recurrent neural networks to learn representation of stack, buffer, sequence of transitions

Research highlight:
Dependency parsing with stack-LSTMs

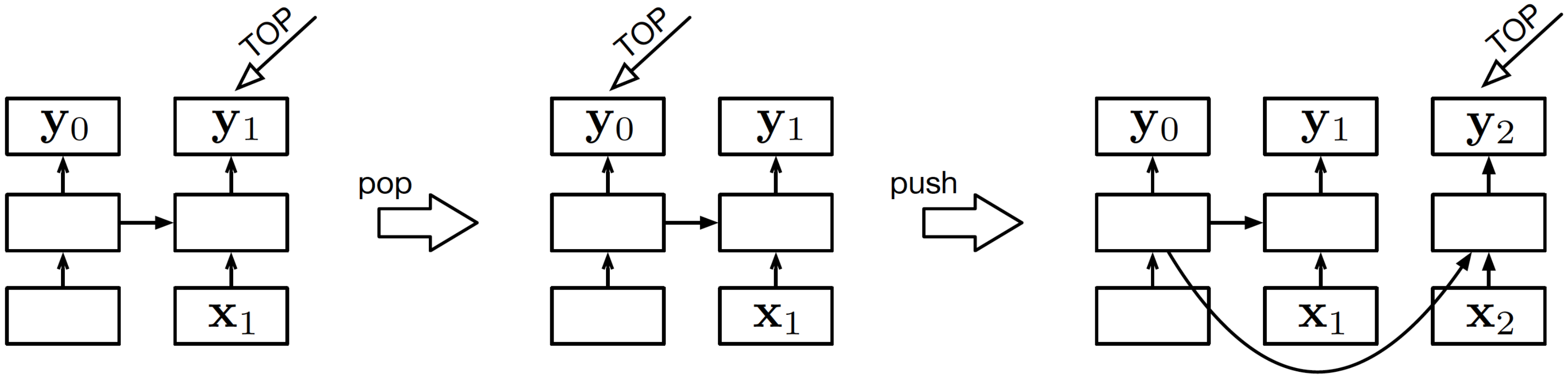


Figure 1: A stack LSTM extends a conventional left-to-right LSTM with the addition of a stack pointer (notated as TOP in the figure). This figure shows three configurations: a stack with a single element (left), the result of a pop operation to this (middle), and then the result of applying a push operation (right). The boxes in the lowest rows represent stack contents, which are the inputs to the LSTM, the upper rows are the outputs of the LSTM (in this paper, only the output pointed to by TOP is ever accessed), and the middle rows are the memory cells (the $\mathbf{c}_t$'s and $\mathbf{h}_t$'s) and gates. Arrows represent function applications (usually affine transformations followed by a nonlinearity), refer to §2.1 for specifics.

Research highlight: Dependency parsing with stack-LSTMs

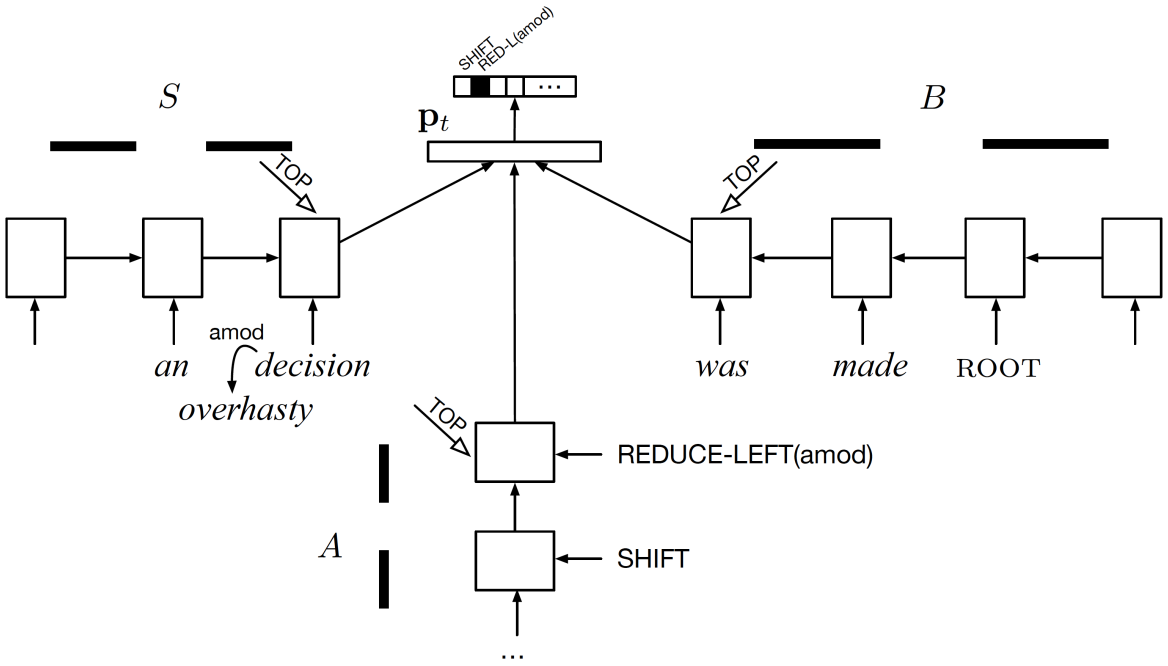


Figure 2: Parser state computation encountered while parsing the sentence "*an overhasty decision was made*." Here S designates the stack of partially constructed dependency subtrees and its LSTM encoding; B is the buffer of words remaining to be processed and its LSTM encoding; and A is the stack representing the history of actions taken by the parser. These are linearly transformed, passed through a ReLU nonlinearity to produce the parser state embedding $\mathbf{p}_t$. An affine transformation of this embedding is passed to a softmax layer to give a distribution over parsing decisions that can be taken.

