Natural Language Dependency Parsing

SPFLODD

19 February 2017

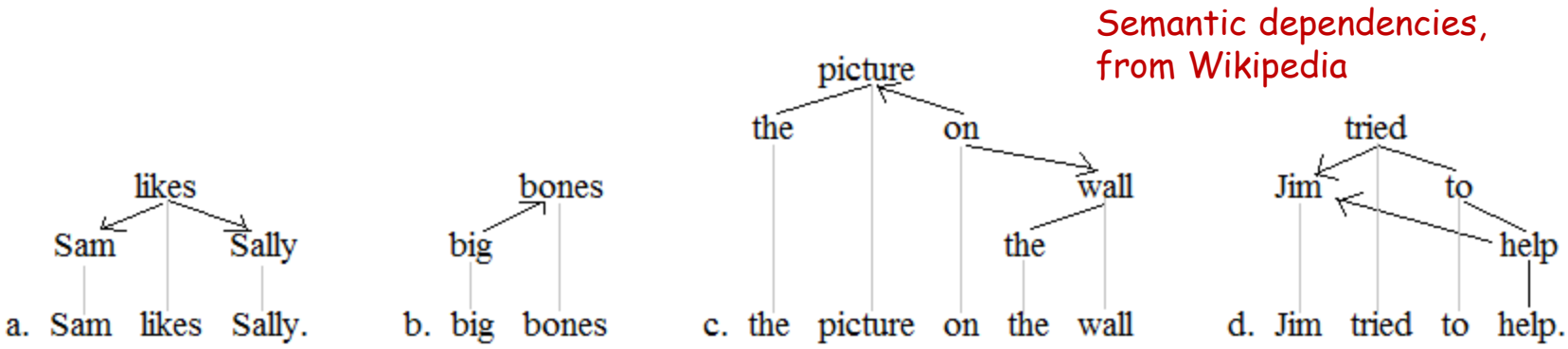
Grammar Structure

- The grammars we've seen so far are instances of *phrase structure* grammars
  - Deal with arrangement of *contiguous phrases* within the grammar
  - Not specific to natural language
    - Works for many other languages, e.g. computer programs, the language of irrationals
  - Natural language:
    - Assumption: Meaning is held in contiguous phrases
    - Phrase structure does not always reveal the entire story
      - Particularly in languages with free word order
  - Modern theories largely trace back to Chomsky..

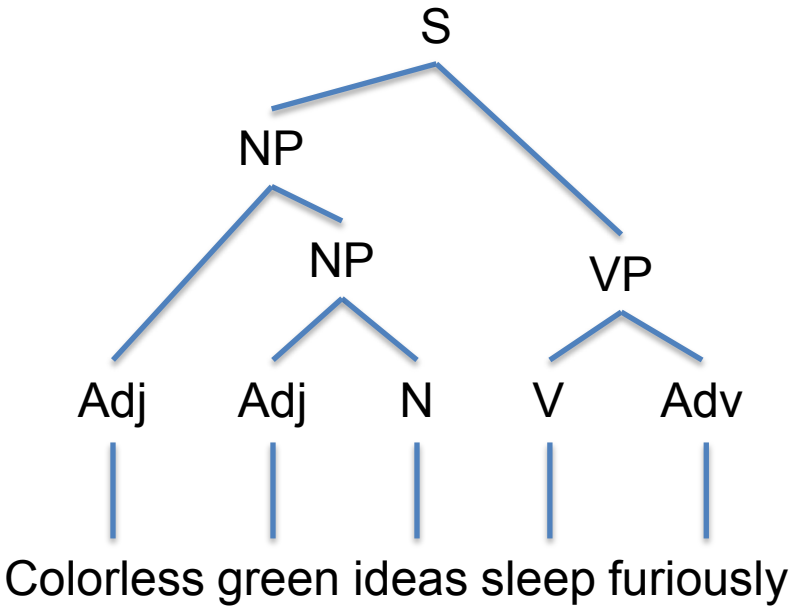
- An alternate approach looks at direct "dependencies" between words
  - Specific to NL..

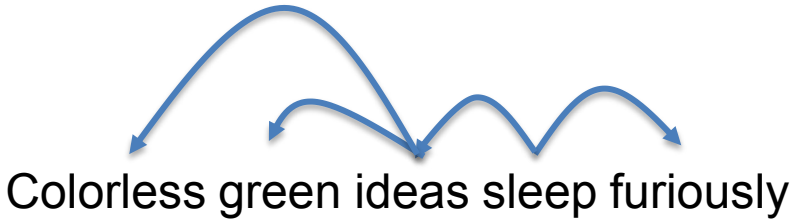
Dependency

- Dependency theory is built on the hypothesis that words are governed by one another directly, rather than through intermediate non-terminal entities
- Identifying these connections helps us interpret a sentence
- "Governance" can be identified based on different factors
  - Syntax, morphology, phonetics, semantics…



Dependency vs Phrase Structure





- But how do we decide which word connects to which?
  - Coming up..

A brief history

- The existence of dependencies between grammatical units was possibly first described by Pāṇini ca 7th century BC
  - I.e. at least as old as constituency theory
- The first grammarian to use the (equivalent of the) word "dependency" was by Ibn Maḍāʾ, in 12th-century Córdoba (Al Andalus, Spain)
- Mentions in the nineteenth century, but serious revival only in the mid-late twentieth century

Dependency

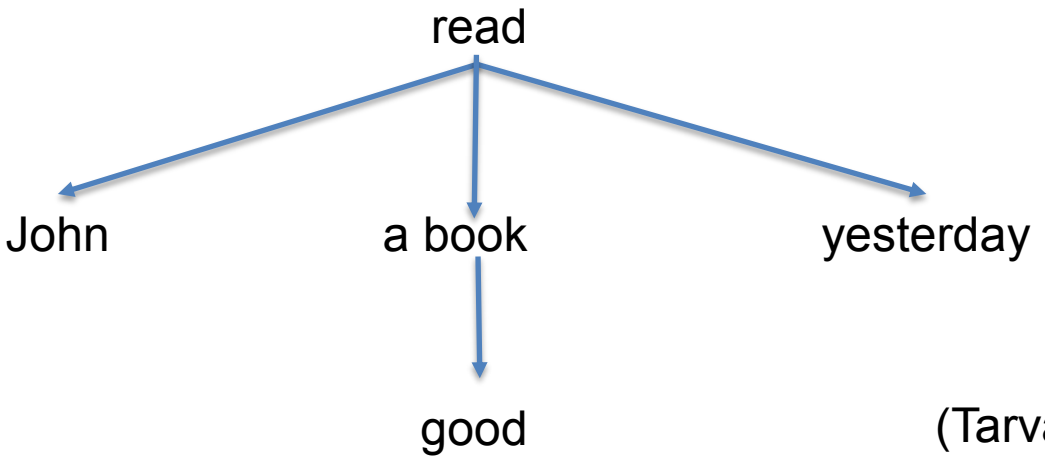
- Modern dependency theory traces back to the work of Tesnière
  - Lucien Tesnière. Éléments de syntaxe structural. Klincksieck, Paris 1959 (now available in English for only USD 175)
- Dependency is the notion that linguistic units (words) are connected to one another by directed links.
- The verb is taken to be the structural center of the clause structure
  - All other words are connected directly or indirectly to the verb through directed links called "*dependencies*"
    - Tarvainen, Kalevi. *Depedenssikielioppi*. Helsinki:1977
    - Matthews, Peter. *Syntax*. Cambridge: Cambridge University Press, 1981

Dependency structure

- Specifically, every word is assumed to be governed by *one* other word in the sentence
  - Though each word may govern multiple words
  - "Parent-child" relationship
  - I.e. *tree* structure

Dependency: hierarchy

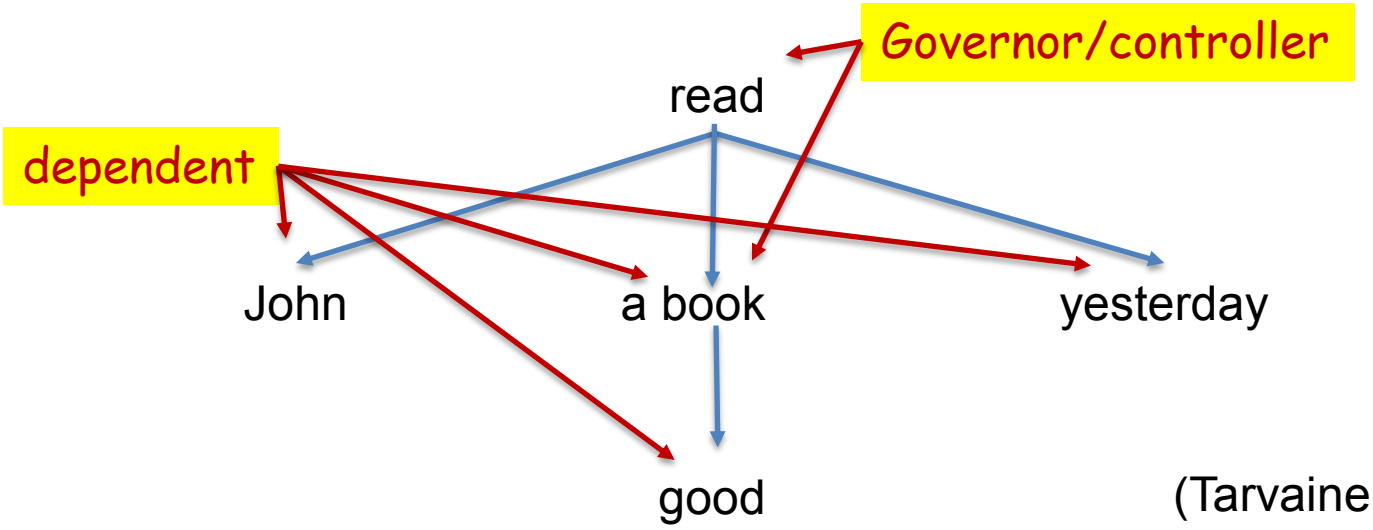
- A sentence is described as a hierarchical structure, where some parts of the sentence have a higher place than others
  - The verb has the highest place



(Tarvainen, ‘87)

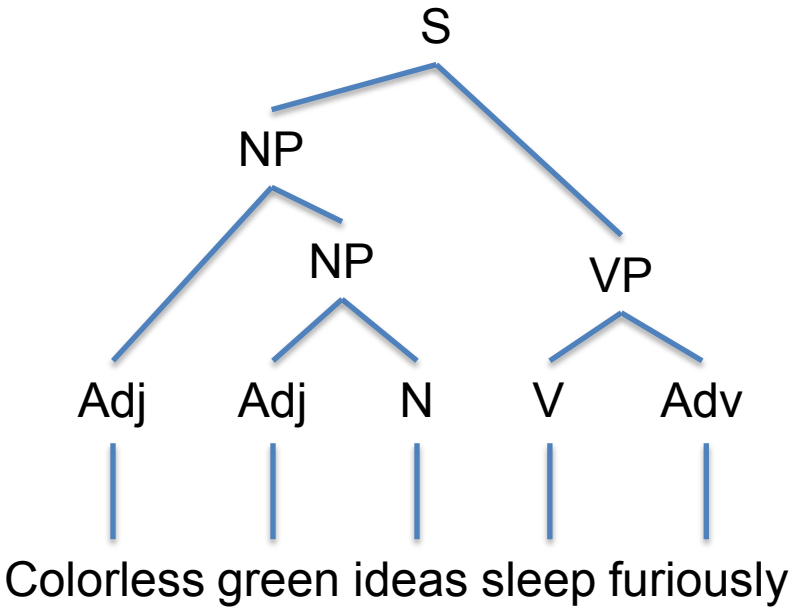
Dependency: hierarchy

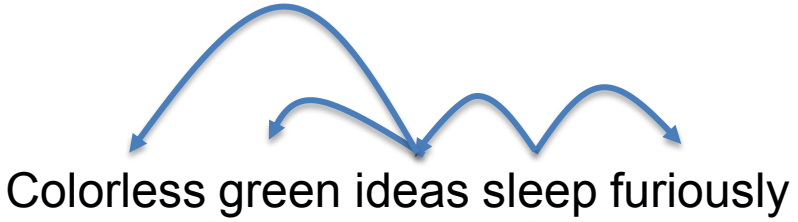
- A sentence is described as a hierarchical structure, where some parts of the sentence have a higher place than others
  - The verb has the highest place



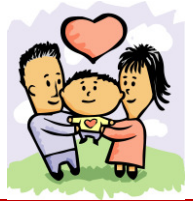
(Tarvainen, '87)

The hidden story

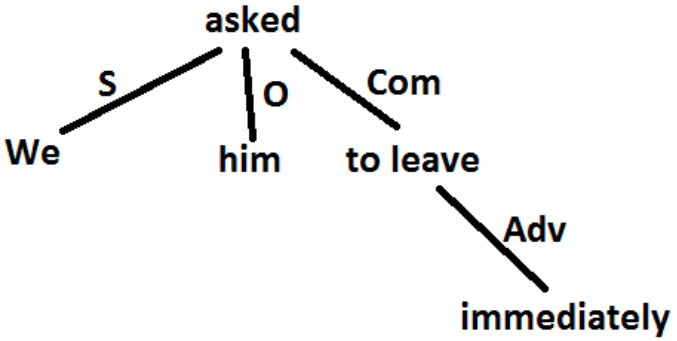


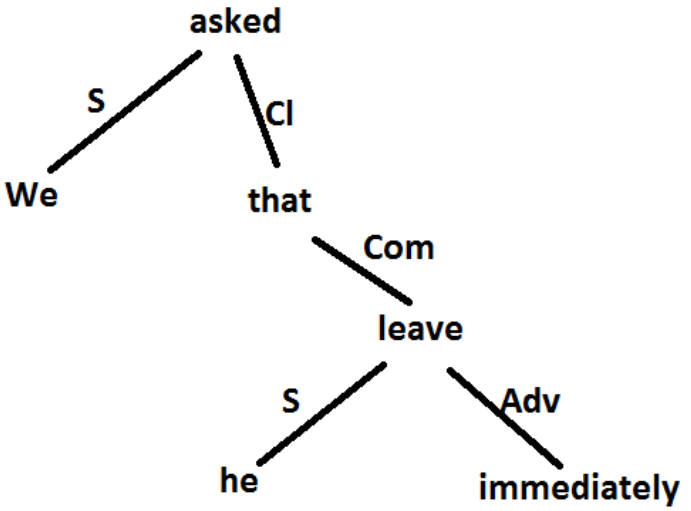


- Modern linguistic theories (both Chomsky and Tesnière) assume parent-child relationships
  - Single parent, (possibly) multiple children
  - It is this assumption that makes computational modelling tractable
    - Like all assumptions, not universally valid



Dependency

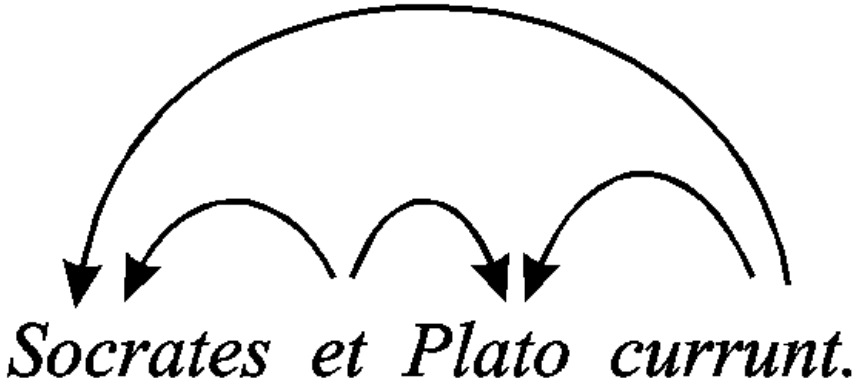




https://www.linkedin.com/pulse/parent-child-principle-grammar-wei-li

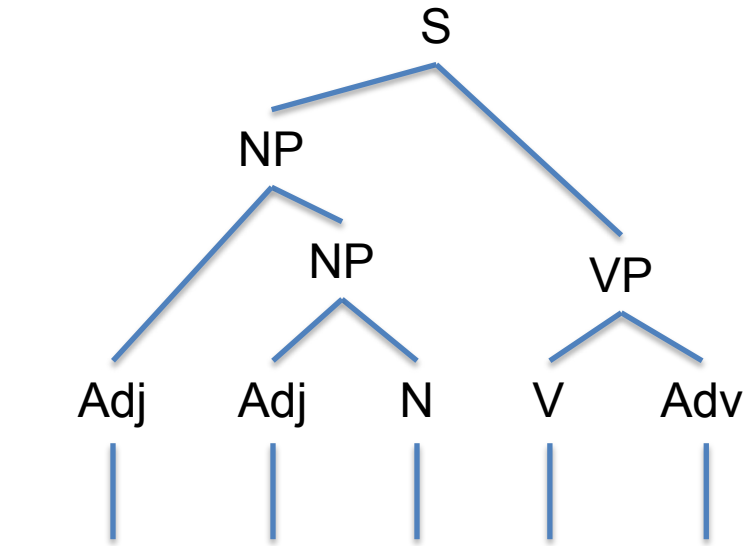
- Can change without changing meaning

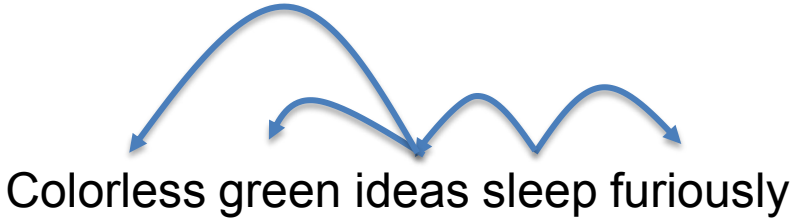
Even conjugation is a problem



- Even simple things like conjugation can cause problems
  - Example from Radulphus Brito, ca 1300
    - Brito was one of the “modistae”, a group of grammarians in the late 1200s who also tried to formalize language
    - They employed dependency structure, but did not separate semantics from syntax, leading to odd structures

So: why dependency?



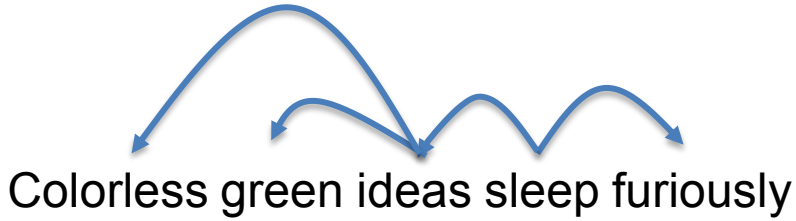


- Dependency trees are much smaller than phrase-structure trees
  - Easier to represent
- If appropriately plotted, can capture information that most (tractable) phrase structure grammars cannot
  - E.g. for non-configurational languages..

Equivalence

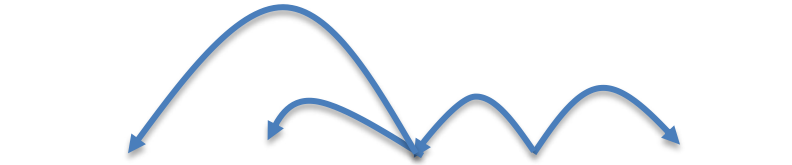
- "In general, it can be shown that for any *dependency grammar* there is a phrase structure grammar which will generate an identical set of sentences; likewise for any phrase structure grammar (or any PSF limited to the form of rule which we have illustrated) the same set of sentences can be generated by a dependency grammar. In that sense the two are said to be weakly equivalent"
  - From "Syntax", Peter Hugoe Matthews, Cambridge University Press, 1981
- Does not mean the trees will be identical
  - If they were, the two would be "strongly equivalent"
  - They are *not* strongly equivalent
    - Descriptions you get for dependency grammars are not all derivable from PS grammars and vice versa

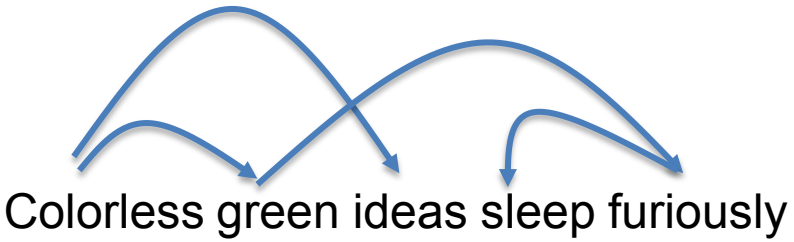
The problem

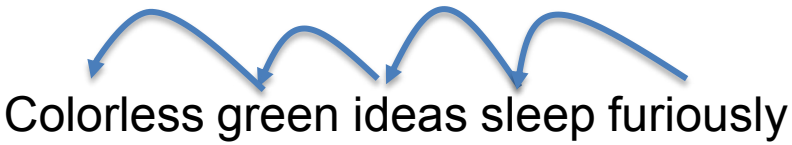


- Determining a meaningful dependency structure in a sentence helps us interpret it
  - Assumption: structure is somehow related to meaning
- Problem: How do we determine the dependency structure?

