UPPSALA
UNIVERSITET

Dependency grammar and dependency parsing

Syntactic analysis (5LN455)

2015-12-09

Sara Stymne
Department of Linguistics and Philology

Based on slides from Marco Kuhlmann

UPPSALA UNIVERSITET

Activities - dependency parsing

- 3 lectures (December)
- 1 literature seminar (January 13)
- 2 assignments (DL: January 17)
    - Written assignment
    - Try and evaluate a state-of-the-art system, MaltParser
- Supervision on demand, by email or book a meeting

UPPSALA UNIVERSITET

Overview

- Dependency parsing in general
- Arc-factored dependency parsing
  - Collins' algorithm
  - Eisner's algorithm
- Transition-based dependency parsing
  - The arc-standard algorithm
- Evaluation of dependency parsers

UPPSALA UNIVERSITET

Dependency grammar

Dependency grammar

- The term ‘dependency grammar’ does not refer to a specific grammar formalism.
- Rather, it refers to a specific way to describe the syntactic structure of a sentence.

The notion of dependency

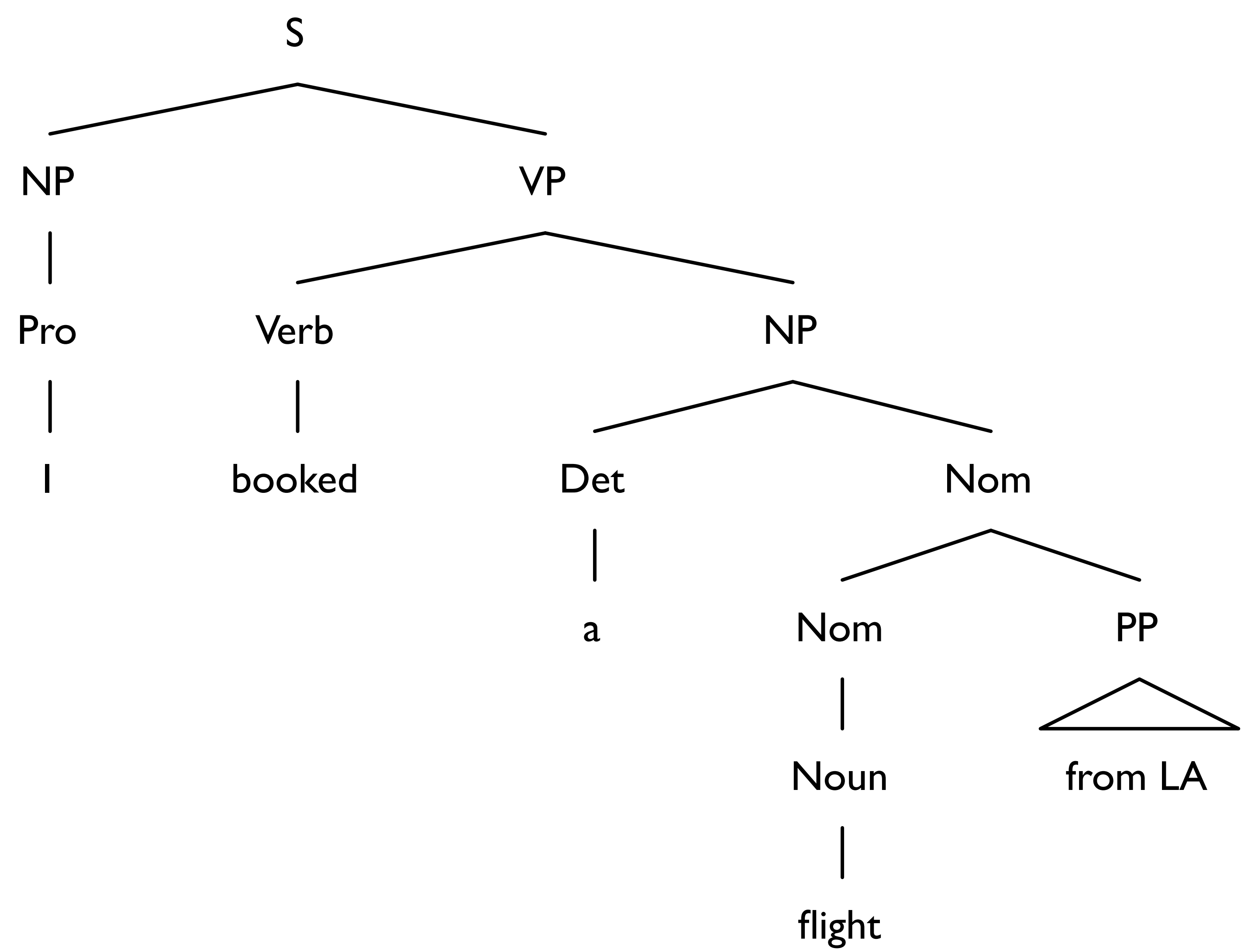
- The basic observation behind constituency is that groups of words may act as one unit.

  Example: noun phrase, prepositional phrase

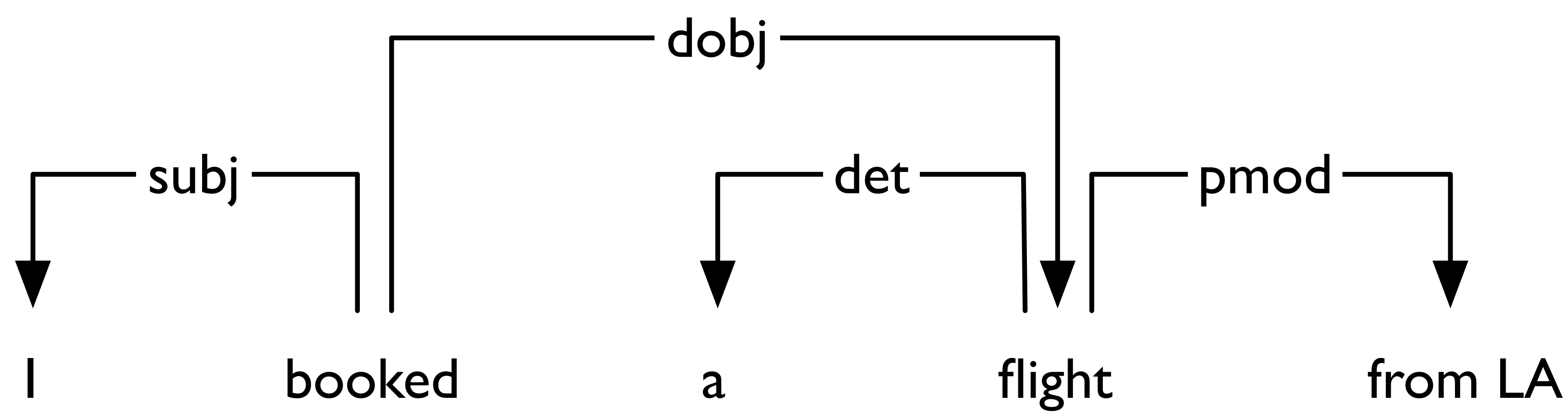
- The basic observation behind dependency is that words have grammatical functions with respect to other words in the sentence.

  Example: subject, modifier

Phrase structure trees



Dependency trees



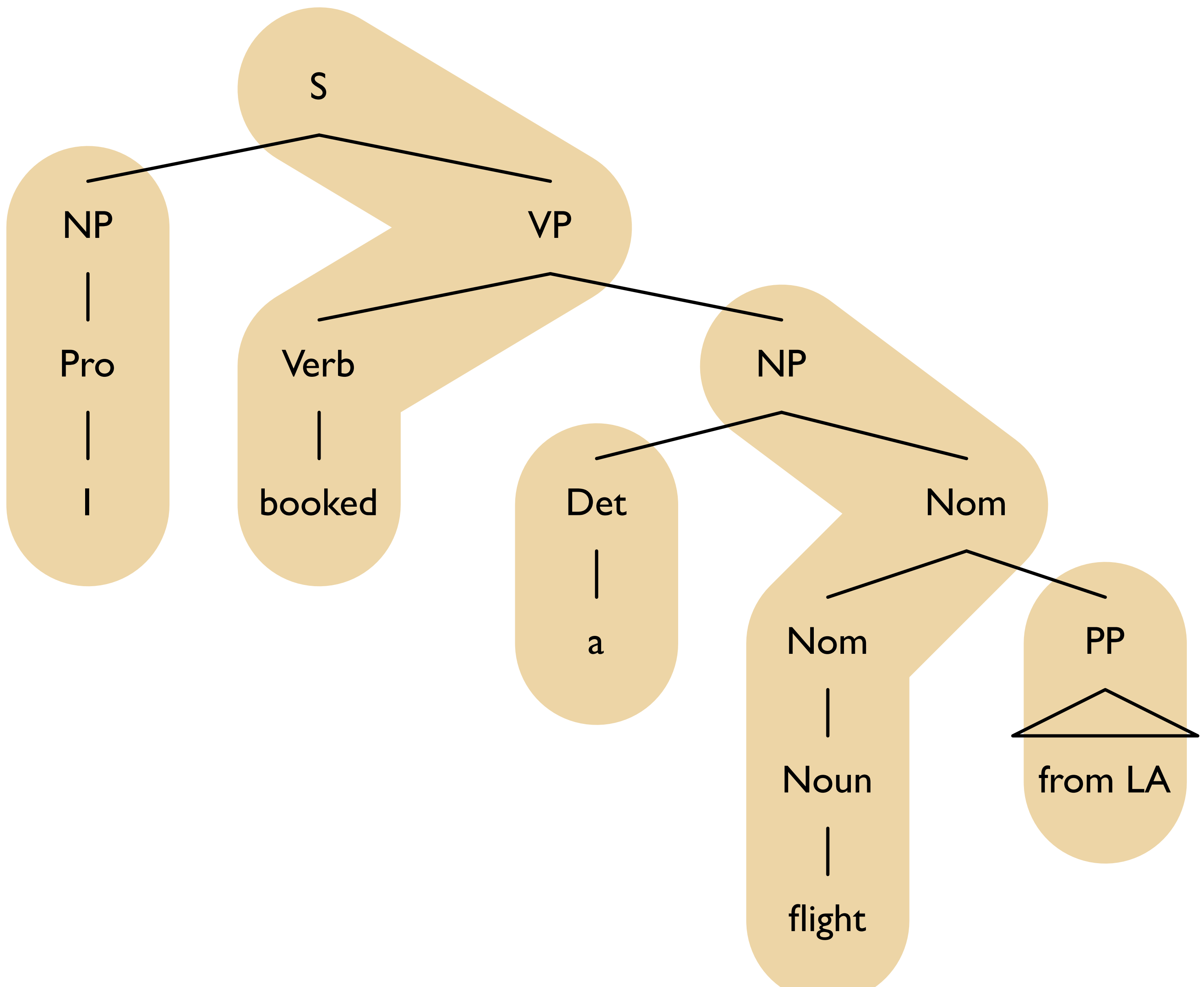
- In an arc $h \rightarrow d$, the word h is called the head, and the word d is called the dependent.
- The arcs form a rooted tree.
- Each arc has a label, l, and an arc can be described as (h, d, l)

Heads in phrase structure grammar

- In phrase structure grammar, ideas from dependency grammar can be found in the notion of heads.
- Roughly speaking, the head of a phrase is the most important word of the phrase: the word that determines the phrase function.