Alternate Transition Systems

Note: A different way of writing arc-standard transition system

- Transitions:
  - Left-Arc$_k$:

    $$(\sigma|i, j|\beta, A) \Rightarrow (\sigma, j|\beta, A \cup \{(j, i, k)\})$$
  - Right-Arc$_k$:

    $$(\sigma|i, j|\beta, A) \Rightarrow (\sigma, i|\beta, A \cup \{(i, j, k)\})$$
  - Shift:

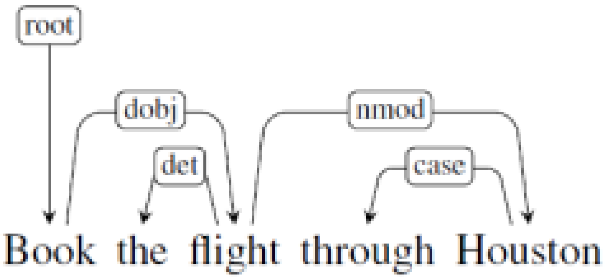
    $$(\sigma, i|\beta, A) \Rightarrow (\sigma|i, \beta, A)$$
- Preconditions:
  - Left-Arc$_k$:

    $$\neg[i = 0]$$
    $$\neg\exists i' \exists k' [(i', i, k') \in A]$$
  - Right-Arc$_k$:

    $$\neg\exists i' \exists k' [(i', j, k') \in A]$$

A weakness of arc-standard parsing

Right dependents cannot be attached to their head until all their dependents have been attached



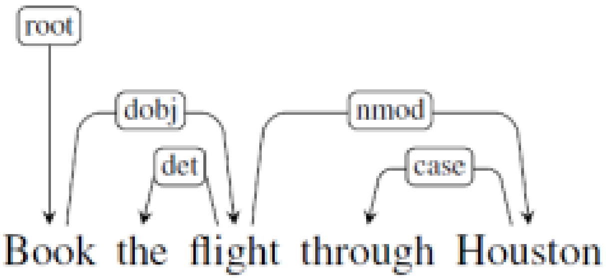
Step	Stack	Word List	Predicted Action
0	[root]	[book, the, flight, through, houston]	SHIFT
1	[root, book]	[the, flight, through, houston]	SHIFT
2	[root, book, the]	[flight, through, houston]	SHIFT
3	[root, book, the, flight]	[through, houston]	LEFTARC
4	[root, book, flight]	[through, houston]	SHIFT
5	[root, book, flight, through]	[houston]	SHIFT
6	[root, book, flight, through, houston]	[]	LEFTARC
7	[root, book, flight, houston]	[]	RIGHTARC
8	[root, book, flight]	[]	RIGHTARC
9	[root, book]	[]	RIGHTARC
10	[root]	[]	Done

Figure 14.8 Generating training items consisting of configuration/predicted action pairs by simulating a parse with a given reference parse.

Arc Eager Parsing

- LEFT-ARC:
  - Create head-dependent rel. between word at front of buffer and word at top of stack
  - pop the stack
- RIGHT-ARC:
  - Create head-dependent rel. between word on top of stack and word at front of buffer
  - Shift buffer head to stack
- SHIFT
  - Remove word at head of input buffer
  - Push it on the stack
- REDUCE
  - Pop the stack

Arc Eager Parsing Example



Step	Stack	Word List	Action	Relation Added
0	[root]	[book, the, flight, through, houston]	RIGHTARC	(root → book)
1	[root, book]	[the, flight, through, houston]	SHIFT	
2	[root, book, the]	[flight, through, houston]	LEFTARC	(the ← flight)
3	[root, book]	[flight, through, houston]	RIGHTARC	(book → flight)
4	[root, book, flight]	[through, houston]	SHIFT	
5	[root, book, flight, through]	[houston]	LEFTARC	(through ← houston)
6	[root, book, flight]	[houston]	RIGHTARC	(flight → houston)
7	[root, book, flight, houston]	[]	REDUCE	
8	[root, book, flight]	[]	REDUCE	
9	[root, book]	[]	REDUCE	
10	[root]	[]	Done	

Figure 14.10 A processing trace of *Book the flight through Houston* using the arc-eager transition operators.

