UPPSALA UNIVERSITET

Graph-based dependency parsing

Syntactic analysis (5LN455)

2024

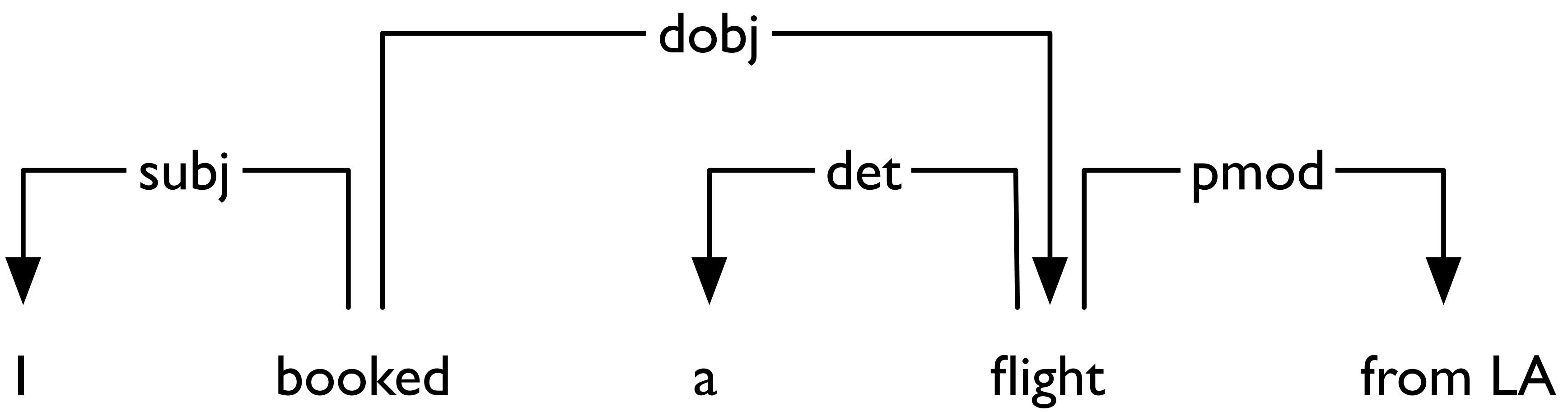
Sara Stymne
Department of Linguistics and Philology

Partially based on slides from Marco Kuhlmann

Arc-factored dependency parsing

Ambiguity

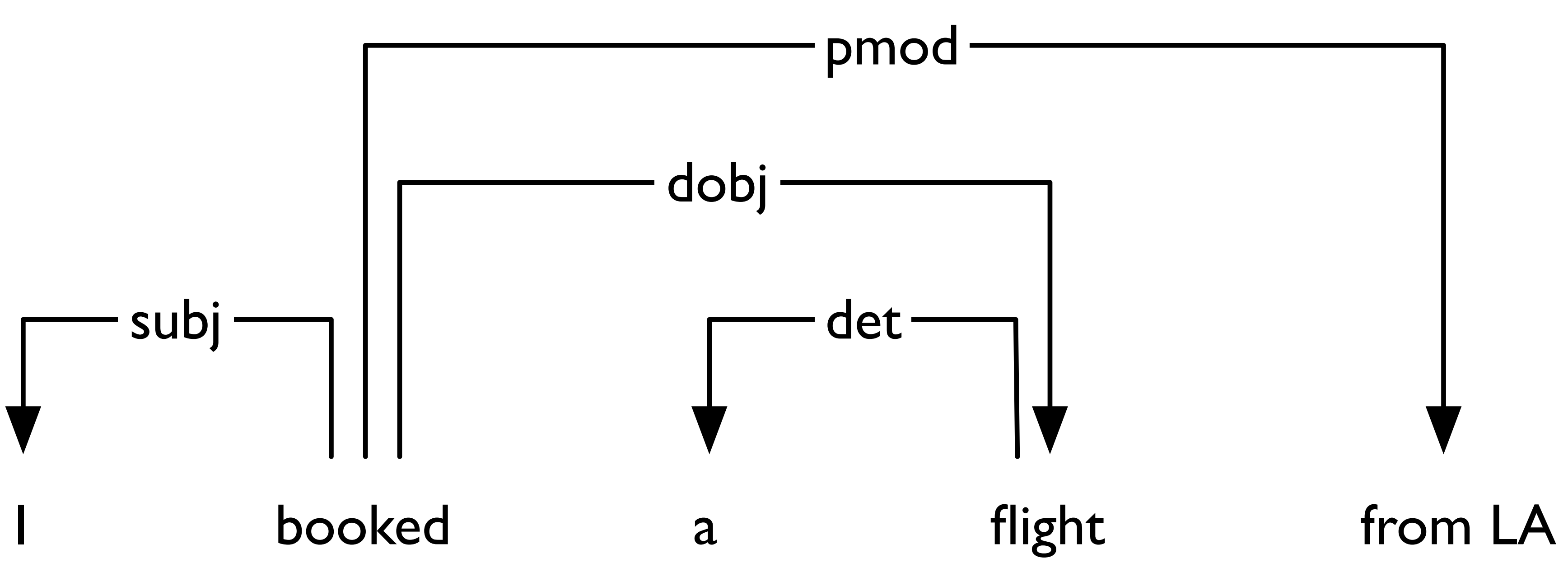
Just like phrase structure parsing, dependency parsing has to deal with ambiguity.



UPPSALA UNIVERSITET

Ambiguity

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



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?

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.

Examples of classic 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.’