Examples: noun in a noun phrase,
preposition in a prepositional phrase

Heads in phrase structure grammar



The history of dependency grammar

- The notion of dependency can be found in some of the earliest formal grammars.
- Modern dependency grammar is attributed to Lucien Tesnière (1893–1954).
- Recent years have seen a revived interest in dependency-based description of natural language syntax.

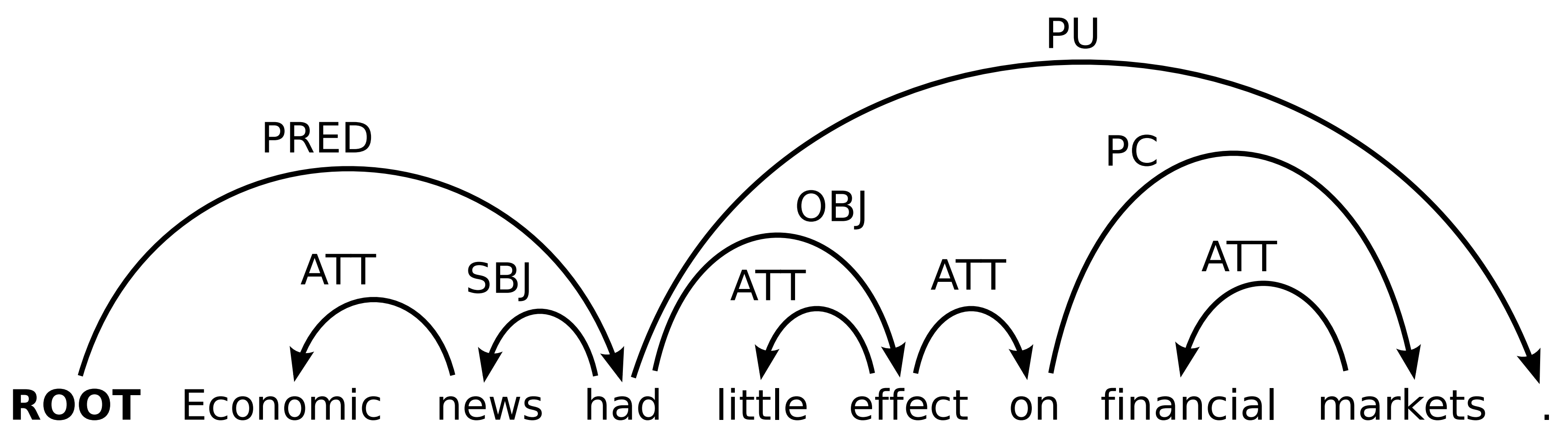
Linguistic resources

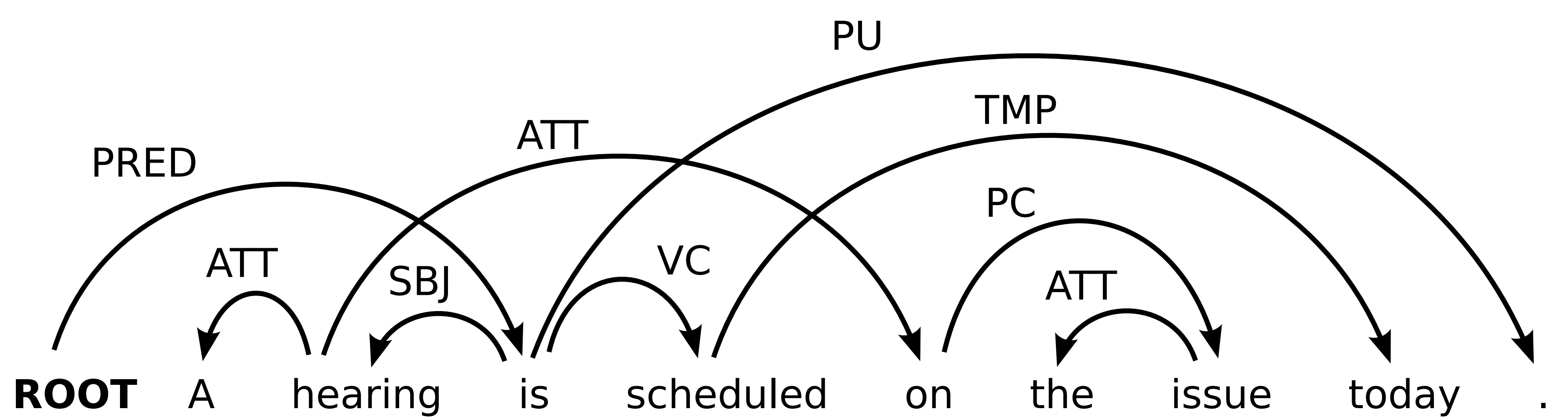
- Descriptive dependency grammars exist for some natural languages.
- Dependency treebanks exist for a wide range of natural languages.
- These treebanks can be used to train accurate and efficient dependency parsers.

Projectivity

- An important characteristic of dependency trees is projectivity
- A dependency tree is projective if:
  - For every arc in the tree, there is a directed path from the head of the arc to all words occurring between the head and the dependent (that is, the arc (i,l,j) implies that $i \to^* k$ for every k such that min(i, j) < k < max(i, j))

Projective and non-projective trees





Projectivity and dependency parsing

- Many dependency parsing algorithms can only handle projective trees
- Non-projective trees do occur in natural language
  - How often depends on the language (and treebank)

UPPSALA UNIVERSITET

Projectivity in the course

- The algorithms we will discuss in detail during the lectures will only concern projective parsing
- Non-projective parsing:
  - Seminar 2: Pseudo-projective parsing
  - Other variants mentioned briefly during the lectures
  - You can read more about it in the course book!

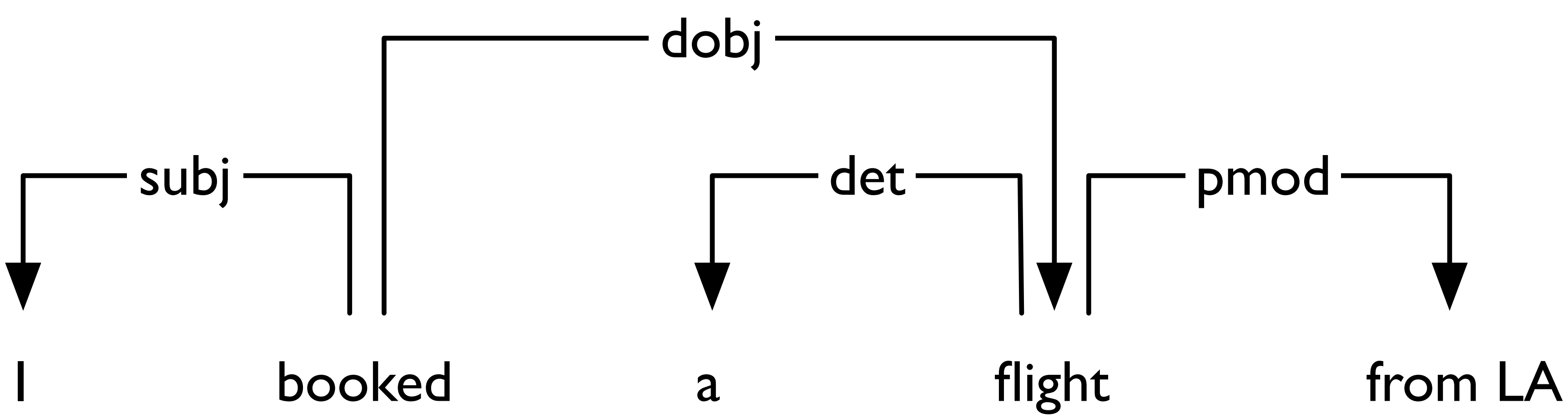
UPPSALA
UNIVERSITET

Arc-factored dependency parsing

UPPSALA UNIVERSITET

Ambiguity

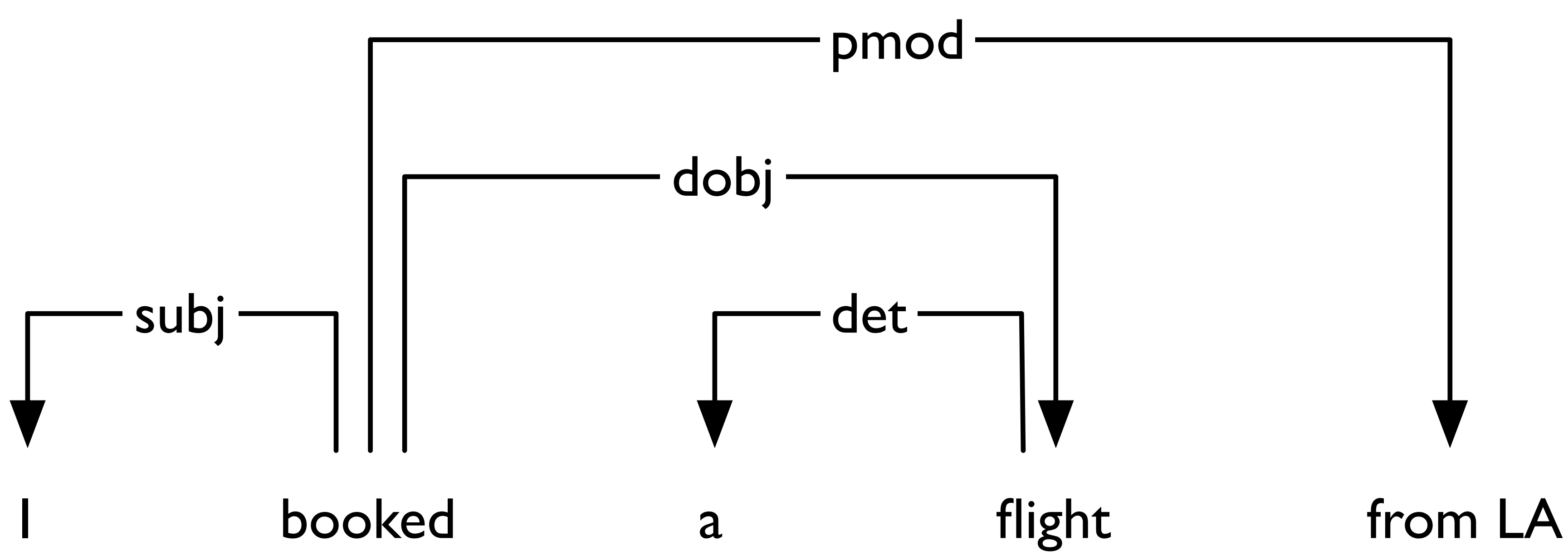
Just like phrase structure parsing, dependency parsing has to deal with ambiguity.



UPPSALA UNIVERSITET

Ambiguity

Just like phrase structure parsing,
dependency parsing has to deal with ambiguity.



UPPSALA UNIVERSITET

Disambiguation

- We need to disambiguate between alternative analyses.
- We develop mechanisms for scoring dependency trees, and disambiguate by choosing a dependency tree with the highest score.