Trees & Forests

- A dependency forest (here) is a dependency graph satisfying
  - Root
  - Single-Head
  - Acyclicity
  - but **not** Connectedness

Properties of this transition-based parsing algorithm

- Correctness
  - For every complete transition sequence, the resulting graph is a projective dependency forest (soundness)
  - For every projective dependency forest G, there is a transition sequence that generates G (completeness)

- Trick: forest can be turned into tree by adding links to $ROOT_0$

Dealing with non-projectivity

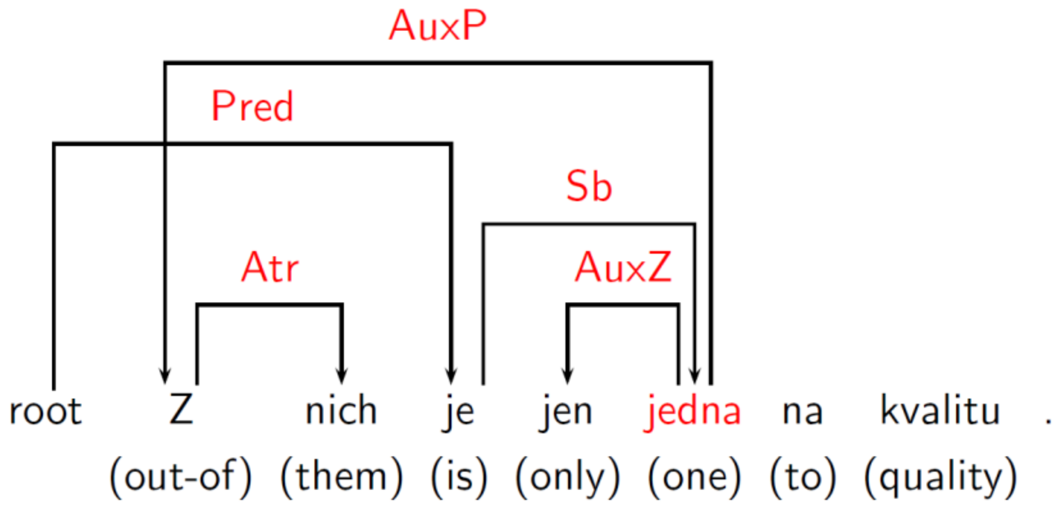
Projectivity

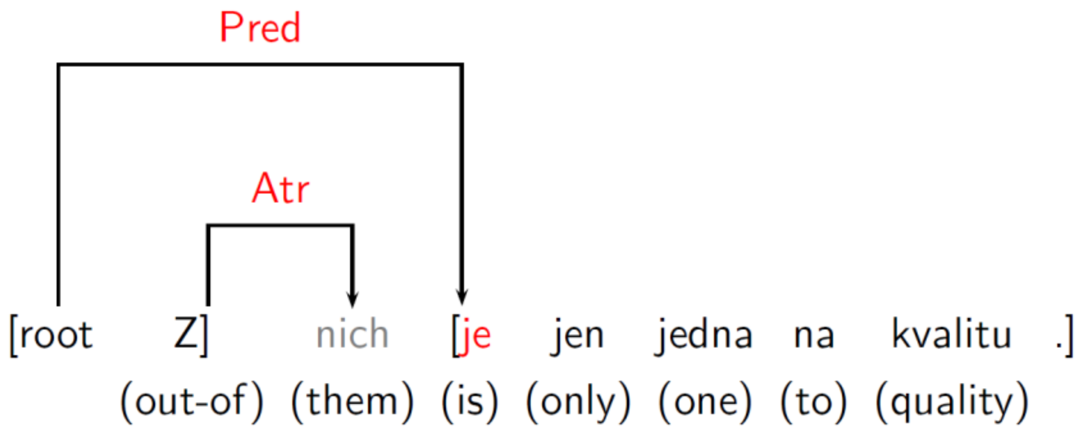
- **Arc** from head to dependent is **projective**
  - If there is a path from head to every word between head and dependent

- **Dependency tree** is **projective**
  - If all arcs are projective
  - Or equivalently, if it can be drawn with no crossing edges

- Projective trees make computation easier
- But most theoretical frameworks do not assume projectivity
  - Need to capture long-distance dependencies, free word order

Arc-standard parsing can't produce non-projective trees




Pred
Atr
[root Z] nich [je jen jedna na kvalitu .]
(out-of) (them) (is) (only) (one) (to) (quality)

How frequent are non-projective structures?

- Statistics from CoNLL shared task
  - NPD = non projective dependencies
  - NPS = non projective sentences

Language	**%NPD**	**%NPS**
Dutch	5.4	36.4
German	2.3	27.8
Czech	1.9	23.2
Slovene	1.9	22.2
Portuguese	1.3	18.9
Danish	1.0	15.6

How to deal with non-projectivity? (1) change the transition system

Transition		Preconditio
NP-Left$_r$	$(\sigma \| w_i \| w_k, w_j \| \beta, A) \Rightarrow (\sigma \| w_k, w_j \| \beta, A \cup \{(w_j, r, w_i)\})$	$i \neq 0$
NP-Right$_r$	$(\sigma \| w_i \| w_k, w_j \| \beta, A) \Rightarrow (\sigma \| w_i, w_k \| \beta, A \cup \{(w_i, r, w_j)\})$	

