Graph-Based Parsing

Miguel Ballesteros

Algorithms for NLP Course.
7-11

Carnegie Mellon

By using some Joakim Nivre's materials from Uppsala University
and Jason Eisner's material from Johns Hopkins University.

Outline

- Arc-Factored Models.
- Eisner's Algorithm.
- Spanning Tree Parsing.
- Higher-order-models
  - Projective Parsing.
  - Non-Projective Parsing.

Arc-Factored Models

- A widely used graph-based model for dependency parsing is the so-called arc-factored (or edge-factored) model.
- The score of a dep.tree is decomposed into the scores of individual arcs:

$$\mathbf{Score}(\boldsymbol{x},\boldsymbol{y}) = \sum_{(i,l,j) \in A_y} \mathbf{Score}(i,l,j,x)$$

  - We use A_y for the arc set of the dependency tree *y* (that is, y=(V_x,A_y) and Score(i,l,j,x) for the score of the arc (i,l,j) in the sentence *x*.

Arc-Factored Models

- This gives us a parsing model, where the generative component maps each sentence x to the set of all spanning trees for the node set V_x (relative to some label set L):

$\mathrm{GEN}(x) = \{y \mid y \text{ is a spanning tree in } G_x = (V_x , V_x \times L \times V_x)\}$

Arc-Factored Models

- The evaluative component then ranks candidate trees according to their arc-factored score and returns the one with the highest score:

$$\mathbf{y^*}_{y \in GEN(X)} = \mathbf{argmax\ EVAL(X,Y)} = \sum_{(i,l,j) \in A_y} \mathbf{Score(i,l,j,x)}$$

- The scoring function is usually of the form:

$$\mathbf{Score(i,l,j,x)} = \sum_{(i,l,j) \in A_y} \mathbf{f_k(i,l,j,x) \cdot w_k}$$

Arc-Factored Models

- The scoring function is usually of the form:

$$\mathbf{Score(i,l,j,x) = \sum_{(i,l,j) \in A_y} f_k(i,l,j,x) \cdot w_k}$$

- In this model, every function $\mathbf{f_k(i, l, j, x)}$ is a numerical (often binary) feature function, representing some salient property of the arc (i, l, j) in the context of x
- $\mathbf{w_k}$ is a real-valued feature weight, reflecting the tendency of $\mathbf{f_i\ (i, l, j, x)}$ to co-occur with good or bad parses.

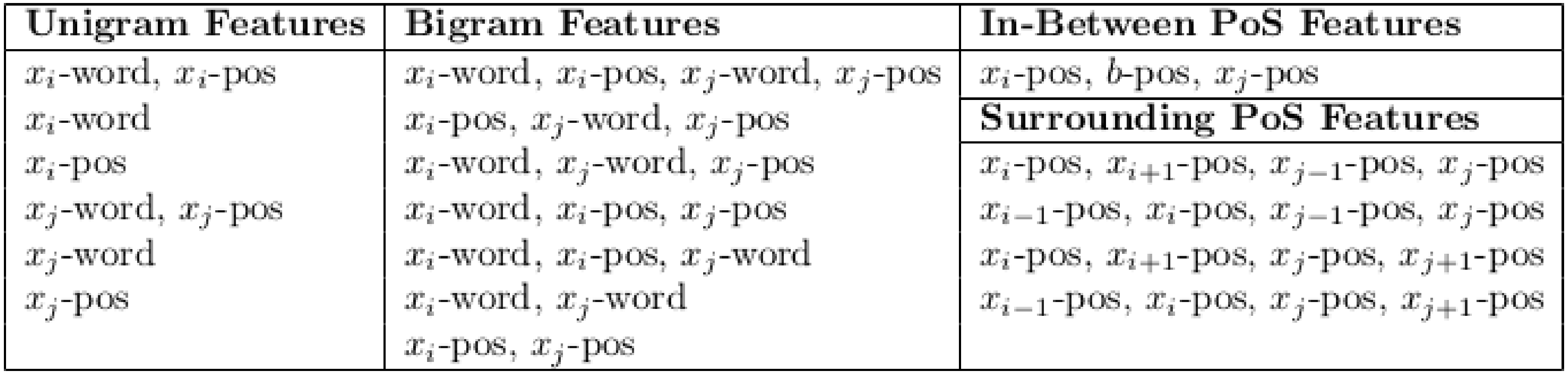
Arc-Factored Models

- Dependency parsers tend to use purely discriminative models instead of probability models
  - but there is no principled reason why this should be the case.
- In a linear discriminative scoring model for arcs, the final form of the arc-factored model is:

$$y^* = \underset{y \in GEN(X)}{\operatorname{argmax}} \; EVAL(x,y) = \underset{y \in GEN(X)}{\operatorname{argmax}} \sum_{(i,l,j) \in A_y} \sum_{k=1}^{K} f_k(i,l,j,x) \cdot w_k$$

Arc-Factored Models

- Since this model contains no grammatical rules at all, parsing accuracy is completely dependent on the choice of good features.
- Here an example of features:



Unigram Features	**Bigram Features**	**In-Between PoS Features**
x_i-word, x_i-pos	x_i-word, x_i-pos, x_j-word, x_j-pos	x_i-pos, b-pos, x_j-pos
x_i-word	x_i-pos, x_j-word, x_j-pos	**Surrounding PoS Features**
x_i-pos	x_i-word, x_j-word, x_j-pos	x_i-pos, x_{i+1}-pos, x_{j-1}-pos, x_j-pos
x_j-word, x_j-pos	x_i-word, x_i-pos, x_j-pos	x_{i-1}-pos, x_i-pos, x_{j-1}-pos, x_j-pos
x_j-word	x_i-word, x_i-pos, x_j-word	x_i-pos, x_{i+1}-pos, x_j-pos, x_{j+1}-pos
x_j-pos	x_i-word, x_j-word	x_{i-1}-pos, x_i-pos, x_j-pos, x_{j+1}-pos
	x_i-pos, x_j-pos	

TABLE 1. Arc-factored feature templates $\mathbf{f}_k(i, l, j, x)$ for the arc (i, l, j) in the context of sentence x (McDonald, 2006). Notation: w-word = word form of w; w-pos = part-of-speech tag of w; b = any word between w_i and w_j. Binarization is used to obtain one binary feature for each possible instantiation of a template.

Online Learning for Arc-Factored

- Learning the weights of an arc-factored model can be done in many ways,
- Popular approach: online learning algorithm that parses one sentence at a time and updates the weights after each training example with the goal of minimizing the number of errors on the training corpus.
- Perceptron Learning.

Eisner's Algorithm

- In order to use the arc-factored model for parsing, we need an efficient algorithm for decoding.
  - That is: finding the highest scoring dependency tree y for a given sentence x
- We also need an efficient decoding algorithm for learning the weights of the model (in the perceptron scenario)

Eisner's Algorithm

- As long as we restrict ourselves to projective trees, we can build dependency trees by a dynamic programming algorithm very similar to the CKY algorithm.

- We can define a head-lexicalized (P)CFG that generates a parse trees isomorphic to dep.trees.

- The Eisner's algorithm is an efficient $O(n^3)$ that does this. Eisner (1996, 2000) (also projective trees)

Eisner's algorithm

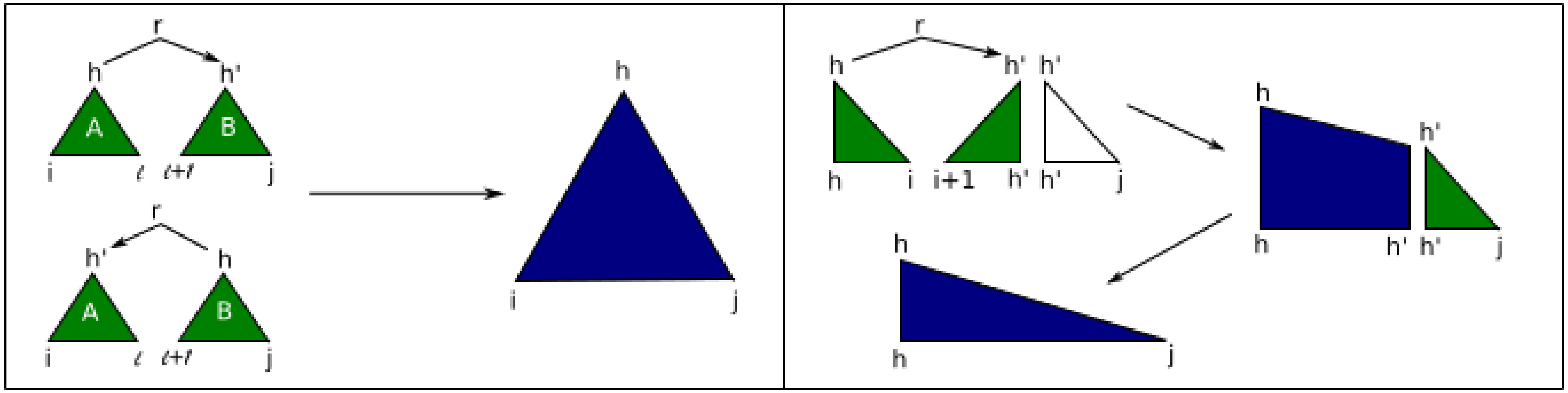


FIGURE 4. Comparison of the chart items in the CKY algorithm (left) and Eisner's algorithm (right).

- Schematic representation of the process (prev slide) for left-headed dependencies and contrasts it with the CKY algorithm (left).