The problem



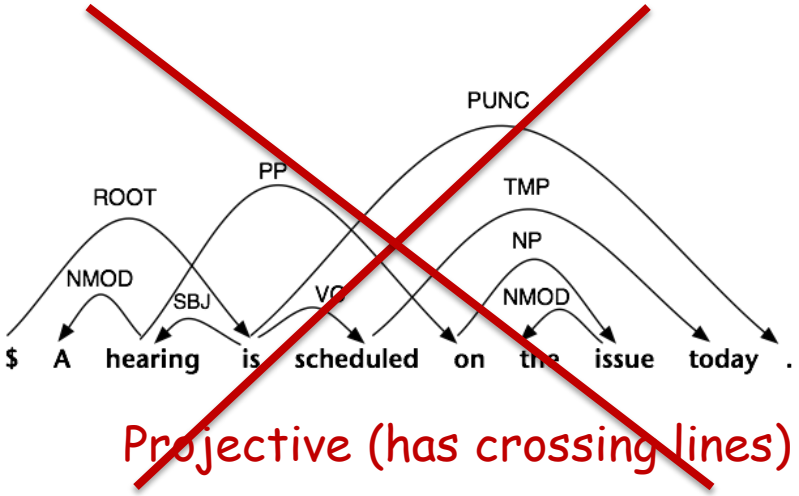




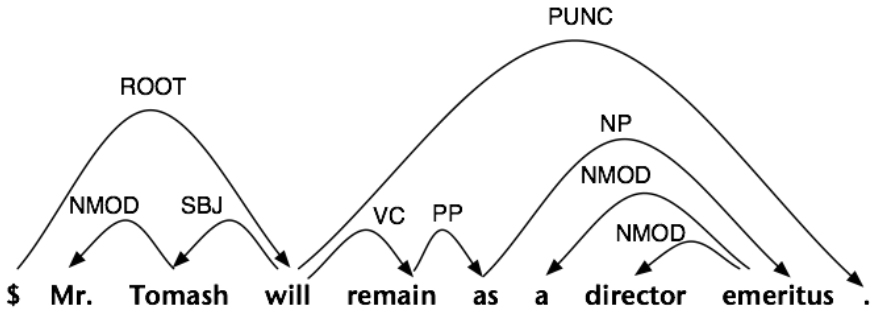
- An exponentially large number of ways of drawing the tree
  - How do we choose the right one
  - How do we assign *probabilities* for the various options?

Finding the dependency tree..

- AKA dependency parsing
- Define a structure to the tree
  - E.g. Exclude some options



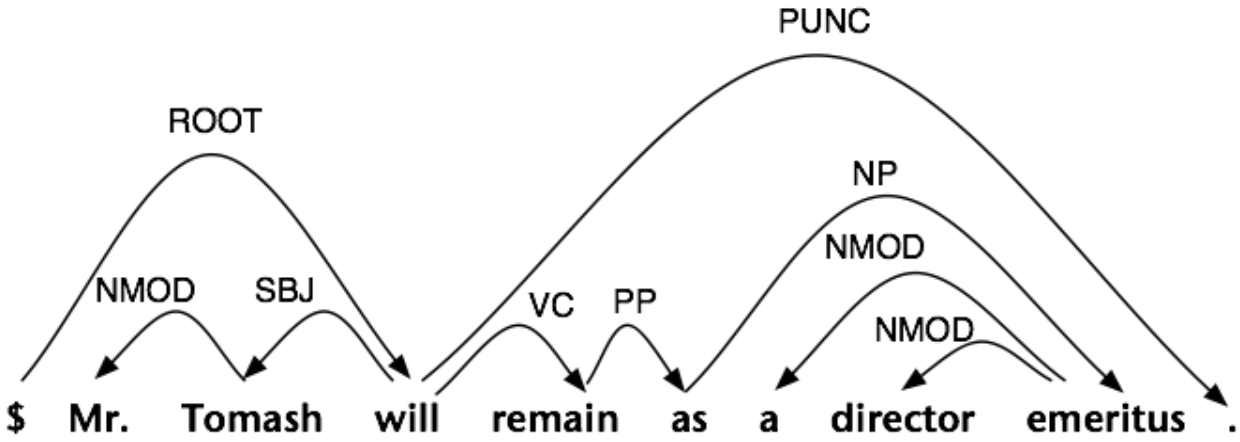
Projective (has crossing lines)



Projective (no crossing lines)

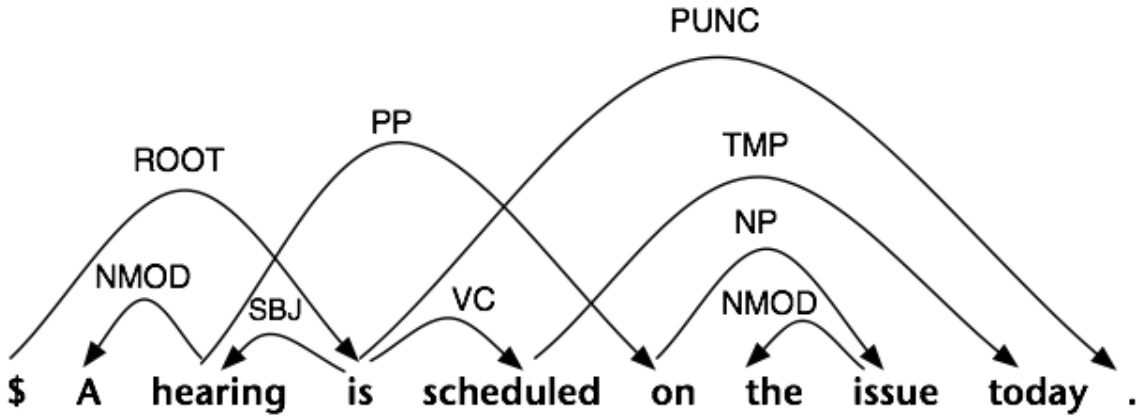
Dependencies and Context-Freeness

- **Projective** dependency trees are ones where edges don't cross
- Projective dependency parsing means searching only for projective trees
- English is mostly projective...



Dependencies and Context-Freeness

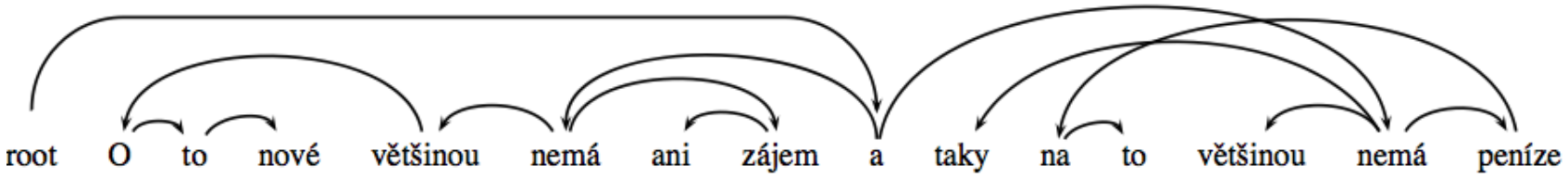
- But not entirely!



- Dependencies constructed through simple means from the Penn treebank will be projective.
  - Parses follow a CFG

Dependencies and Context-Freeness

- Other languages are arguably less projective



- **Projective** dependency grammars generate **context-free** languages
- **Non-projective** dependency grammars can generate **context-sensitive** languages

Finding the dependency tree..

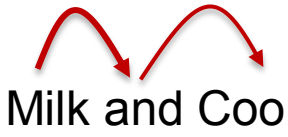
- AKA dependency parsing
- Define a structure to the tree
  - Exclude some options
- Define rules of dependency
  - What kinds of dependencies are possible
    - Or what makes some dependencies more likely than others
  - AKA a dependency grammar!
  - Problem: Need to define potential dependency between every word and every word!
- The manner in which we define these dependencies will determine the tree we will find
  - Learnable from human annotation..

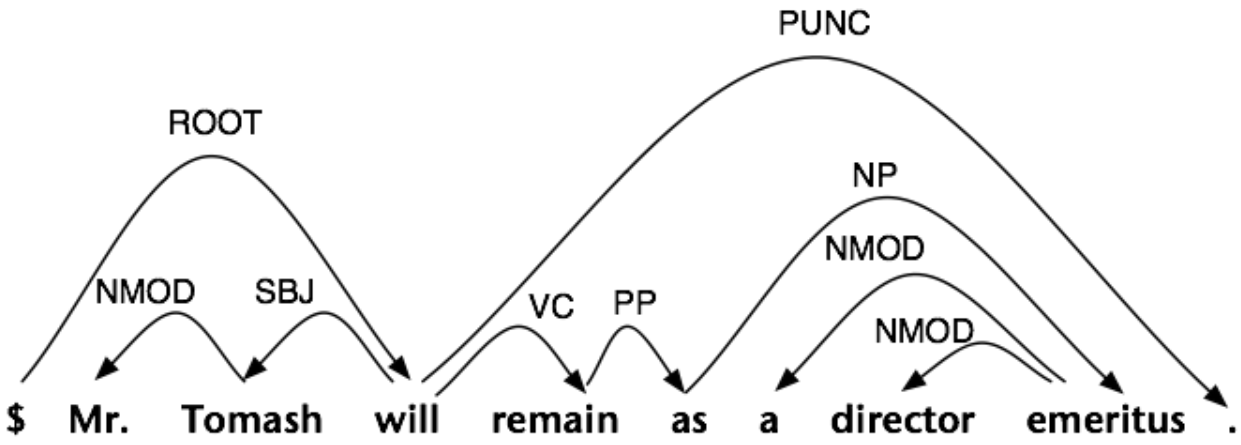
Typical criteria for dependency

- W depends on head H in a structure S if:
  - H determines syntactic category of S
    - And could even replace S
  - H is mandatory, W is optional
  - H determines W and decides if it is mandatory
  - The form of W depends on H
  - The position of W is with reference to H
  - W modifies H semantically, and H assigns semantic to S
- Note: dependency determined not just by the words, but by the structures they are connected to!

- These (and other criteria) can be codified into a model
  - For now we will assume that a simplified model…

Some arbitrary rules





- Conjugation is linear
- Periods attach to the verb

Projective Dependency Parsing

- Major assumption: edge-factored model

$$p(\tau, w_1^n) \propto \prod_{i=1}^{n} \phi(w_1^n, \tau(w_i))$$

- Carroll and Charniak (1992) described a PCFG that has this property
- Eisner (1996) described several stochastic models for generating projective trees like this
- This is a linear model with a certain kind of feature locality
  - Will not go into actual features that have been proposed

Projective Dependency Parsing

- Major assumption: edge-factored model

$$S\left(\tau(w_1^n)\right) = \sum_{e \in \tau} S(e, \tau)$$

- A common form of edge-factored model scores:

$$S(e, \tau) = W^T \Phi(e, \tau)$$

  - $\Phi(e, \tau)$ is a vector of *features* derived from the edge and the tree it resides in
    - Not dealing with this issue just yet, just assume the scores can be computed

- Find the tree with the highest $S\left(\tau(w_1^n)\right)$

Projective Dependency Parsing

$$S\left(\tau(w_1^n)\right) = S(e', \tau) + \sum_{e \in \tau \setminus e'} S(e, \tau)$$

- Removing an edge *splits* the tree into two
  - Property of a tree
- For our dependency graph, every edge connects two words and can be written as: $e = (w_i, w_j)$
- So (since the cost of edges *only* depends on the trees they reside in)

$$S\left(\tau(w_1^n)\right) = S\left(w_i, w_j, \tau\right) + S\left(\tau(w_i)\right) + S\left(\tau\left(w_j\right)\right)$$

  - $S\left(\tau(w_i)\right)$ is the score of the subtree that includes w_i
- If $i = j$, $S\left(\tau(w_1^n)\right) = S(\tau^+(w_i)) + S\left(\tau^-\left(w_j\right)\right)$
  - $\tau^+(w_i)$ is the tree for which w_i is a leaf
  - $\tau^-\left(w_j\right)$ is the tree for which w_j is a head

Projective Dependency Parsing

$$S\left(\tau(w_1^n)\right) = S(e', \tau) + \sum_{e \in \tau \setminus e'} S(e, \tau)$$

- Removing an edge *splits* the tree into two
  - Property of a tree
- For our dependency graph, every edge connects two words and can be written as: $e = (w_i, w_j)$
- So (since the cost of edges *only* depends on the trees they reside in)

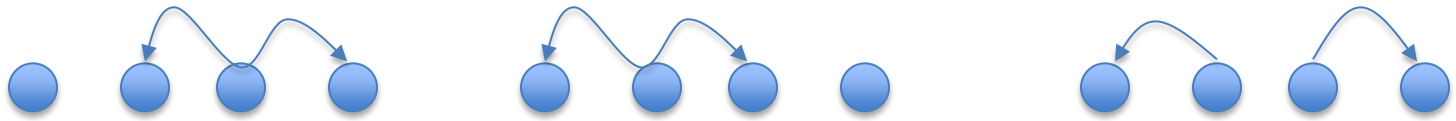
$$S\left(\tau(w_1^n)\right) = S\left(w_i, w_j, \tau\right) + S\left(\tau(w_i)\right) + S\left(\tau\left(w_j\right)\right)$$

  - $S\left(\tau(w_i)\right)$ is the score of the subtree that includes w_i
- If $i = j, S\left(\tau(w_1^n)\right) = S(\tau^+(w_i)) + S\left(\tau^-\left(w_j\right)\right)$
  - $\tau^+(w_i)$ is the tree for which w_i is a leaf
  - $\tau^-\left(w_j\right)$ is the tree for which w_j is a head

This is sufficient to build a DP algorithm!

Basic idea

$$S\left(\tau\left(w_1^n\right)\right) = S\left(w_i, w_j, \tau\right) + S\left(\tau\left(w_i\right)\right) + S\left(\tau\left(w_j\right)\right)$$



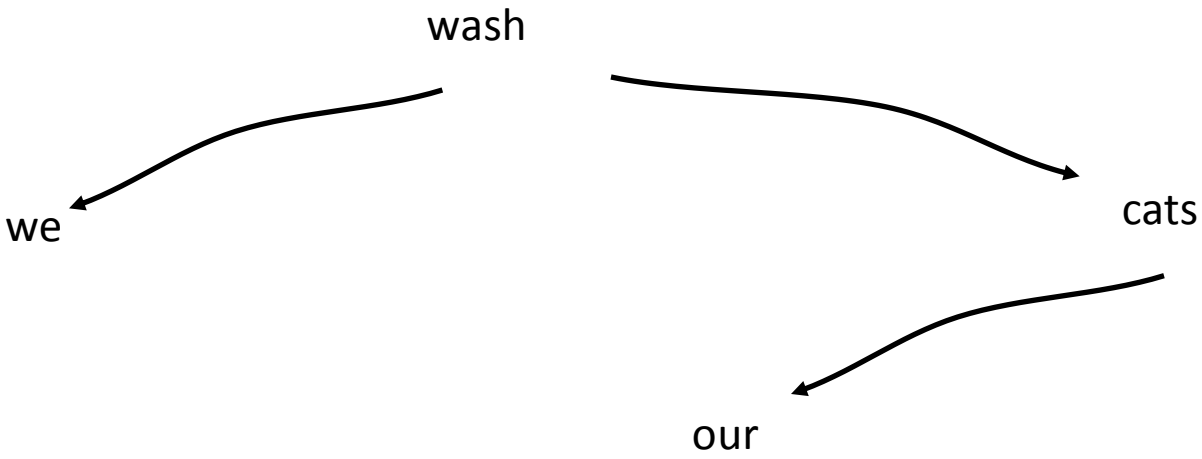
- Given many ways of partitioning a string into two subtrees, the merged tree that gives you the best combined score is the best overall tree
  - We build the DP on this idea

Dependency Grammar

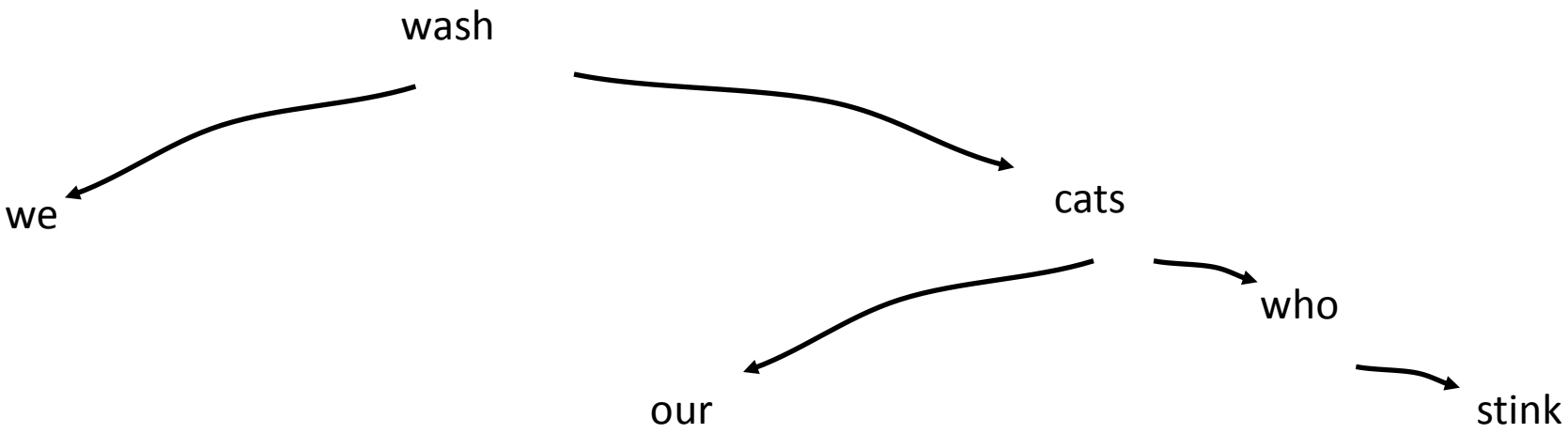
- A variety of theories and formalisms
- Focus on relationship between **words** and their syntactic relationships
- Correlates with study of languages that have free(r) word order (e.g., Czech)
- Lexicalization is central, phrases secondary

- We will talk about **bare bones** dependency trees (Eisner, 1996), then consider adding dependency labels

