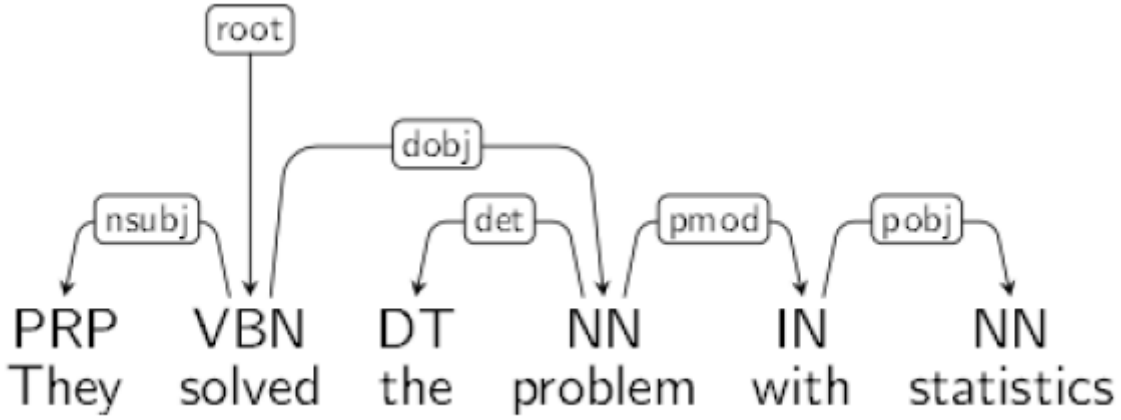
Statistical Processing of Natural Language: Dependency Parsing

Alicia Ageno
ageno@cs.upc.edu
Universitat Politècnica de Catalunya

Dependency Parsing

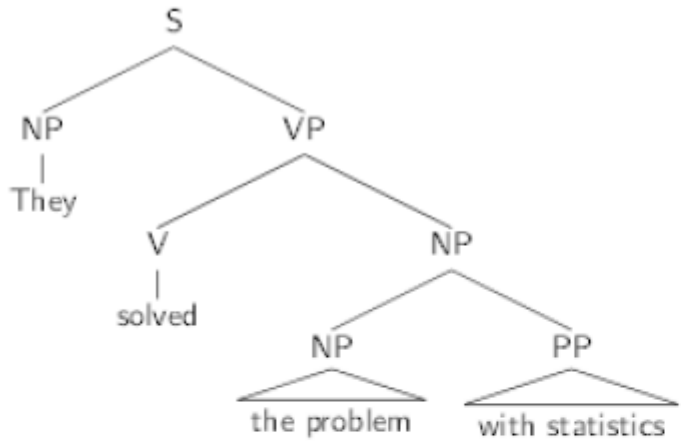
A dependency tree



Dependency Parsing

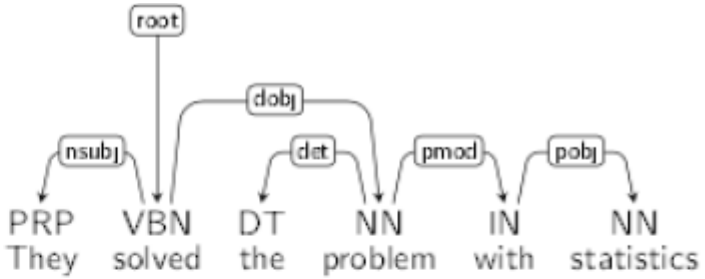
Theories of Syntactic Structure

Constituent trees



- Main element: constituents (or phrases, or bracketings)
- Constituents abstract linguistic units
- Results in nested trees