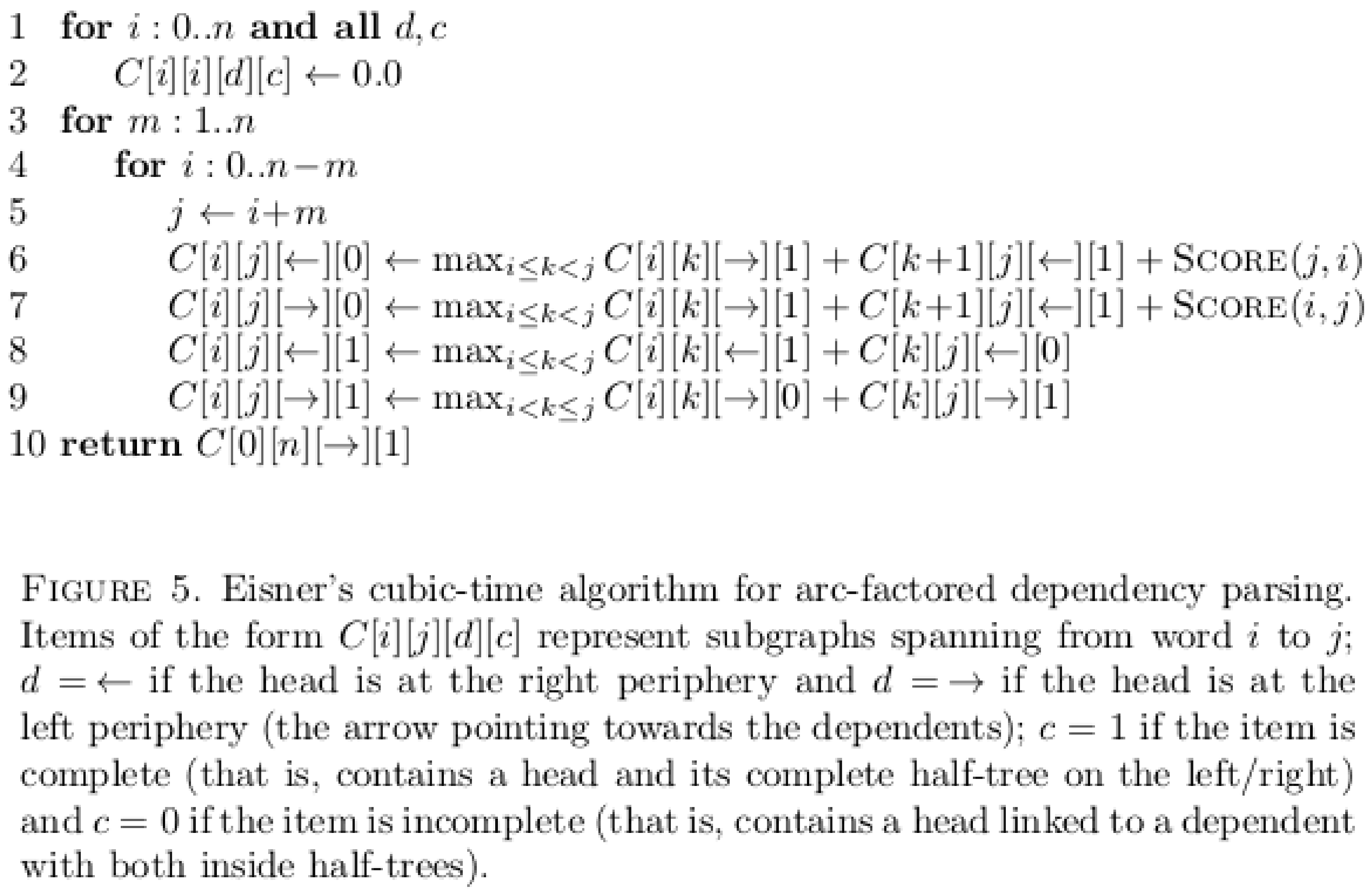
Eisner's Algorithm

- The key idea is to use a split-head representation
  - to let the chart cells represent half-trees instead of trees,
  - we can replace the indices for lexical heads (one in each half of the new subtree) by boolean variables indicating whether the lexical head is at the left or at the right periphery of the respective half-trees.

Eisner's Algorithm

- We need **two combinatory operations**,
  - 1- for adding a dependency arc between the heads of two half-trees to form an incomplete half-tree.
  - 2- for subsequently combining this incomplete half-tree and with a complete half-tree to form a larger complete half-tree.

Eisner's Algorithm



```
for $i : 0..n$ and all $d, c$
    $C[i][i][d][c] \leftarrow 0.0$
for $m : 1..n$
    for $i : 0..n-m$
        $j \leftarrow i+m$
        $C[i][j][\leftarrow][0] \leftarrow \max_{i \leq k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1] + \textsc{Score}(j, i)$
        $C[i][j][\rightarrow][0] \leftarrow \max_{i \leq k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1] + \textsc{Score}(i, j)$
        $C[i][j][\leftarrow][1] \leftarrow \max_{i \leq k < j} C[i][k][\leftarrow][1] + C[k][j][\leftarrow][0]$
        $C[i][j][\rightarrow][1] \leftarrow \max_{i < k \leq j} C[i][k][\rightarrow][0] + C[k][j][\rightarrow][1]$
return $C[0][n][\rightarrow][1]$
```

FIGURE 5. Eisner's cubic-time algorithm for arc-factored dependency parsing. Items of the form $C[i][j][d][c]$ represent subgraphs spanning from word i to j; $d = \leftarrow$ if the head is at the right periphery and $d = \rightarrow$ if the head is at the left periphery (the arrow pointing towards the dependents); $c = 1$ if the item is complete (that is, contains a head and its complete half-tree on the left/right) and $c = 0$ if the item is incomplete (that is, contains a head linked to a dependent with both inside half-trees).

Eisner's Algorithm

```
for $i : 0..n$ and all $d, c$
    $C[i][i][d][c] \leftarrow 0.0$
```

Parsing is initialized by setting the score of every possible one-word item
(left-headed, right-headed, complete and incomplete) to zero
(C is for complete half tree or incomplete: true or false)

FIGURE 5. Eisner's cubic-time algorithm for arc-factored dependency parsing. Items of the form $C[i][j][d][c]$ represent subgraphs spanning from word i to j; $d = \leftarrow$ if the head is at the right periphery and $d = \rightarrow$ if the head is at the left periphery (the arrow pointing towards the dependents); $c = 1$ if the item is complete (that is, contains a head and its complete half-tree on the left/right) and $c = 0$ if the item is incomplete (that is, contains a head linked to a dependent with both inside half-trees).