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
    - e.g. Chu-Liu-Edmunds algorithm (CLE)

UPPSALA
UNIVERSITET

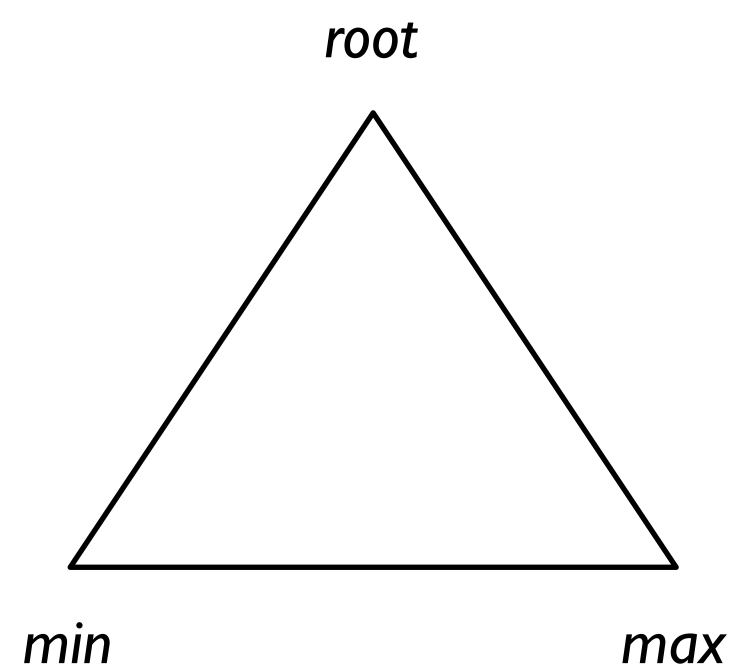
Collins’ algorithm

UPPSALA UNIVERSITET

Collins' algorithm

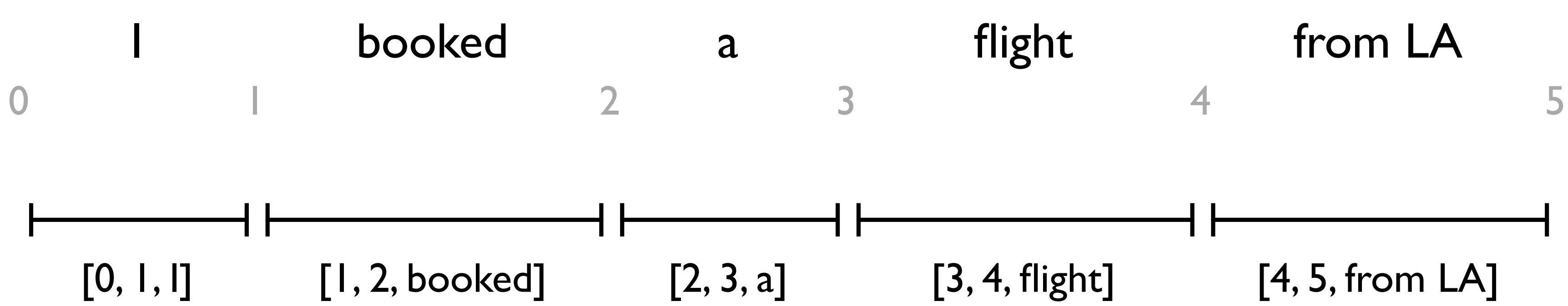
- Collin's algorithm is a simple algorithm for computing the highest-scoring dependency tree under an arc-factored scoring model.
- It can be understood as an extension of the CKY algorithm to dependency parsing.
- Like the CKY algorithm, it can be characterized as a bottom-up algorithm based on dynamic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

Signatures, Collins’

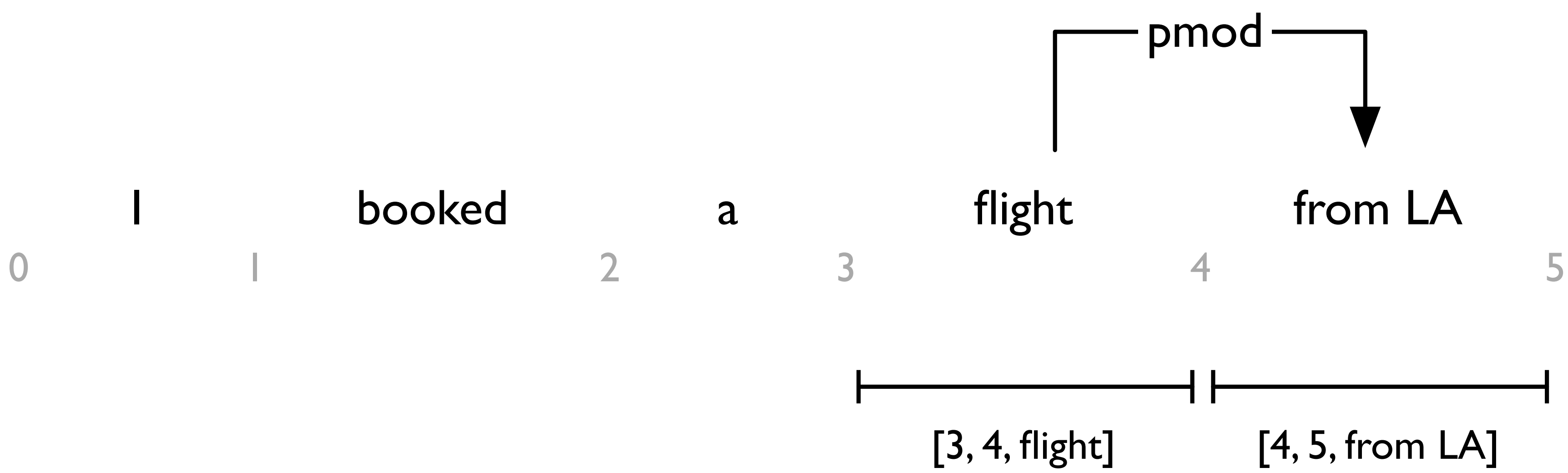


[*min*, *max*, *root*]