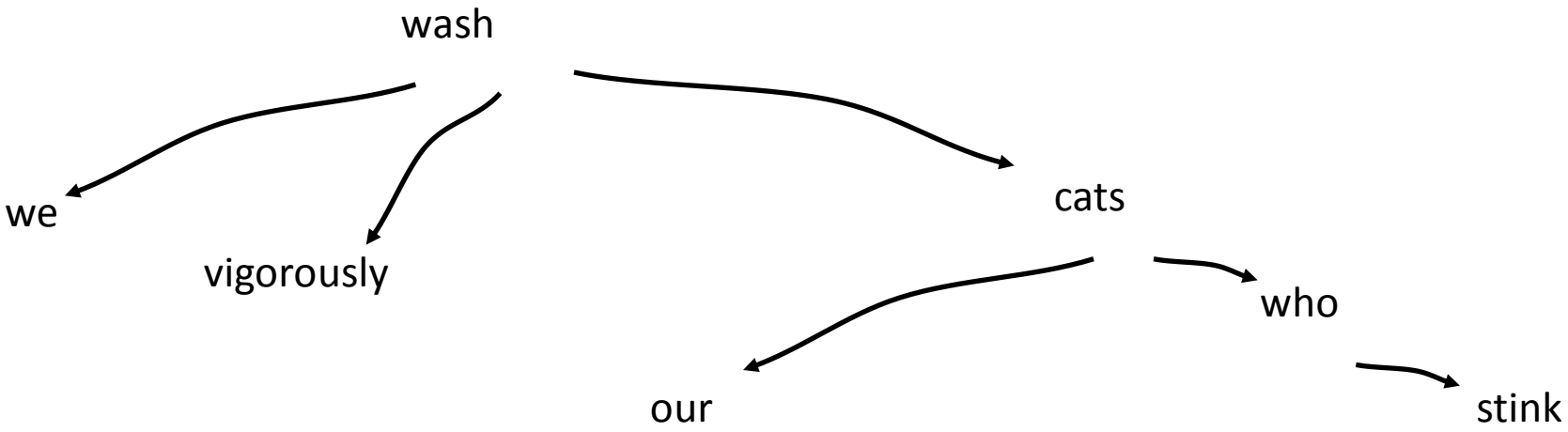
Bare Bones Dependency Parse



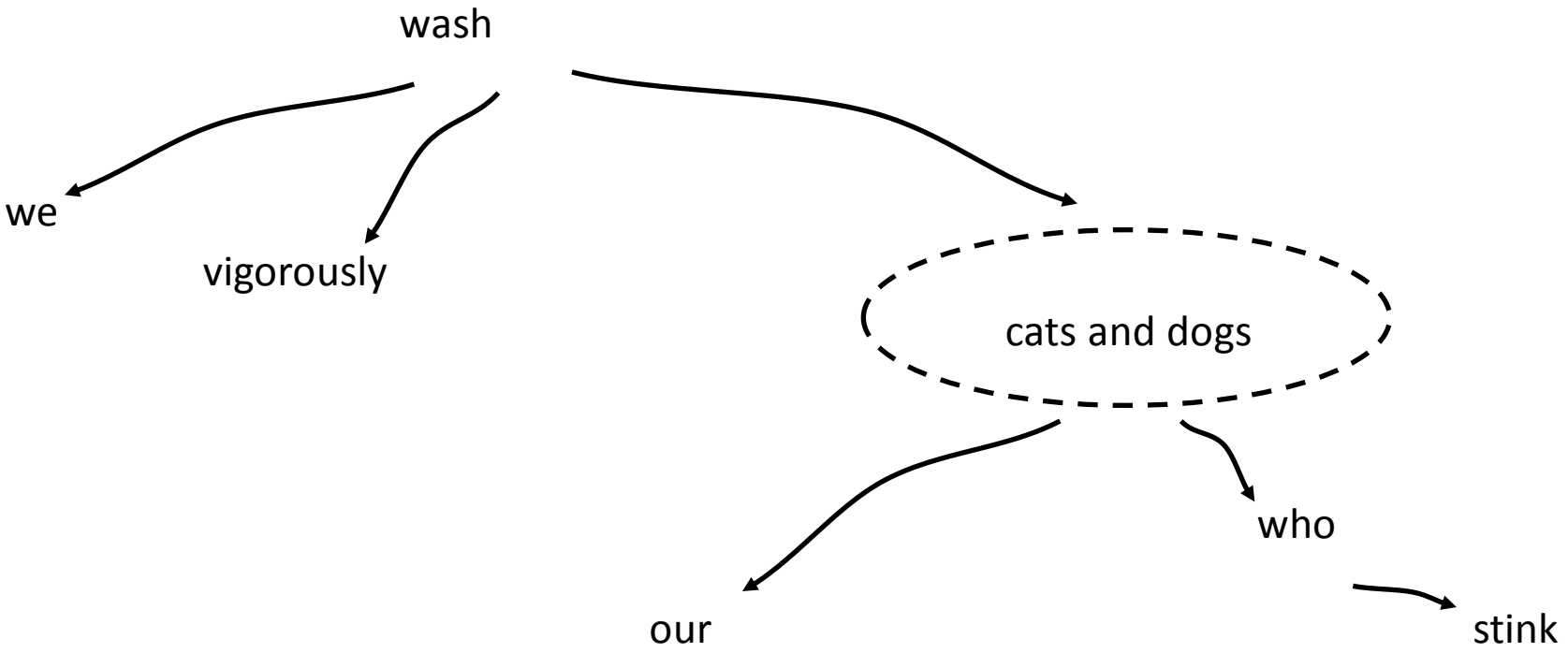
Bare Bones Dependency Parse



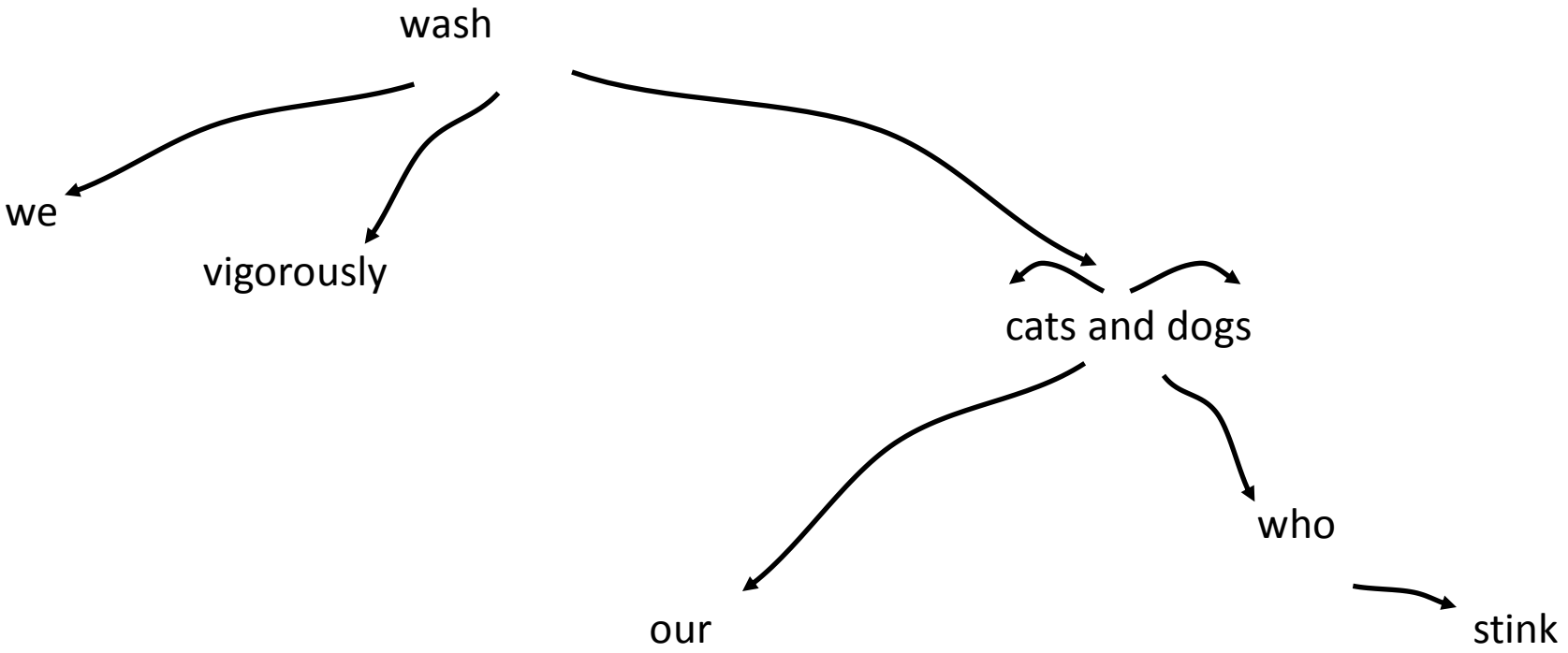
Bare Bones Dependency Parse



Bare Bones Dependency Parse



Bare Bones Dependency Parse



Graph-based vs. Transition-based

- All models above optimize a global score and resort to local features
  - These are sometimes known as **graph-based models.**
- Just like in the phrase-structure/constituent world, there are also approaches that use shift-reduce algorithms.
- With good statistical learning methods, you can get very high performance using **greedy** search without back-tracking!
- "Local decisions, global features"
  - These are known as **transition-based models**; they reduce a structured problem to a lot of classification decisions, kind of like MEMMs.
  - See work by J. Nivre.

Algorithms != Models

- As in HMMs, PCFGs, etc., the algorithms we need depend on the independence assumptions, **not** on the specific formulation of the statistical scores.
- We assume, from here on, that the scores are factored by dependency tree edges.

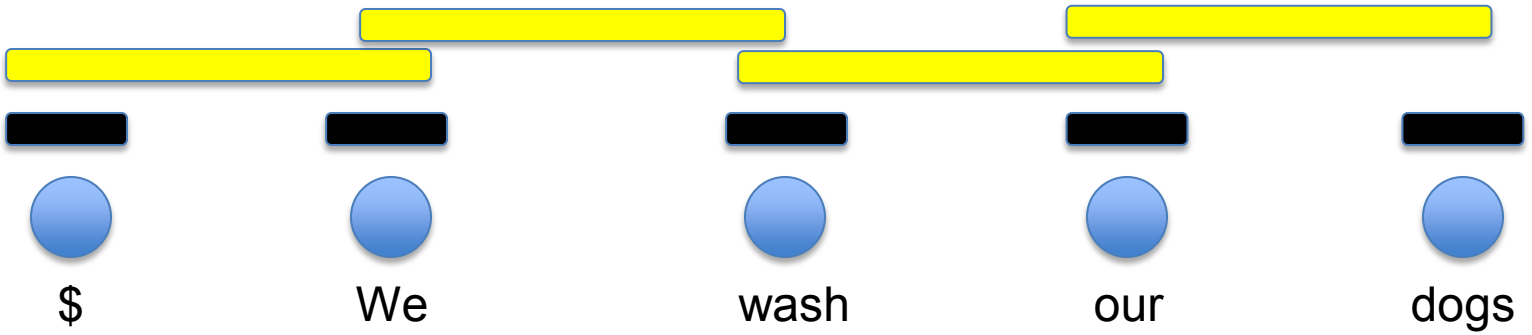
$$s(\tau) = \sum_{i=1}^{n} s(w_1^n, \tau(w_i))$$

- Projective algorithm (Eisner, 1996)
- Non-projective algorithm (McDonald et al., 2005)

The naive algorithm

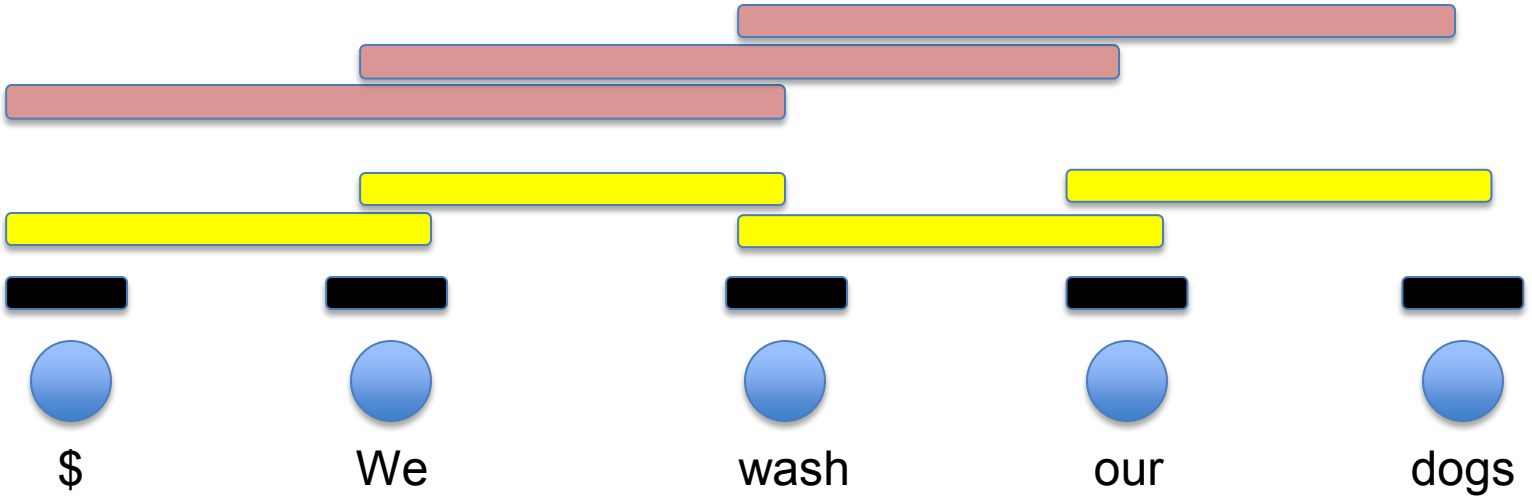
- For each possible “span” (length of string)
  - Starting from each position
    - Compute and save all possible trees (with different headwords) from lower-span trees

The naive algorithm



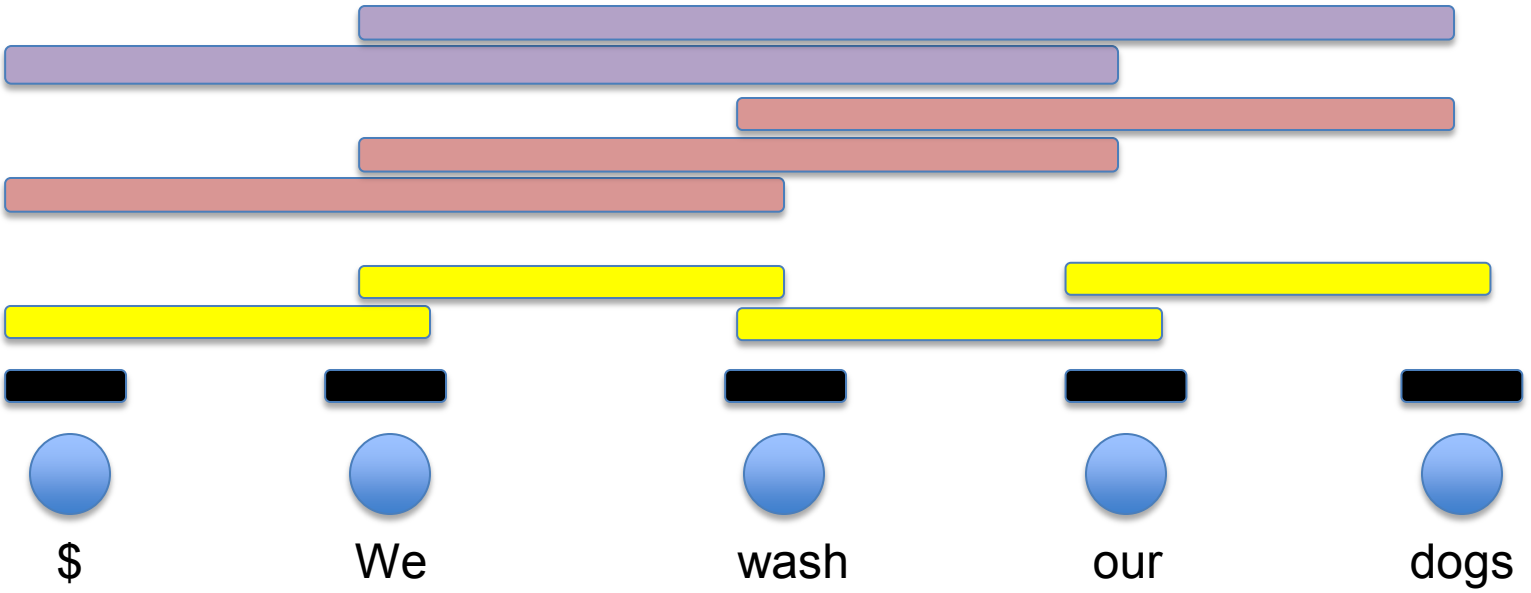
- For each possible "span" (length of string)
  - Starting from each position
    - Compute and save all possible trees (with different headwords) from lower-span trees

The naive algorithm



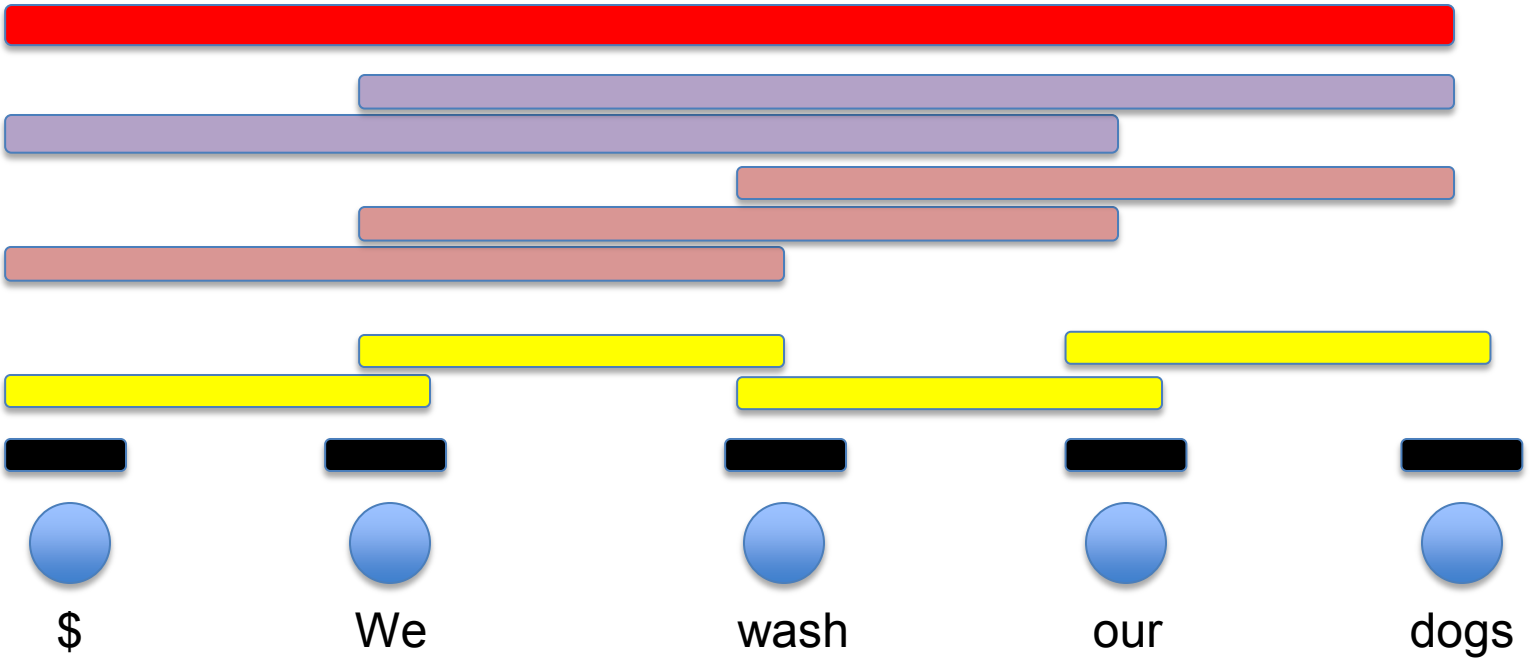
- For each possible "span" (length of string)
  - Starting from each position
    - Compute and save all possible trees (with different headwords) from lower-span trees