Eisner's algorithm

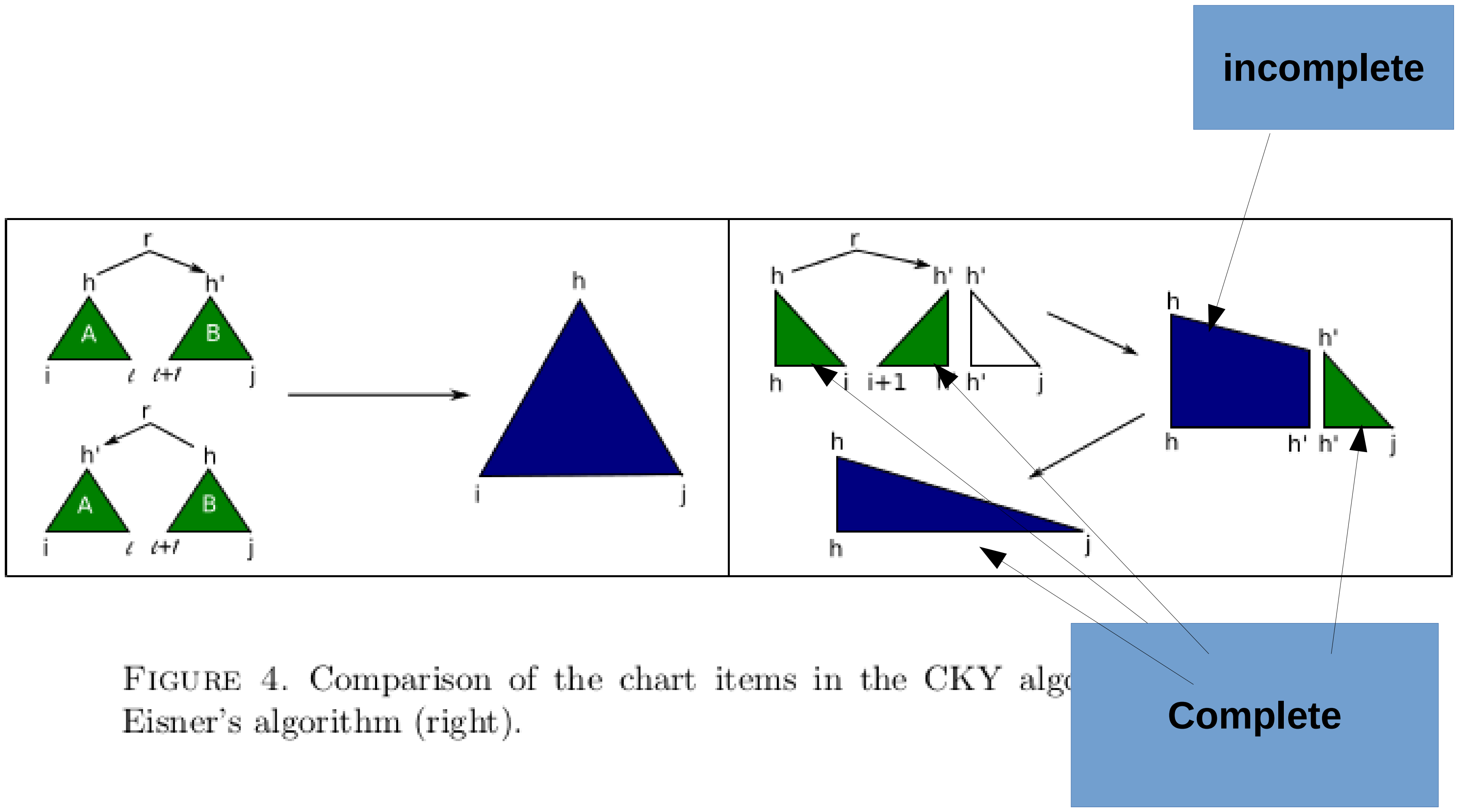
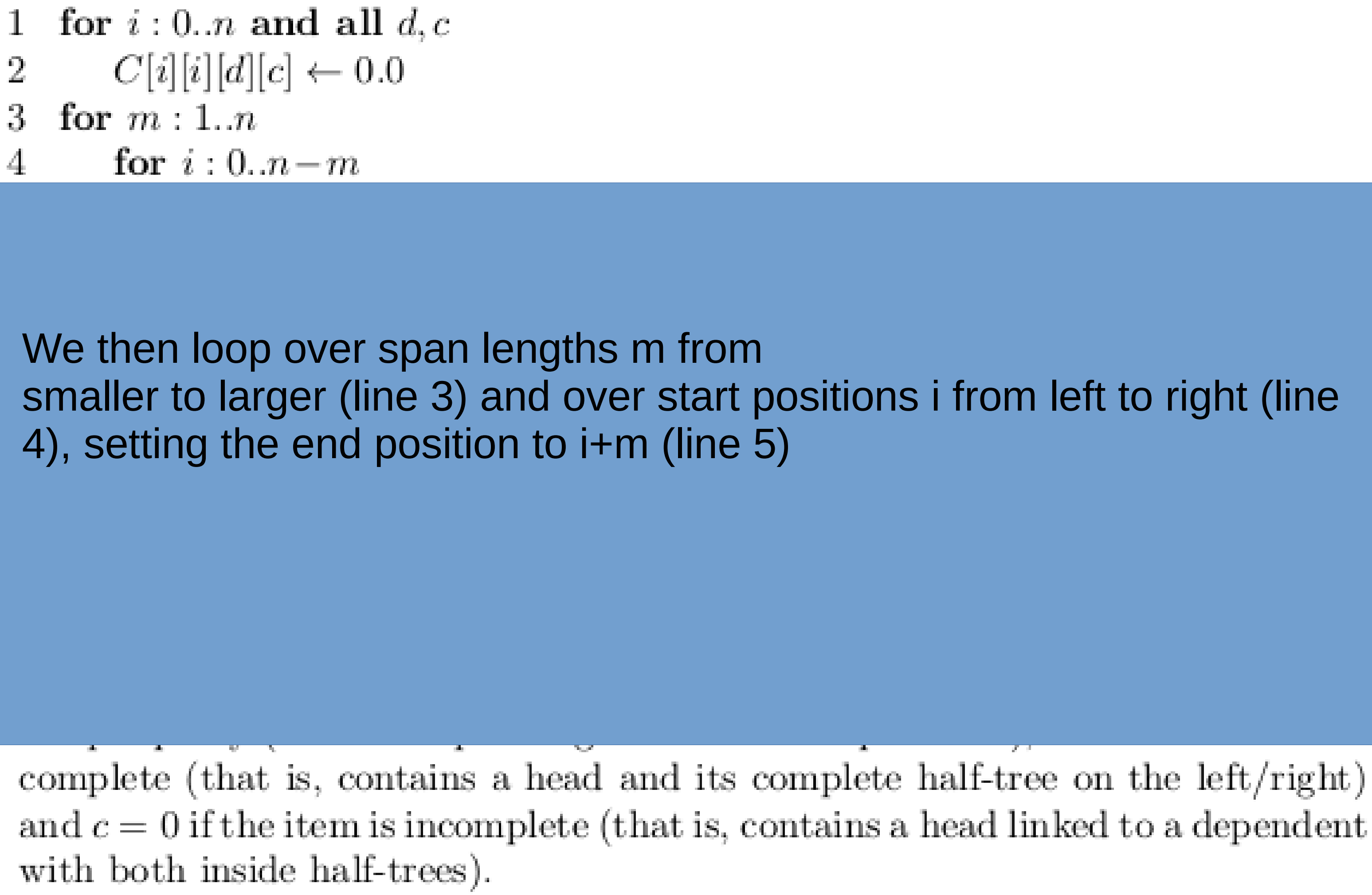


FIGURE 4. Comparison of the chart items in the CKY algo
Eisner's algorithm (right).

Eisner's Algorithm



```
1  for $i : 0..n$ and all $d, c$
2      $C[i][i][d][c] \leftarrow 0.0$
3  for $m : 1..n$
4      for $i : 0..n-m$
```

We then loop over span lengths m from
smaller to larger (line 3) and over start positions i from left to right (line
4), setting the end position to i+m (line 5)

complete (that is, contains a head and its complete half-tree on the left/right) and $c = 0$ if the item is incomplete (that is, contains a head linked to a dependent with both inside half-trees).

Eisner's Algorithm

```
1  for $i : 0..n$ and all $d, c$
2      $C[i][i][d][c] \leftarrow 0.0$
3  for $m : 1..n$
4      for $i : 0..n-m$
5          $j \leftarrow i+m$
6          $C[i][j][\leftarrow][0] \leftarrow \max_{i \leq k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1] + \textsc{Score}(j, i)$
7          $C[i][j][\rightarrow][0] \leftarrow \max_{i \leq k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1] + \textsc{Score}(i, j)$
```

We first find the best incomplete items from i to j by finding the best split position k and adding up the scores of the left and right halves plus the score of the new arc, either (j, i) (line 6) or (i, j) (line 7).

Note that the two (complete) items being combined must have i and j as their heads.

We are simply creating an arc from i to j (or j to i)

Eisner's Algorithm

```
1 for $i : 0..n$ and all $d, c$
2     $C[i][i][d][c] \leftarrow 0.0$
3 for $m : 1..n$
4     for $i : 0..n-m$
5         $j \leftarrow i+m$
6         $C[i][j][\leftarrow][0] \leftarrow \max_{i\leq k<j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1] + \textsc{Score}(j, i)$
7         $C[i][j][\rightarrow][0] \leftarrow \max_{i\leq k<j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1] + \textsc{Score}(i, j)$
8         $C[i][j][\leftarrow][1] \leftarrow \max_{i\leq k<j} C[i][k][\leftarrow][1] + C[k][j][\leftarrow][0]$
9         $C[i][j][\rightarrow][1] \leftarrow \max_{i<k\leq j} C[i][k][\rightarrow][0] + C[k][j][\rightarrow][1]$
```

Next, we find the best complete items from i to j by again
finding the best split position k but this time only adding up the scores of one incomplete and one complete item having the same head, producing a complete item with the head to the right (line 8) or to the left (line 9).

Eisner's Algorithm

```
for $i : 0..n$ and all $d, c$
    $C[i][i][d][c] \leftarrow 0.0$
for $m : 1..n$
    for $i : 0..n-m$
        $j \leftarrow i+m$
        $C[i][j][\leftarrow][0] \leftarrow \max_{i \leq k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1] + \textsc{Score}(j, i)$
        $C[i][j][\rightarrow][0] \leftarrow \max_{i \leq k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1] + \textsc{Score}(i, j)$
        $C[i][j][\leftarrow][1] \leftarrow \max_{i \leq k < j} C[i][k][\leftarrow][1] + C[k][j][\leftarrow][0]$
        $C[i][j][\rightarrow][1] \leftarrow \max_{i < k \leq j} C[i][k][\rightarrow][0] + C[k][j][\rightarrow][1]$
return $C[0][n][\rightarrow][1]$
```

Finally, we return the highest-scoring complete item spanning 0 to n and with its head to the left

Eisner's Algorithm Complexity

```
for $i : 0..n$ and all $d, c$
    $C[i][i][d][c] \leftarrow 0.0$
for $m : 1..n$
    for $i : 0..n-m$
        $j \leftarrow i+m$
        $C[i][j][\leftarrow][0] \leftarrow \max_{i \le k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1] + \textsc{Score}(j, i)$
        $C[i][j][\rightarrow][0] \leftarrow \max_{i \le k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1] + \textsc{Score}(i, j)$
        $C[i][j][\leftarrow][1] \leftarrow \max_{i \le k < j} C[i][k][\leftarrow][1] + C[k][j][\leftarrow][0]$
        $C[i][j][\rightarrow][1] \leftarrow \max_{i < k \le j} C[i][k][\rightarrow][0] + C[k][j][\rightarrow][1]$
return $C[0][n][\rightarrow][1]$
```

It is easy to see that the parsing complexity is O(n^3) since

-two nested **for** loops

-a loop inside (the **max** operation all of which takes O(n) time)

(max basically calculates the highest scoring one in the chart)

Eisner's Algorithm

- Eisner’s algorithm was first put to use in Eisner (1996), who combined it with a generative probability model to create one of the first statistical dependency parsers.
- A later version of it by McDonald et al. (2005), using an arc-factored discriminative model and online learning reached state-of-the-art for some languages.