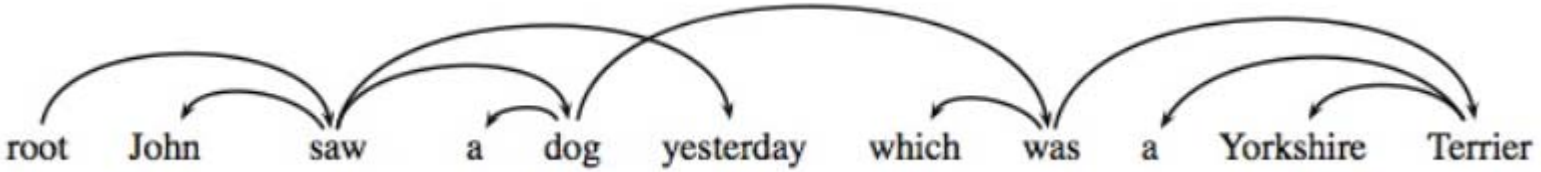
Dependency trees

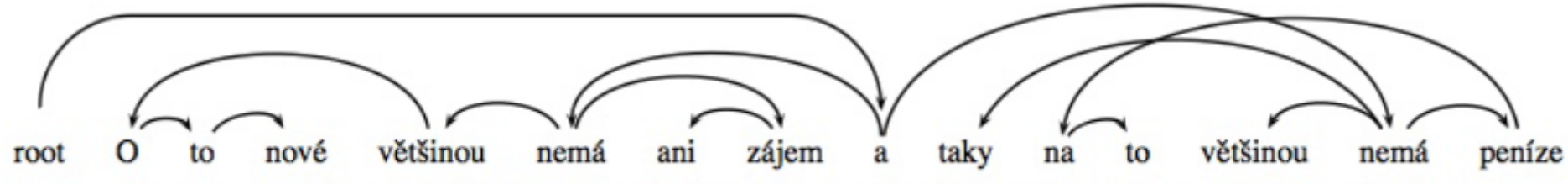


- Main element: dependency
- Focus on relations between words
- Handles **free word order** nicely. Exercise: parse **He saw the dog yesterday that barked all night**

Dependency Parsing

Non-projective dependency trees



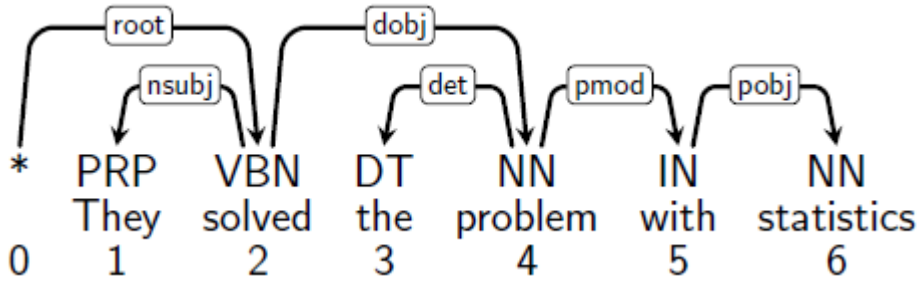


He is mostly not even interested in the new things and in most cases, he has no money for it either.

(from Czech Prague Dependency Treebank)

Dependency Parsing

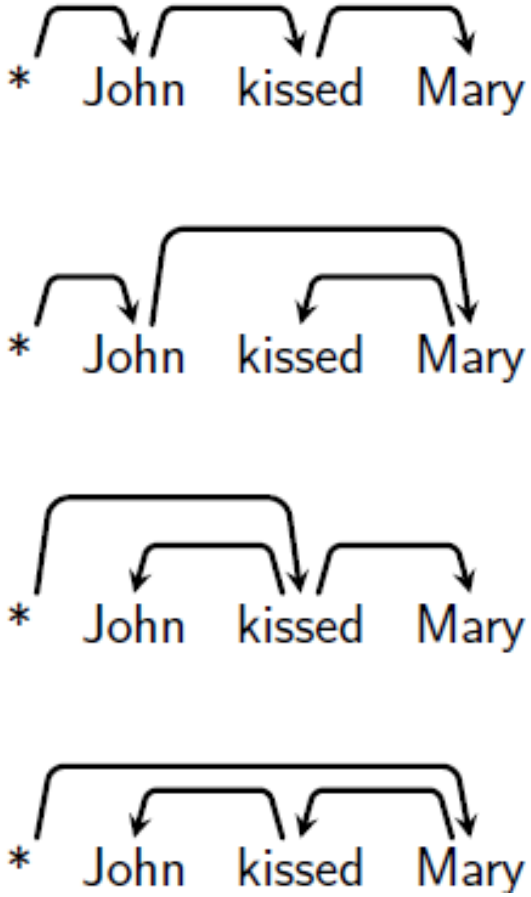
Dependency trees

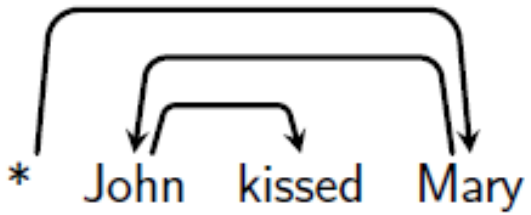


- * is a special *root* symbol
- Each dependency is a tuple *(h,m,l)* where
  - *h* is the index of the head word (root is 0)
  - *m* is the index of the modifier word
  - *l* is a dependency label
  - e.g.: (0,2,root), (2,1,nsubj), (2,4,dobj), (4,3,det), (4,5,pmod), (5,6,pobj)
- Sometimes we just consider unlabeled dependencies