The naive algorithm



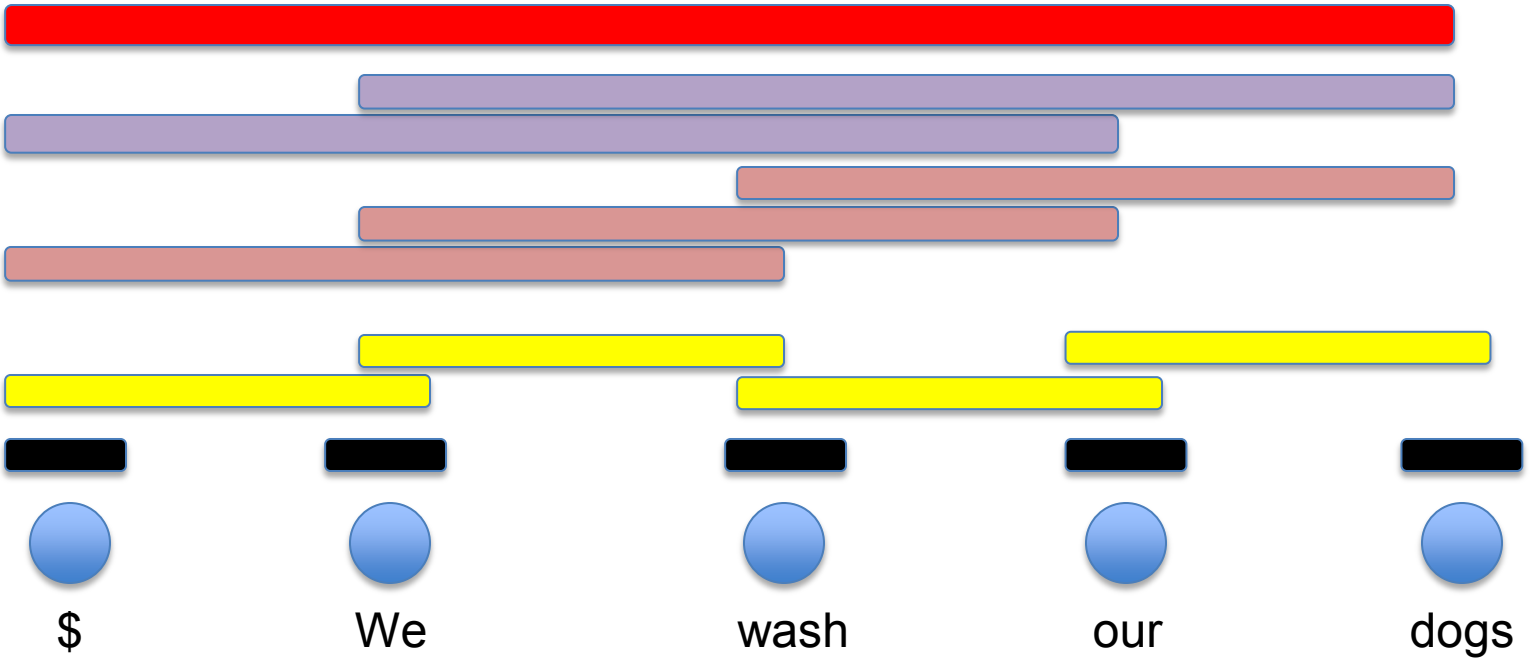
- For each possible "span" (length of string)
  - Starting from each position
    - Compute and save all possible trees (with different headwords) from lower-span trees

The naive algorithm



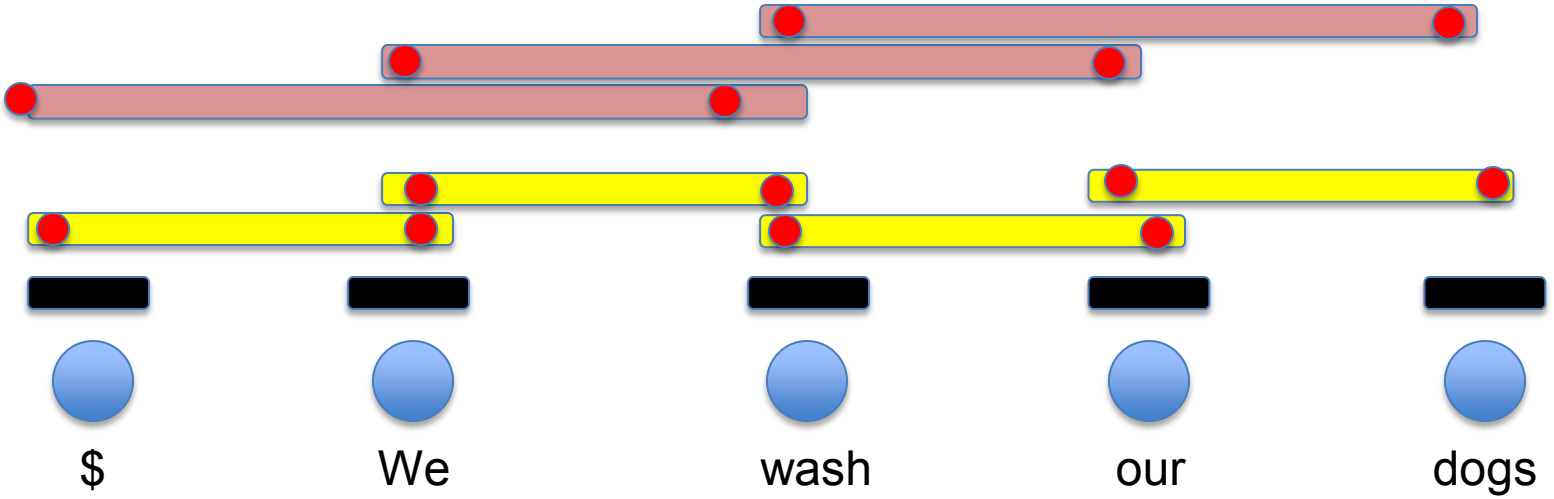
- For each possible "span" (length of string)
  - Starting from each position
    - Compute and save all possible trees (with different headwords) from lower-span trees

The naive algorithm



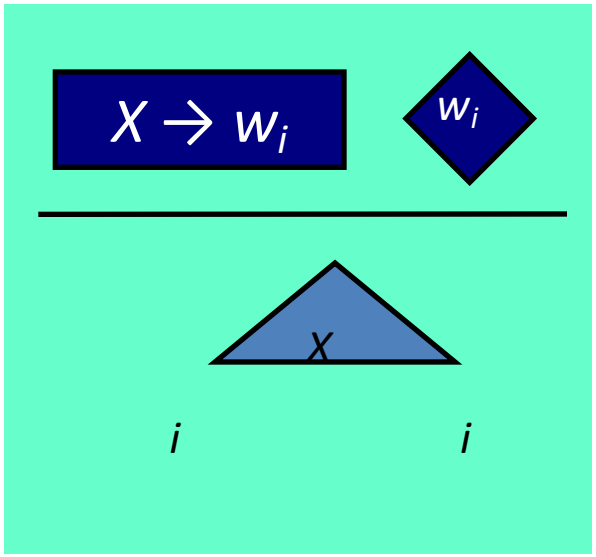
- Each span of K words has K possible head words
  - Representing K trees
  - Building the next-level span trees must evaluate all K of these for composition
- Overall cost of algorithm $O(N^5)$

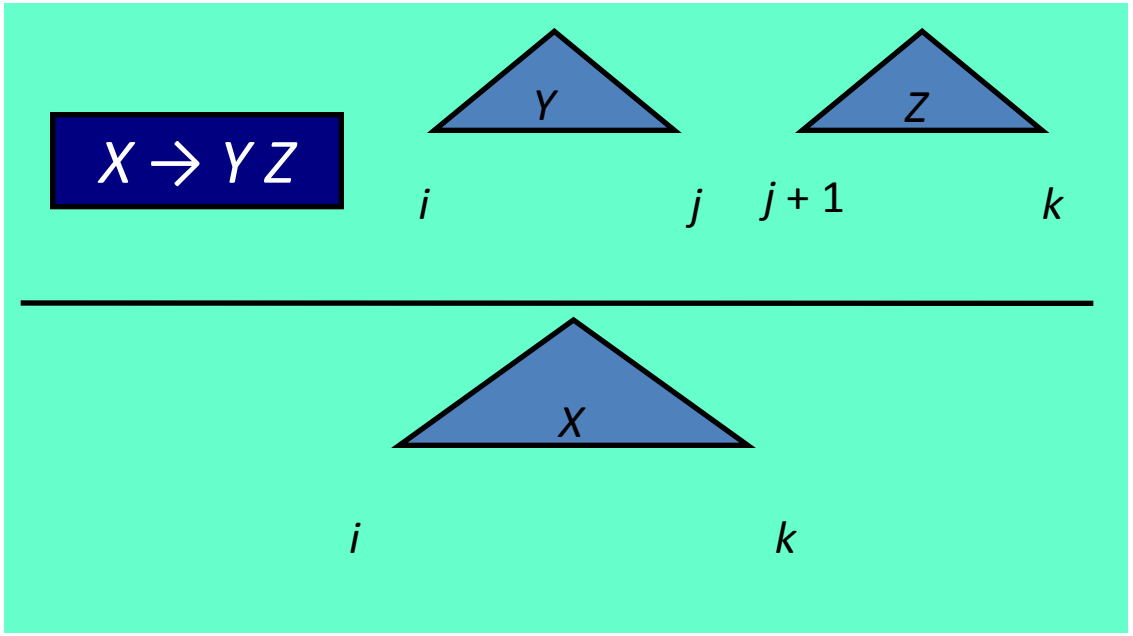
Eisner '96

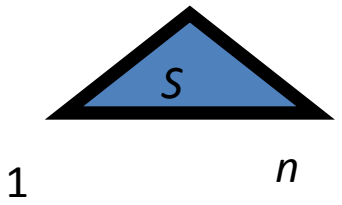


- Realization: You don't need to store and evaluate all K possible trees in the K-span
  - If carefully done, sufficient to only consider subtrees with roots at the end!

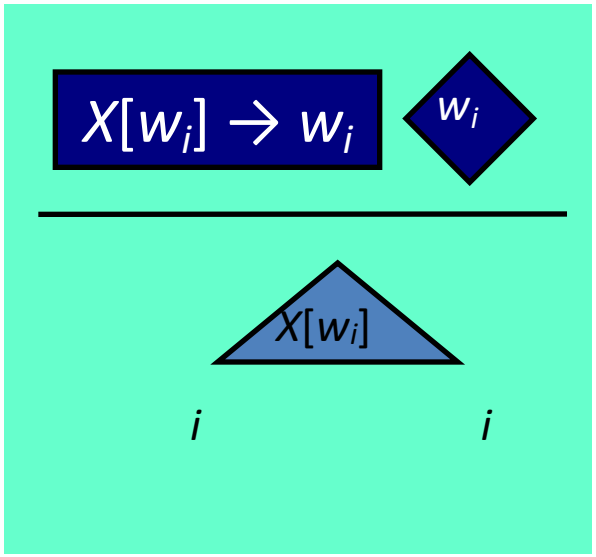
Remember CKY

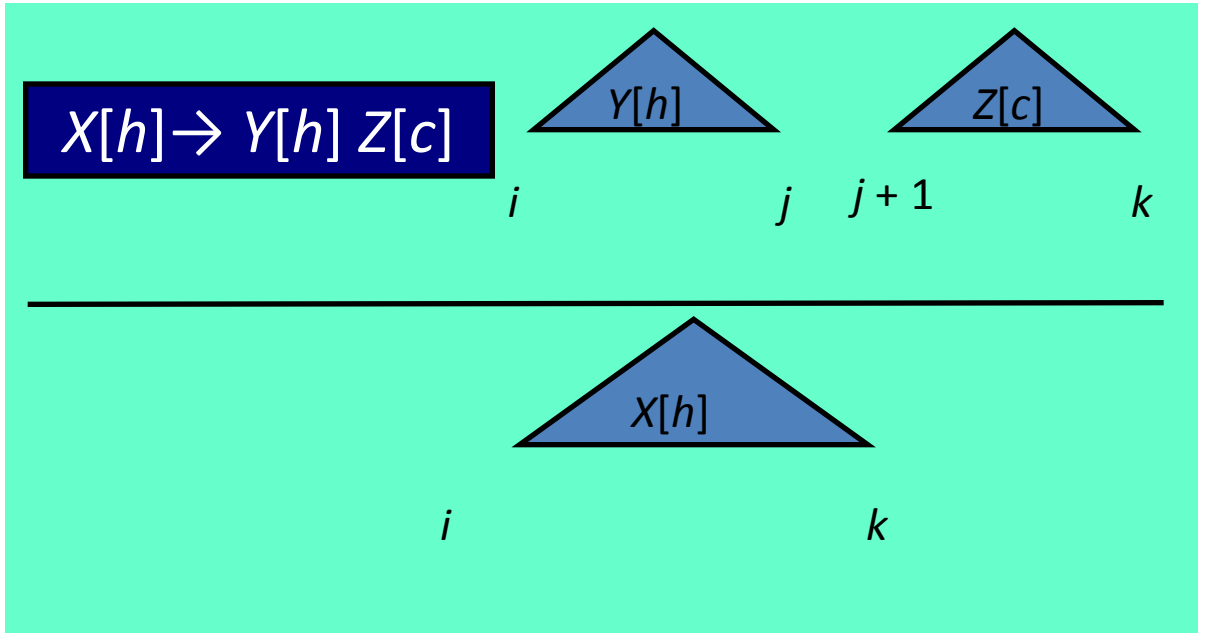


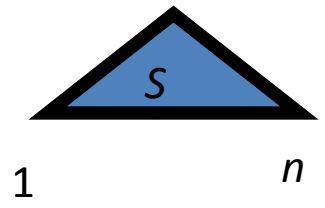




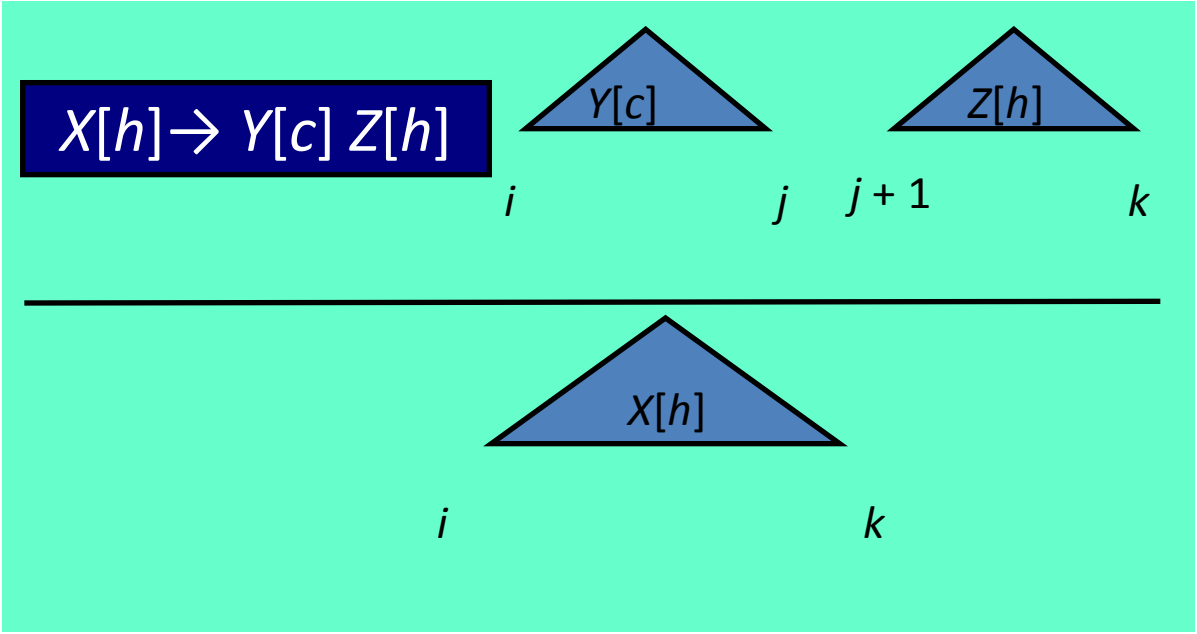
CKY with Heads



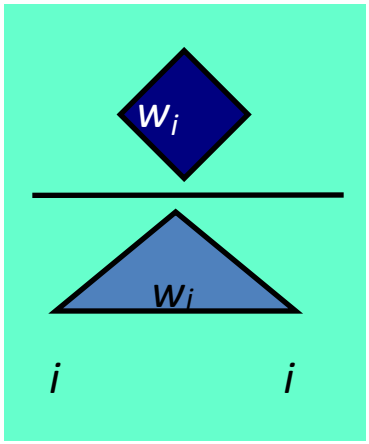


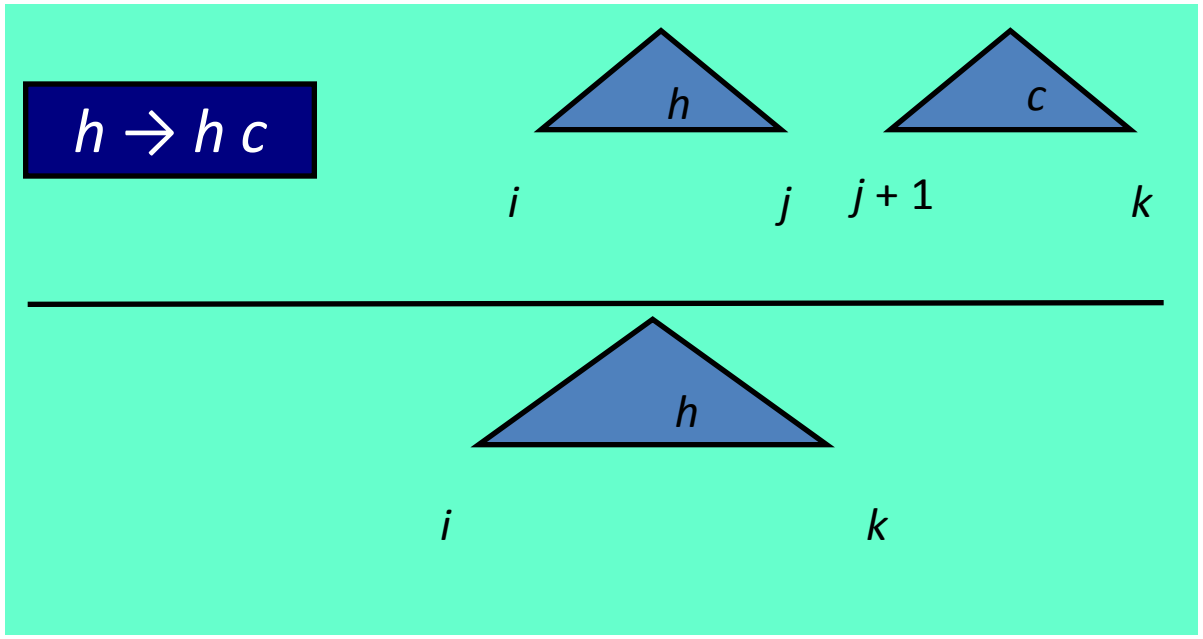


CKY with Heads (one more rule)



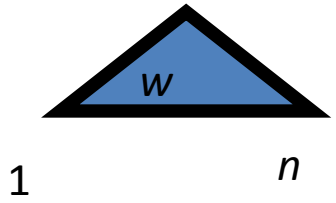
CKY with Heads, without Nonterminals



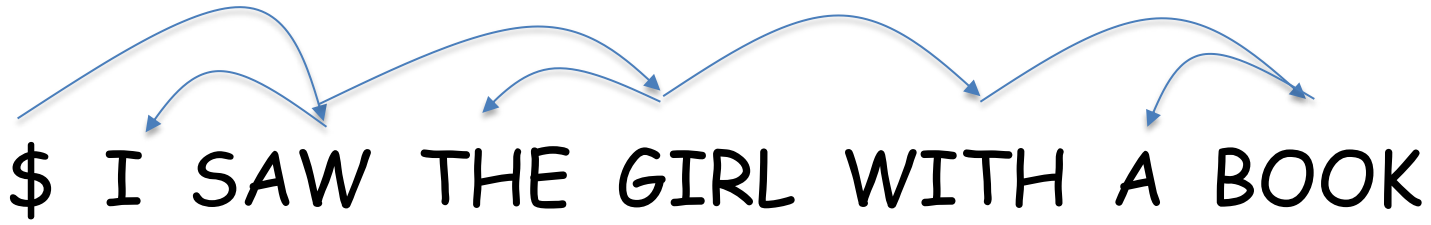


*Plus the rule for $h \rightarrow c\ h$.

What's the runtime?