- Add new transitions
  - That apply to 2nd word of the stack
  - Top word of stack is treated as context

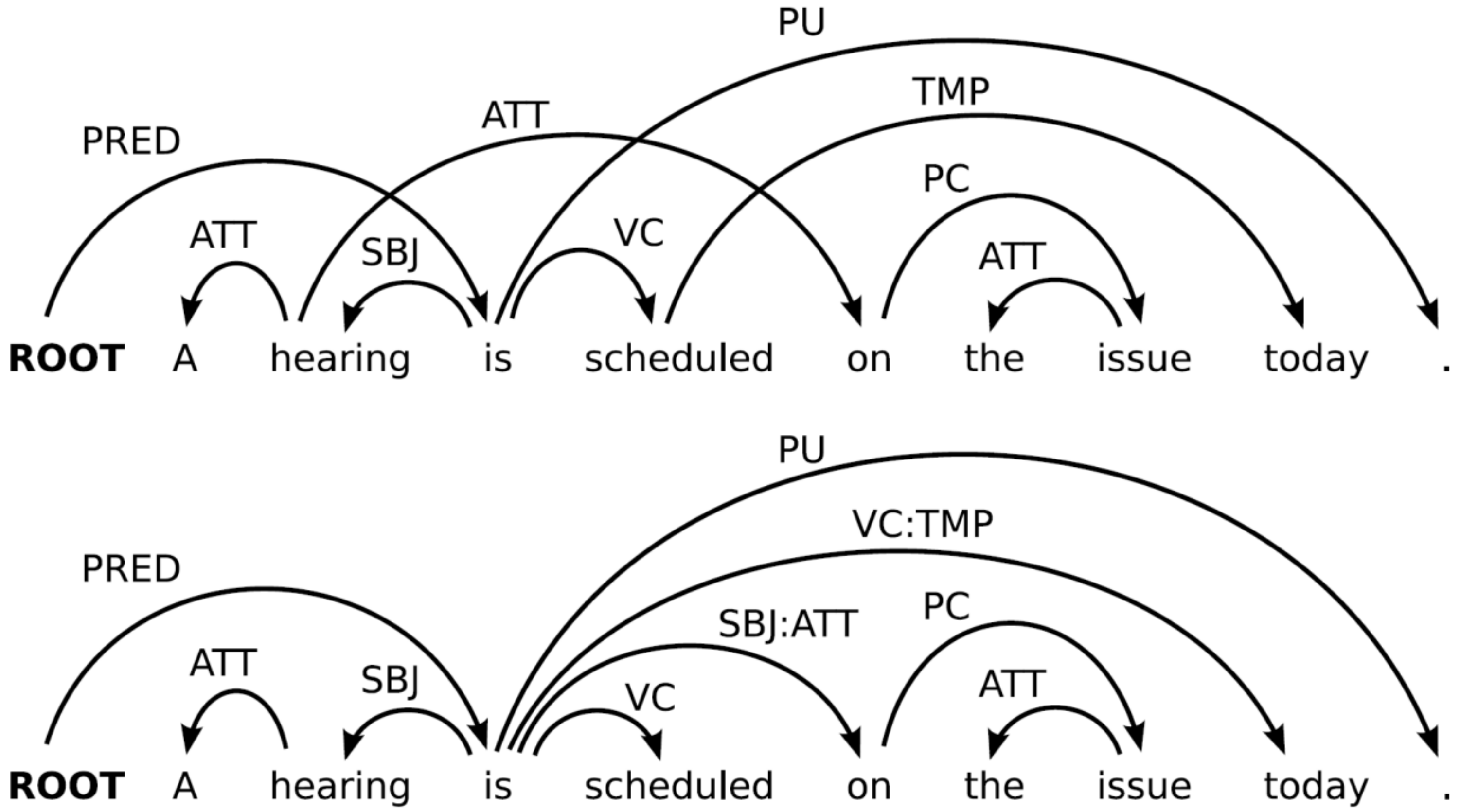
[Attardi 2006]

How to deal with non-projectivity? (2) pseudo-projective parsing

Solution:

- “projectivize” a non-projective tree by creating new projective arcs
- That can be transformed back into non-projective arcs in a post-processing step

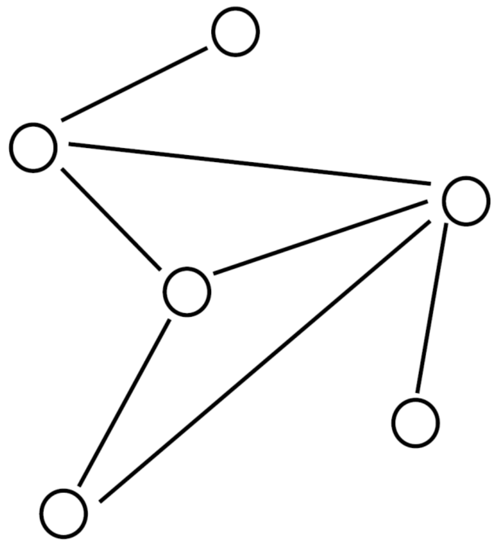
How to deal with non-projectivity?
(2) pseudo-projective parsing