UPPSALA UNIVERSITET

Scoring models and parsing algorithms

Distinguish two aspects:

- Scoring model:
  How do we want to score dependency trees?
- Parsing algorithm:
  How do we compute a highest-scoring dependency tree under the given scoring model?

UPPSALA UNIVERSITET

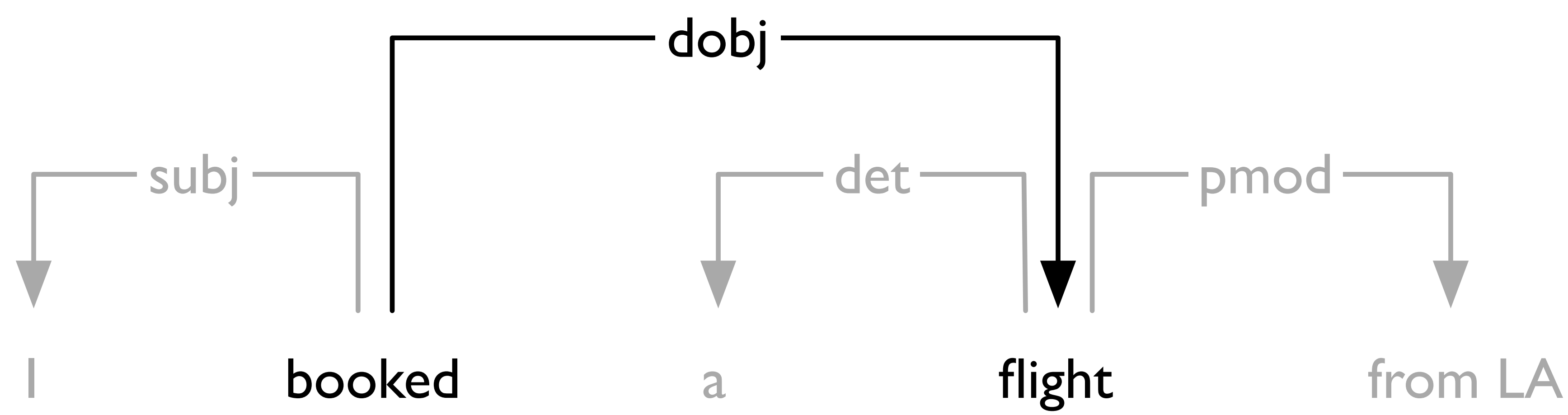
The arc-factored model

- Split the dependency tree t into parts $p_1, \ldots, p_n$, score each of the parts individually, and combine the score into a simple sum.

  $$\text{score}(t) = \text{score}(p_1) + \ldots + \text{score}(p_n)$$

- The simplest scoring model is the arc-factored model, where the scored parts are the arcs of the tree.

Features



- To score an arc, we define features that are likely to be relevant in the context of parsing.
- We represent an arc by its feature vector.

Examples of features

Examples of features

- ‘The head is a verb.’

Examples of features

- ‘The head is a verb.’
- ‘The dependent is a noun.’

Examples of features

- ‘The head is a verb.’
- ‘The dependent is a noun.’
- ‘The head is a verb
  and the dependent is a noun.’

Examples of features

- ‘The head is a verb.’
- ‘The dependent is a noun.’
- ‘The head is a verb
  and the dependent is a noun.’
- ‘The head is a verb
  and the predecessor of the head is a pronoun.’

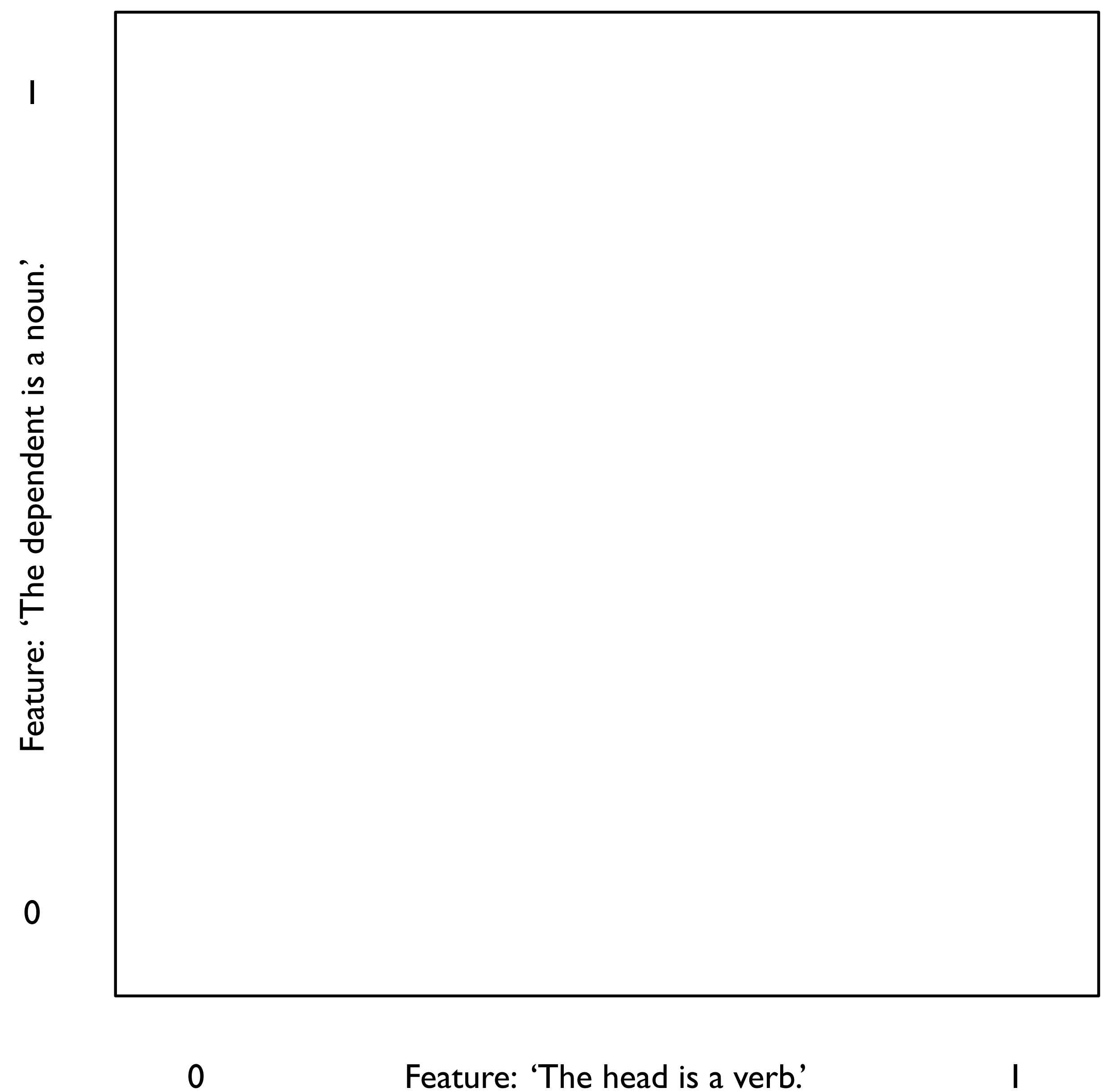
Examples of features

- ‘The head is a verb.’
- ‘The dependent is a noun.’
- ‘The head is a verb
  and the dependent is a noun.’
- ‘The head is a verb
  and the predecessor of the head is a pronoun.’
- ‘The arc goes from left to right.’

Examples of features

- ‘The head is a verb.’
- ‘The dependent is a noun.’
- ‘The head is a verb
  and the dependent is a noun.’
- ‘The head is a verb
  and the predecessor of the head is a pronoun.’
- ‘The arc goes from left to right.’
- ‘The arc has length 2.’

Feature vectors



Arc-factored dependency parsing

Feature vectors

Implementation of feature vectors

- We assign each feature a unique number.
- For each arc, we collect the numbers of those features that apply to that arc.
- The feature vector of the arc is the list of those numbers.

Example: [1, 2, 42, 313, 1977, 2008, 2010]

Feature weights

- Arc-factored dependency parsers require a training phase.
- During training, our goal is to assign, to each feature f_i, a feature weight w_i.
- Intuitively, the weight w_i quantifies the effect of the feature f_i on the likelihood of the arc.

How likely is it that we will see an arc with this feature in a useful dependency tree?

UPPSALA UNIVERSITET

Feature weights

We define the score of an arc $h \rightarrow d$ as the weighted sum of all features of that arc:

$$\text{score}(h \rightarrow d) = f_1 w_1 + \ldots + f_n w_n$$

Training using structured prediction

- Take a sentence *w* and a gold-standard dependency tree *g* for *w*.
- Compute the highest-scoring dependency tree under the current weights; call it *p*.
- Increase the weights of all features that are in *g* but not in *p*.
- Decrease the weights of all features that are in *p* but not in *g*.

Training using structured prediction

- Training involves repeatedly parsing (treebank) sentences and refining the weights.
- Hence, training presupposes an efficient parsing algorithm.

Higher order models

- The arc-factored model is a first-order model, because scored subgraphs consist of a single arc.
- An nth-order model scores subgraphs consisting of (at most) n arcs.
- Second-order: siblings, grand-parents
- Third-order: tri-siblings, grand-siblings
- Higher-order models capture more linguistic structure and give higher parsing accuracy, but are less efficient

Parsing algorithms

- Projective parsing
  - Inspired by the CKY algorithm
    - Collins’ algorithm
    - Eisner’s algorithm
- Non-projective parsing:
  - Minimum spanning tree (MST) algorithms

Graph-based parsing

- Arc-factored parsing is an instance of graph-based dependency parsing
- Because it scores the dependency graph (tree)
- Graph-based models are often contrasted with transition-based models (next Wednesday)
- There are also grammar-based methods, which we will not discuss

UPPSALA UNIVERSITET

Summary

- The term ‘arc-factored dependency parsing’ refers to dependency parsers that score a dependency tree by scoring its arcs.
- Arcs are scored by defining features and assigning weights to these features.
- The resulting parsers can be trained using structured prediction.
- More powerful scoring models exist.