Dependency Parsing

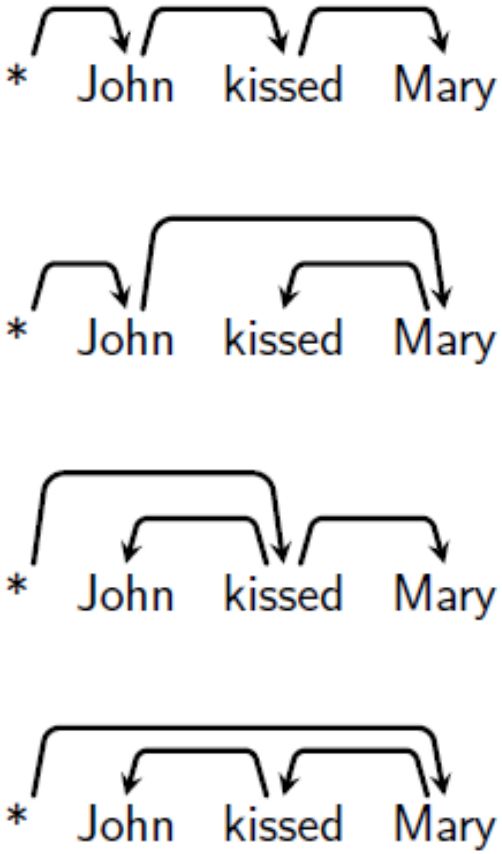
Dependency trees for “John kissed Mary”

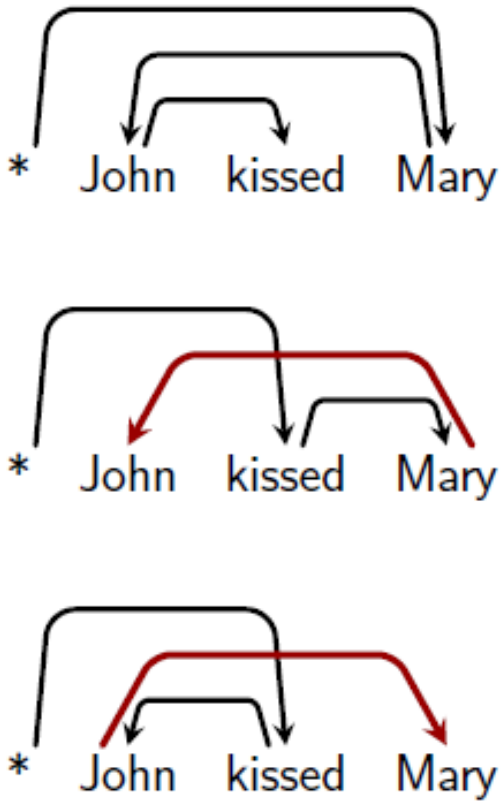




Dependency Parsing

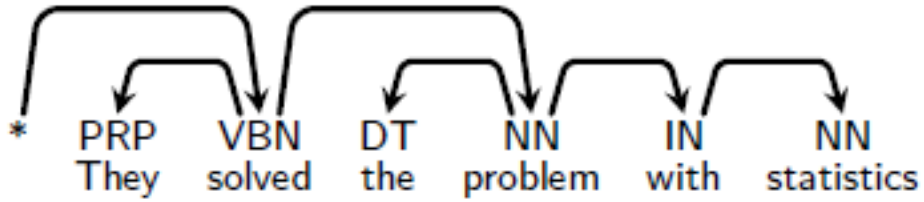
Dependency trees for “John kissed Mary”





Dependency Parsing

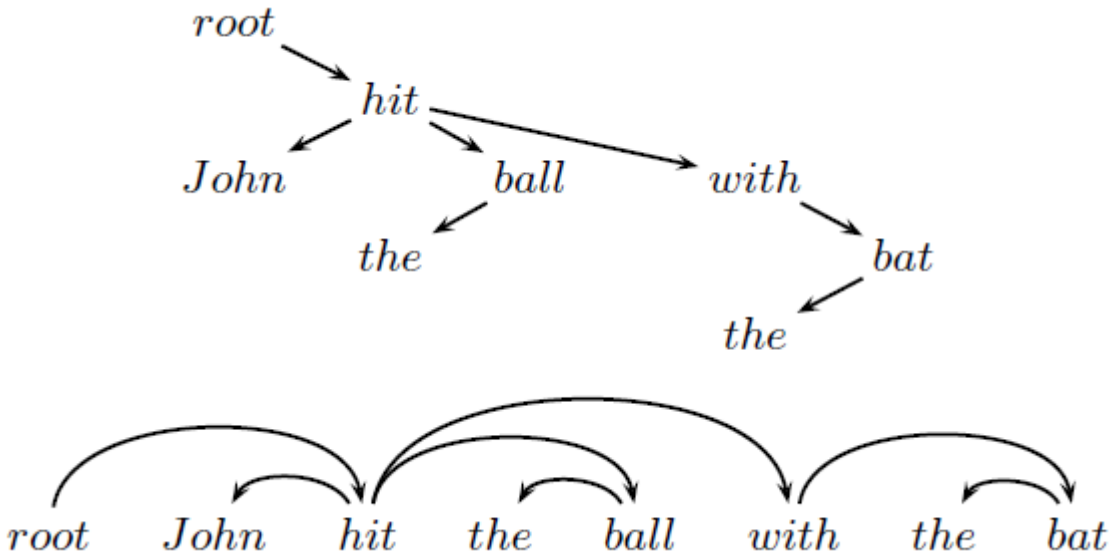
Conditions on Dependency Structures



- Dependency tree
  a) Each non-root token has exactly one incoming arc
  b) All tokens are weakly connected
  c) There are no cycles
  - That is, dependency arcs form a directed tree rooted at *
- Projective dependency trees
  - There are no crossing dependencies
- Non-projective dependency trees, no further constraints
  - Hence a projective tree is also a non-projective set