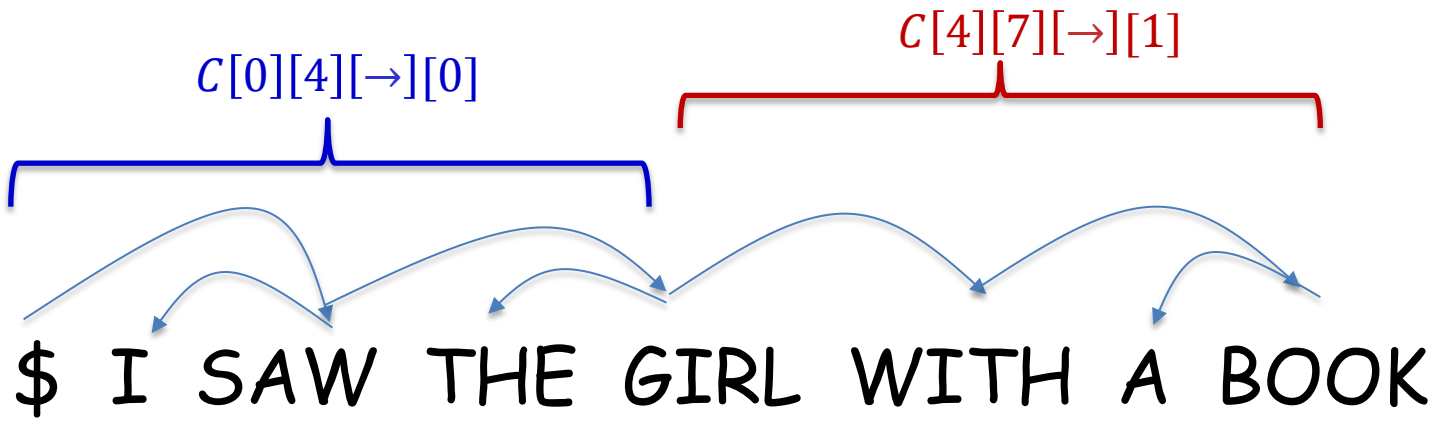


Eisner's Algorithm



- An instance of a *maximum spanning tree* algorithm
  - Or a minimum spanning tree, depending on whether we consider a score or a cost
- Note: The "root" symbol at the beginning
- Finds a spanning tree from the root that has the highest score (or probability)

Eisner's Algorithm

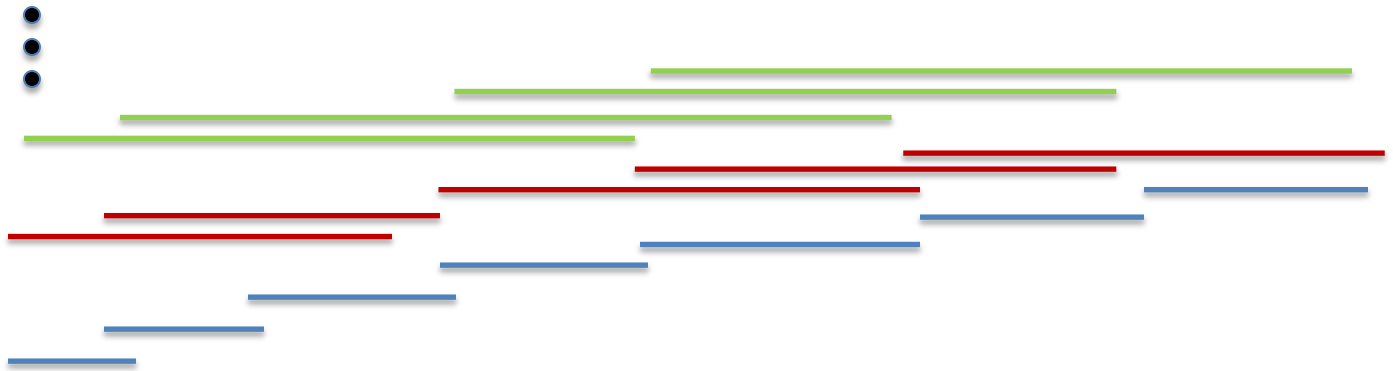


- Notation:

$$C[i][j][d][c]$$

- Best cost of tree spanning symbols $[i, j]$
  - $d = \rightarrow$: Root at i
  - $d = \leftarrow$: Root at j
  - $c = 0$: Incomplete subtree: Can be extended from boundary
  - $c = 1$: Complete subtree: Boundary cannot be extended, must be extended from root

Eisner's Algorithm



$ I SAW THE GIRL WITH A BOOK

- Recursive, for every span, find
  - Best left-to-right incomplete tree
  - Best right-to-left incomplete tree
  - Best left-to-right complete tree
  - Best right-to-left complete tree

Eisner's Algorithm

for $i : 0 \ldots n$ **and all** d, c:
    $C[i][i][d][c] = 0$
for $l: 1 \ldots n$:
    for $s: 0 \ldots n - l$:
        $t = s + l$

$$C[s][t][\rightarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_s, w_t)$$

$$C[s][t][\leftarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_t, w_s)$$

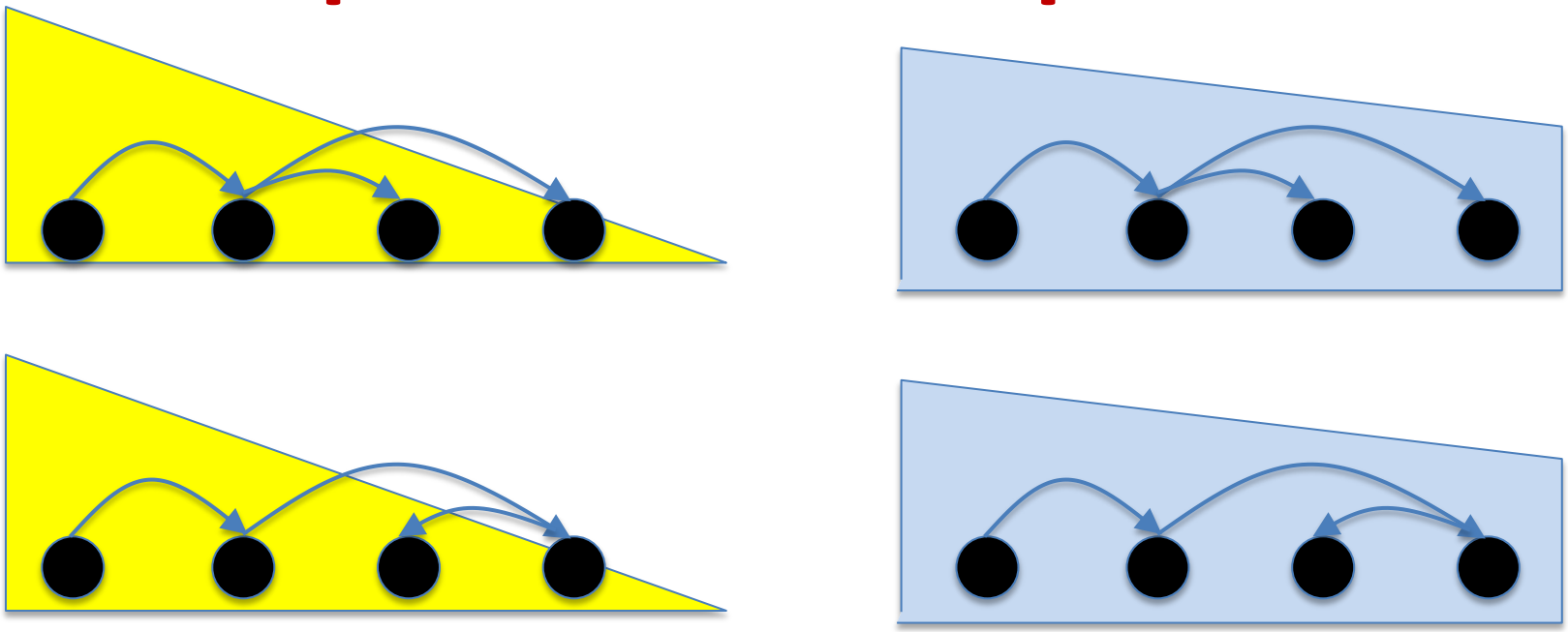
$$C[s][t][\rightarrow][1] = \max_{s < j \le t} C[s][j][\rightarrow][1] + C[j][t][\rightarrow][1]$$

$$C[s][t][\leftarrow][1] = \max_{s \le j < t} C[s][j][\leftarrow][1] + C[j][t][\leftarrow][1]$$

return $C[0][n][\rightarrow][1]$

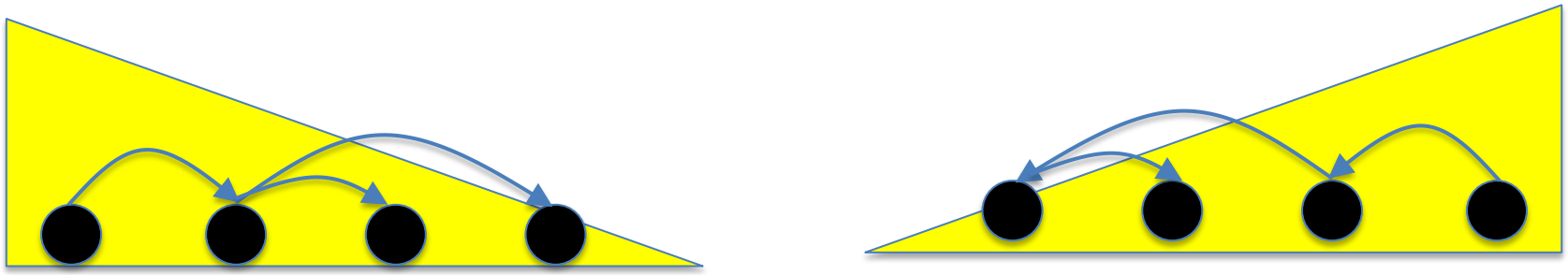
- Note dynamic programming structure
- Uses best length-N trees to find best length N+1 trees
  - "Score" may be computed by any model
- Algorithm is $O(N^3)$

Explanation with pictures



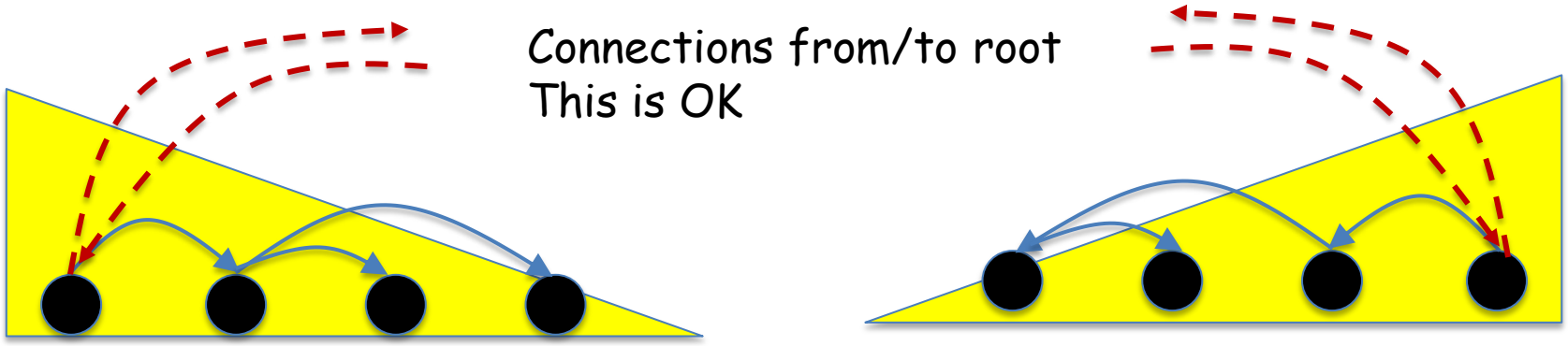
- Right triangle: Complete tree, with root at left
- Right trapezium: Incomplete tree with root at left
- Left triangle and left trapezium are corresponding structures with the root to the right

Explanation with pictures



- Complete right subtree: Cannot take additional right children
  - Can have arbitrary substructure within subtree
  - **Root to extreme left**
- Complete left subtree: Cannot take additional left children
  - Can have arbitrary substructure within subtree
  - **Root to extreme right**

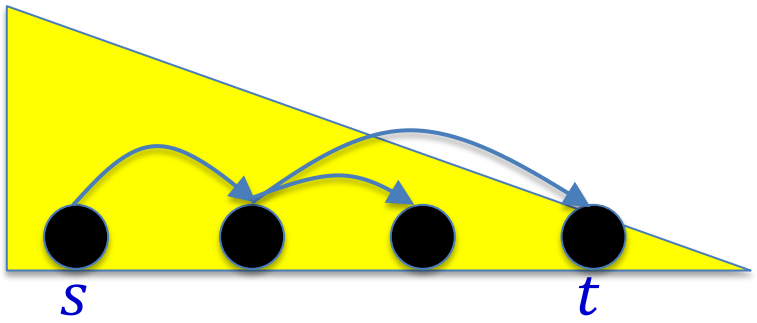
Explanation with pictures



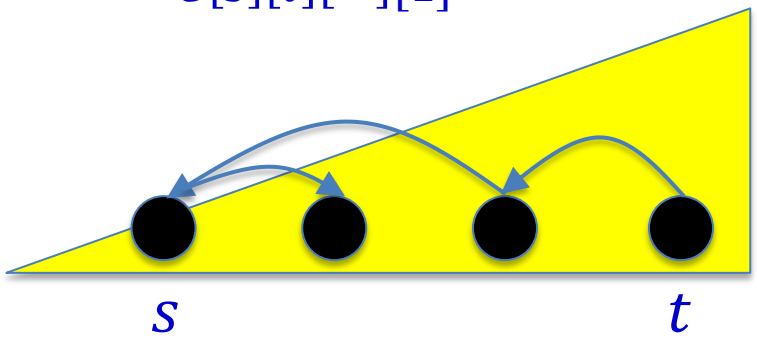
- Complete right subtree: Cannot take additional right children
  - Can have arbitrary substructure within subtree
  - **Root to extreme left**
- Complete left subtree: Cannot take additional left children
  - Can have arbitrary substructure within subtree
  - **Root to extreme right**

Explanation with pictures

$C[s][t][\rightarrow][1]$



$C[s][t][\leftarrow][1]$

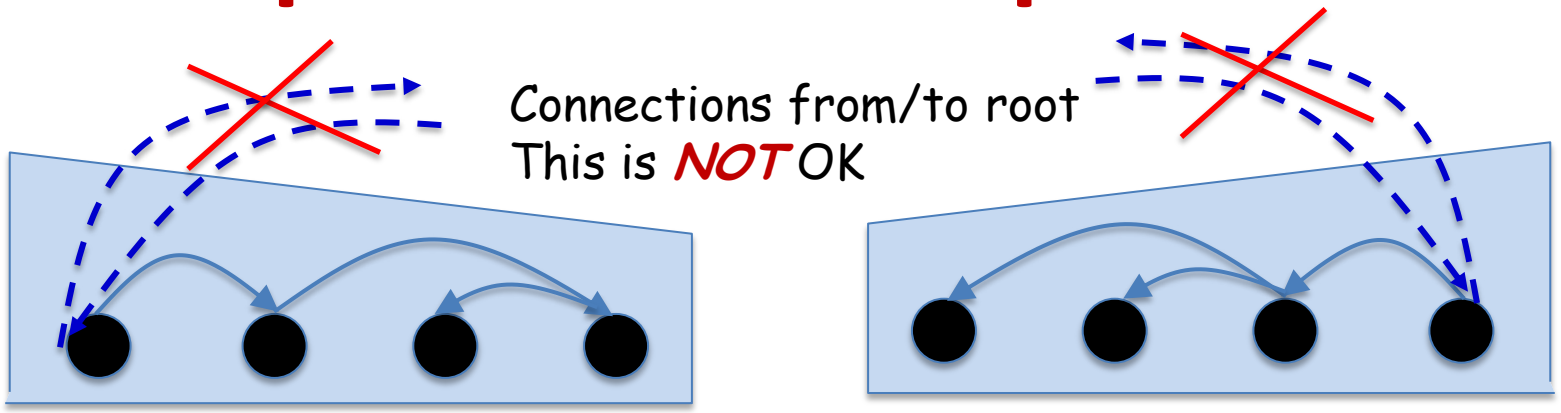


- Note representation of best score

Explanation with pictures

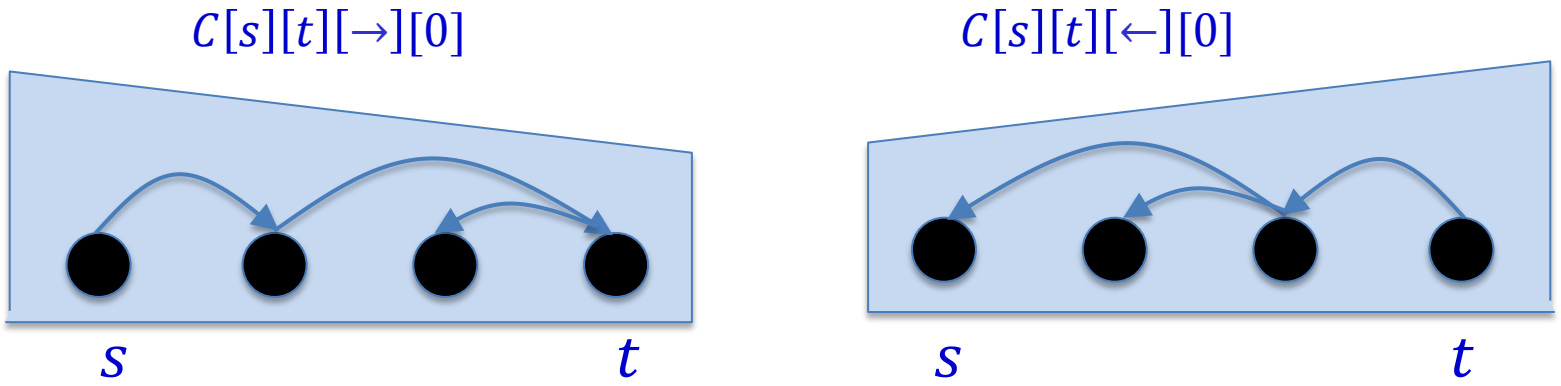
- Incomplete right subtree: Can extend to the right from rightmost node
  - But not from any other node
  - Root at extreme left
- Incomplete left subtree: Can extend to the left from leftmost node
  - But not from any other node
  - Root at extreme right

Explanation with pictures



- Incomplete right subtree: Can extend to the right from rightmost node
  - But not from any other node
  - Root at extreme left
- Incomplete left subtree: Can extend to the left from leftmost node
  - But not from any other node
  - Root at extreme right

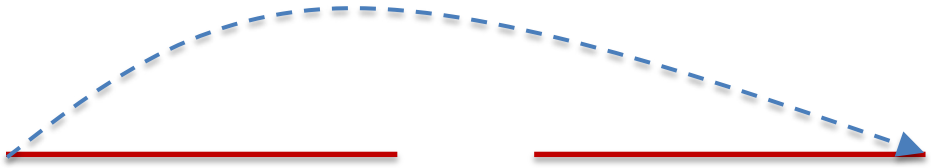
Explanation with pictures



- Note representation of best score

Logic behind recursion

$$C([1][3][*][?] \circ [4][5][*][?]) = C[1][3][*][?] + C[4][5][*][?] + \varphi(w_1, w_5)$$



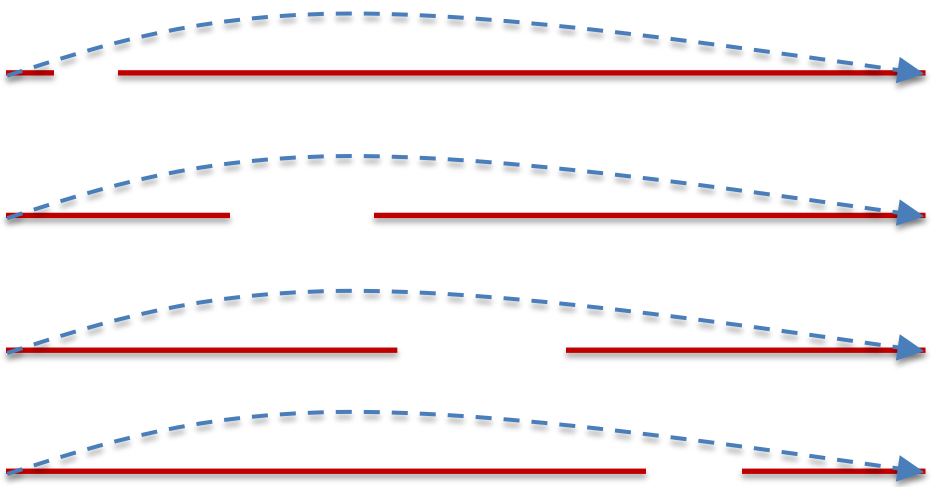
$ I SAW THE GIRL WITH A BOOK

Case 1: Connecting the heads of two spans one word apart

- By our model: The score of connecting two spans is the sum of the score of the individual spans plus the score of the connection