Spanning Tree Parsing

- Given an arc-factored model, finding the highest scoring dependency tree y for a sentence x is equivalent to the graph-theoretic problem of finding a weighted maximum spanning tree in G_x
- Where the weight of an arc (i, l, j) is simply Score(i, l, j, x)
- From this perspective, Eisner's algorithm can be understood as an algorithm for finding a **projective** maximum spanning tree.
  - G_x is the complete graph $G_x=(V_x, V_x \times L \times V_x)$

Spanning Tree Parsing

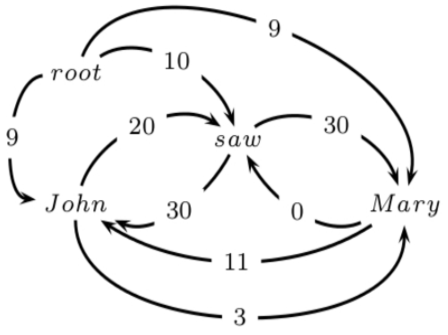
- So, what about **non-projective trees**?
- Finding a maximum spanning tree without the projectivity constraint is a harder problem.
- We need to use standard algorithms from graph theory for finding a MST in a directed graph
  - **Chu-Liu-Edmonds Algorithm**
    (Chu & Liu 1965, Edmonds 1967)

Spanning Tree Parsing

- This is a greedy recursive algorithm that starts by building a graph where every node (except the designated root node) has its highest-scoring incoming arc.
  - It this graph is a tree, it is the maximum spanning tree.
    We are done.
  - If it is not a tree, it contains at least one cycle. The algorithm identifies the cycle, contracts the cycle into a single node and calls itself recursively on the smaller graph.
    - A MST of the smaller graph can be expanded to a MST of the larger initial graph.
    - How? When the recursion bottoms out we can construct successively larger solutions until we have the maximum spanning tree for the original graph.

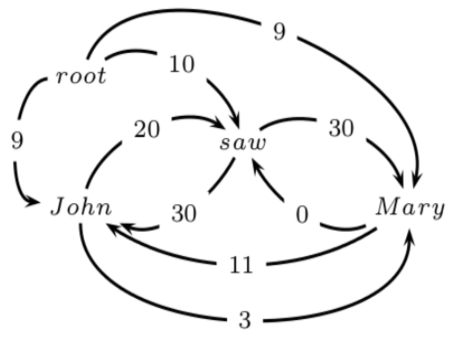
      We are done.

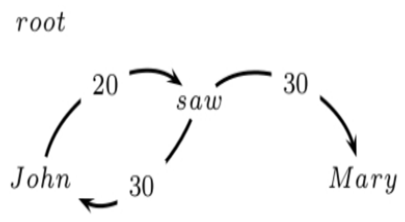
Spanning Tree Parsing



Arc-factored complete graph Gx

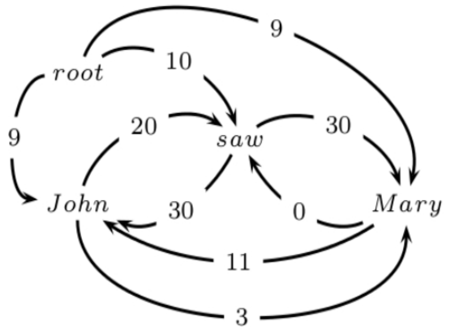
Spanning Tree Parsing

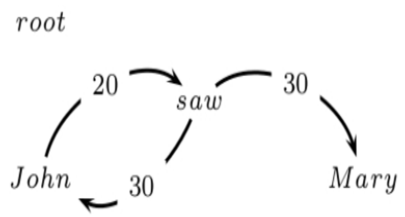




Arc-factored complete graph Gx

Spanning Tree Parsing

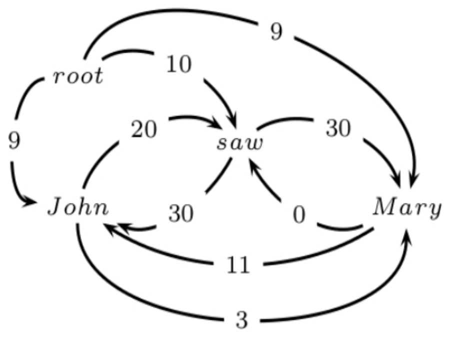


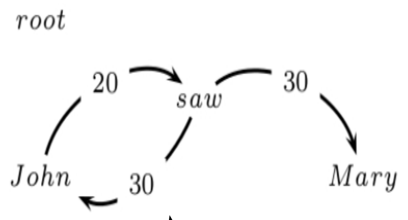


Arc-factored complete graph Gx

Contains a cycle?

Spanning Tree Parsing

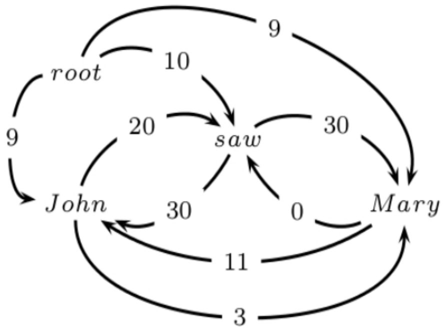


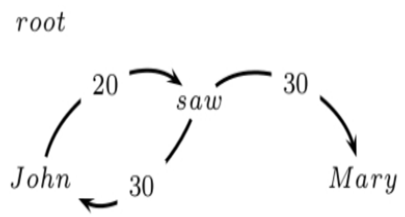


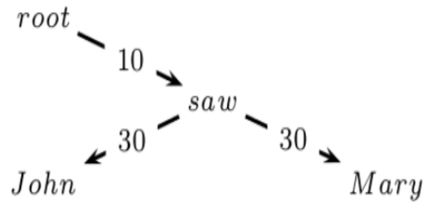
Arc-factored complete graph Gx

Contains a cycle?
YES

Spanning Tree Parsing







Arc-factored complete graph Gx

Contains a cycle?
YES

Recursive call

MST

Spanning Tree Parsing

- Every recursive call takes $O(n^2)$ time, and there can be at most $O(n)$ recursive calls, which means that the Chu-Liu-Edmonds algorithm can be used to perform non-projective dependency parsing with the arc-factored model in $O(n^3)$ time.

- Note that there is an optimization due to Tarjan (1977) that brings the complexity down to $O(n^2)$ (faster than projective parsing with Eisner's)

Spanning Tree Parsing

- The most famous parser that implements this strategy is **MSTParser** by Ryan McDonald that using what I'm going to present in the next slides won the CoNLL 2006 and 2007 Shared Tasks.

Higher-Order Graph-Based Models

Higher-Order Graph-Based Models

- The arc-factored model is often referred to as a **first-order graph-based model**, because it decomposes the score of a tree into the scores of one arc at a time.

$$\mathbf{y^*_{y \in GEN(X)} = argmax\ EVAL(X,Y) = \sum_{(i,l,j) \in A_y} Score(i,l,j,x)}$$

From a linguistic point of view this is a rather drastic assumption, similar to the independence assumptions we used to have in the simpler PCFG models.

Even more drastic assumption because we do not have a grammar that makes linguistic constraints.

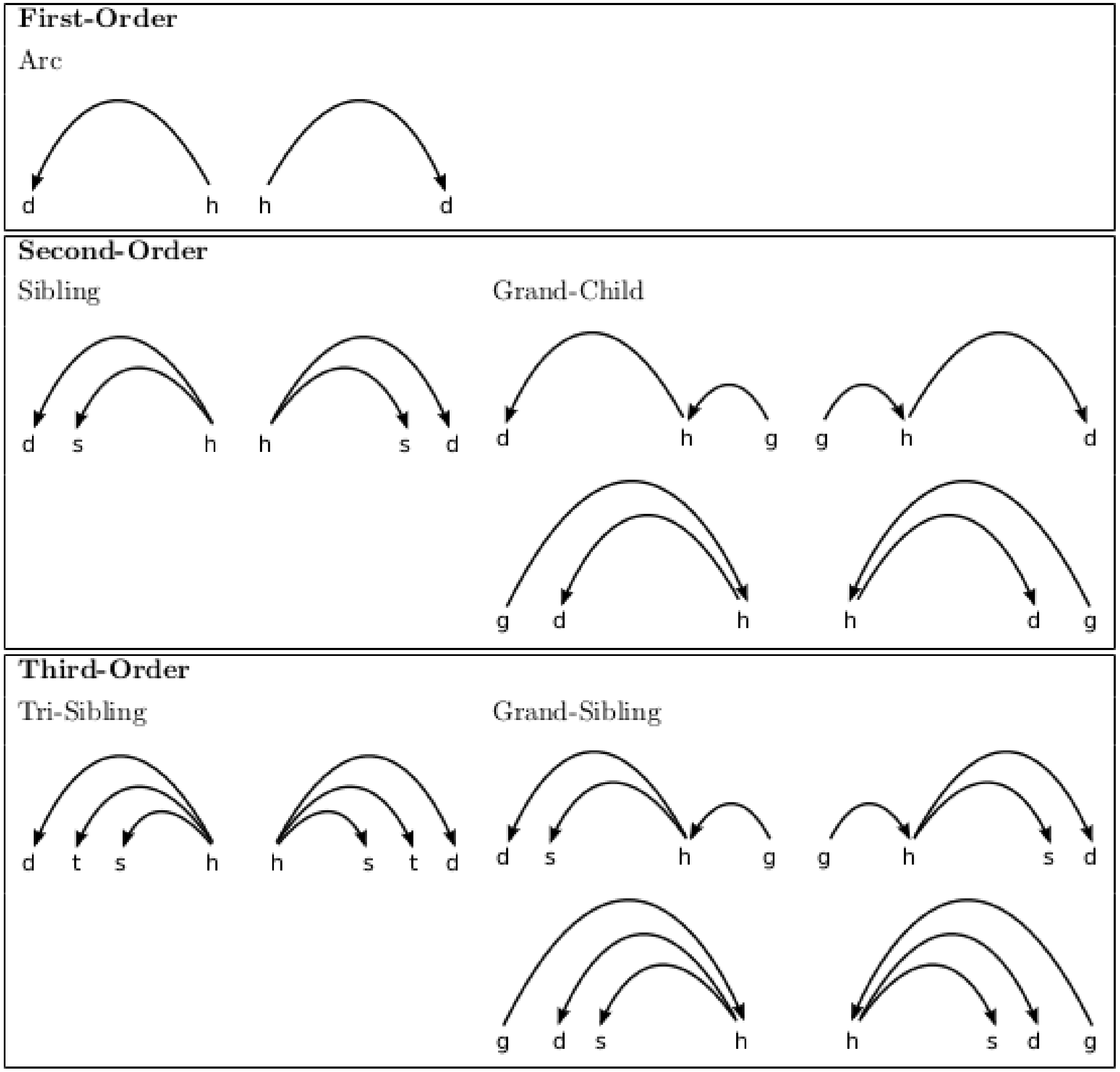
A first-order model might make stupid mistakes like attaching two subjects to a verb.