Dependency Parsing

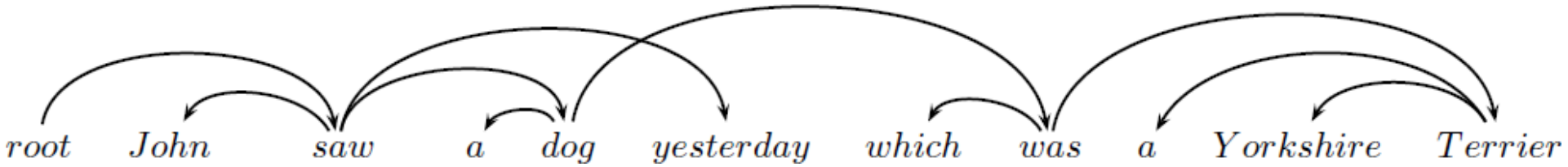
Projective Structures



- A projective tree is one that can be written with all words in a predefined linear order and all edges drawn on the plane with none crossing
- We can say a tree is projective if and only if and edge from word w to word u implies that w is an ancestor of all words between w and u

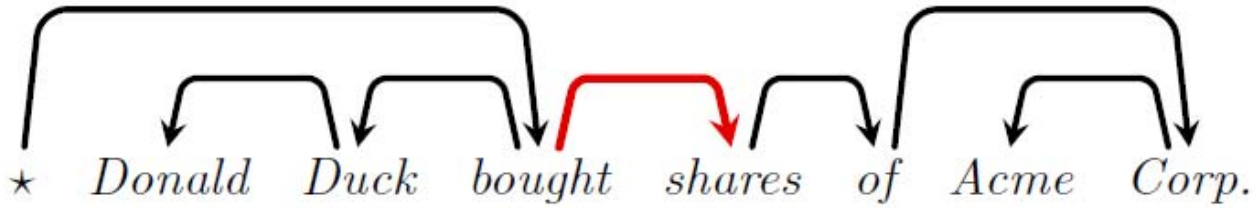
Dependency Parsing

Non-Projective Structures



- There is no way to draw the dependency tree for this sentence in the plane with no crossing edges
- Non-projective dependencies are more frequent in languages with more flexible word order (German, Dutch, Czech,...)

Dependency Parsing



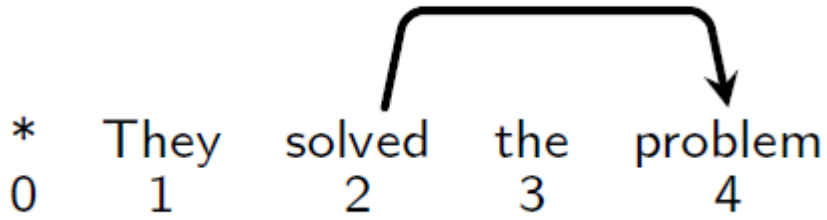
- Directed arcs represent dependencies between
  - a head word (e.g, *bought*)
  - a modifier word (e.g, *shares*)
- The parsing problem

$$\text{parse}(x) = \underset{y \in \mathcal{Y}(x)}{\text{argmax}}\, \text{score}(x, y)$$

  - x is a sentence, $\mathcal{Y}(x)$ is the set of all dependency trees
  - $\text{score}(x, y)$ evaluates compatibility of x and y
- How to score? How to solve the argmax?

Dependency Parsing

Some notation



Given a sentence with *n* words:

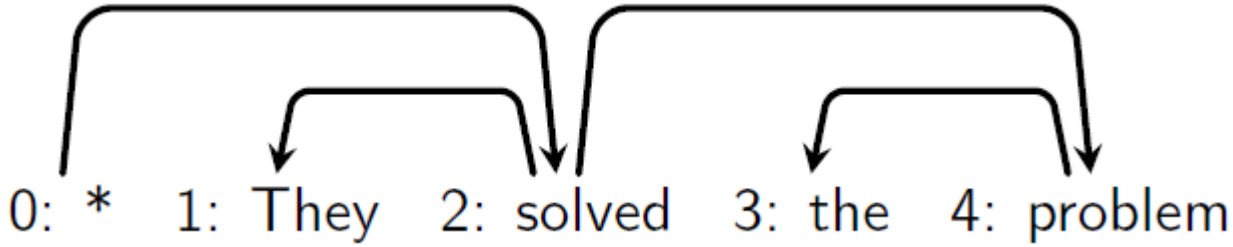
- $\mathcal{D}$ Is the set of all possible dependencies that can be assigned to the sentence. E.g.