UPPSALA UNIVERSITET

Overview

- Arc-factored dependency parsing
  - Collins' algorithm
  - Eisner's algorithm
- Transition-based dependency parsing
  - The arc-standard algorithm
- Dependency treebanks
- Evaluation of dependency parsers

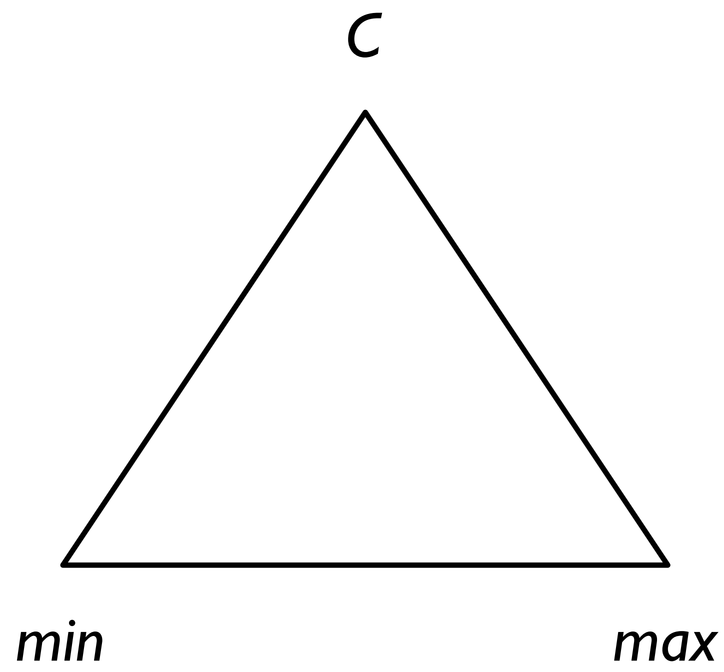
Collins’ algorithm

UPPSALA UNIVERSITET

Collins' algorithm

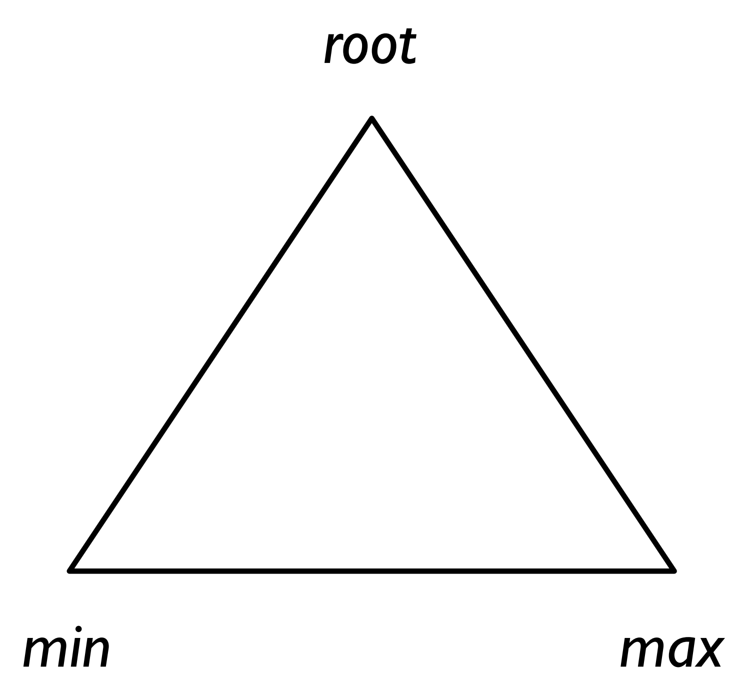
- Collin's algorithm is a simple algorithm for computing the highest-scoring dependency tree under an arc-factored scoring model.
- It can be understood as an extension of the CKY algorithm to dependency parsing.
- Like the CKY algorithm, it can be characterized as a bottom-up algorithm based on dynamic programming.

Signatures, CKY



$[min, max, C]$

Signatures, Collins'



[*min*, *max*, *root*]

Initialization

I	booked	a	flight	from LA

0 1 2 3 4 5

UPPSALA UNIVERSITET

Initialization

I	booked	a	flight	from LA
[0, 1, I]	[1, 2, booked]	[2, 3, a]	[3, 4, flight]	[4, 5, from LA]

0 1 2 3 4 5

Adding a left-to-right arc

I	booked	a	flight	from LA

0 1 2 3 4 5

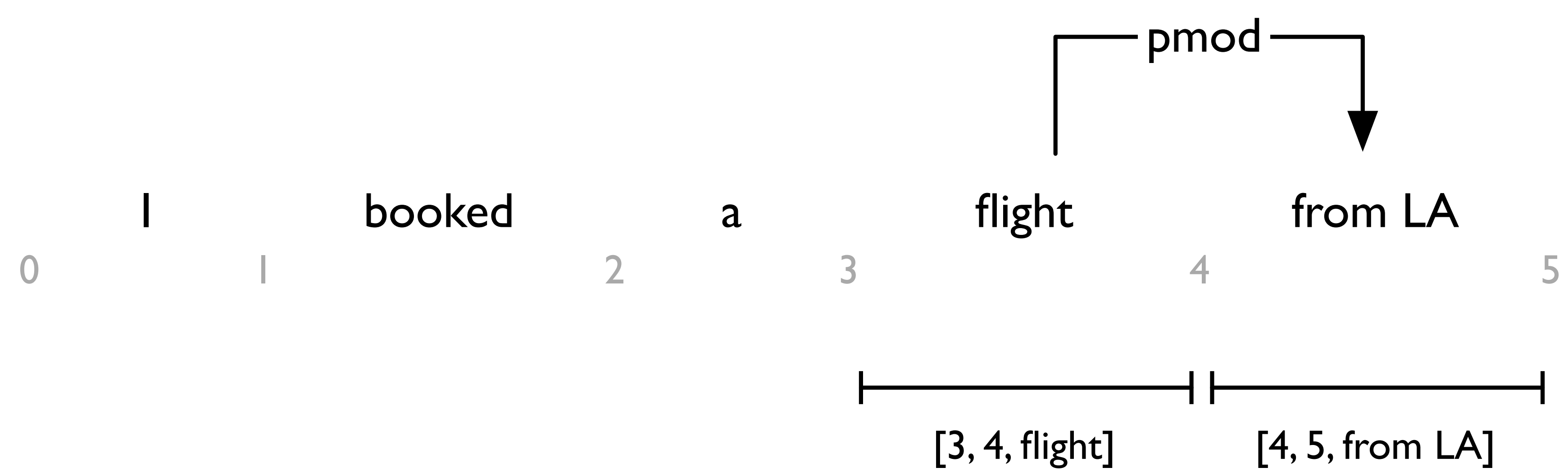
Adding a left-to-right arc

I booked a flight from LA

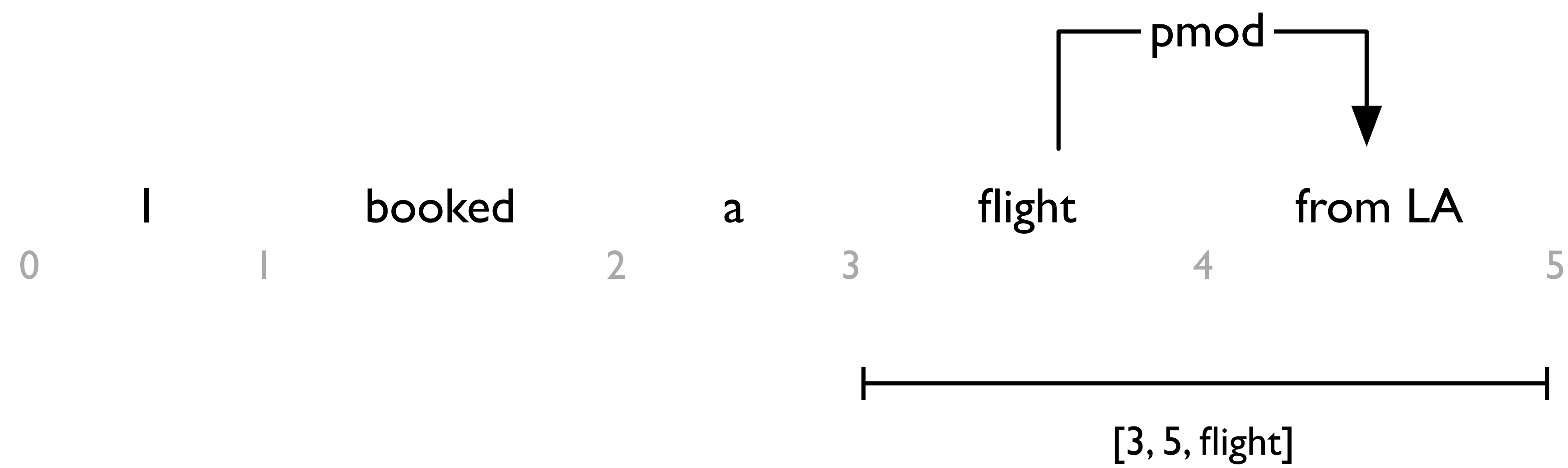
0 1 2 3 4 5

[3, 4, flight] [4, 5, from LA]