Logic behind recursion

$$C[1][5][*][?] = \max_{j} \; C[1][j][*][?] + C[j][5][*][?] + \varphi(w_1, w_5)$$



$ I SAW THE GIRL WITH A BOOK

Connecting the heads of two spans one word apart

- The *best* score for composing a span from $i \rightarrow j$ is the score derived from the best junction

Logic behind recursion

$$C([1][3][*][?] \circ [3][5][*][?]) = C[1][3][*][?] + C[4][5][*][?]$$

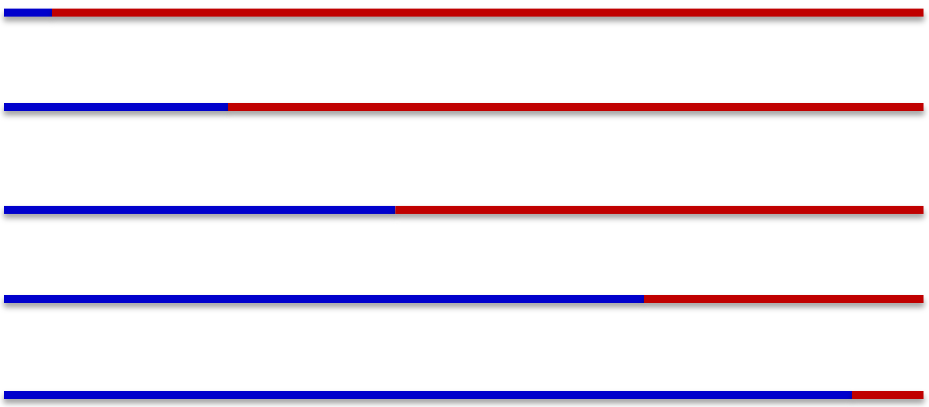
$ I SAW THE GIRL WITH A BOOK

Case 2: Connecting two spans at a boundary

- By our model: The score of connecting two spans is the sum of the scores of the individual spans

Logic behind recursion

$$C[1][5][*][?] = \max_{j} \; C[1][j][*][?] + C[j][5][*][?]$$

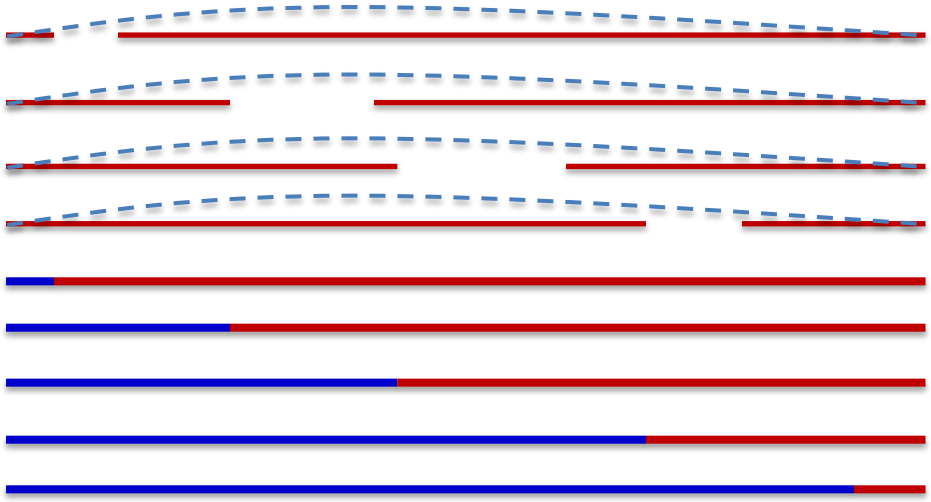


$ I SAW THE GIRL WITH A BOOK

Connecting the heads of two spans one word apart

- The *best* score for composing a span from $i \rightarrow j$ is the score derived from the best junction

Logic behind recursion

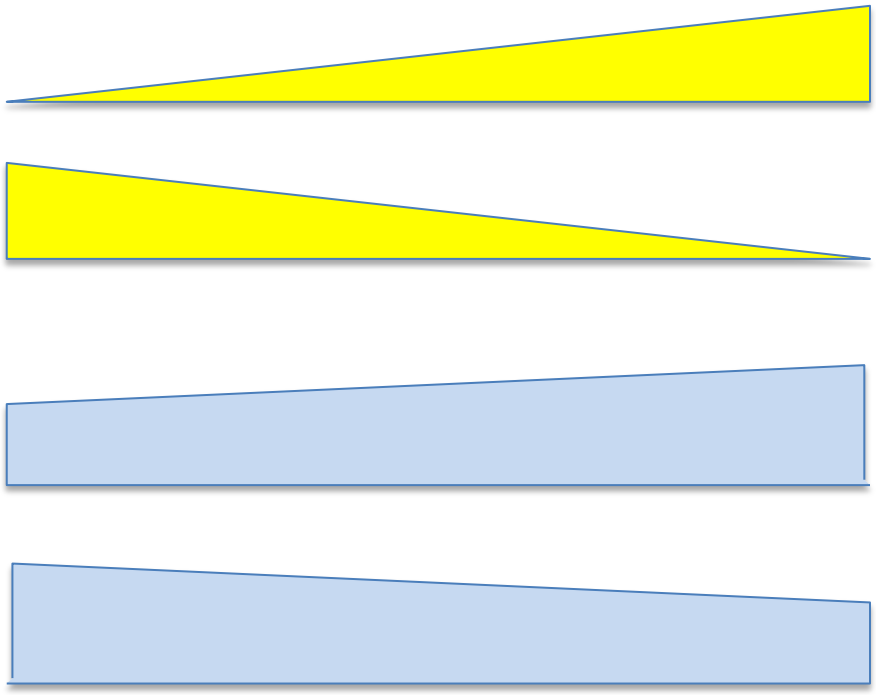


$ I SAW THE GIRL WITH A BOOK

Connecting the heads of two spans one word apart

- All possible ways of forming the span

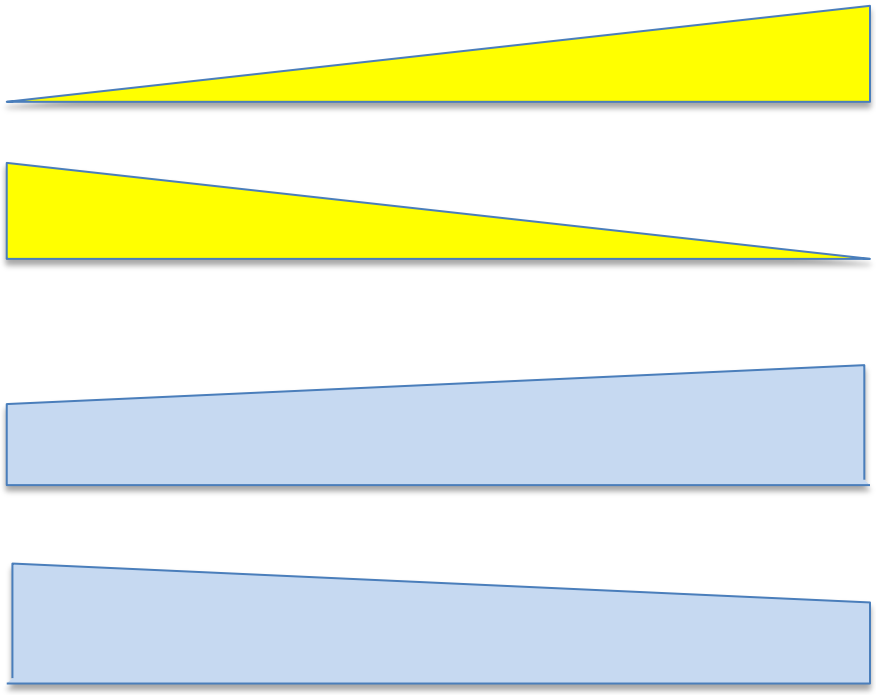
In figures



$ I SAW THE GIRL WITH A BOOK

- There are four kinds of subtrees we can compose within any span with the root at a boundary
- Lets see how we can compose these

In figures



If we do this right, we never have to worry about composing any subtree with a root in the middle

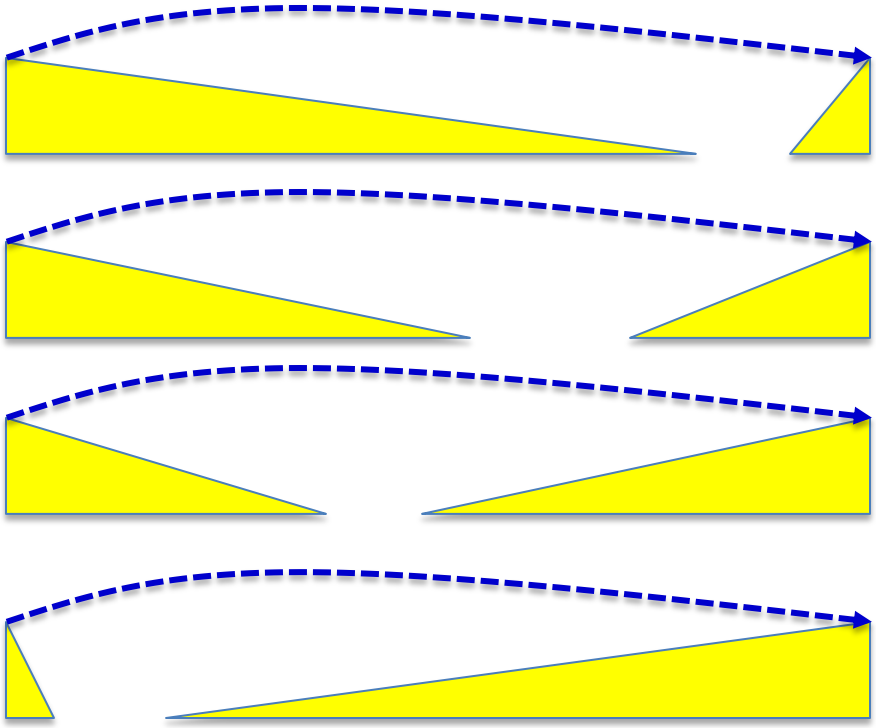
It will naturally fall out of the algorithm

$ I SAW THE GIRL WITH A BOOK

- There are four kinds of subtrees we can compose within any span with the root at a boundary
- Lets see how we can compose these

$$C[s][t][\rightarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_s, w_t)$$

Left to right



Select the best one

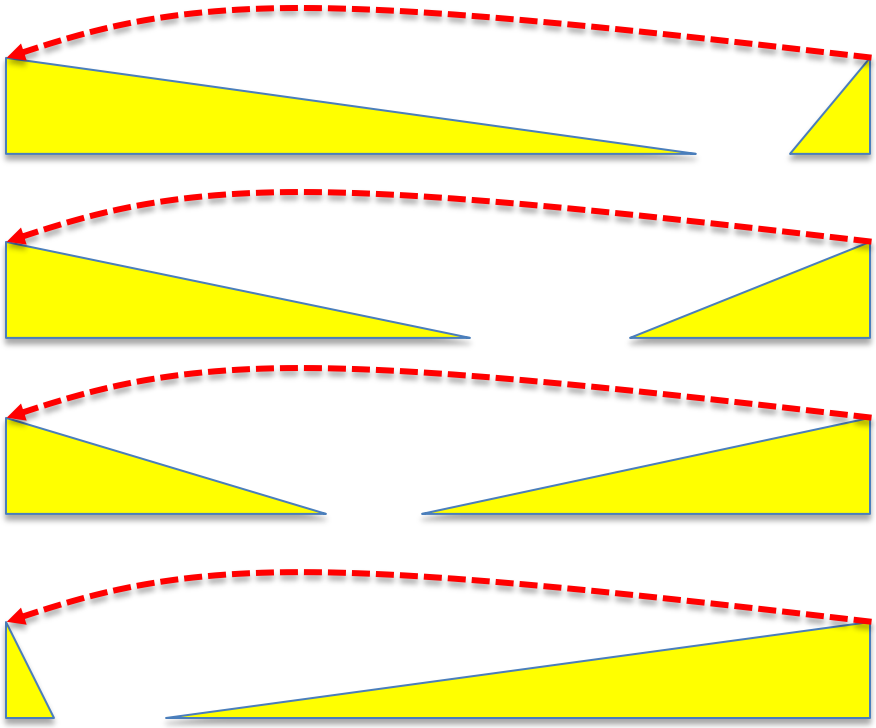
$ I SAW THE GIRL WITH A BOOK

Outcome

- Connecting two competed trees to get an incomplete tree
  - Showing left to right only (connection from root of left tree to root of right tree)

$$C[s][t][\leftarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_t, w_s)$$

Right to left



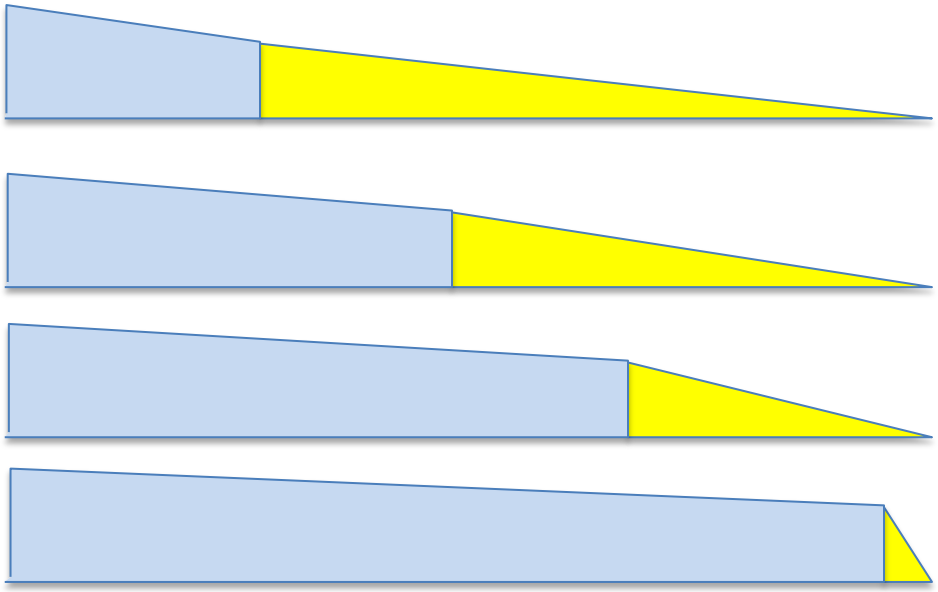
Select the best one

$ I SAW THE GIRL WITH A BOOK

Outcome

- Connecting two competed trees to get an incomplete tree
  - Showing right to left

$$C[s][t][\rightarrow][1] = \max_{s<j\leq t} C[s][j][\rightarrow][0] + C[j][t][\rightarrow][1]$$



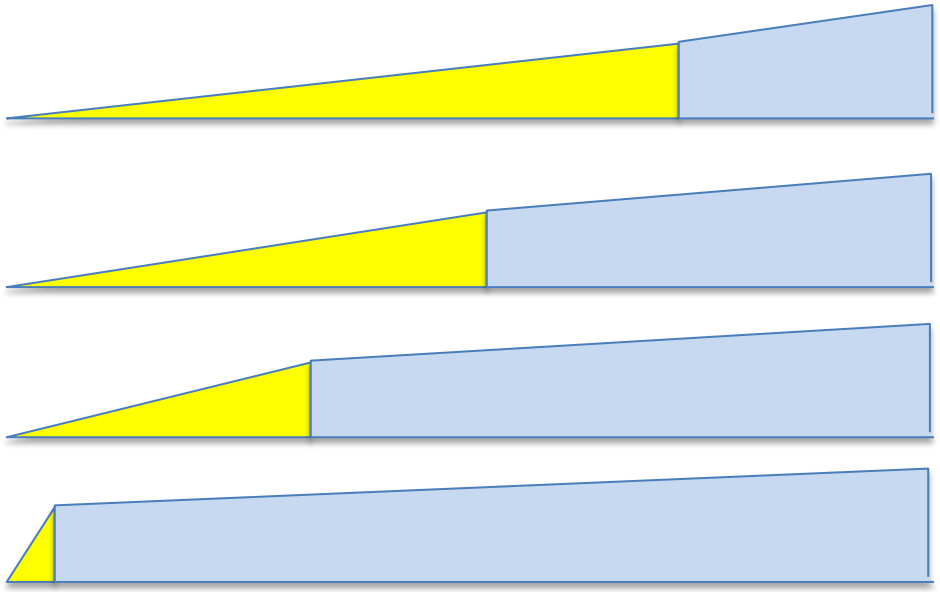
Select the best one

$ I SAW THE GIRL WITH A BOOK

Outcome

- Connecting an incomplete tree and a complete tree to get a complete tree
  - Showing left to right only (root to left)

$$C[s][t][\leftarrow][1] = \max_{s \le j < t} C[s][j][\leftarrow][0] + C[j][t][\leftarrow][1]$$



Select the best one

$ I SAW THE GIRL WITH A BOOK

Outcome

- Connecting an incomplete tree and a complete tree to get a complete tree
  - Showing Right to Left (Root is to the right)

An Example

$ I SAW THE GIRL WITH A BOOK

- The sequence of operations

Eisner's Algorithm

for $i : 0 \ldots n$ **and all** d, c:
    $C[i][i][d][c] = 0$
for $l: 1 \ldots n$:
    for $s: 0 \ldots n - l$:
        $t = s + l$

$$C[s][t][\rightarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_s, w_t)$$

$$C[s][t][\leftarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_t, w_s)$$

$$C[s][t][\rightarrow][1] = \max_{s < j \le t} C[s][j][\rightarrow][1] + C[j][t][\rightarrow][1]$$

$$C[s][t][\leftarrow][1] = \max_{s \le j < t} C[s][j][\leftarrow][1] + C[j][t][\leftarrow][1]$$

return $C[0][n][\rightarrow][1]$

- Note dynamic programming structure
- Uses best length-N trees to find best length N+1 trees
  - "Score" may be computed by any model
- Algorithm is $O(N^3)$

Eisner's Algorithm

for $i : 0 \ldots n$ **and all** d, c:
    $C[i][i][d][c] = 0$

for $l: 1 \ldots n$:
    for $s: 0 \ldots n - l$:
        $t = s + l$

$$C[s][t][\rightarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_s, w_t)$$

$$C[s][t][\leftarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_t, w_s)$$

$$C[s][t][\rightarrow][1] = \max_{s < j \le t} C[s][j][\rightarrow][1] + C[j][t][\rightarrow][1]$$

$$C[s][t][\leftarrow][1] = \max_{s \le j < t} C[s][j][\leftarrow][1] + C[j][t][\leftarrow][1]$$

return $C[0][n][\rightarrow][1]$

- Note dynamic programming structure
- Uses best length-N trees to find best length N+1 trees
  - "Score" may be computed by any model
- Algorithm is $O(N^3)$

An Example

$$C[i][i][\rightarrow][1] = C[i][i][\leftarrow][1] = 0$$

- The sequence of operations
- Level 0: every word is both a left and right directed tree with score 0

Eisner's Algorithm

for $i : 0 \ldots n$ **and all** d, c:
  $C[i][i][d][c] = 0$
for $l: 1 \ldots n$:
  for $s: 0 \ldots n - l$:
    $t = s + l$