Higher-Order Graph-Based Models

- Much better parsing accuracy can be achieved if there are more factors taken into account.
  - Instead of only considering first-order features, that is scores between simpler arcs.
  - Bigger subgraphs are considered.
  - Remove lack of context (as in better PCFGs).

- This is what people call 2^{nd} order and 3^{rd} order models for graph-based parsing.

Higher-Order Graph-Based Models

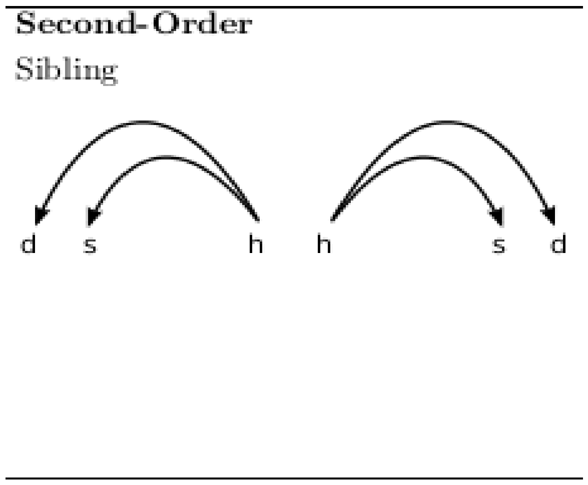


Higher-Order Graph-Based Models

- Of course, when using higher-order models it is important to manage the complexity.

Second-Order Graph-Based Models

- McDonald & Pereira (2006)
- In this model, the trees are decomposed into two-arc subgraphs consisting of a head and two adjacent dependents on the same side of the head word.
  - This is called a sibling factor.



Second-Order Graph-Based Models

- When scoring dependency trees in this model, it is common to also include the simpler first-order arc factors from the (first-order) arc-factored model.
- In this way, the relation between a head and a dependent is scored both in isolation and in the context of the nearest preceding sibling.

$$\mathbf{Score\ (x,y)} = \sum_{(h,d) \in A_y} \mathbf{Score(h,d)} + \sum_{(h,s,d) \in A_y} \mathbf{Score(h,s,d)}$$

Second-Order Graph-Based Models

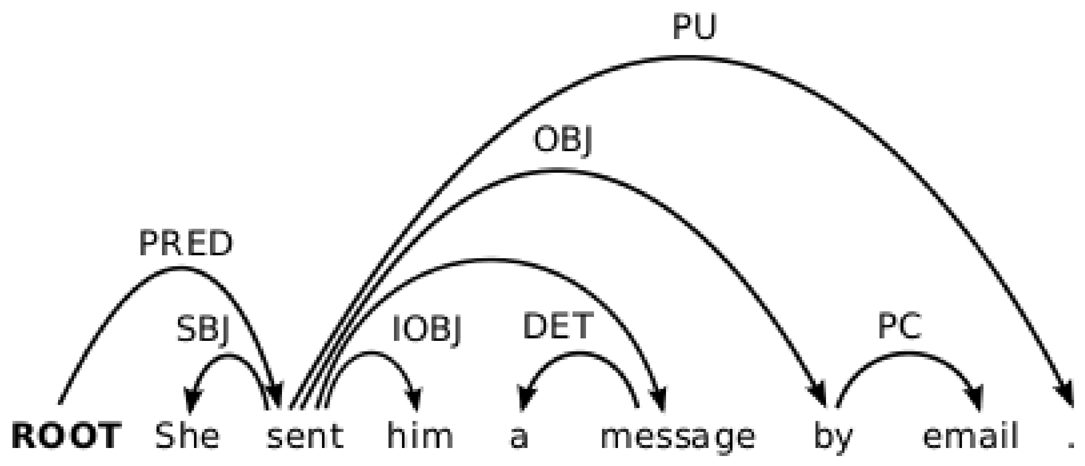
- When scoring dependency trees in this model, it is common to also include the simpler first-order arc factors from the (first-order) arc-factored model.
- In this way, the relation between a head and a dependent is scored both in isolation and in the context of the nearest preceding sibling.

$$\text{Score}(x,y) = \sum_{(h,d) \in A_y} \text{Score}(h,d) + \sum_{(h,s,d) \in A_y} \text{Score}(h,s,d)$$

This takes the form a linear model:

$$\text{Score}(x,y) = \sum_{k=1} \sum_{(h,d) \in A_y} f_k(h,d) \cdot w_k + \sum_{k=1} \sum_{(h,s,d) \in A_y} f_k(h,s,d) \cdot w_k$$

Second-Order Graph-Based Models



Score(x, y) = Score_1 (root, sent) + Score_1 (sent, She) +

(first-order) Score_1 (sent, him) + Score_1 (sent, message) +

Score_1 (sent, by) + Score_1 (sent, .) +

Score_1 (message, a) + Score_1 (by, email) +

(second-order) Score_2 (root, −, sent) + Score_2 (sent, −, She) +

Score_2 (sent, −, him) + Score_2 (sent, him, message) +

Score_2 (sent, message, by) + Score_2 (sent, by, .) +

Score_2 (message, −, by) + Score_2 (by, −, email)

Second-Order Graph-Based Models

- Sibling factors enable more complex feature templates.
- In particular (McDonald, 2006)