Adding a left-to-right arc

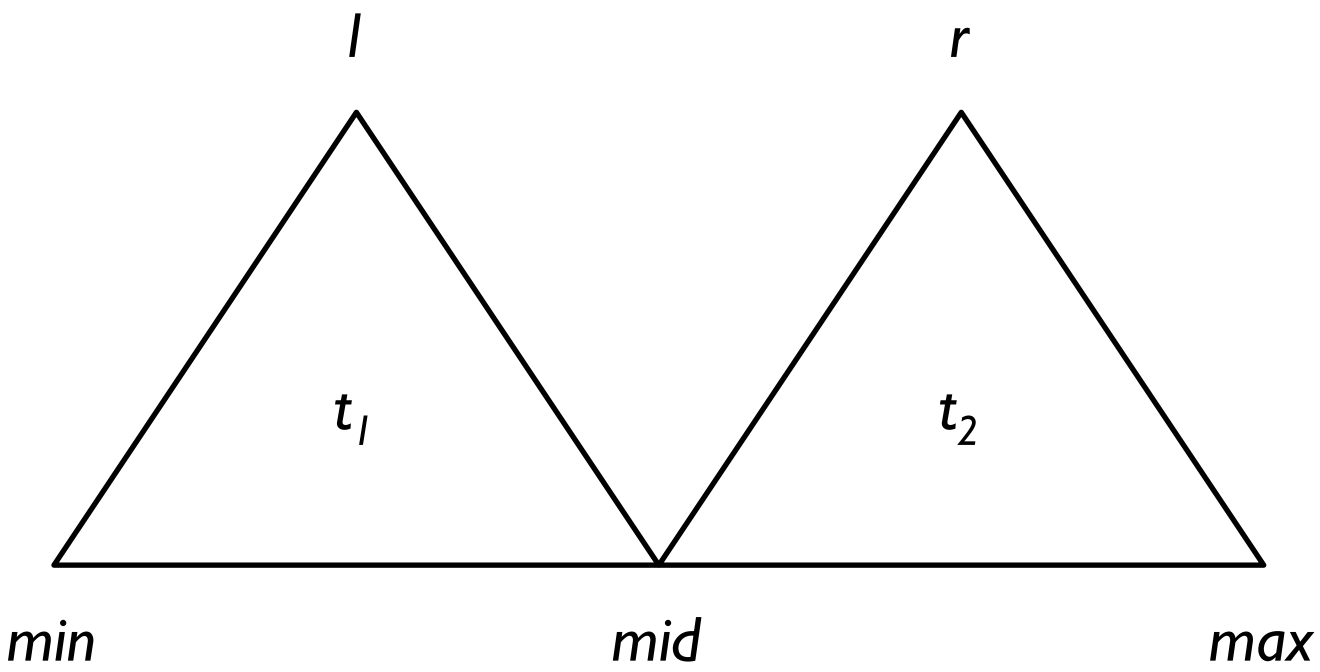


Adding a left-to-right arc

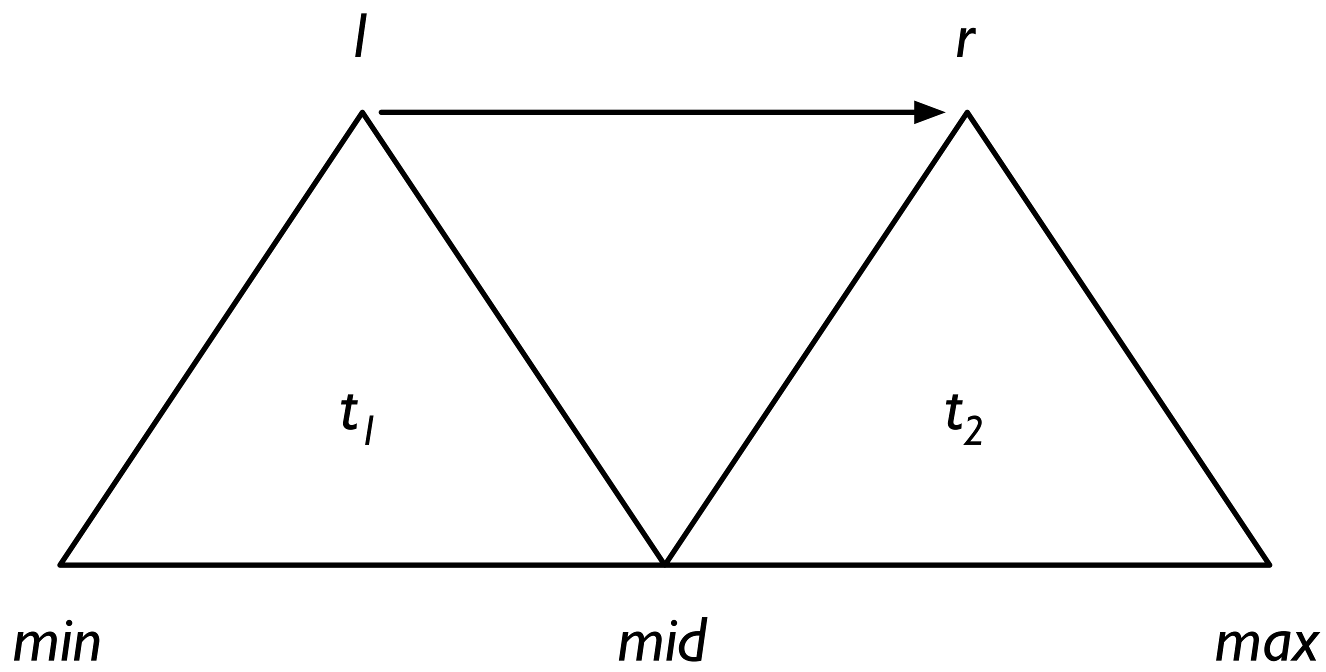


Adding a left-to-right arc

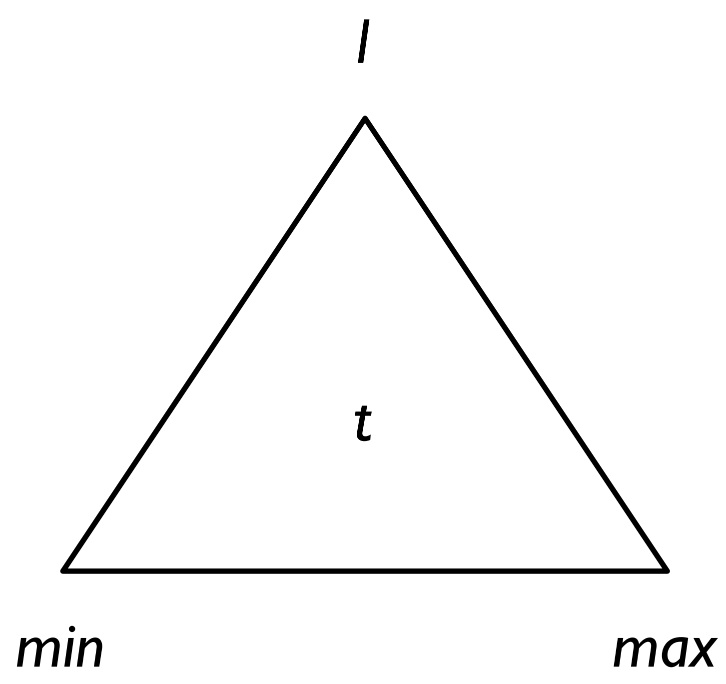
Adding a left-to-right arc



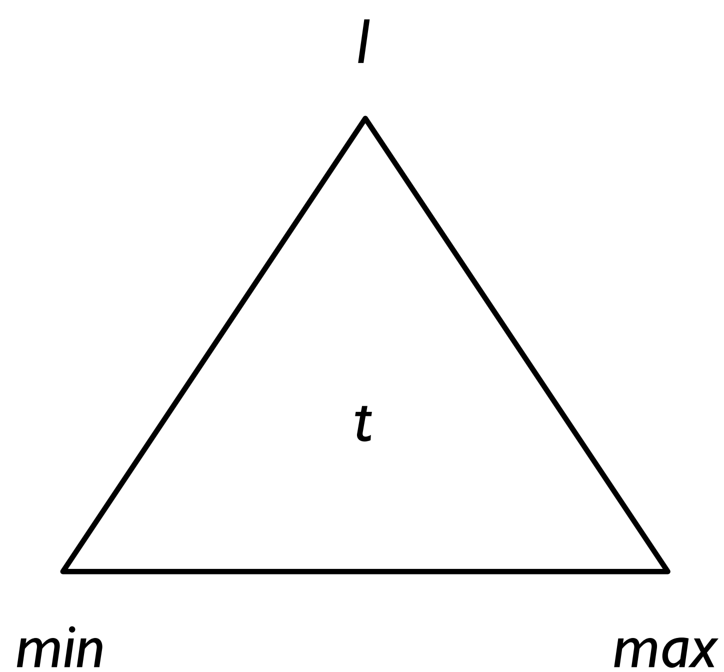
Adding a left-to-right arc



Adding a left-to-right arc



Adding a left-to-right arc

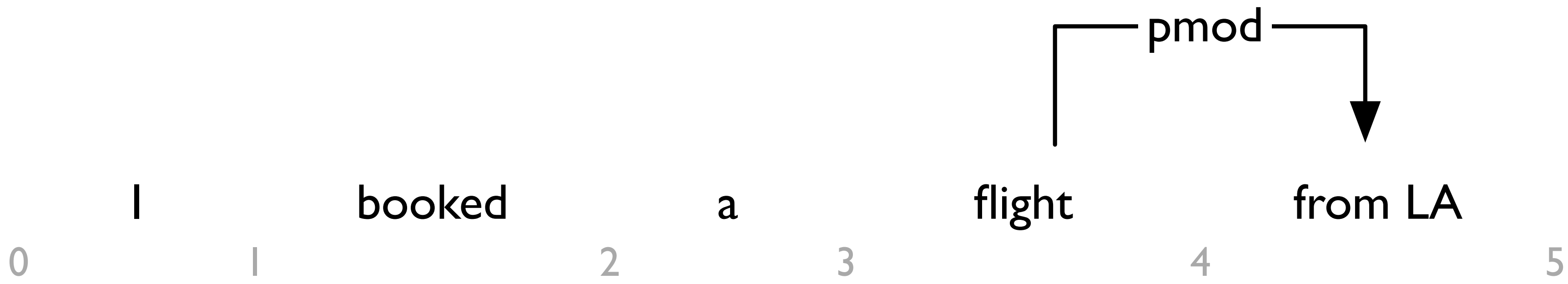


$$\text{score}(t) = \text{score}(t_1) + \text{score}(t_2) + \text{score}(l \rightarrow r)$$

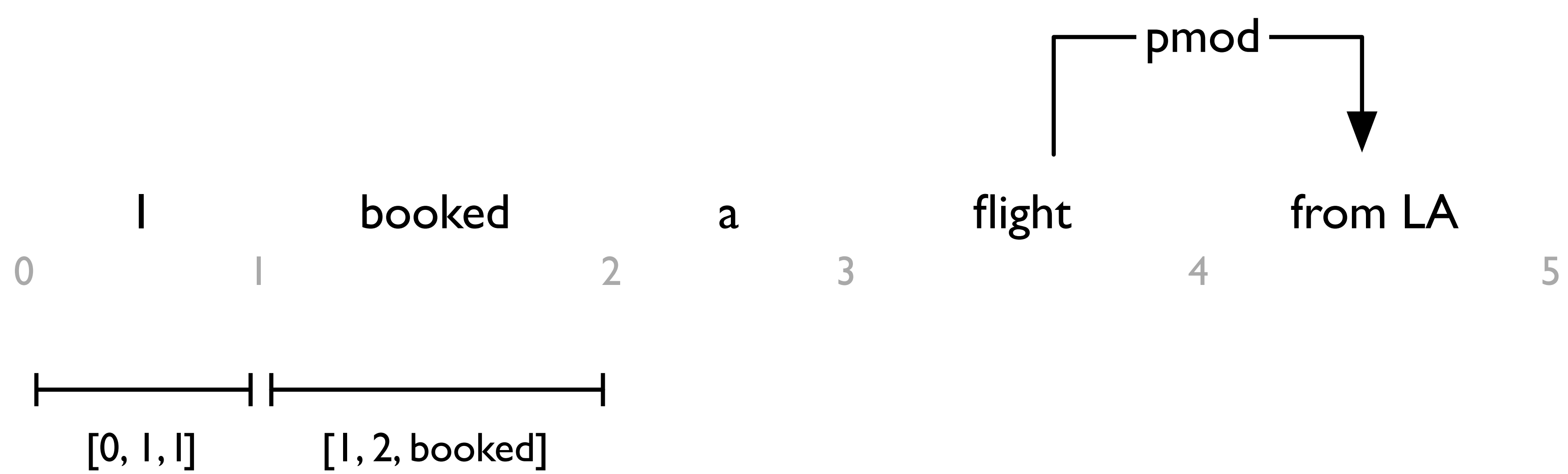
Adding a left-to-right arc

```
for each [min, max] with max - min > 1 do
  for each l from min to max - 2 do
    double best = score[min][max][l]
    for each r from l + 1 to max - 1 do
      for each mid from l + 1 to r do
        t₁ = score[min][mid][l]
        t₂ = score[mid][max][r]
        double current = t₁ + t₂ + score(l → r)
        if current > best then
          best = current
    score[min][max][l] = best
```