  $\mathcal{D}$ = { (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,1), (2,3), (2,4), (3,1), (3,2), (3,4), (4,1), (4,2), (4,3)}

- *y* is a dependency tree
  - $y \subseteq \mathcal{D}$
  - dependencies in *y* form a directed tree (projective or not)
- $\mathcal{Y}$ is the set of all valid dependency trees for the sentence (projective or not)

Dependency Parsing

Probabilistic Arc-factored Dependency Parsing



- Assume we have *p*(modifier word | head word)
- In a probabilistic **arc-factored** model:

$$
\begin{aligned}
p(\mathbf{x}, \mathbf{y}) &= p(\mathbf{x},\ (*,2),(2,1),(2,4),(4,3)\ ) \\
&= p(\mathbf{x}_2,\ (*,2)\ ) \times p(\mathbf{x},\ (2,1),(2,4),(4,3) \mid \mathbf{x}_2,(*,2)\ ) \\
&= p(*) \times p(\mathbf{x}_2 \mid *) \times p(\mathbf{x},\ (2,1),(2,4),(4,3) \mid \mathbf{x}_2, 2) \\
&= p(\mathbf{x}_2 \mid *) \times p(\mathbf{x}_1 \mid \mathbf{x}_2) \times p(\mathbf{x}_4 \mid \mathbf{x}_2) \times p(\mathbf{x}_3 \mid \mathbf{x}_4) \\
&= \prod_{(h,m) \in \mathbf{y}} p(\mathbf{x}_m \mid \mathbf{x}_h)
\end{aligned}
$$

Dependency Parsing

Towards Linear Arc-factored Dependency Parsing

- Consider an arc-factored probabilistic model:

$$p(\mathbf{x}, \mathbf{y}) = \prod_{(h,m)\in\mathbf{y}} p(\mathbf{x}_m \mid \mathbf{x}_h)$$

- Prediction is:

$$\begin{aligned}
\operatorname*{argmax}_{\mathbf{y}\in\mathcal{Y}} p(\mathbf{x}, \mathbf{y}) &= \operatorname*{argmax}_{\mathbf{y}} \prod_{(h,m)\in\mathbf{y}} p(\mathbf{x}_m \mid \mathbf{x}_h) \\
&= \operatorname*{argmax}_{\mathbf{y}} \exp\left\{ \sum_{(h,m)\in\mathbf{y}} \log p(\mathbf{x}_m \mid \mathbf{x}_h) \right\} \\
&= \operatorname*{argmax}_{\mathbf{y}} \sum_{(h,m)\in\mathbf{y}} \log p(\mathbf{x}_m \mid \mathbf{x}_h) \\
&= \operatorname*{argmax}_{\mathbf{y}} \sum_{(h,m)\in\mathbf{y}} \text{score}(\mathbf{x}, h, m)
\end{aligned}$$

where $\text{score}(\mathbf{x}, h, m) = \log p(\mathbf{x}_m \mid \mathbf{x}_h)$

Dependency Parsing

A CRF for Arc-factored Dependency Parsing

- A log-linear distribution of trees **y** given **x**

$$p(\mathbf{y} \mid \mathbf{x}; \mathbf{w}) = \frac{\exp\left\{\sum_{(h,m,l)\in\mathbf{y}} \mathbf{w} \cdot \mathbf{f}(\mathbf{x}, h, m, l)\right\}}{Z(\mathbf{x}; \mathbf{w})}$$

- $\mathbf{f}(\mathbf{x},h,m)$ is a vector of d features of (h,m,l) assigned to x
- $\mathbf{w} \in \mathbb{R}^d$ are the parameters of the model
- $Z(\mathbf{x}; \mathbf{w}) = \sum_{\mathbf{y}\in\mathcal{Y}} \exp\left\{\sum_{(h,m,l)\in\mathbf{y}} \mathbf{w} \cdot \mathbf{f}(\mathbf{x}, h, m, l)\right\}$
- Prediction is linear:

$$\begin{aligned}\underset{\mathbf{y}\in\mathcal{Y}^*}{\operatorname{argmax}}\, P(\mathbf{y}|\mathbf{x}; \mathbf{w}) &= \frac{\exp\left\{\sum_{(h,m,l)\in\mathbf{y}} \mathbf{w} \cdot \mathbf{f}(\mathbf{x}, h, m, l)\right\}}{Z(\mathbf{x}; \mathbf{w})} \\ &= \sum_{(h,m,l)\in\mathbf{y}} \mathbf{w} \cdot \mathbf{f}(\mathbf{x}, h, m, l)\end{aligned}$$