  xh-pos – xs-pos – xd-pos

  xs-pos – xd-pos

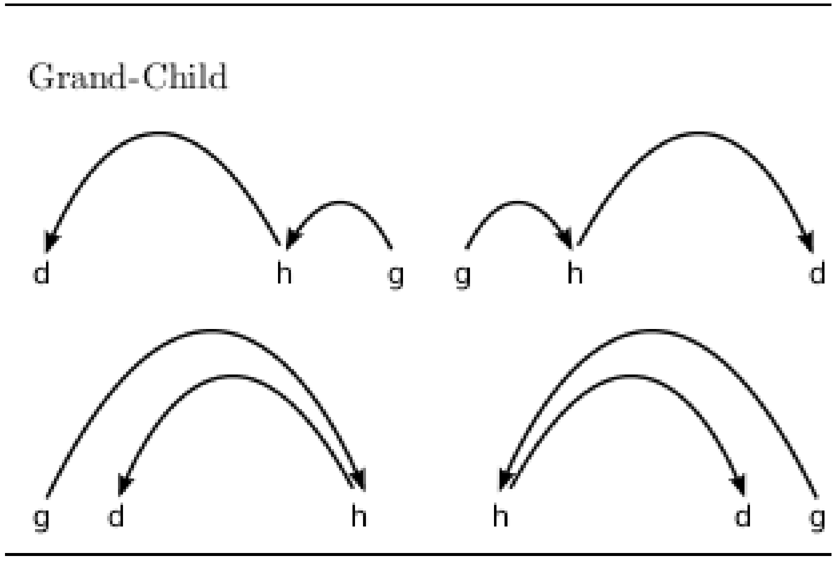
  xs-word – xd-word

  xs-word – xd-pos

  xs-pos – xd-word

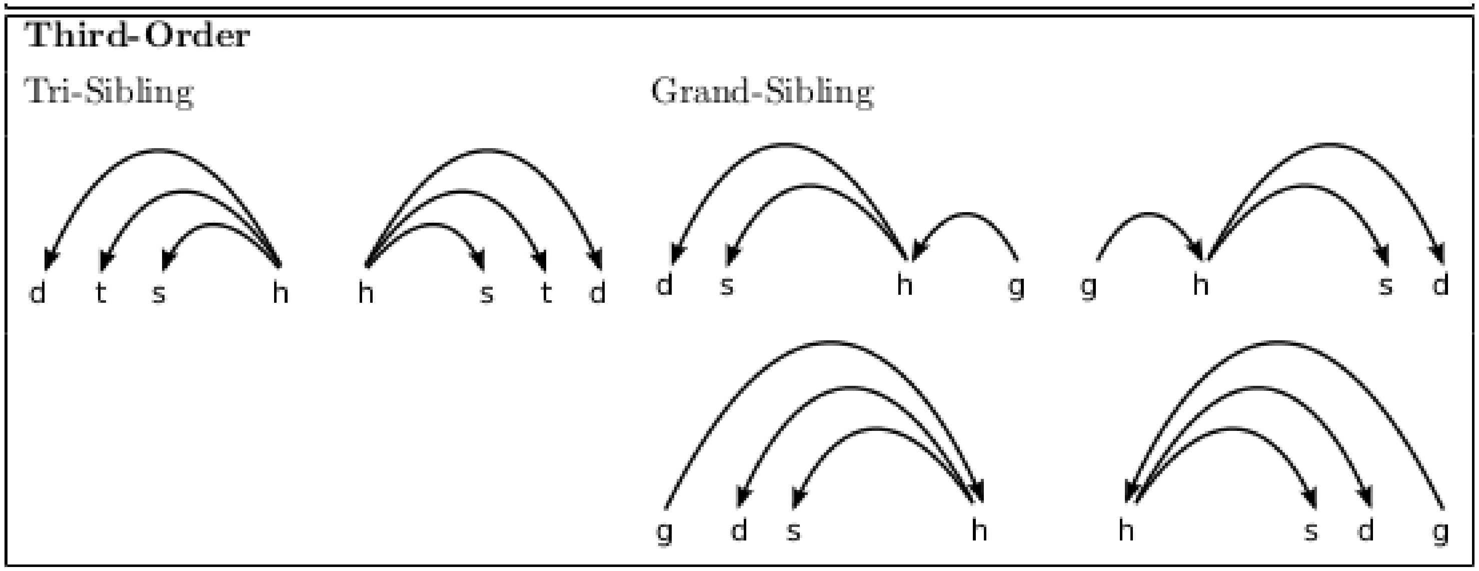
Second-Order Graph-Based Models

- Another type of second-order factor, due to Carreras (2007) (in the reading list), instead extends in the vertical dimension by including a dependent, its head, and the head of the head.
- They are called grand-child factors, and they are scored in the same way.



Third-Order Graph-Based Models

- Koo & Collins (2010) (in the reading list) proposed the use of third-order features, incorporating both tri-sibling and grand-sibling factors.



Third-Order Graph-Based Models

- They produce state-of-the-art accuracy for and they compute the scores as follows:

$$\begin{aligned}\textbf{Score}(x,y) = & \sum_{(h,d) \in A_y} \textbf{Score}_1(h,d) + \\ & \sum_{(h,s,d) \in A_y} \textbf{Score}_{2s}(h,s,d) + \\ & \sum_{(g,h,d) \in A_y} \textbf{Score}_{2g}(g,h,d) + \\ & \sum_{(h,s,t,d) \in A_y} \textbf{Score}_{3s}(h,s,t,d) + \\ & \sum_{(g,h,s,d) \in A_y} \textbf{Score}_{3g}(g,h,s,d) +\end{aligned}$$

Third-Order Graph-Based Models

- They produce state-of-the-art accuracy for and they compute the scores as follows:

$$\begin{aligned} \text{Score}(x,y) = & \sum_{(h,d) \in A_y} \text{Score}_1(h,d) + \\ & \sum_{(h,s,d) \in A_y} \text{Score}_{2s}(h,s,d) + \\ & \sum_{(g,h,d) \in A_y} \text{Score}_{2g}(g,h,d) + \\ & \sum_{(h,s,t,d) \in A_y} \text{Score}_{3s}(h,s,t,d) + \\ & \sum_{(g,h,s,d) \in A_y} \text{Score}_{3g}(g,h,s,d) + \end{aligned}$$

First-order
Arc-factored

Third-Order Graph-Based Models

- They produce state-of-the-art accuracy for and they compute the scores as follows:

$$\begin{aligned}\textbf{Score}(x,y) = &\sum_{(h,d)\in A_y} \textbf{Score}_1(h,d) + \\ &\sum_{(h,s,d)\in A_y} \textbf{Score}_{2s}(h,s,d) + \\ &\sum_{(g,h,d)\in A_y} \textbf{Score}_{2g}(g,h,d) + \\ &\sum_{(h,s,t,d)\in A_y} \textbf{Score}_{3s}(h,s,t,d) + \\ &\sum_{(g,h,s,d)\in A_y} \textbf{Score}_{3g}(g,h,s,d) +\end{aligned}$$

2nd-order sibling McDonald & Pereira

Third-Order Graph-Based Models

- They produce state-of-the-art accuracy for and they compute the scores as follows:

$$
\begin{aligned}
\text{Score}(x,y) = & \sum_{(h,d) \in A_y} \text{Score}_1(h,d) + \\
& \sum_{(h,s,d) \in A_y} \text{Score}_{2s}(h,s,d) + \\
& \sum_{(g,h,d) \in A_y} \text{Score}_{2g}(g,h,d) + \\
& \sum_{(h,s,t,d) \in A_y} \text{Score}_{3s}(h,s,t,d) + \\
& \sum_{(g,h,s,d) \in A_y} \text{Score}_{3g}(g,h,s,d) +
\end{aligned}
$$

2nd order grand-child Carreras. 2007

Third-Order Graph-Based Models

- They produce state-of-the-art accuracy for and they compute the scores as follows:

$$\begin{aligned}\text{Score}(x,y) = &\sum_{(h,d)\in A_y} \text{Score}_1(h,d) + \\ &\sum_{(h,s,d)\in A_y} \text{Score}_{2s}(h,s,d) + \\ &\sum_{(g,h,d)\in A_y} \text{Score}_{2g}(g,h,d) + \\ &\sum_{(h,s,t,d)\in A_y} \text{Score}_{3s}(h,s,t,d) + \\ &\sum_{(g,h,s,d)\in A_y} \text{Score}_{3g}(g,h,s,d) +\end{aligned}$$

3rd order sibling
Koo & Collins

Third-Order Graph-Based Models

- They produce state-of-the-art accuracy for and they compute the scores as follows:

$$\begin{aligned}\mathbf{Score\ (x,y)} = & \textstyle\sum_{(h,d) \in A_y} \mathrm{Score}_1(h,d) + \\ & \textstyle\sum_{(h,s,d) \in A_y} \mathrm{Score}_{2s}(h,s,d) + \\ & \textstyle\sum_{(g,h,d) \in A_y} \mathrm{Score}_{2g}(g,h,d) + \\ & \textstyle\sum_{(h,s,t,d) \in A_y} \mathrm{Score}_{3s}(h,s,t,d) + \\ & \textstyle\sum_{(g,h,s,d) \in A_y} \mathrm{Score}_{3g}(g,h,s,d) +\end{aligned}$$

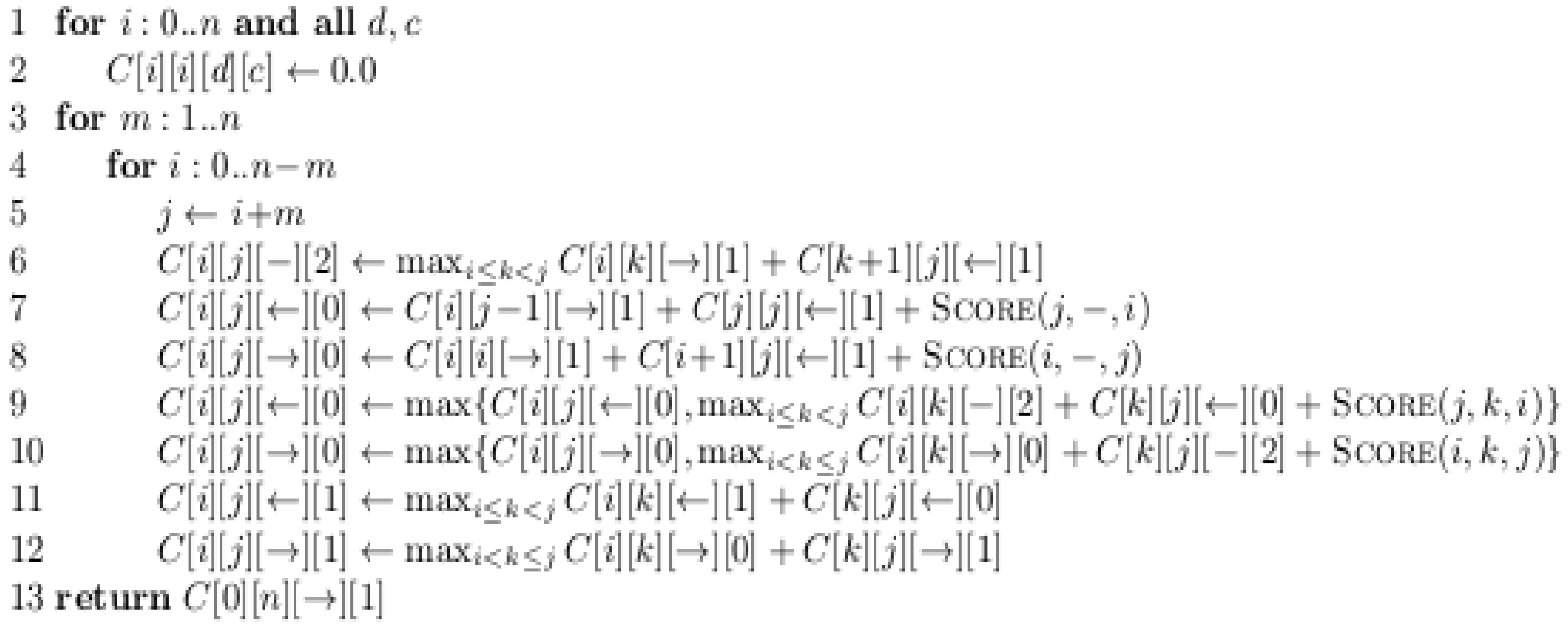
3rd order grand-child Koo & Collins

Projective Parsing for Higher-Order Models

- If we only want to parse projective trees, then decoding for higher-order models can be performed with adaptations of **Eisner's algorithm.**
- The algorithm is still cubic **$O(n^3)$** for sibling second-order factors.
- While we require **$O(n^4)$** for third-oder factors because of an extra loop (over grandparent or the second sibling)