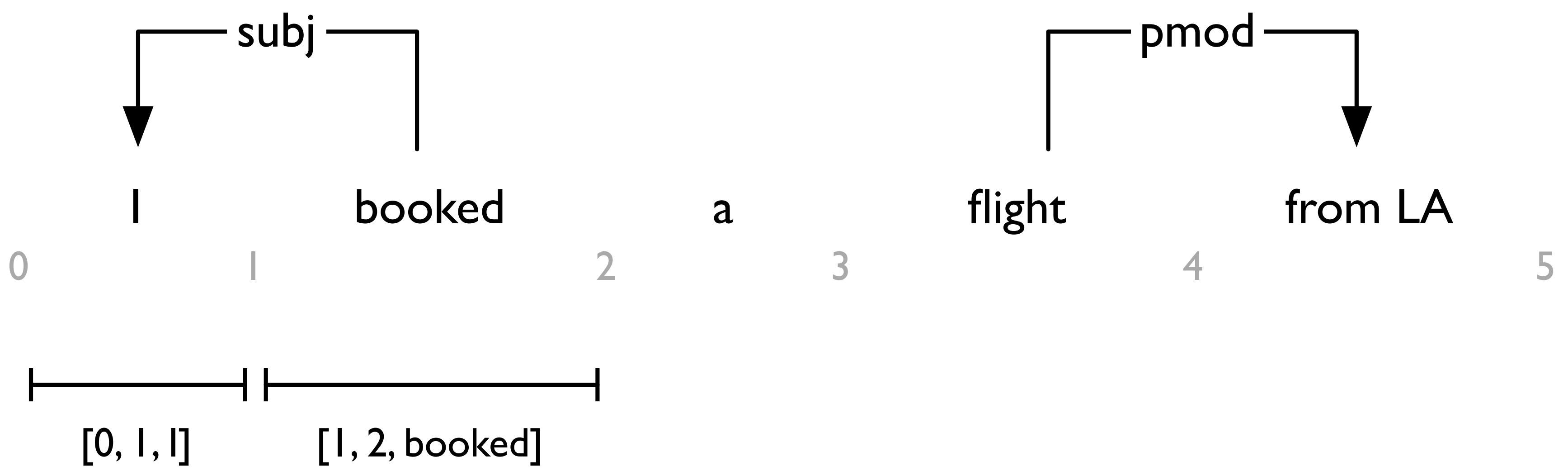
Adding a right-to-left arc



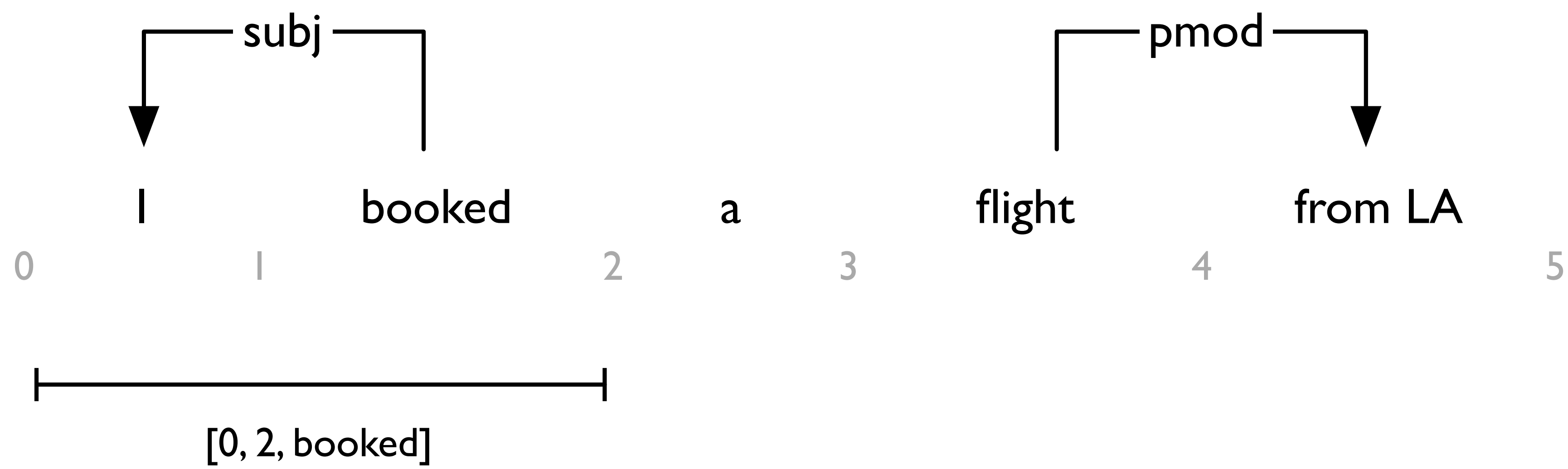
Adding a right-to-left arc



Adding a right-to-left arc

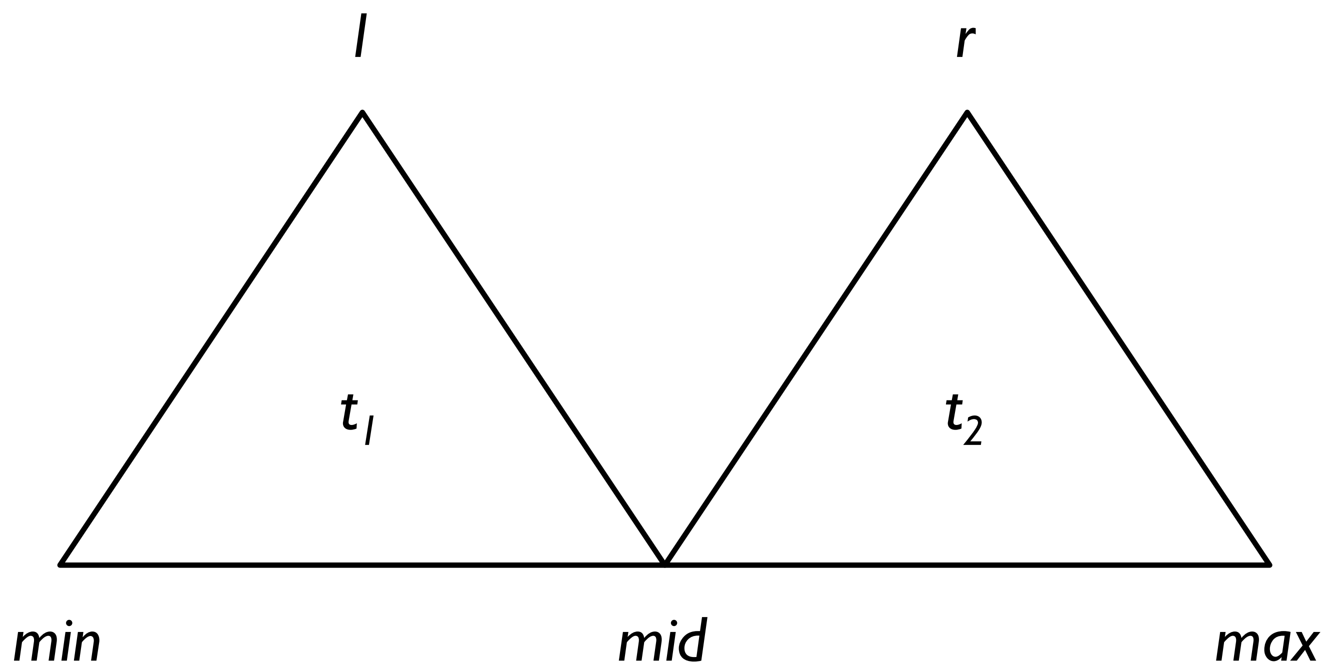


Adding a right-to-left arc

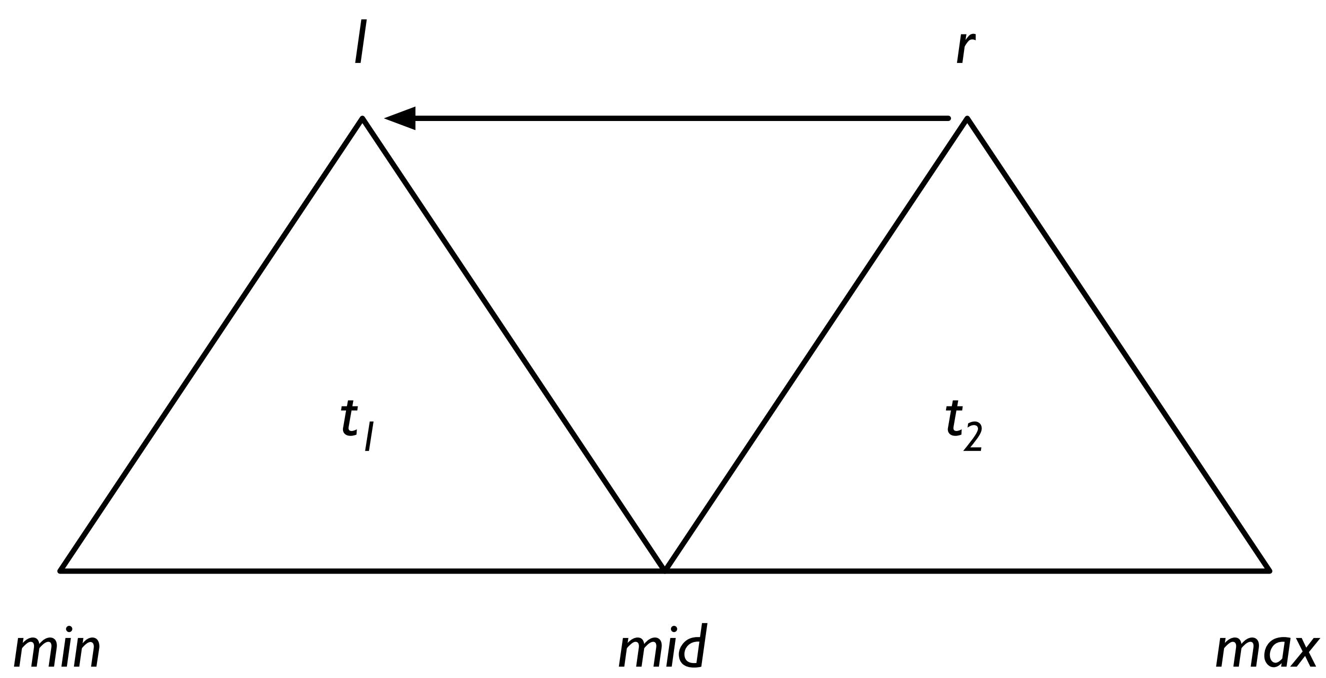


Adding a right-to-left arc

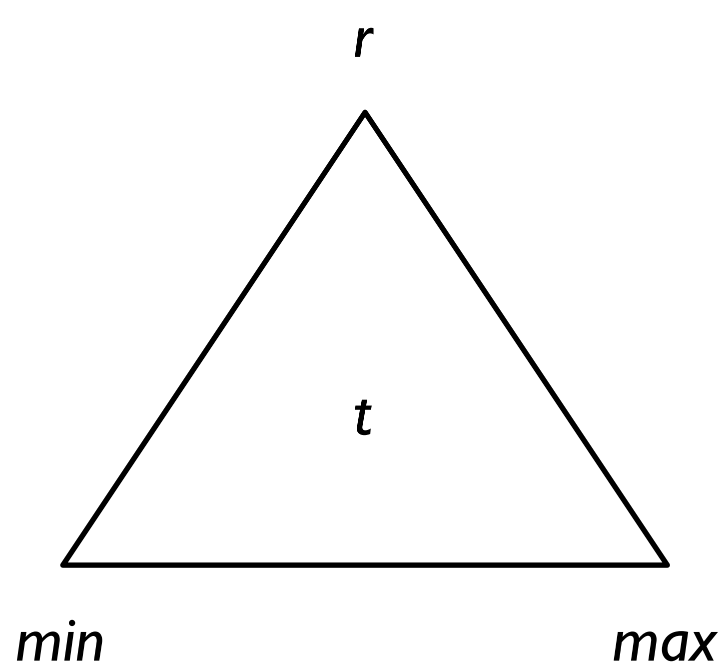
Adding a right-to-left arc



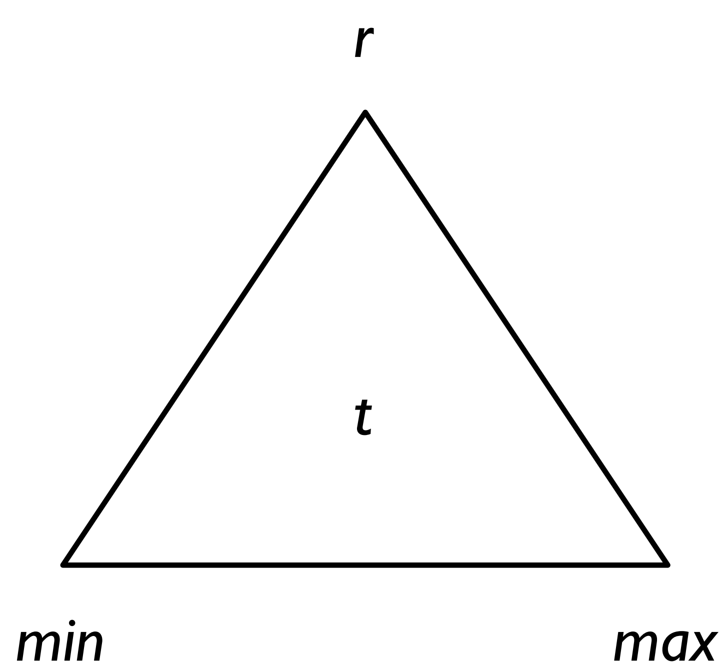
Adding a right-to-left arc



Adding a right-to-left arc



Adding a right-to-left arc



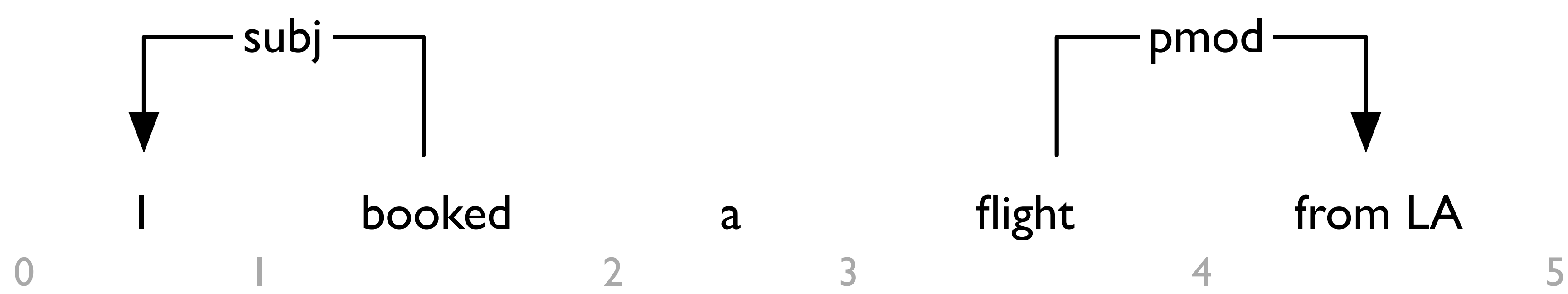
$$\text{score}(t) \;=\; \text{score}(t_1) + \text{score}(t_2) + \text{score}(r \rightarrow l)$$

Adding a right-to-left arc

```
for each [min, max] with max - min > 1 do
  for each r from min + 1 to max - 1 do
    double best = score[min][max][r]
    for each l from min to r - 1 do
      for each mid from l + 1 to r do
        t1 = score[min][mid][l]
        t2 = score[mid][max][r]
        double current = t1 + t2 + score(r → l)
        if current > best then
          best = current
    score[min][max][r] = best
```