Dependency Parsing

Features in Arc-factored Dependency Parsing

f(**x**,*h*,*m*): a vector of *d* features of (*h*,*m*,*l*) assigned to *x*

- As in tagging, we tipically use indicator features
- Templates in (McDonald et al, 2005):

word features
h-word, h-pos
h-word
h-pos
m-word, m-pos
m-word
m-pos

dependency features
h-word, h-pos, m-word, m-pos
h-pos, m-word, m-pos
h-word, m-word, m-pos
h-word, h-pos, m-pos
h-word, h-pos, m-word
h-word, m-word
h-pos, m-pos

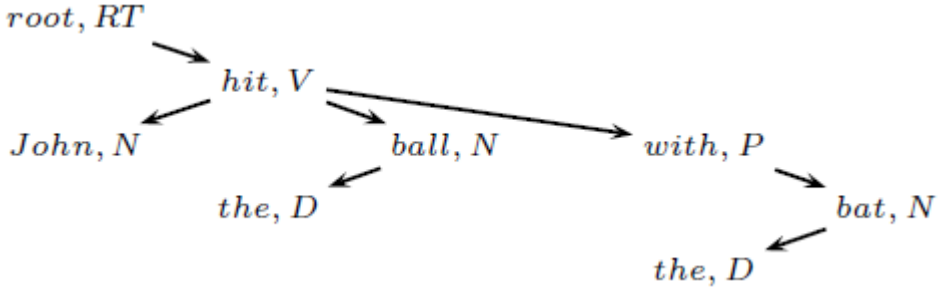
- Example: (feature template + dependency direction)

$$\mathbf{f}_j(\mathbf{x}, h, m, l) = \begin{cases} 1 & \text{if word}(h) = \textit{solve} \text{ and word}(m) = \textit{problem} \\ & \quad \text{and } l = \textit{dobj} \text{ and } h < m \\ 0 & \text{otherwise} \end{cases}$$

Dependency Parsing

Features in Arc-factored Dependency Parsing

Example:



f(hit,with)

h-word="hit", h-pos="V", m-word="with", m-pos="P"

h-pos="V", m-word="with", m-pos="P"

h-word="hit", m-word="with", m-pos="P"

h-word="hit", h-pos="V", m-pos="P"

h-word="hit", h-pos="V", m-word="with"

h-word="hit", m-word="with"

h-pos="V", m-pos="P"

h-word="hit", h-pos="V"

m-word="with", m-pos="P"

h-word="hit"

h-pos="V"

m-word="with"

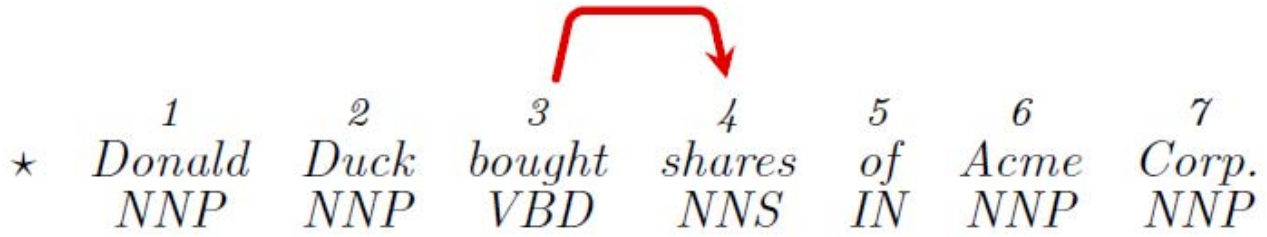
m-pos="P"

+ the direction of attachment and the distance between words:

h-word("hit"), m-word("with"), dir=R, dist=2

Dependency Parsing

Features in Arc-factored Dependency Parsing



- Some features:

$$\mathbf{f}(\mathbf{x}, 3, 4) = [\![\text{“bought”} \rightarrow \text{“shares”}, \; VBD \rightarrow NNS, \; \text{“bought”}/VBD \rightarrow \text{“shares”}/NNS, \; \text{“bought”} \rightarrow NNS, \; VBD \rightarrow \text{“shares”}, \; \text{adjacent}, \; \text{head is left of mod}, \ldots]\!]$$

- In general **f** can capture any property of the dependency and the entire **x**

Dependency Parsing

A CRF for Arc-factored Dependency Parsing

$$p(\mathbf{y} \mid \mathbf{x}; \mathbf{w}) = \frac{\exp\left\{\sum_{(h,m,l)\in\mathbf{y}} \mathbf{w} \cdot \mathbf{f}(\mathbf{x}, h, m, l)\right\}}{Z(\mathbf{x}; \mathbf{w})}$$