Projective Parsing for Higher-Order Models

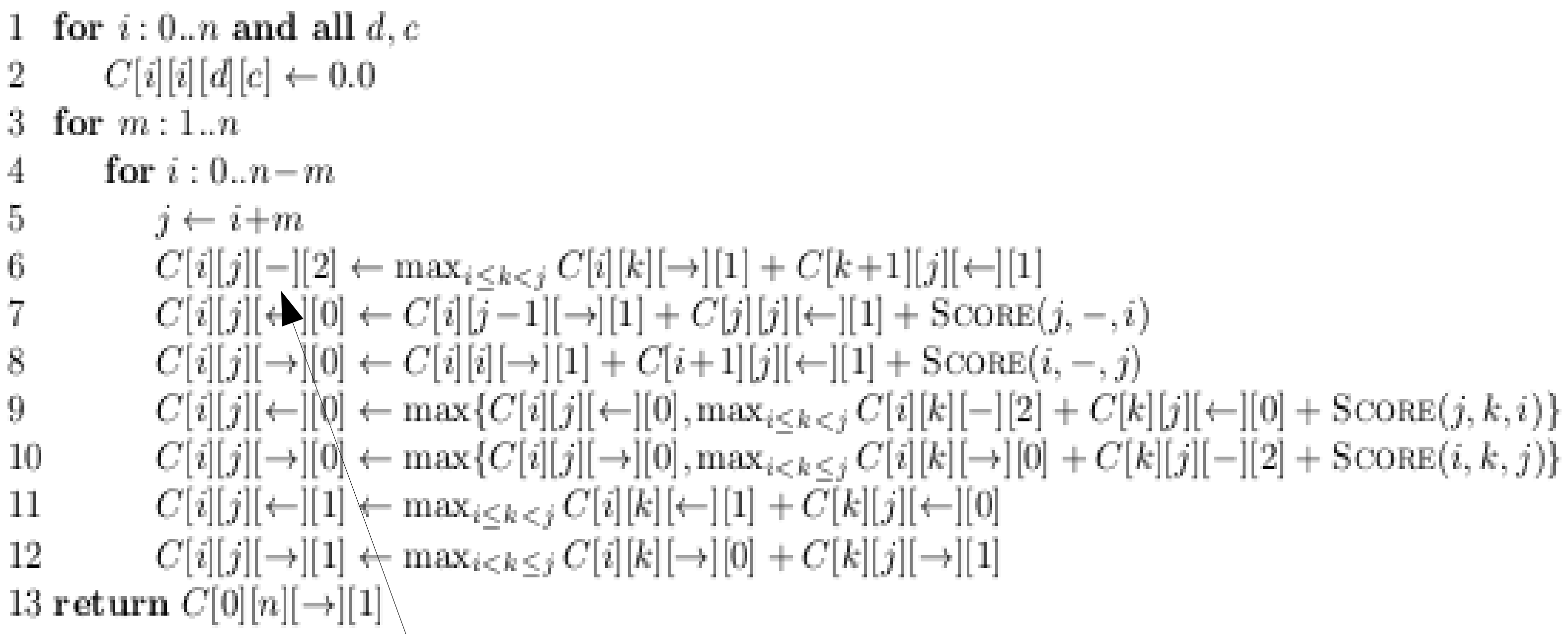
Eisner's algorithm with Second-order sibling factors.



```
**for** $i : 0..n$ **and all** $d, c$
    $C[i][i][d][c] \leftarrow 0.0$
**for** $m : 1..n$
    **for** $i : 0..n-m$
        $j \leftarrow i+m$
        $C[i][j][-][2] \leftarrow \max_{i \leq k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1]$
        $C[i][j][\leftarrow][0] \leftarrow C[i][j-1][\rightarrow][1] + C[j][j][\leftarrow][1] + \textsc{Score}(j, -, i)$
        $C[i][j][\rightarrow][0] \leftarrow C[i][i][\rightarrow][1] + C[i+1][j][\leftarrow][1] + \textsc{Score}(i, -, j)$
        $C[i][j][\leftarrow][0] \leftarrow \max\{C[i][j][\leftarrow][0], \max_{i \leq k < j} C[i][k][-][2] + C[k][j][\leftarrow][0] + \textsc{Score}(j, k, i)\}$
        $C[i][j][\rightarrow][0] \leftarrow \max\{C[i][j][\rightarrow][0], \max_{i < k \leq j} C[i][k][\rightarrow][0] + C[k][j][-][2] + \textsc{Score}(i, k, j)\}$
        $C[i][j][\leftarrow][1] \leftarrow \max_{i \leq k < j} C[i][k][\leftarrow][1] + C[k][j][\leftarrow][0]$
        $C[i][j][\rightarrow][1] \leftarrow \max_{i < k \leq j} C[i][k][\rightarrow][0] + C[k][j][\rightarrow][1]$
**return** $C[0][n][\rightarrow][1]$
```

Projective Parsing for Higher-Order Models

Eisner's algorithm with Second-order sibling factors.



```
for $i: 0..n$ and all $d, c$
    $C[i][i][d][c] \leftarrow 0.0$
for $m: 1..n$
    for $i: 0..n-m$
        $j \leftarrow i+m$
        $C[i][j][-][2] \leftarrow \max_{i \le k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1]$
        $C[i][j][\leftarrow][0] \leftarrow C[i][j-1][\rightarrow][1] + C[j][j][\leftarrow][1] + \textsc{Score}(j, -, i)$
        $C[i][j][\rightarrow][0] \leftarrow C[i][i][\rightarrow][1] + C[i+1][j][\leftarrow][1] + \textsc{Score}(i, -, j)$
        $C[i][j][\leftarrow][0] \leftarrow \max\{C[i][j][\leftarrow][0], \max_{i \le k < j} C[i][k][-][2] + C[k][j][\leftarrow][0] + \textsc{Score}(j, k, i)\}$
        $C[i][j][\rightarrow][0] \leftarrow \max\{C[i][j][\rightarrow][0], \max_{i < k \le j} C[i][k][\rightarrow][0] + C[k][j][-][2] + \textsc{Score}(i, k, j)\}$
        $C[i][j][\leftarrow][1] \leftarrow \max_{i \le k < j} C[i][k][\leftarrow][1] + C[k][j][\leftarrow][0]$
        $C[i][j][\rightarrow][1] \leftarrow \max_{i < k \le j} C[i][k][\rightarrow][0] + C[k][j][\rightarrow][1]$
return $C[0][n][\rightarrow][1]$
```

Sibling item, this is new and it is required. Now we have left, right and sibling.
A sibling item is like an incomplete item in that it consists in two adjacent half-trees (complete items).

Projective Parsing for Higher-Order Models

Eisner's algorithm with Second-order sibling factors.