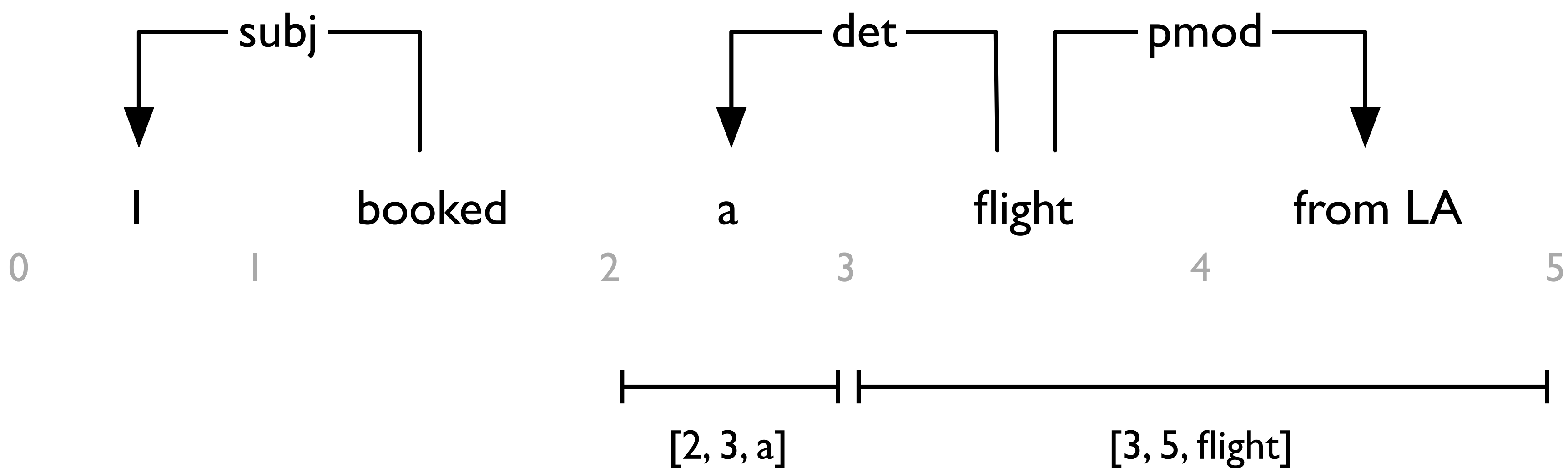
Collins' algorithm

Finishing up

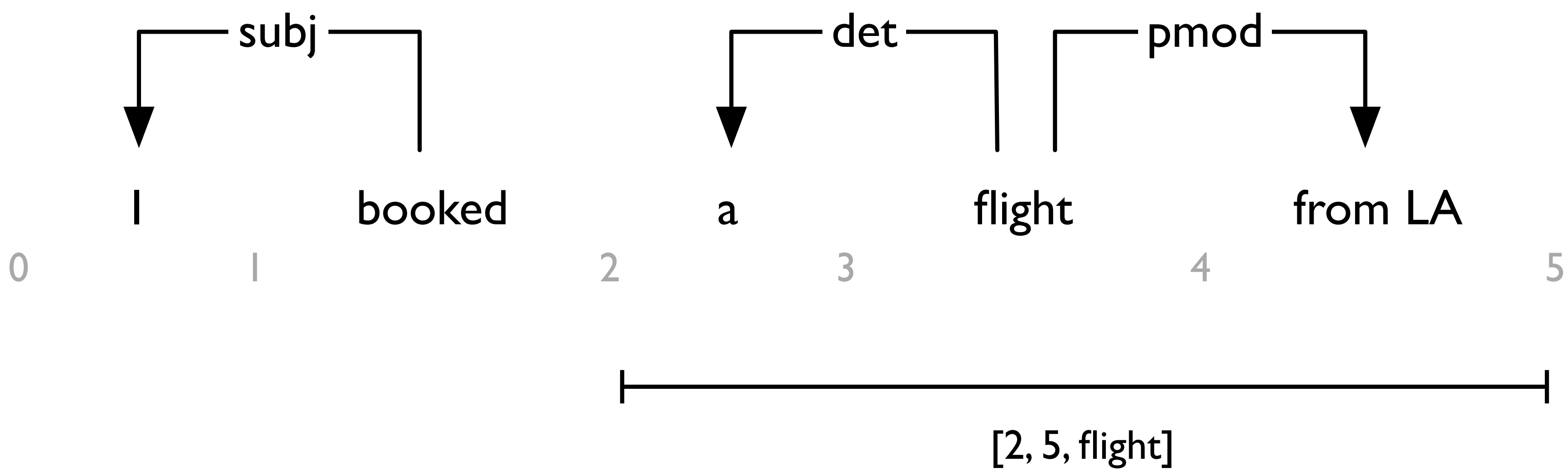


Collins' algorithm

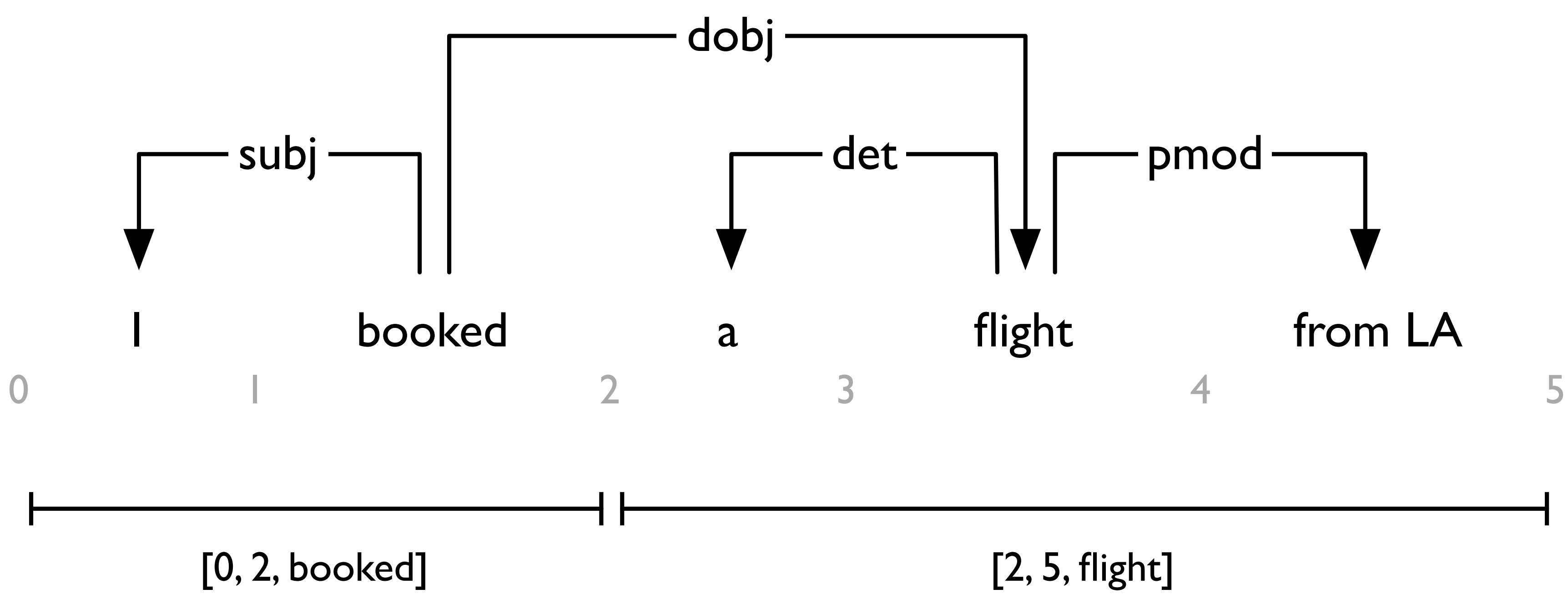
Finishing up



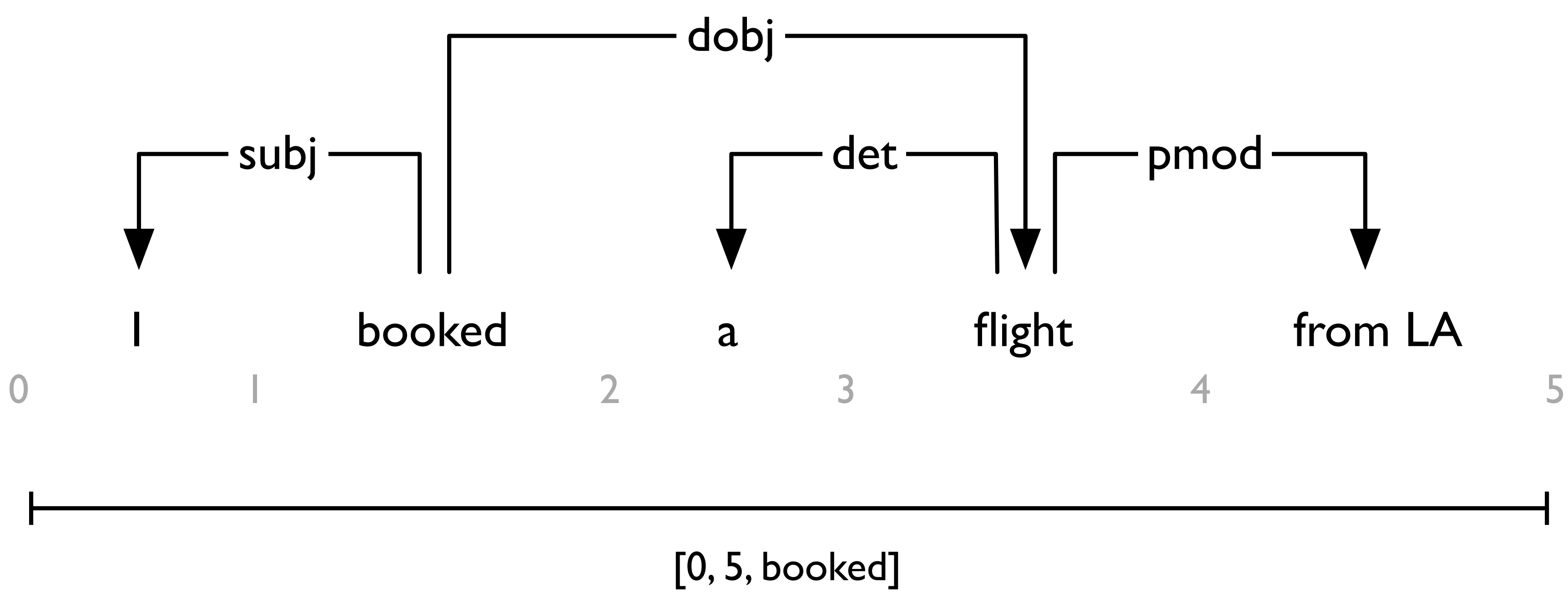
Finishing up



Finishing up



Finishing up



Collins' algorithm

Complexity analysis

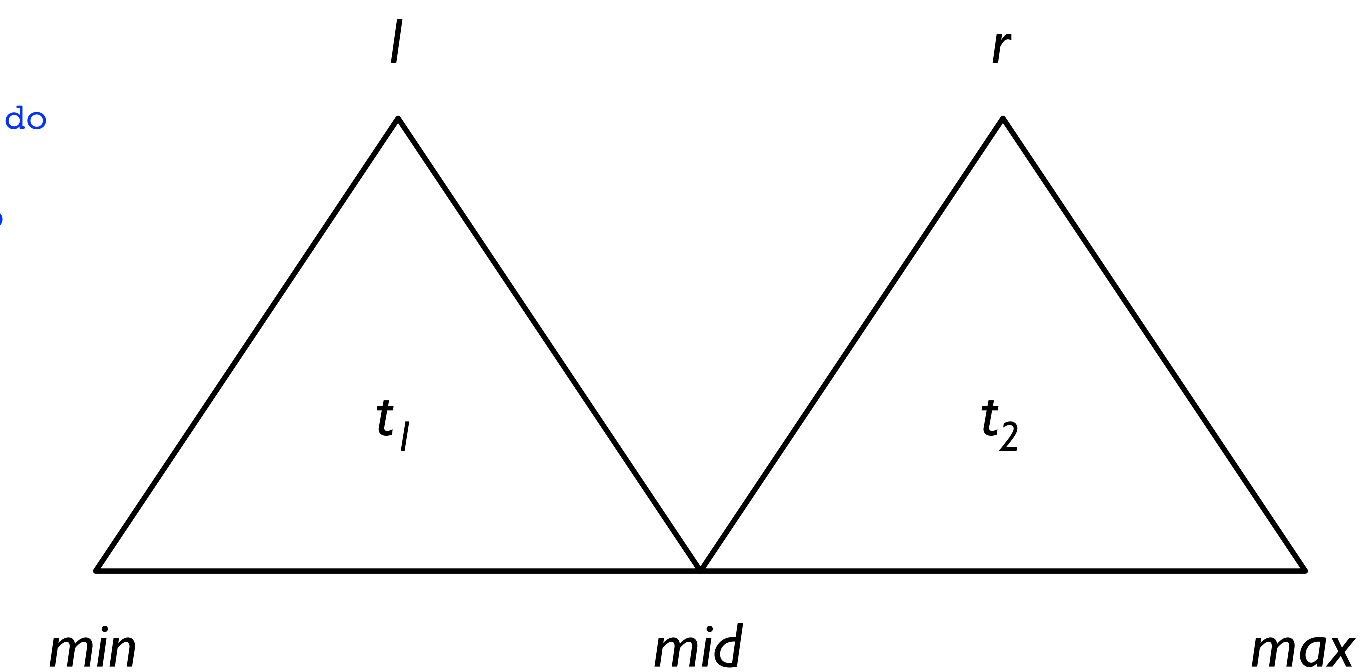
- Runtime?
- Space?

```
for each [min, max] with max - min > 1 do
  for each r from min + 1 to max - 1 do
    double best = score[min][max][r]
    for each l from min to r - 1 do
      for each mid from l + 1 to r do
        t₁ = score[min][mid][l]
        t₂ = score[mid][max][r]
        double current = t₁ + t₂ + score(r → l)
        if current > best then
          best = current
    score[min][max][r] = best
```

Complexity analysis

- Runtime?
- Space?

```
for each [min, max] with max - min > 1 do
  for each r from min + 1 to max - 1 do
    double best = score[min][max][r]
    for each l from min to r - 1 do
      for each mid from l + 1 to r do
        t₁ = score[min][mid][l]