Initialization

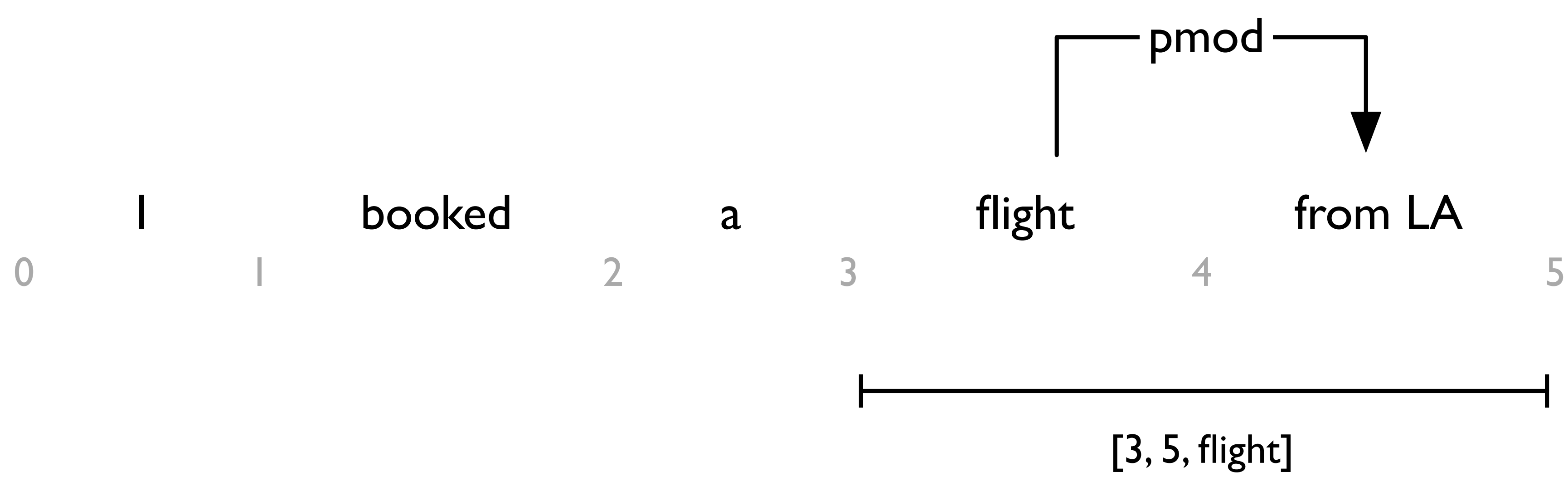


Adding a left-to-right arc

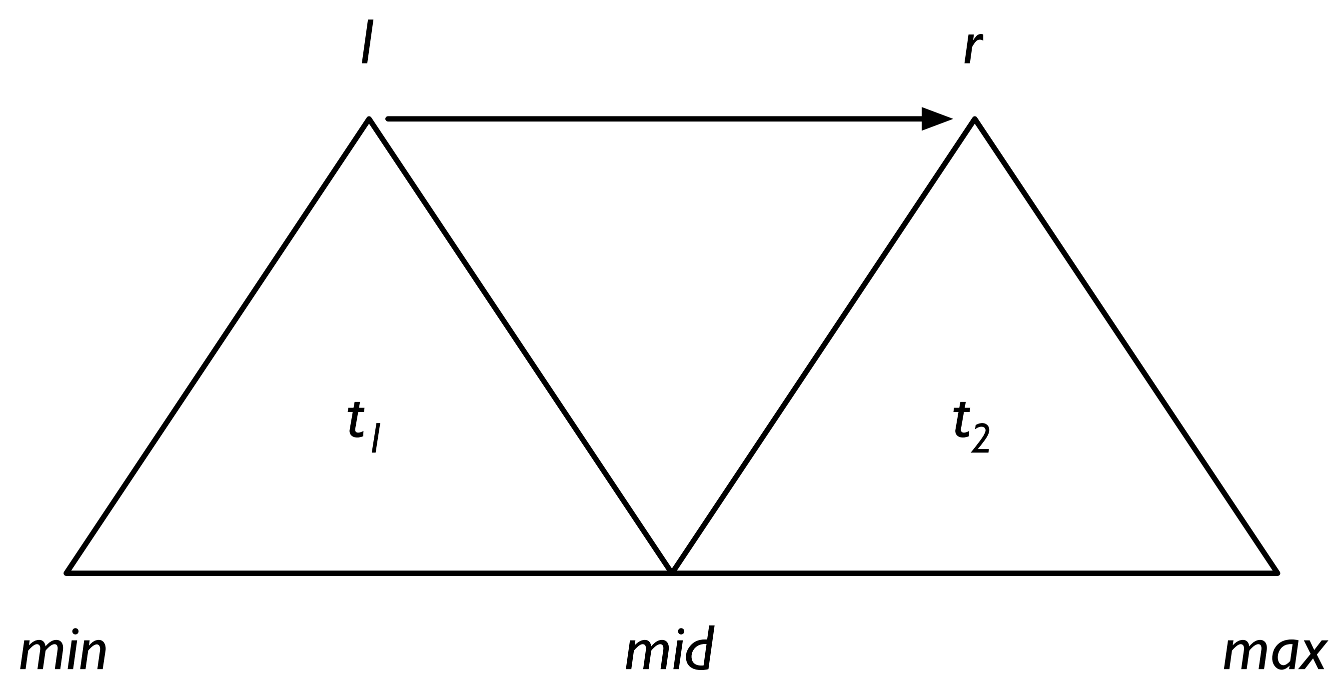


Collins' algorithm

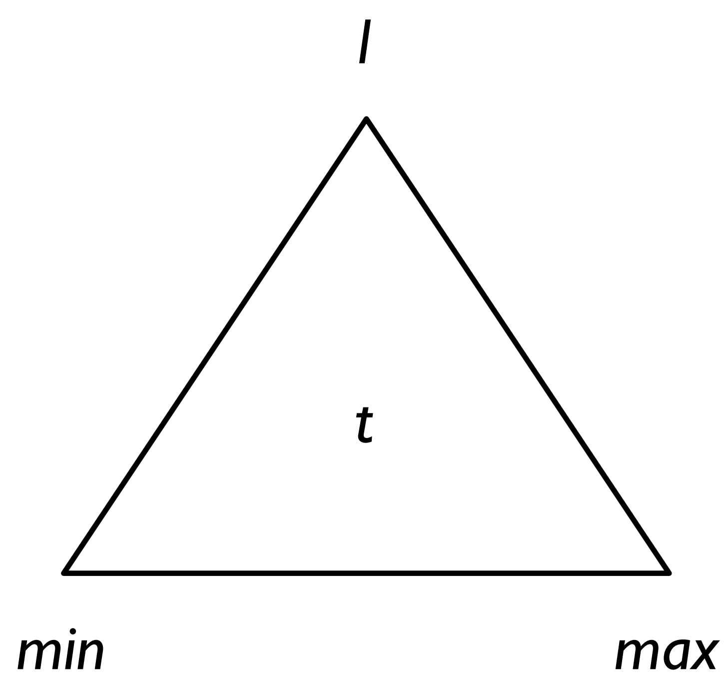
Adding a left-to-right arc



Adding a left-to-right arc



Adding a left-to-right arc



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

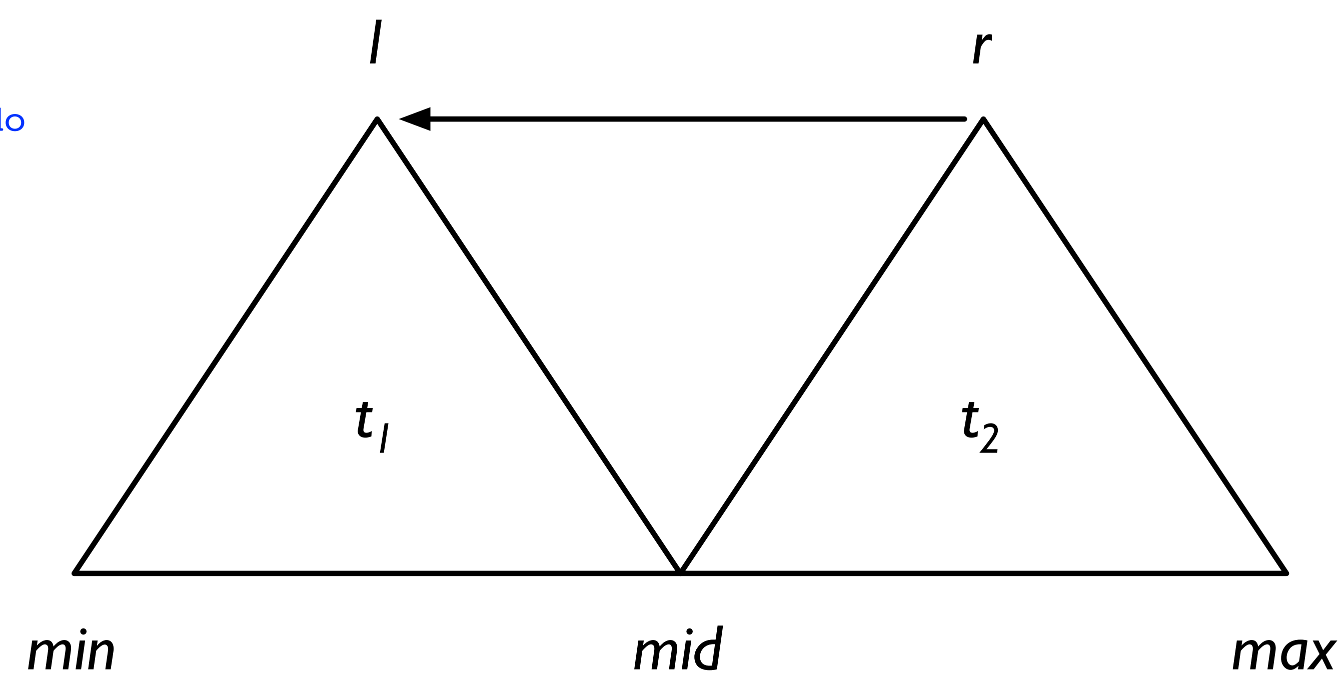
Adding a left-to-right arc

```
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
```

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:
  $O(|w|^5)$

Extension to the labeled case

- It is important to distinguish dependencies of different types between the same two words.

  Example: subj, dobj

- For this reason, practical systems typically deal with labeled arcs.
- The question then arises how to extend Collins' algorithm to the labeled case.