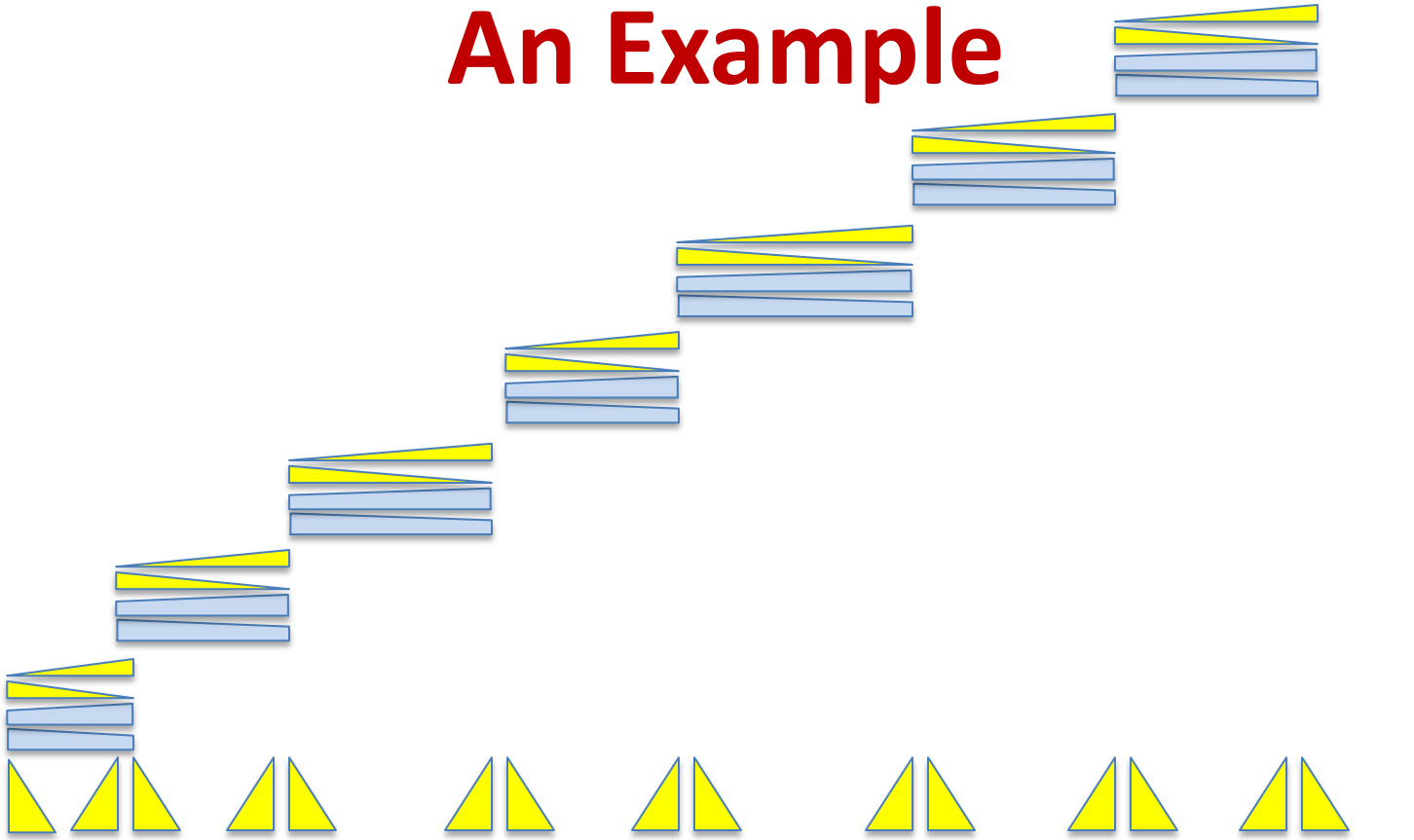
$l = 1$

$$C[s][t][\rightarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_s, w_t)$$
$$C[s][t][\leftarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_t, w_s)$$
$$C[s][t][\rightarrow][1] = \max_{s < j \le t} C[s][j][\rightarrow][1] + C[j][t][\rightarrow][1]$$
$$C[s][t][\leftarrow][1] = \max_{s \le j < t} C[s][j][\leftarrow][1] + C[j][t][\leftarrow][1]$$

return $C[0][n][\rightarrow][1]$

- Note dynamic programming structure
- Uses best length-N trees to find best length N+1 trees
  - "Score" may be computed by any model
- Algorithm is $O(N^3)$

An Example



- Level 1: Using level 0 trees, compose two-word span trees from every word
  - Left incomplete tree
  - Right incomplete tree
  - Left complete tree
  - Right complete tree

Eisner's Algorithm

for $i : 0 \ldots n$ **and all** d, c:
  $C[i][i][d][c] = 0$
for $l: 1 \ldots n$:
  for $s: 0 \ldots n - l$:
    $t = s + l$

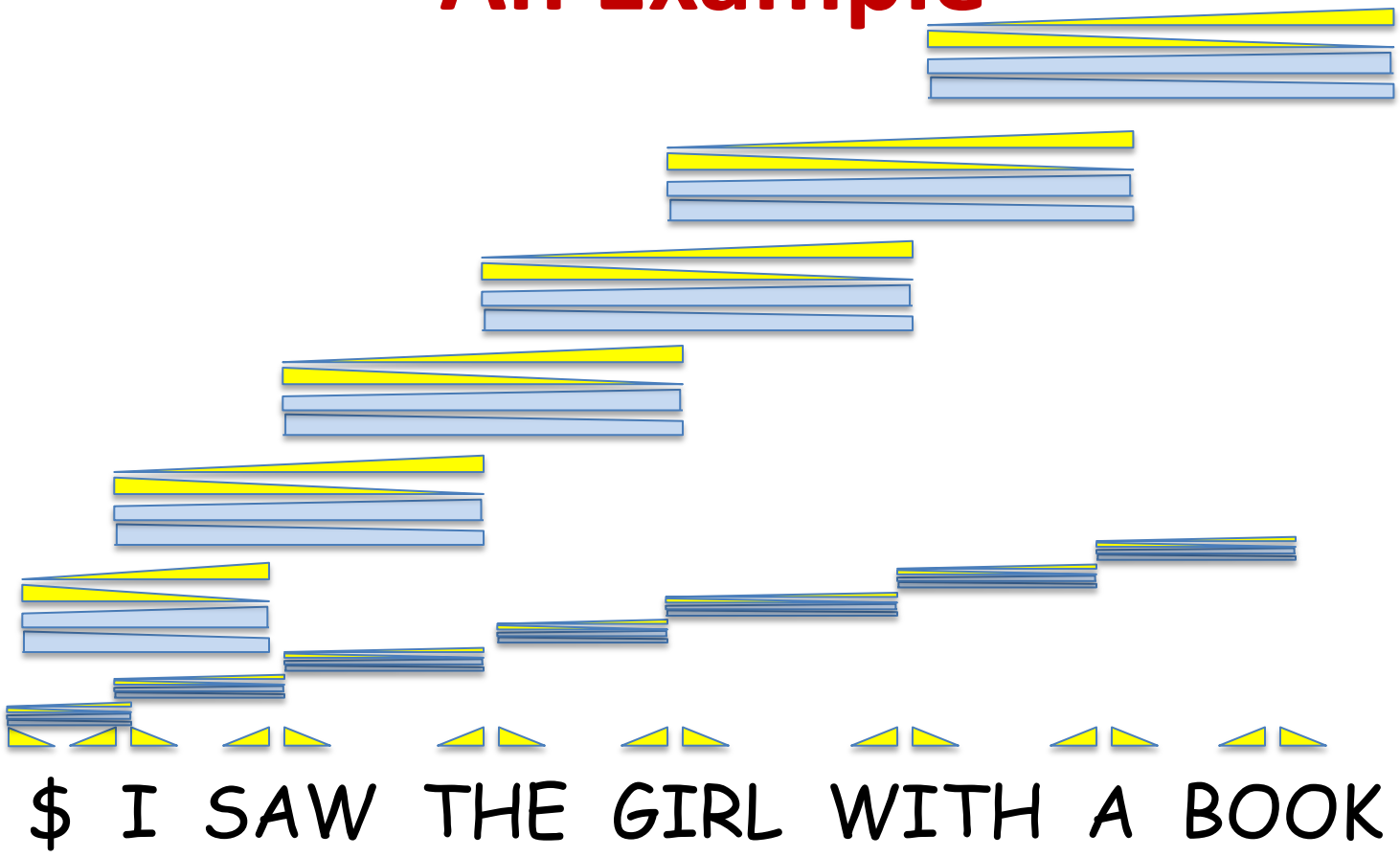
$$C[s][t][\rightarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_s, w_t)$$
$$C[s][t][\leftarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_t, w_s)$$
$$C[s][t][\rightarrow][1] = \max_{s < j \le t} C[s][j][\rightarrow][1] + C[j][t][\rightarrow][1]$$
$$C[s][t][\leftarrow][1] = \max_{s \le j < t} C[s][j][\leftarrow][1] + C[j][t][\leftarrow][1]$$

return $C[0][n][\rightarrow][1]$

$l = 2$

- Note dynamic programming structure
- Uses best length-N trees to find best length N+1 trees
  - "Score" may be computed by any model
- Algorithm is $O(N^3)$

An Example



- Level 1: Using level 0 and level 1 trees, compose all four types of three-word span trees, starting at every word
  - Left incomplete tree
  - Right incomplete tree
  - Left complete tree
  - Right complete tree

Eisner's Algorithm

for $i : 0 \ldots n$ **and all** d, c:
  $C[i][i][d][c] = 0$
for $l: 1 \ldots n$:
  for $s: 0 \ldots n - l$:
    $t = s + l$

$l = 3$

$$C[s][t][\rightarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_s, w_t)$$
$$C[s][t][\leftarrow][0] = \max_{s \le j < t} C[s][j][\rightarrow][1] + C[j+1][t][\leftarrow][1] + \varphi(w_t, w_s)$$
$$C[s][t][\rightarrow][1] = \max_{s < j \le t} C[s][j][\rightarrow][1] + C[j][t][\rightarrow][1]$$
$$C[s][t][\leftarrow][1] = \max_{s \le j < t} C[s][j][\leftarrow][1] + C[j][t][\leftarrow][1]$$

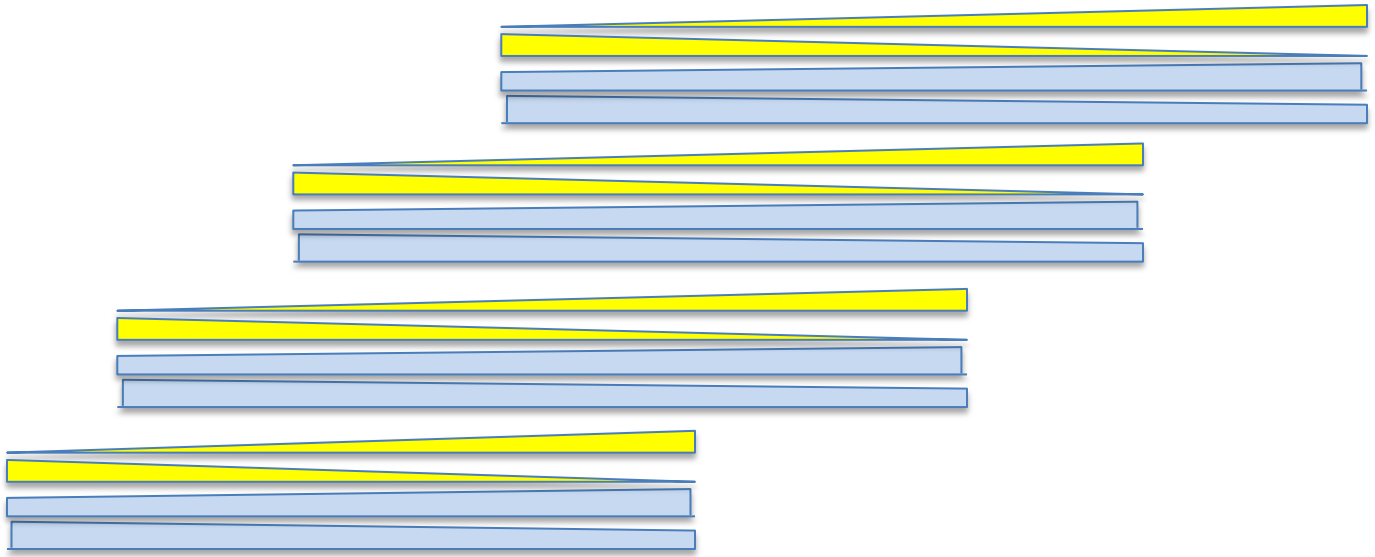
return $C[0][n][\rightarrow][1]$

- Note dynamic programming structure
- Uses best length-N trees to find best length N+1 trees
  - "Score" may be computed by any model
- Algorithm is $O(N^3)$

An Example

- Level 1: Using level 0, level 1 and level 2 trees, compose all four types of four-word span trees, starting at every word
  - Left incomplete tree
  - Right incomplete tree
  - Left complete tree
  - Right complete tree

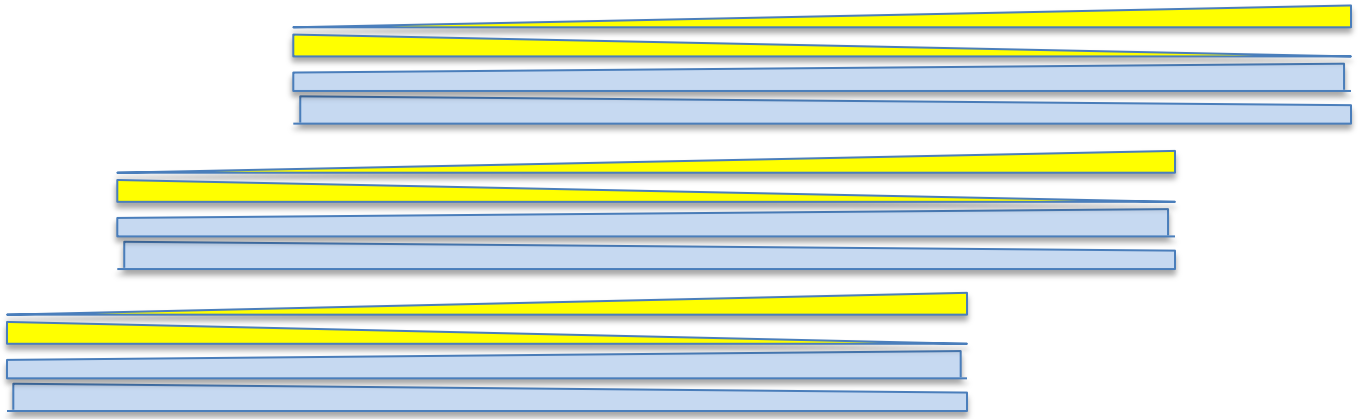
An Example



$ I SAW THE GIRL WITH A BOOK

- Construct all five-word trees from lower-level trees, starting at each position
  - As the length of the tree increases, the number of trees decrease

An Example



$ I SAW THE GIRL WITH A BOOK

- Recursively increase the span
  - Increasing span decreases the number of trees

An Example

$ I SAW THE GIRL WITH A BOOK

- Recursively increase the span
  - Increasing span decreases the number of trees

An Example

$ I SAW THE GIRL WITH A BOOK

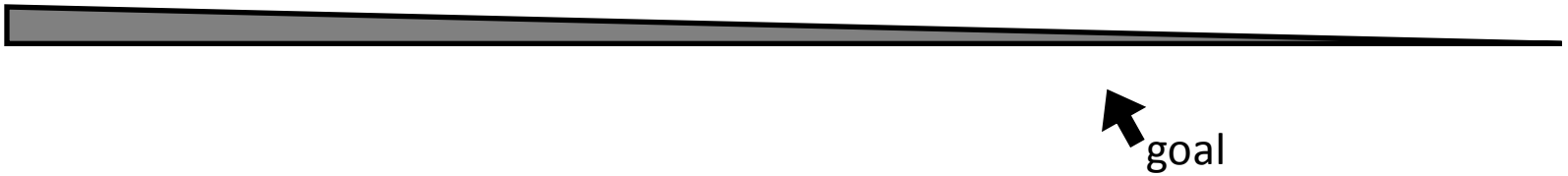
- Recursively increase the span
  - Increasing span decreases the number of trees

An Example

$ I SAW THE GIRL WITH A BOOK

- Eventually we should have a single complete LR tree, starting from the initial $ (the root)
  - If we don't, the parse has failed

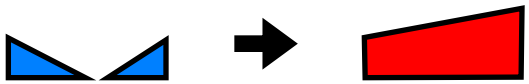
Example of the Eisner Algorithm

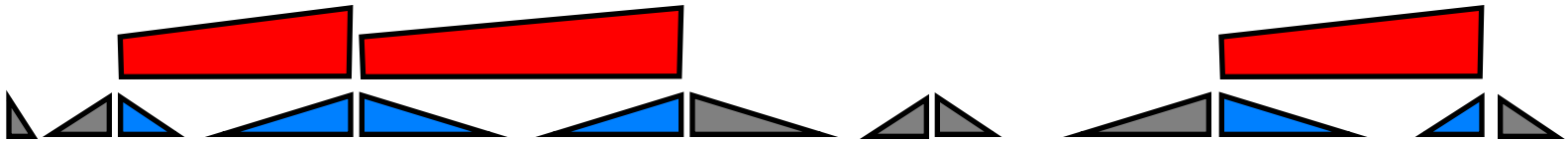


$ The professor chuckled with unabashed glee

Example of the Eisner Algorithm

Attach:



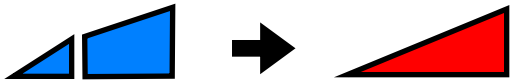


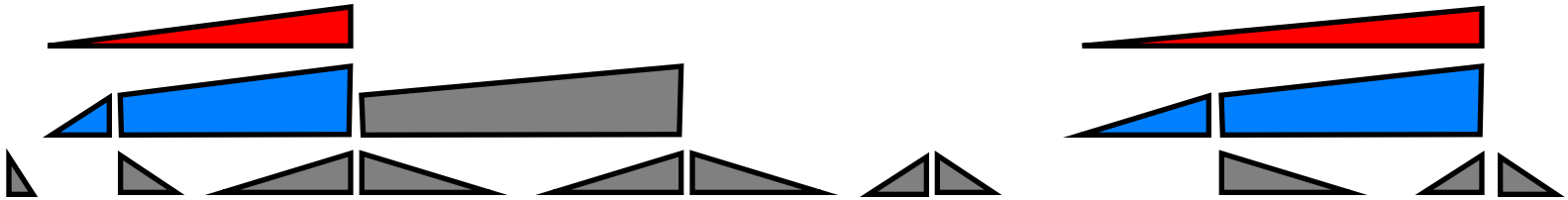
$ The professor chuckled with unabashed glee

Example of the Eisner Algorithm

Example of the Eisner Algorithm

Complete:

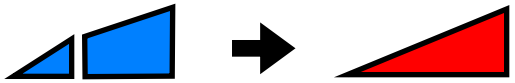


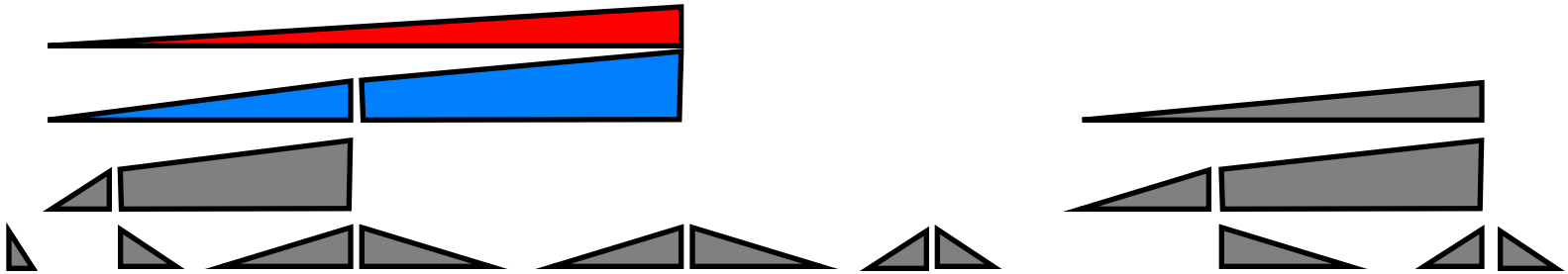


$ The professor chuckled with unabashed glee

Example of the Eisner Algorithm

Complete:

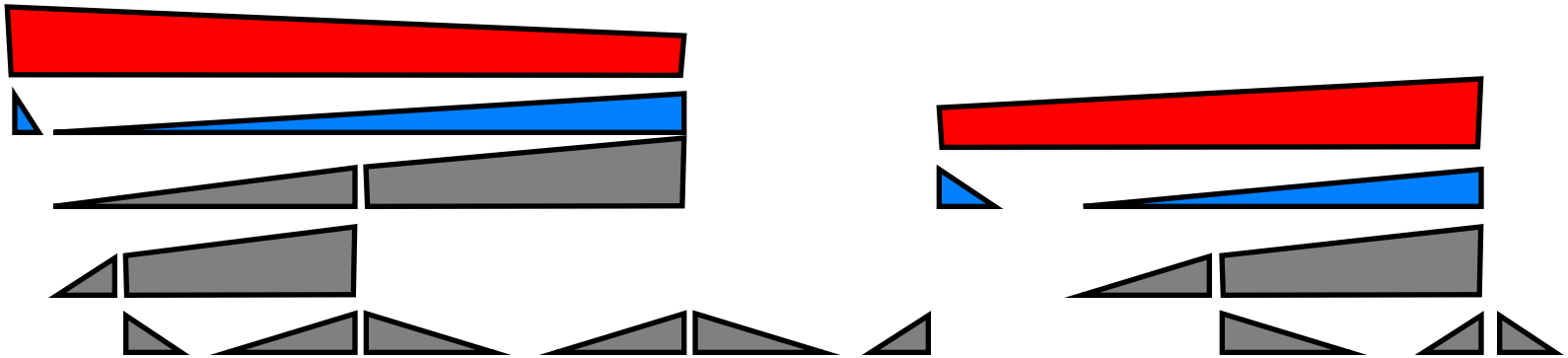




$ The professor chuckled with unabashed glee

Example of the Eisner Algorithm

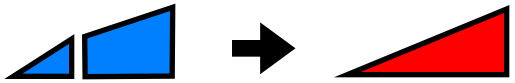
Attach:

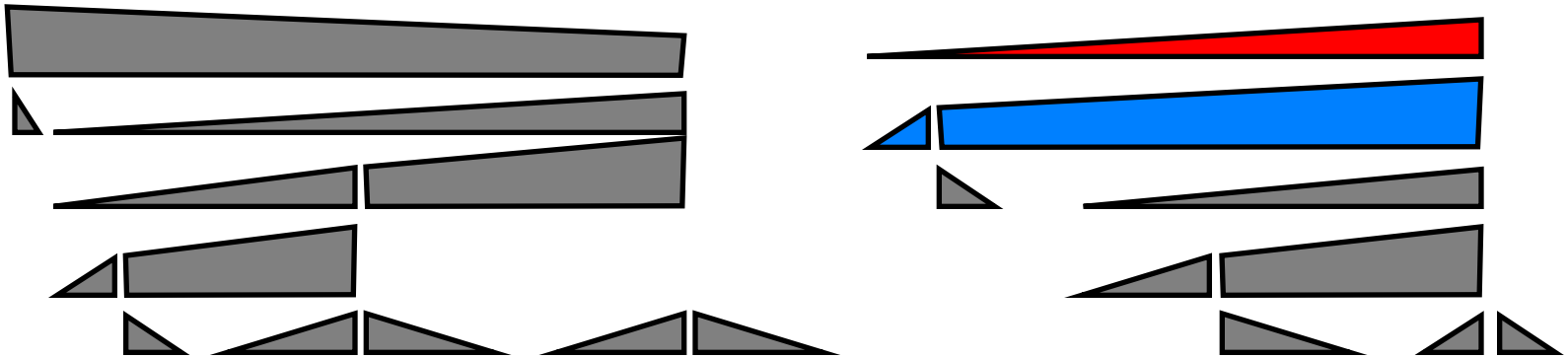


$ The professor chuckled with unabashed glee

Example of the Eisner Algorithm

Complete:

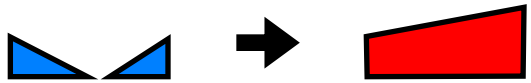


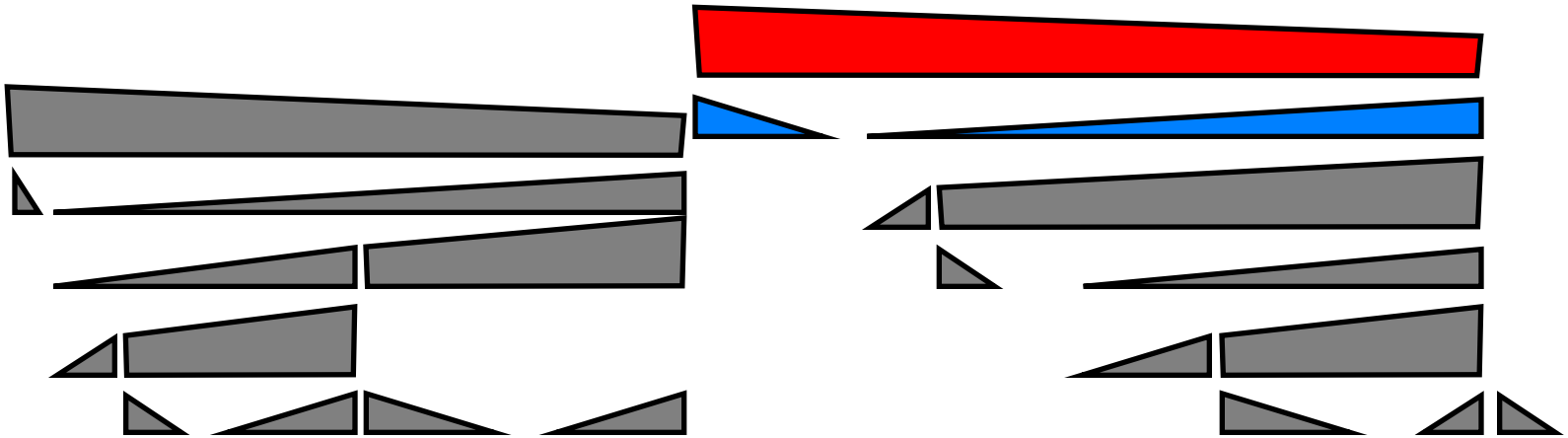


$ The professor chuckled with unabashed glee

Example of the Eisner Algorithm

Attach:

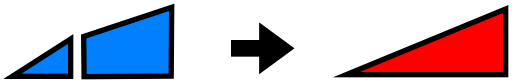


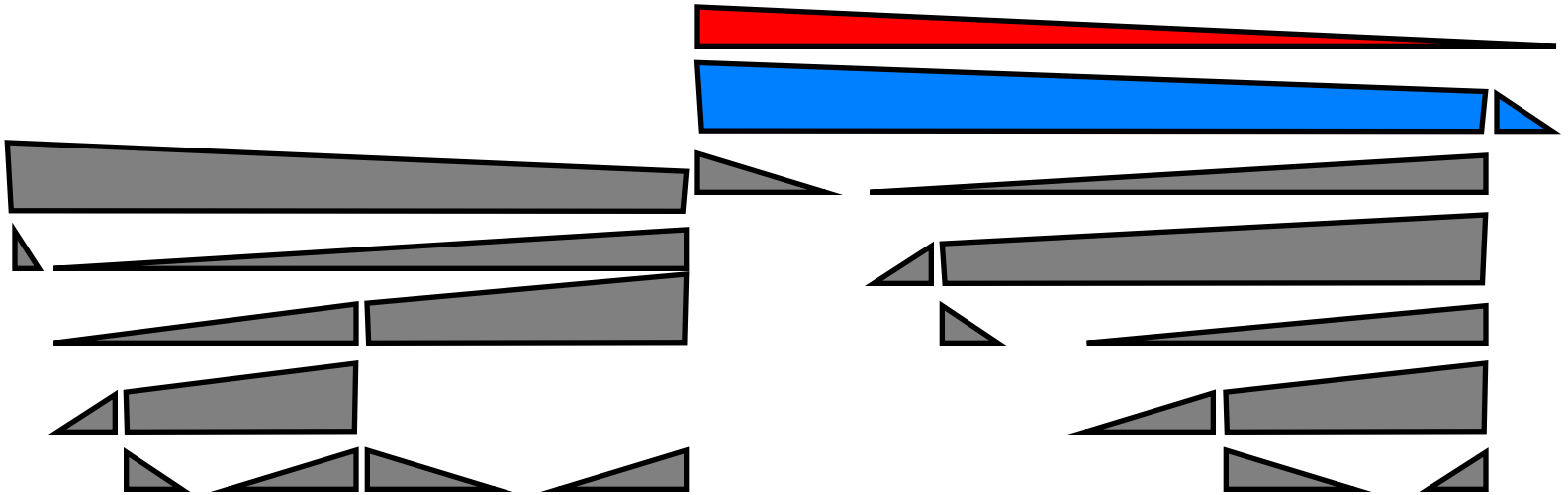


$ The professor chuckled with unabashed glee

Example of the Eisner Algorithm

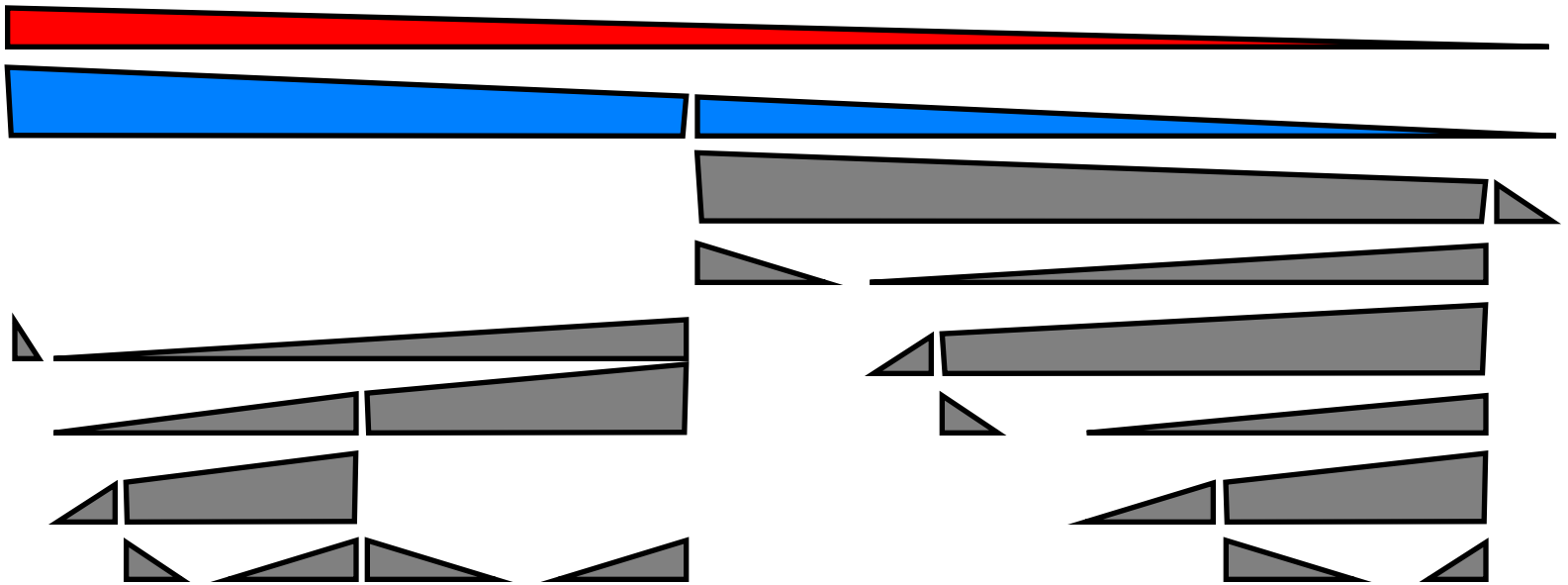
Complete:





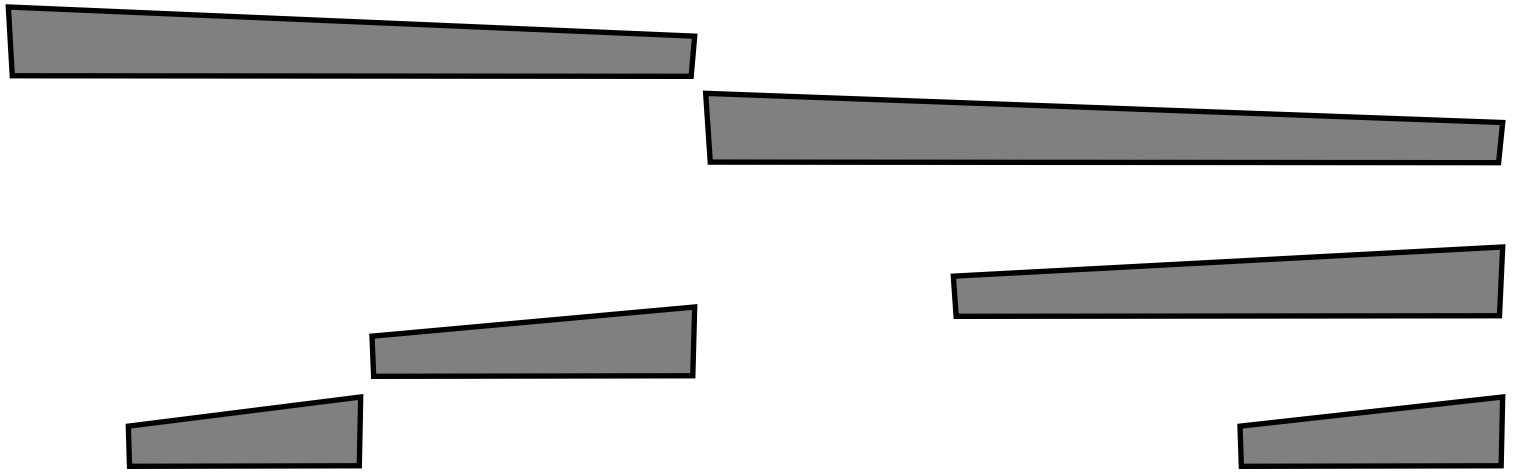
$ The professor chuckled with unabashed glee

Example of the Eisner Algorithm



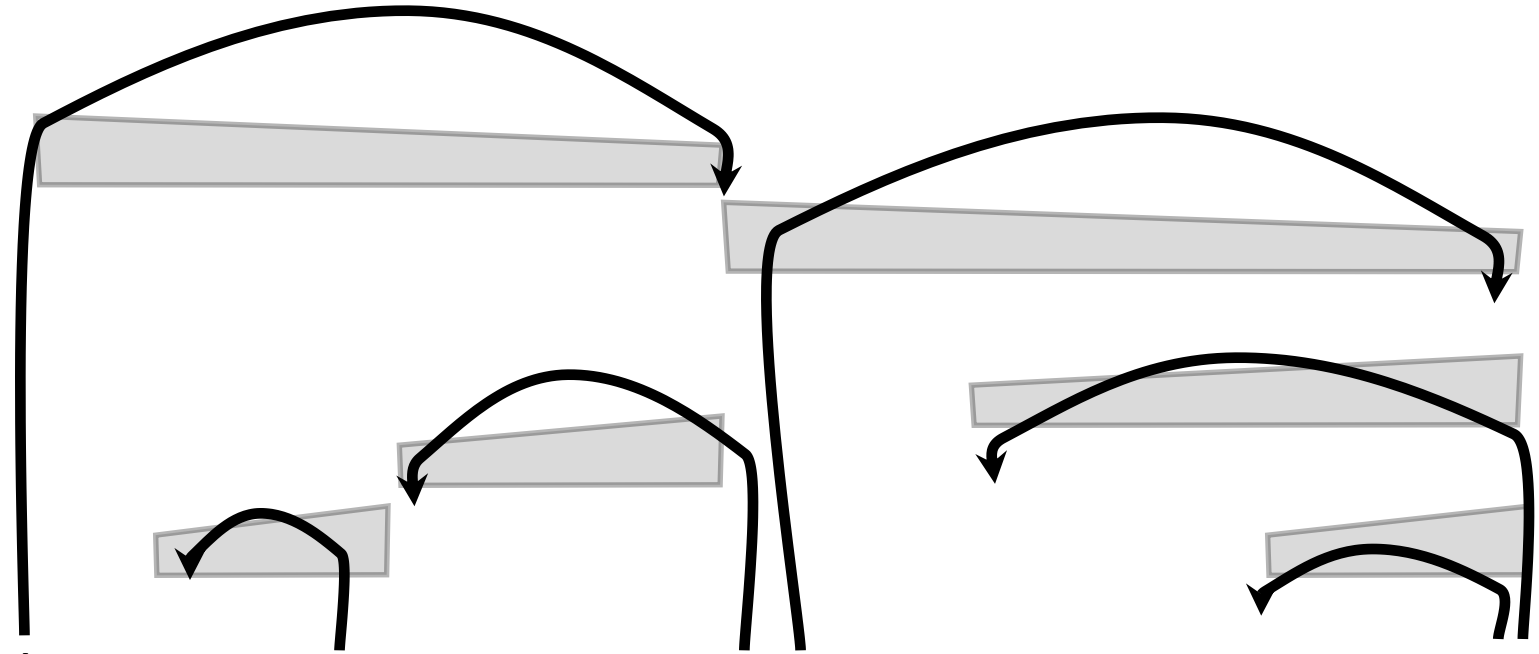
$ The professor chuckled with unabashed glee

Example of the Eisner Algorithm



$ The professor chuckled with unabashed glee

Example of the Eisner Algorithm



$ The professor chuckled with unabashed glee

Bare Bones Dependencies and Labels

- The way to represent a lot of phenomena is clear (predicate-argument and modifier relationships)
- Conjunctions pose a problem
- Sometimes words that "should" be connected are not, because of the single-parent rule
- From bare bones to **labels**:
  - consider labeled edges
  - most algorithms can be easily extended for labeled dependency parsing
- Linguistically imperfect, but computationally attractive

Evaluation

- Attachment accuracy
  - Labeled
  - Unlabeled

Bottomline

- Rethinking the algorithm in terms of attachments rather than constituents gives us an asymptotic savings!
- Bare bones, projective dependency parsing is $O(n^3)$

- What about non-projectivity?

Non-projective Dependency Parsing (McDonald et al., 2005)

- Key idea: a non-projective dependency parse is a **directed spanning tree** where
  - vertices = words
  - directed edges = parent-to-child relations
- Well-known problem: minimum-cost spanning tree
- Solution: Chu-Liu-Edmonds algorithm (cubic)
  - Good news: fast! can now recover non-projective trees!
  - Bad news: much larger search space, potential for error

Breaking Independence Assumptions

- Adding labels doesn't fundamentally change Eisner or MST
- What about edge-factoring?
- Projective case: local statistical dependence among same-side children of a given head - still cubic (Eisner and Satta, 1999).
- Non-projective parsing with any kind of second-order features (e.g., on adjacent edges) is NP-hard.
  - McDonald explored approximations in his thesis
  - Find the best projective parse and then rearrange the edges as long as the score improves - $O(n^3)$
  - ILP (Martins et al., 2009)

CoNLL Tasks

- Dependency parsing is now more popular than constituency parsing
  - Just showed a very basic framework
  - But much current work builds on similar frameworks
- 2006 and 2007: dependency parsing on a variety of languages was the shared task at CoNLL - a few dozen systems.
  - Many of the datasets are freely available.
  - Parsing papers now typically evaluate on most or all of these datasets.
- CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies
  - Followup in CoNLL 2018d

Parsing in 2011

Current Research Directions

- Better learning for parsing (e.g., max margin as in Taskar et al., 2004; CRFs as in Finkel et al., 2008; many other "parsing" papers that are really about learning for structured prediction)
- Integrating learning and search (Petrov, 2009)
- Synchronous grammars in machine translation; bilingual parsing
- Richer formalisms (CCG, TAG, unification-based grammars)
- Integrating parsing with morphological or semantic analysis