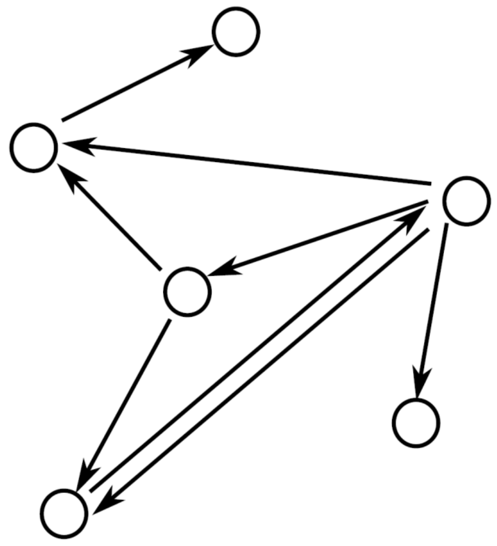


Graph-based parsing

Graph concepts refresher

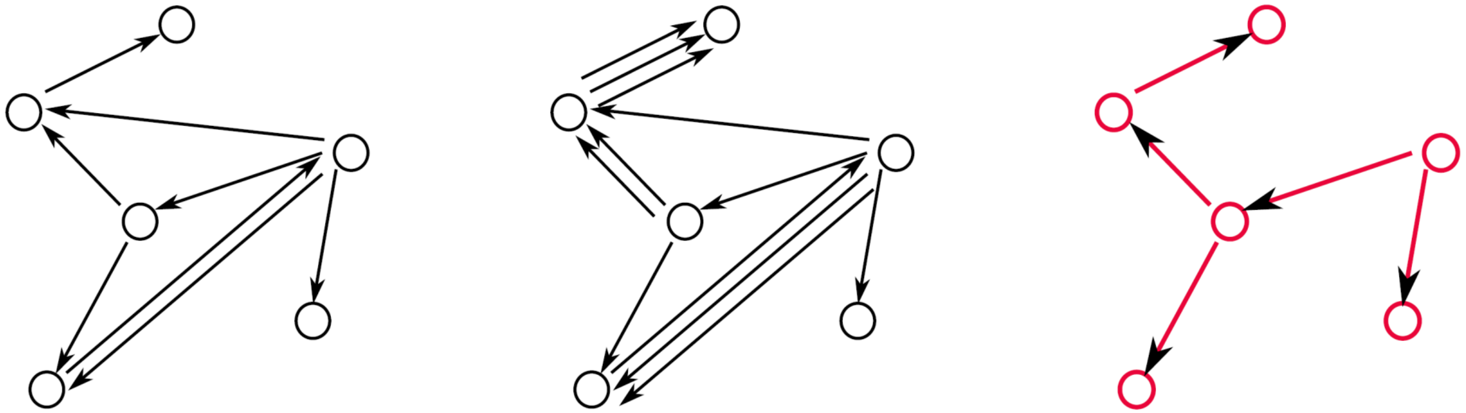
- A graph $G = (V, A)$ is a set of verteces V and arcs $(i, j) \in A$, where $i, j \in V$
- Undirected graphs: $(i, j) \in A \Leftrightarrow (j, i) \in A$
- **Directed graphs (digraphs)**: $(i, j) \in A \not\Rightarrow (j, i) \in A$





Directed Spanning Trees

- A directed spanning tree of a (multi-)digraph $G = (V, A)$, is a subgraph $G' = (V', A')$ such that:
  - $V' = V$
  - $A' \subseteq A$, and $|A'| = |V'| - 1$
  - G' is a tree (acyclic)
- A spanning tree of the following (multi-)digraphs