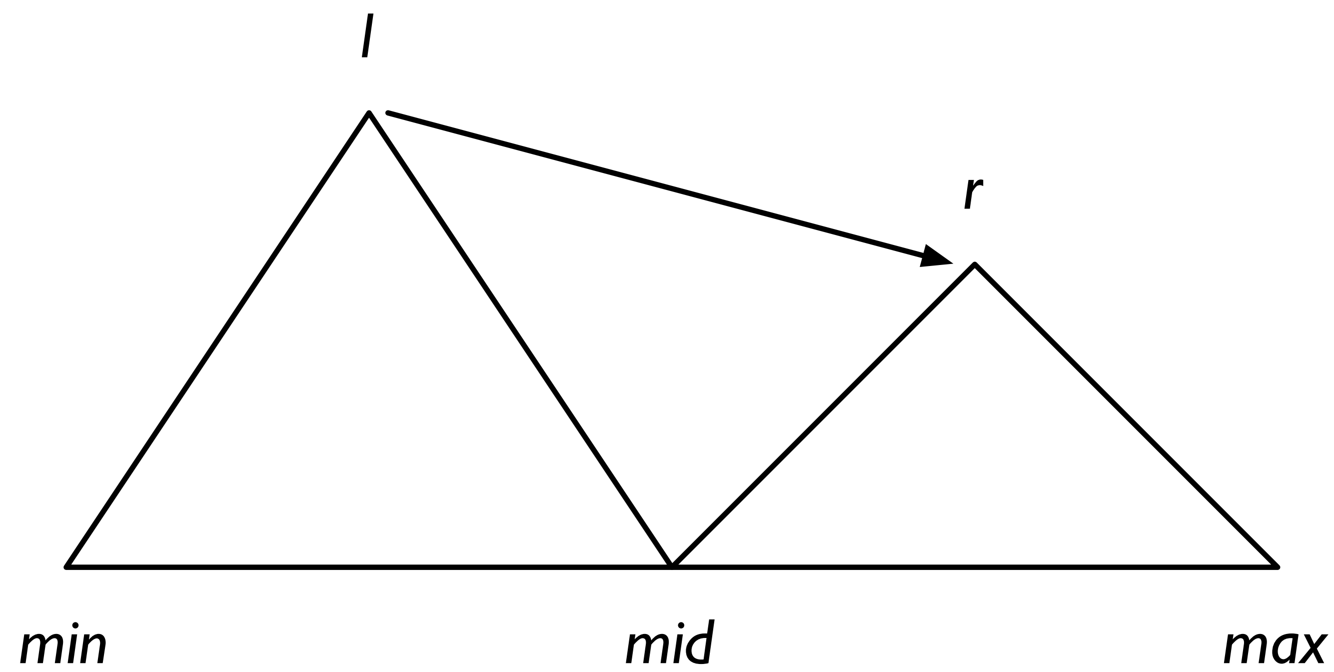
Smart approach

- Before parsing, compute a table that lists, for each head-dependent pair (h, d), the label that maximizes the score of arcs $h \rightarrow d$.
  - This is guaranteed to be the arcs that could be used in a highest-scoring tree
- During parsing, simply look up the best label in the pre-computed table.
- This adds (not multiplies!) a factor of $|L||w|^2$ to the overall runtime of the algorithm.

Eisner's algorithm

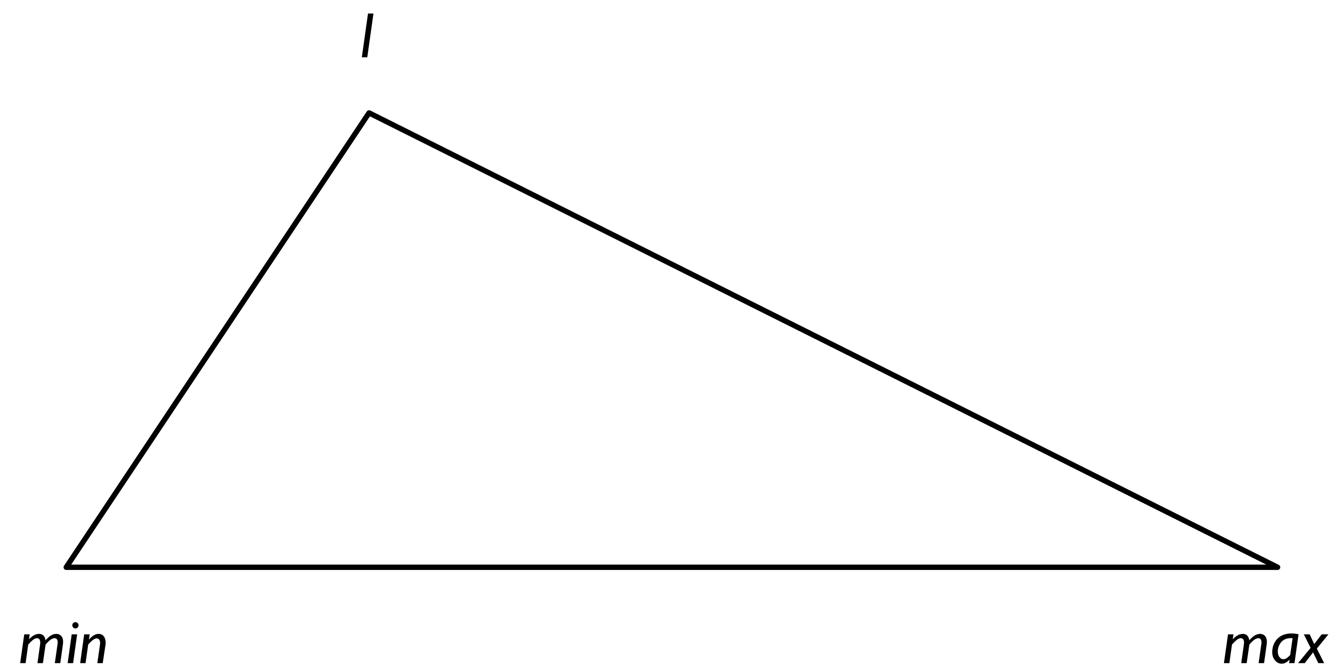
- With its runtime of $O(|w|^5)$, Collins' algorithm may not be of much use in practice.
- With Eisner's algorithm we will be able to solve the same problem in $O(|w|^3)$.
  - Intuition: collect left and right dependents independently