        t₂ = score[mid][max][r]
        double current = t₁ + t₂ + score(r → l)
        if current > best then
          best = current
    score[min][max][r] = best
```

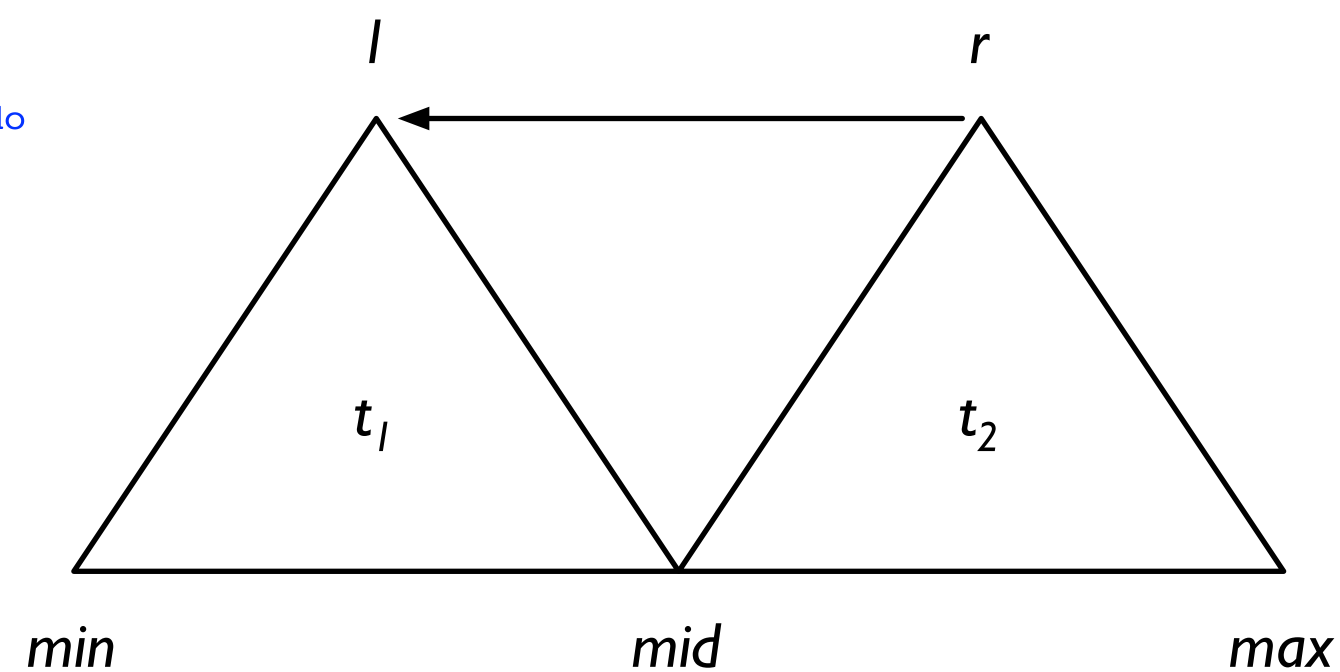


Collins' algorithm

Complexity analysis

- Runtime?
- Space?

```
for each [min, max] with max - min > 1 do
  for each r from min + 1 to max - 1 do
    double best = score[min][max][r]
    for each l from min to r - 1 do
      for each mid from l + 1 to r do
        t₁ = score[min][mid][l]
        t₂ = score[mid][max][r]
        double current = t₁ + t₂ + score(r → l)
        if current > best then
          best = current
    score[min][max][r] = best
```



Complexity analysis

- Space requirement:
  $O(|w|^3)$
- Runtime requirement:
  $O(|w|^5)$

Summary

- Collins' algorithm is a CKY-style algorithm for computing the highest-scoring dependency tree under an arc-factored scoring model.
- It runs in time $O(|w|^5)$.
  This may not be practical for long sentences.
- We have not discussed labels yet - we will do that next lecture