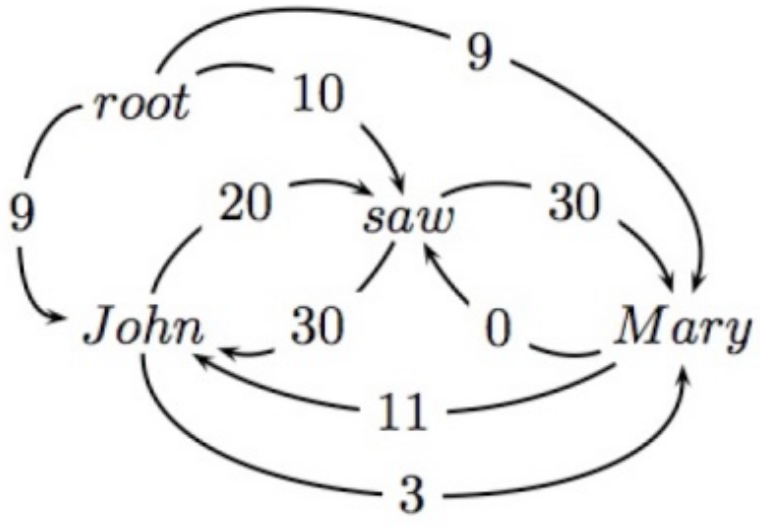
Maximum Spanning Tree

- Assume we have an **arc factored** model

  i.e. weight of graph can be factored as sum or product of weights of its arcs

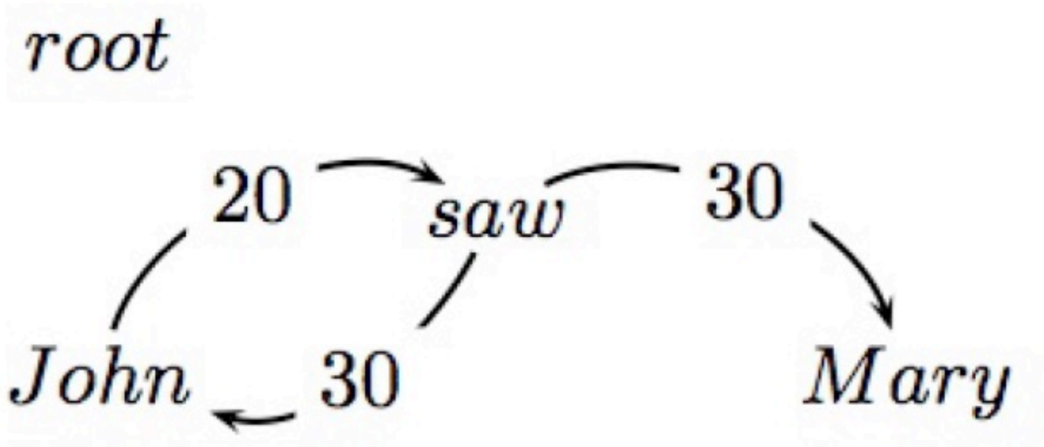
- Chu-Liu-Edmonds algorithm can find the maximum spanning tree for us!
  - Greedy recursive algorithm
  - Naïve implementation: O(n^3)

Chu-Liu-Edmonds illustrated



Chu-Liu-Edmonds illustrated

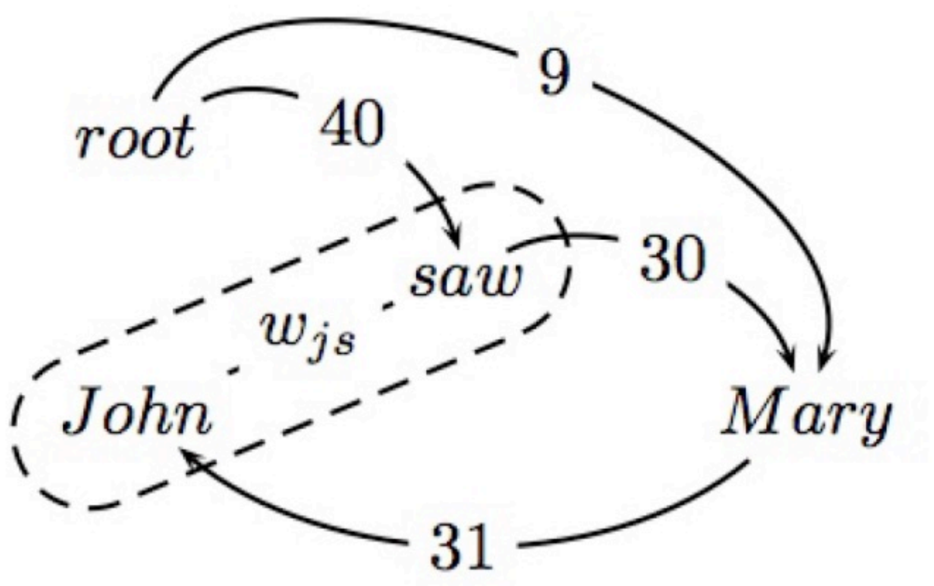
- Find highest scoring incoming arc for each vertex



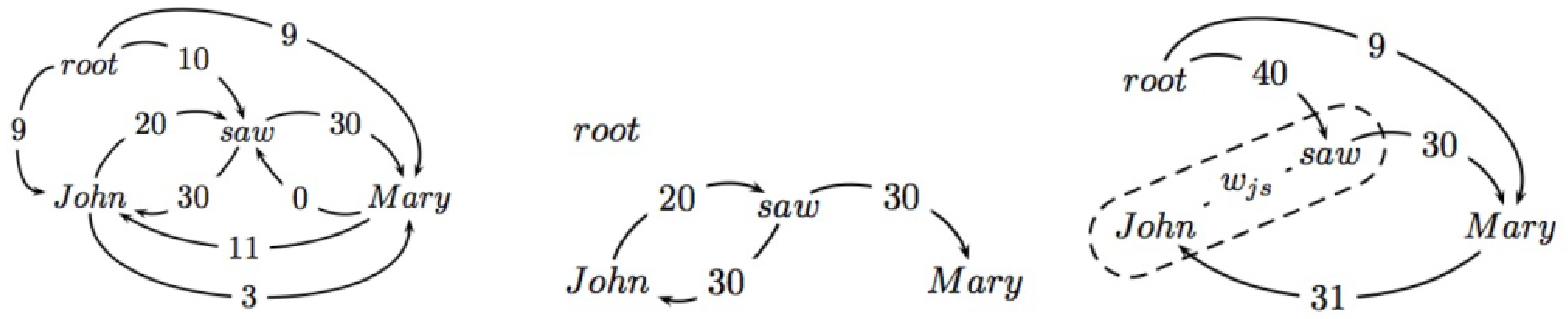
- If this is a tree, then we have found MST!!