Basic idea



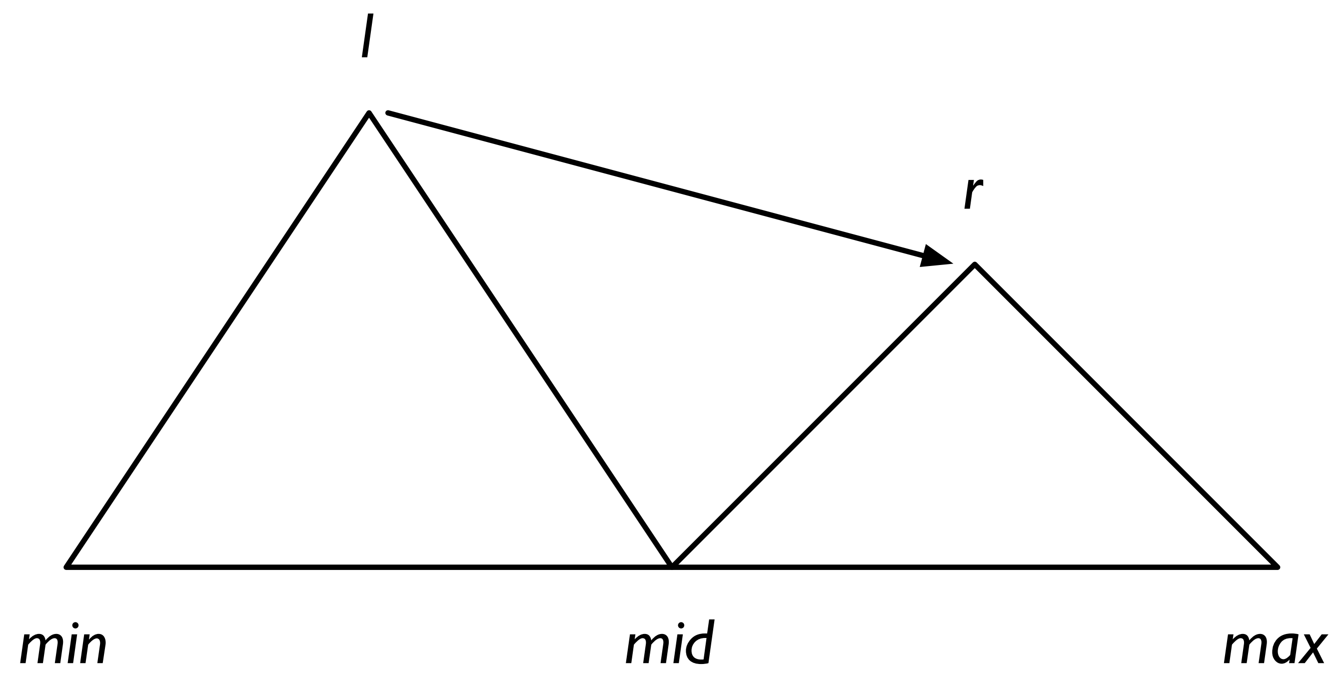
In Collins' algorithm, adding a left-to-right arc is done in one single step, specified by 5 positions.

Basic idea



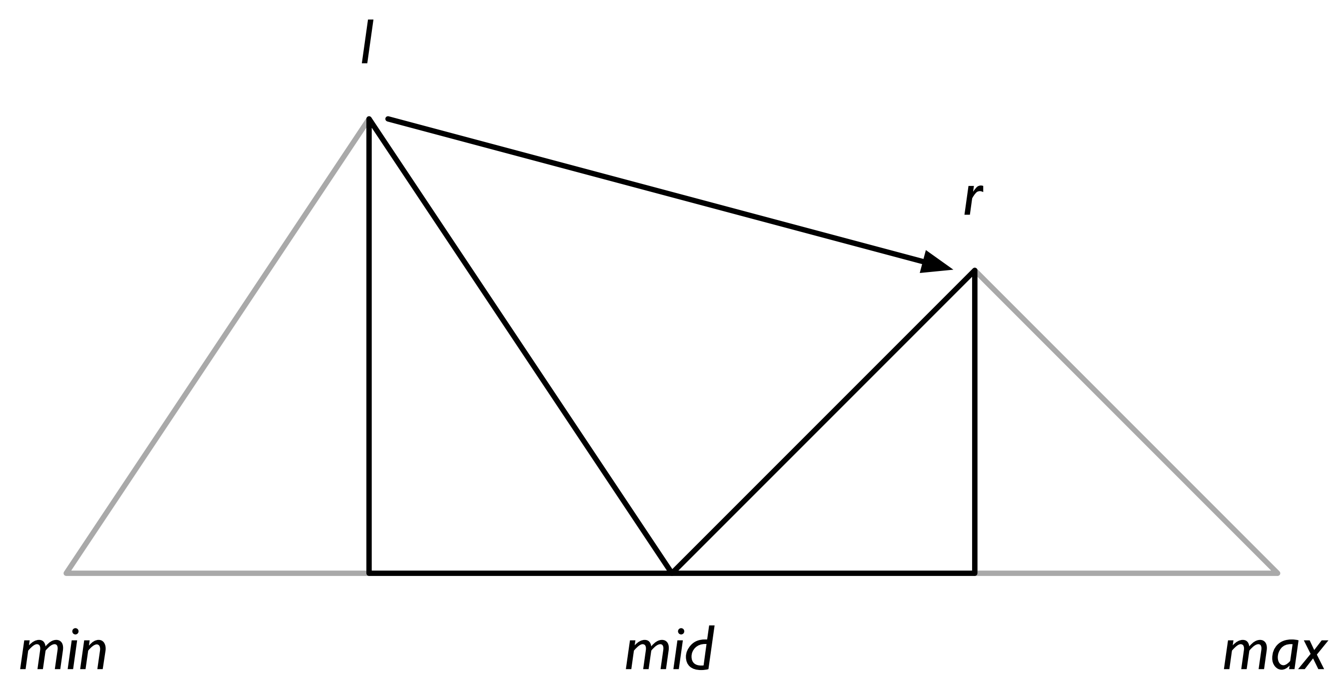
In Collins’ algorithm, adding a left-to-right arc is done in one single step, specified by 5 positions.