- **Decoding**: predict the best dependency tree for **x**

  $$\operatorname*{argmax}_{\mathbf{y}\in\mathcal{Y}} \mathrm{p}(\mathbf{y}|\mathbf{x}; \mathbf{w})$$

  when
  - $\mathcal{Y}$ is the set of projective trees for **x**
  - $\mathcal{Y}$ is the set of non-projective trees for **x**
- **Parameter estimation**: given training data

  $$\left\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \ldots, (\mathbf{x}^{(m)}, \mathbf{y}^{(m)})\right\}$$

  learn parameters **w**

Dependency Parsing

Parameter Estimation: CRF for Parsing

…analogous to CRF for tagging

- Given a training set

$$\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \ldots, (\mathbf{x}^{(m)}, \mathbf{y}^{(m)})\}$$

  estimate **w**

- Define the conditional log-likelihood of the data:

$$L(\mathbf{w}) = \frac{1}{m} \sum_{k=1}^{m} \log \mathrm{p}(\mathbf{y}^{(k)} | \mathbf{x}^{(k)}; \mathbf{w})$$

- $L(\mathbf{w})$ measures how well **w** explains the data. A good value for **w** will give a high value for $\mathrm{p}(\mathbf{y}^{(k)} | \mathbf{x}^{(k)}; \mathbf{w})$ for all $k = 1 \ldots m$
- We want **w** that maximizes $L(\mathbf{w})$

Dependency Parsing

Parsing Dependency Structures

- Arc based factorization: $s(i,j) = \mathbf{w} \cdot \mathbf{f}(i,j)$

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

- **Maximum Spanning Tree** of a graph *G=(V,E)*: tree *y* that maximizes the value $\sum_{(i,j)\in y} s(i,j)$ such that *(i,j)* ϵ *E* and every vertex in *V* is used in the construction of *y*

- For each sentence *x* we can define a directed graph *Gx = (Vx, Ex)* where:

$$V_{\boldsymbol{x}} = \{x_0 = \text{root}, x_1, \ldots, x_n\} \text{ and}$$
$$E_{\boldsymbol{x}} = \{(i, j) : x_i \neq x_j, x_i \in V_{\boldsymbol{x}}, x_j \in V_{\boldsymbol{x}} - \text{root}\}$$

Dependency Parsing

Parsing Dependency Structures

- Finding the dependency tree of highest score is equivalent to finding the MST in *Gx* rooted at the artificial root

- For parsing, we can use efficient algorithms for finding MSTs in undirected graphs:
  - **Projective structures**

    Eisner algorithm

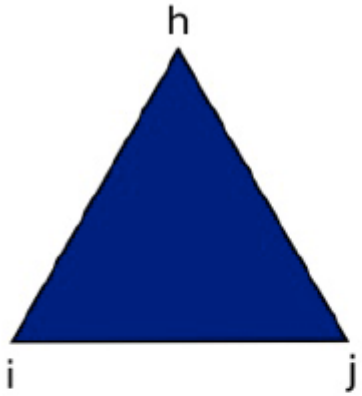
    Modified version of CKY
  - **Non-projective structures**