Chu-Liu-Edmonds illustrated

- If not a tree, identify cycle and contract
- Recalculate arc weights into and out-of cycle

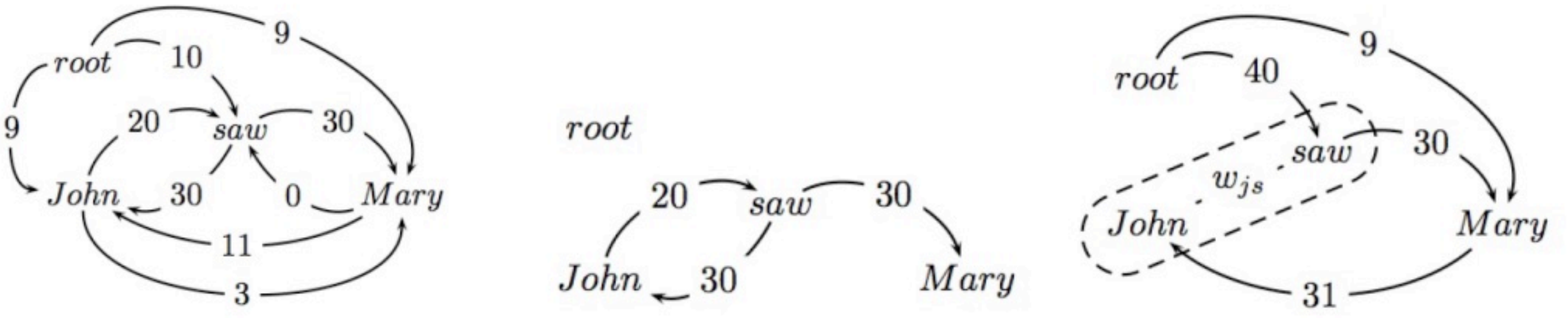


Chu-Liu-Edmonds illustrated



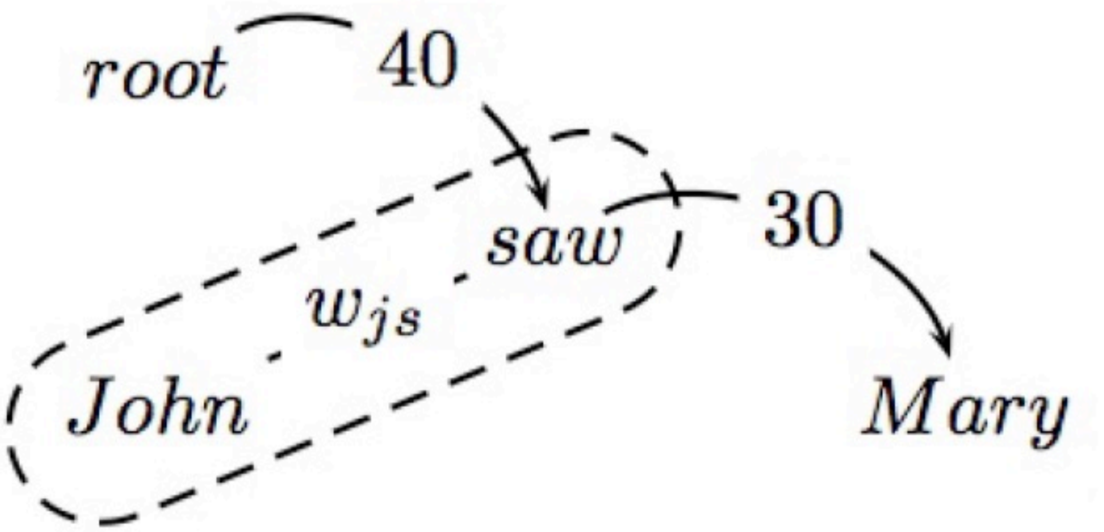
- Outgoing arc weights
  - Equal to the max of outgoing arc over all vertexes in cycle
  - e.g., John $\rightarrow$ Mary is 3 and saw $\rightarrow$ Mary is 30

Chu-Liu-Edmonds illustrated



- Incoming arc weights
  - Equal to the weight of best spanning tree that includes head of incoming arc, and all nodes in cycle
  - root → saw → John is 40 (**)
  - root → John → saw is 29

- This is a tree and the MST for the contracted graph!!