Basic idea



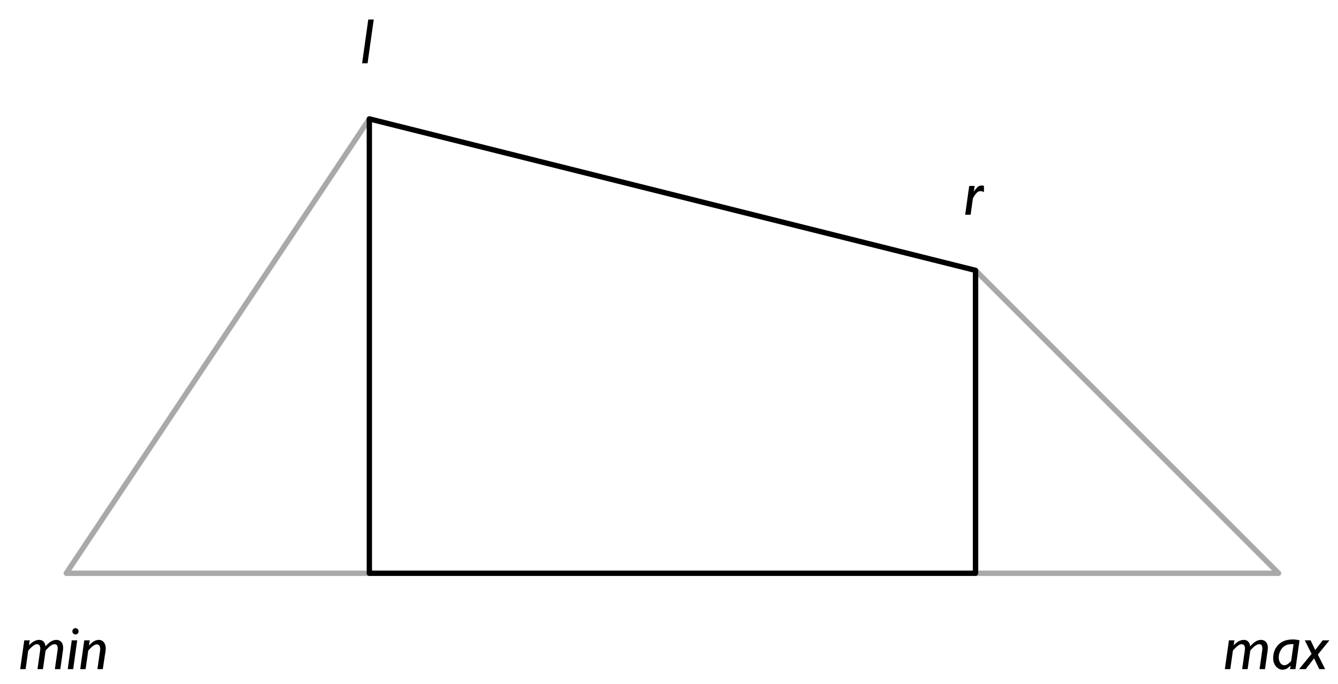
In Eisner’s algorithm, the same thing is done in three steps, each one specified by 3 positions.

Basic idea



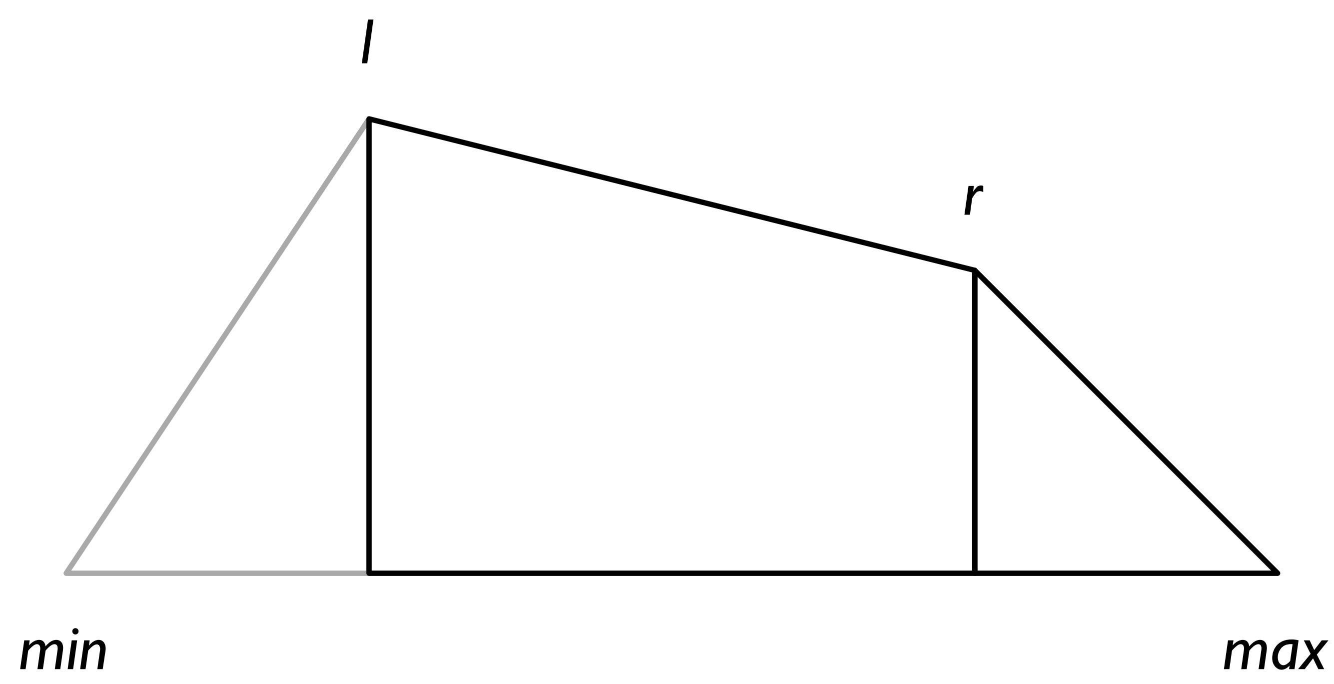
In Eisner’s algorithm, the same thing is done in three steps, each one specified by 3 positions.