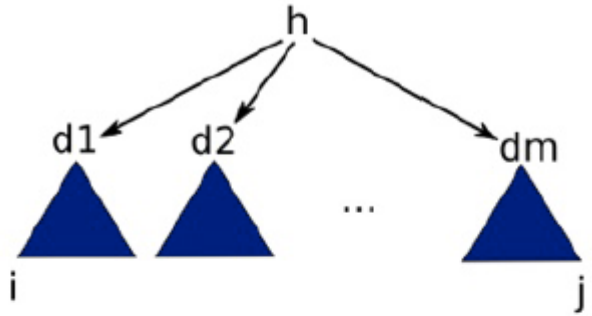
    Chu-Liu-Edmonds algorithm

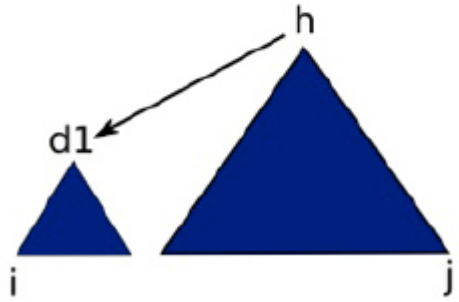
Dependency Parsing

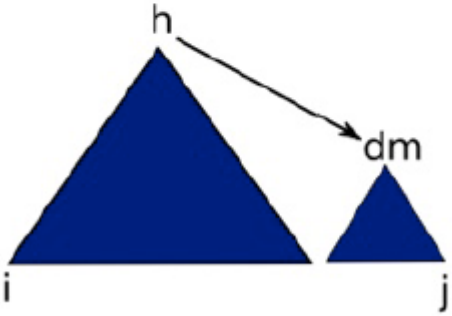
Parsing Projective Structures

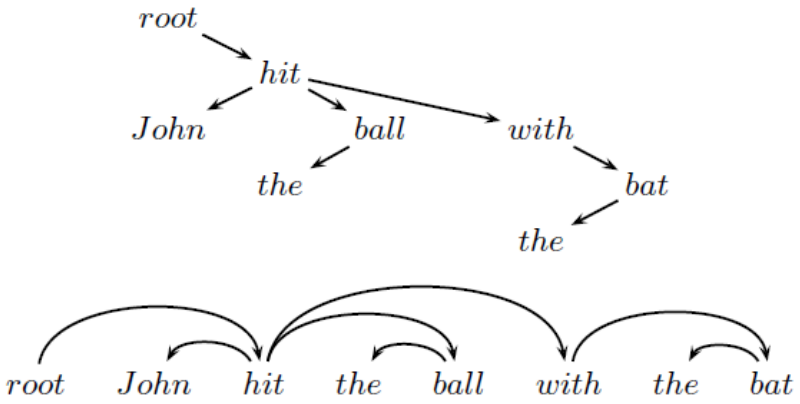
- Any projective tree can be written as a combination of:
  - two smaller *adjacent* projective trees
  - a dependency connecting their roots







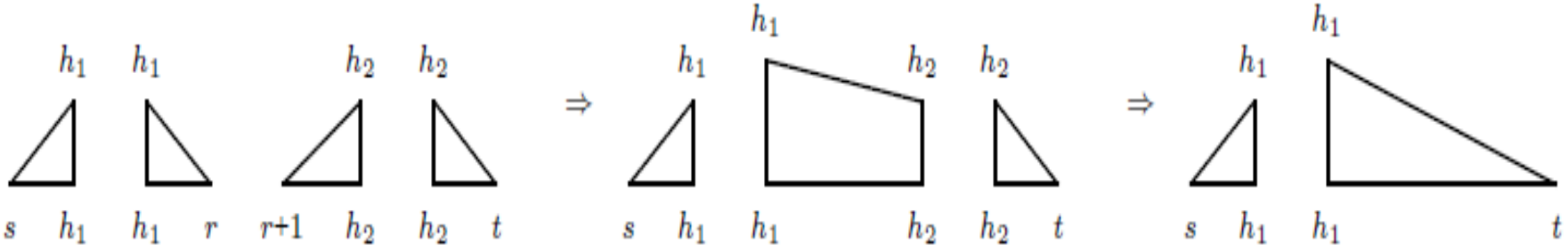




Dependency Parsing

Parsing Projective Structures

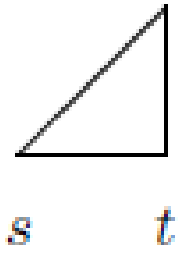
- The algorithm (Eisner 1996) (Eisner 2000) is a variation of CKY
- Main idea: parse the left and right dependents of a word independently, and combine them at a later stage
  - Var indicating the direction of the item (gathering left or right dependents)
  - Var indicating whether an item is complete (available to gather more dependents)



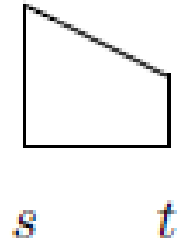
Dependency Parsing

Parsing Projective Structures

- C[s][t][d][c]: dynamic programming table that stores the score of the best subtree from position s to position t (s≤t) with direction d and complete value c
- Ex: C[s][t][←][1] would be the score of the best subtree represented by the item



and C[s][t][→][0] for the item



Dependency Parsing

Parsing Projective Structures

Initialization: $C[s][s][d][c] = 0.0 \quad \forall s, d, c$

for $k : 1..n$

  for $s : 1..n$

    $t = s + k$

    if $t > n$ then break

$$C[s][t][\leftarrow][0] = max_{s \leq r < t}(C[s][r][\rightarrow][1] + C[r+1][t][\leftarrow][1] + s(t, s))$$
$$C[s][t][\rightarrow][0] = max_{s \leq r < t}(C[s][r][\rightarrow][1] + C[r+1][t][\leftarrow][1] + s(s, t))$$
$$C[s][t][\leftarrow][1] = max_{s \leq r < t}(C[s][r][\leftarrow][1] + C[r][t][\leftarrow][0])$$
$$C[s][t][\rightarrow][1] = max_{s < r \leq t}(C[s][r][\rightarrow][0] + C[r][t][\rightarrow][1])$$

  end for

end for

- Score of the best tree for the entire sentence: C[1][n][→][1]
- Best dependency tree: need to maintain backpointers
- Run time: O(n3)

Dependency Parsing

Dependency Parsing nowadays

- Two main approaches:
  - Graph-based: better results
  - Transition-based: far faster
- Google uses T-b dependency parsing for their MT systems
- Current state-of-the-art results: 93.9%
  - Merge of transition-based and graph-based
  - With features of 2nd and 3rd level
  - The precision “jump” occurs when you look at the history (what the parser has done so far) in order to make decisions