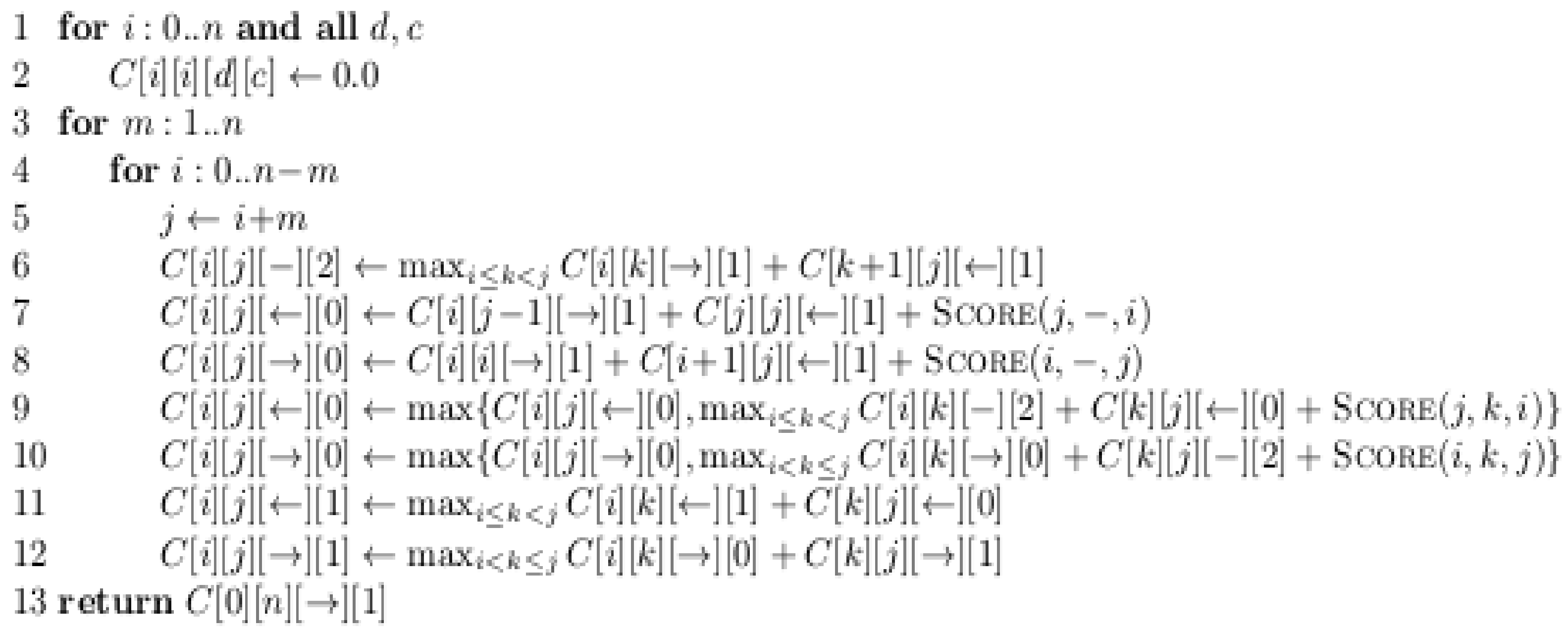
```
1  for $i : 0..n$ and all $d, c$
2      $C[i][i][d][c] \leftarrow 0.0$
3  for $m : 1..n$
4      for $i : 0..n-m$
5          $j \leftarrow i+m$
6          $C[i][j][-][2] \leftarrow \max_{i \leq k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1]$
7          $C[i][j][\leftarrow][0] \leftarrow C[i][j-1][\rightarrow][1] + C[j][j][\leftarrow][1] + \textsc{Score}(j, -, i)$
8          $C[i][j][\rightarrow][0] \leftarrow C[i][i][\rightarrow][1] + C[i+1][j][\leftarrow][1] + \textsc{Score}(i, -, j)$
9          $C[i][j][\leftarrow][0] \leftarrow \max\{C[i][j][\leftarrow][0], \max_{i \leq k < j} C[i][k][-][2] + C[k][j][\leftarrow][0] + \textsc{Score}(j, k, i)\}$
10         $C[i][j][\rightarrow][0] \leftarrow \max\{C[i][j][\rightarrow][0], \max_{i < k \leq j} C[i][k][\rightarrow][0] + C[k][j][-][2] + \textsc{Score}(i, k, j)\}$
11         $C[i][j][\leftarrow][1] \leftarrow \max_{i \leq k < j} C[i][k][\leftarrow][1] + C[k][j][\leftarrow][0]$
12         $C[i][j][\rightarrow][1] \leftarrow \max_{i < k \leq j} C[i][k][\rightarrow][0] + C[k][j][\rightarrow][1]$
13 return $C[0][n][\rightarrow][1]$
```

We use them here. Which is a new way of building incomplete items. It finds the optimal split position k for which we maximize the score combining an incomplete item with a sibling item.

The second max is to compare with lines 7 and 8.

Projective Parsing for Higher-Order Models

Eisner's algorithm with Second-order sibling factors.



```
for $i : 0..n$ and all $d, c$
    $C[i][i][d][c] \leftarrow 0.0$
for $m : 1..n$
    for $i : 0..n-m$
        $j \leftarrow i+m$
        $C[i][j][-][2] \leftarrow \max_{i \leq k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1]$
        $C[i][j][\leftarrow][0] \leftarrow C[i][j-1][\rightarrow][1] + C[j][j][\leftarrow][1] + \textsc{Score}(j, -, i)$
        $C[i][j][\rightarrow][0] \leftarrow C[i][i][\rightarrow][1] + C[i+1][j][\leftarrow][1] + \textsc{Score}(i, -, j)$
        $C[i][j][\leftarrow][0] \leftarrow \max\{C[i][j][\leftarrow][0], \max_{i \leq k < j} C[i][k][-][2] + C[k][j][\leftarrow][0] + \textsc{Score}(j, k, i)\}$
        $C[i][j][\rightarrow][0] \leftarrow \max\{C[i][j][\rightarrow][0], \max_{i < k \leq j} C[i][k][\rightarrow][0] + C[k][j][-][2] + \textsc{Score}(i, k, j)\}$
        $C[i][j][\leftarrow][1] \leftarrow \max_{i \leq k < j} C[i][k][\leftarrow][1] + C[k][j][\leftarrow][0]$
        $C[i][j][\rightarrow][1] \leftarrow \max_{i < k \leq j} C[i][k][\rightarrow][0] + C[k][j][\rightarrow][1]$
return $C[0][n][\rightarrow][1]$
```

Of course, if you have 3rd order features and so on... this grows and become more complicated.

Non-Projective Parsing for Higher-Order Models

- The previous method works with higher order factors and projective trees.
- In slides 25 – 33 we studied non-projective parsing with an arc-factored model = first order factors.
- Non-projective parsing with with higher-order factors is harder and actually NP hard
(McDonald & Satta. 2007)

Non-Projective Parsing for Higher-Order Models

- Recent research has focused on finding good approximate decoding algorithms.
- They are likely to return a good (or even optimal) tree in a high proportion of cases.
- State-of-the-art accuracy has been achieved doing that.

Non-Projective Parsing for Higher-Order Models

- McDonald & Pereira (2006) proposed a two-step approach:
  - 1st step:find the best projective tree (using Eisner’s algorithm).
  - 2nd step: iteratively substitute arcs (for non-projective arcs) in the tree as long as the score of the tree improves.

Non-Projective Parsing for Higher-Order Models

- McDonald & Pereira (2006) proposed a two-step approach:
  - 1st step:find the best projective tree (using Eisner's algorithm).
  - 2nd step: iteratively substitute arcs (for non-projective arcs) in the tree as long as the score of the tree improves.

A variant of the 2nd step is to add new arcs instead of substituting them, and basically output acyclic directed graphs with projective dependencies and some non-projective arcs.

Non-Projective Parsing for Higher-Order Models

- Other approaches are based on
  - Integer Linear Programming
    (Riedel & Clarke, 2006) (Martins et al. 2009)
  - Belief propagation
    (Smith & Eisner, 2008)
  - Dual Decomposition (we will see this in detail)
    (Koo et al. 2010)

Non-Projective Parsing for Higher-Order Models

- Dual decomposition is a technique for optimization of hard problems through joint optimization of simpler problems for which exact algorithms exist.
- Higher-order non-projective dependency parsing is modeled as the joint optimization of two model scores:

$$y^*, z^* = \underset{y \in GEN(X),\ z \in DG(X),\ y=z}{\text{argmax}}\ Score_1(x, y) + Score_2(x, z)$$

Non-Projective Parsing for Higher-Order Models

- Dual decomposition is a technique for optimization of hard problems through joint … for which exact

First model: arc-factored model where the highest-scoring tree is computed following Chu-Liu-Edmonds algorithm.

Any analysis returned is a tree.

- … ndency parsing is modeled as the joint optimization of two model scores:

$$y^*, z^* = \underset{y \in GEN(X),\ z \in DG(X),\ y=z}{\operatorname{argmax}}\ Score_1 (x, y) + Score_2 (x, z)$$

Non-Projective Parsing for Higher-Order Models

- Dual decomposition is a technique for optimization of hard problems through joint [illegible] for which exact [illegible]
- [illegible] ndency parsing is modeled as the joint optimization of two model scores:

First model: arc-factored model where the highest-scoring tree is computed following Chu-Liu-Edmonds algorithm.

Any analysis returned is a tree.

$$y^*, z^* = \underset{y \in \mathrm{GEN}(X),\ z \in \mathrm{DG}(X),\ y=z}{\operatorname{argmax}} \mathrm{Score}_1 (x, y) + \mathrm{Score}_2 (x, z)$$

Note that it can be Non-projective

Non-Projective Parsing for Higher-Order Models

- Dual decomposition is a technique for optimization of hard problems through joint

Second model: higher-order model
(using second-and third-order factors),
where the highest scoring tree cannot be computed
efficiently but where we can use simple head automata
to find the highest scoring set of dependents for each word,
which gives us a dependency graph $z \in DG(X)$

- is modeled as the joint optimization of two model scores:

$$y^*, z^* = \underset{y \in GEN(X),\ z \in DG(X),\ y=z}{\operatorname{argmax}} Score_1(x, y) + Score_2(x, z)$$

All the graphs in which we do not impose the tree constraint
DG(X) contains GEN(x)

Non-Projective Parsing for Higher-Order Models

- Dual decomposition is a technique for

By constraining y and z to be the same, we get an analysis that it is a tree (using Score$_1$ (x, y)) and
is likely to be good (using higher-order factors inherent in Score$_2$ (x, z)).

- is modeled as the joint optimization of two model scores:

$$y^*, z^* = \underset{y \in GEN(X),\ z \in DG(X),\ y=z}{\operatorname{argmax}} Score_1 (x, y) + Score_2 (x, z)$$

All the graphs in which we do not impose the tree constraint

Non-Projective Parsing for Higher-Order Models

- The dual decomposition algorithm first calculates y* and z* separately.
  - If y*=z* then we have found the optimal dependency tree.
  - If not, we impose penalty weights on arcs that are in y* but not in z* (and the other way around).
  - We want y* and z* to be the same, so we iterate until we find it.