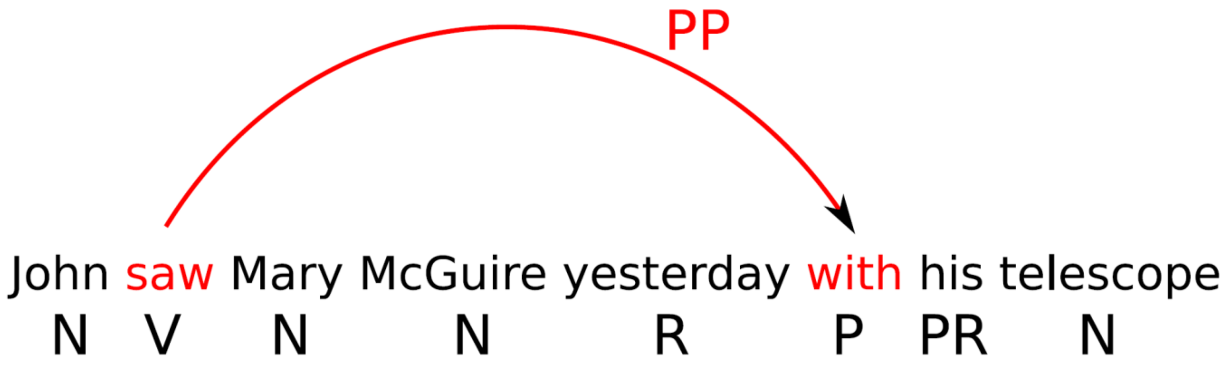
- Go back up recursive call and reconstruct final graph

Arc weights as linear classifiers

$$w_{ij}^{k} = e^{\mathbf{w} \cdot \mathbf{f}(i,j,k)}$$

- Arc weights are a linear combination of features of the arc, $\mathbf{f}$, and a corresponding weight vector $\mathbf{w}$
- Raised to an exponent (simplifies some math ...)
- What arc features?
- [McDonald et al. 2005] discuss a number of binary features

Example of classifier features



- Features from [McDonald et al. 2005]:
  - Identities of the words w_i and w_j and the label l_k

head=saw & dependent=with

How to score a graph G using features?

Arc-factored model assumption

By definition of arc weights as linear classifiers

$$
\begin{aligned}
G &= \operatorname*{arg\,max}_{G \in T(G_x)} \prod_{(i,j,k) \in G} w_{ij}^{k} = \operatorname*{arg\,max}_{G \in T(G_x)} \prod_{(i,j,k) \in G} e^{\mathbf{w} \cdot \mathbf{f}(i,j,k)} \\
&= \operatorname*{arg\,max}_{G \in T(G_x)} \log \prod_{(i,j,k) \in G} e^{\mathbf{w} \cdot \mathbf{f}(i,j,k)} \\
&= \operatorname*{arg\,max}_{G \in T(G_x)} \sum_{(i,j,k) \in G} \mathbf{w} \cdot \mathbf{f}(i,j,k) \\
&= \operatorname*{arg\,max}_{G \in T(G_x)} \mathbf{w} \cdot \sum_{(i,j,k) \in G} \mathbf{f}(i,j,k) = \operatorname*{arg\,max}_{G \in T(G_x)} \mathbf{w} \cdot \mathbf{f}(G)
\end{aligned}
$$

How can we learn the classifier from data?

e.g., The Perceptron

Training data: $\mathcal{T} = \{(x_t, G_t)\}_{t=1}^{|\mathcal{T}|}$

1. $\mathbf{w}^{(0)} = 0;\ i = 0$
2. for $n : 1..N$
3. for $t : 1..T$
4. Let $G' = \arg\max_{G'} \mathbf{w}^{(i)} \cdot \mathbf{f}(G')$
5. if $G' \neq G_t$
6. $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} + \mathbf{f}(G_t) - \mathbf{f}(G')$
7. $i = i + 1$
8. return $\mathbf{w}^i$

Dependency Parsing: what you should know

- Formalizing dependency trees
- Transition-based dependency parsing
  - Shift-reduce parsing
  - Transition system: arc standard, arc eager
  - Oracle
  - Learning/predicting parsing actions
- Graph-based dependency parsing
- A flexible framework that allows many extensions
  - RNNs vs feature engineering, non-projectivity

Extension: dynamic oracle

Problem with standard classifier-based oracle:

- It is “static”
  - ie tied to optimal config sequence that produces gold tree
- What if there are multiple sequences for a single gold tree?
- How can we recover if the parser deviates from gold sequence?

One solution: “dynamic oracle” [Goldberg & Nivre 2012]

See also Locally Optimal Learning to Search [Chang et al. ICML 2015]

Extension: dynamic oracle

Algorithm 3 Online training with a dynamic oracle

$\mathbf{w} \leftarrow 0$
for $I = 1 \rightarrow$ ITERATIONS **do**
  for sentence x with gold tree G_{gold} in corpus **do**
    $c \leftarrow c_s(x)$
    while c is not terminal **do**
      $t_p \leftarrow \arg\max_t \mathbf{w} \cdot \phi(c, t)$
      $\boxed{\text{ZERO\_COST} \leftarrow \{t \mid o(t; c, G_{\text{gold}}) = \textbf{true}\}}$
      $t_o \leftarrow \arg\max_{t \in \text{ZERO\_COST}} \mathbf{w} \cdot \phi(c, t)$
      if $t_p \notin$ ZERO_COST **then**
        $\mathbf{w} \leftarrow \mathbf{w} + \phi(c, t_o) - \phi(c, t_p)$
      $\boxed{t_n \leftarrow \text{CHOOSE\_NEXT}(I, t_p, \text{ZERO\_COST})}$
      $c \leftarrow t_n(c)$
return w

See [Goldberg & Nivre 2012] for details