Basic idea



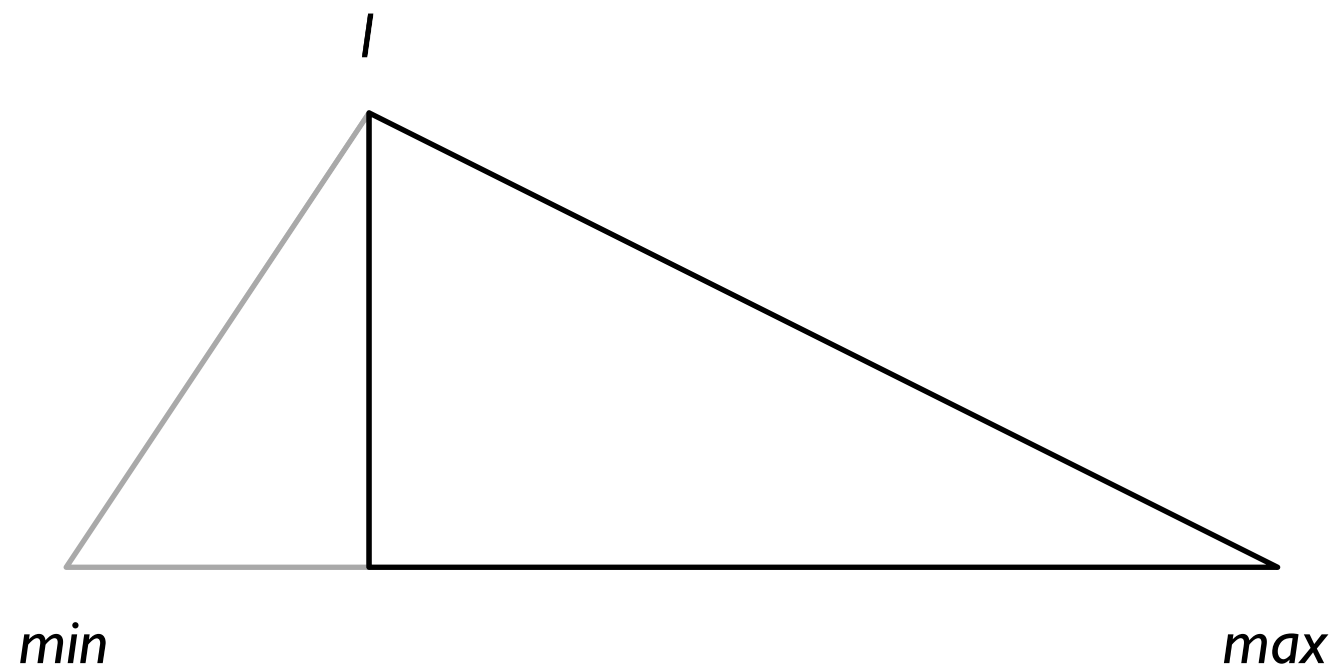
In Eisner's algorithm, the same thing is done in three steps, each one specified by 3 positions.

Basic idea



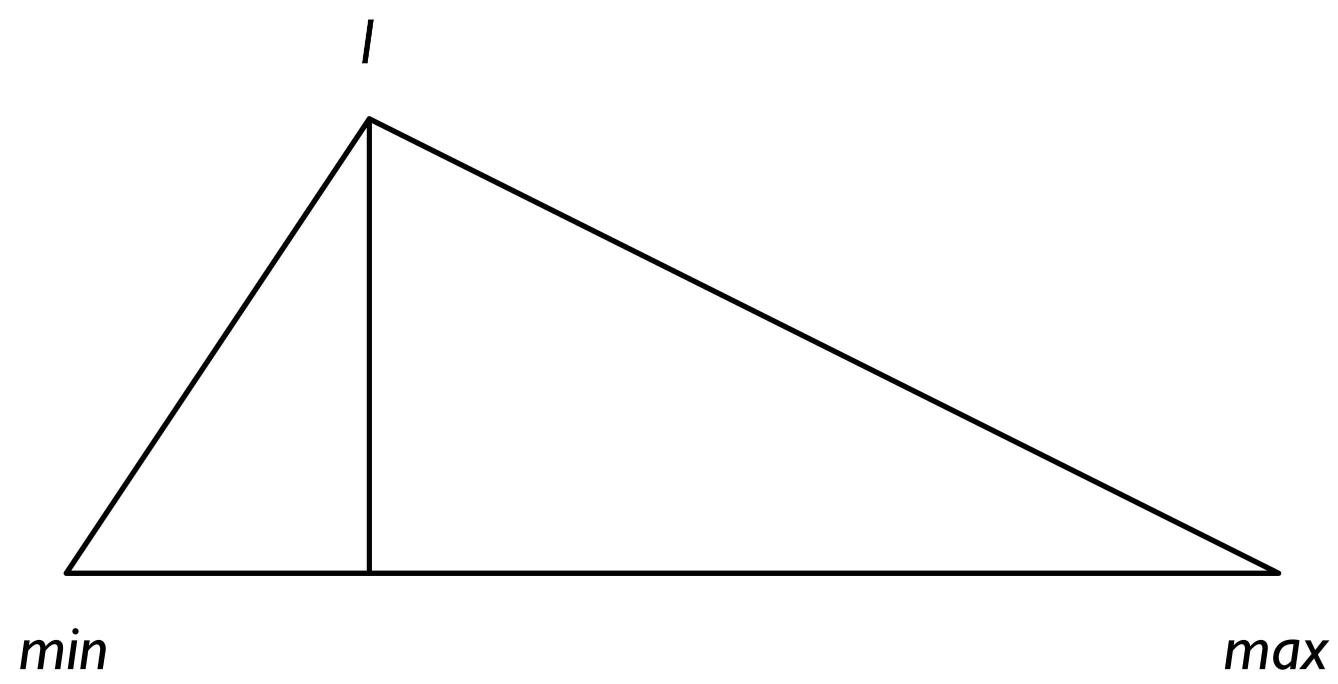
In Eisner’s algorithm, the same thing is done
in three steps, each one specified by 3 positions.

Basic idea



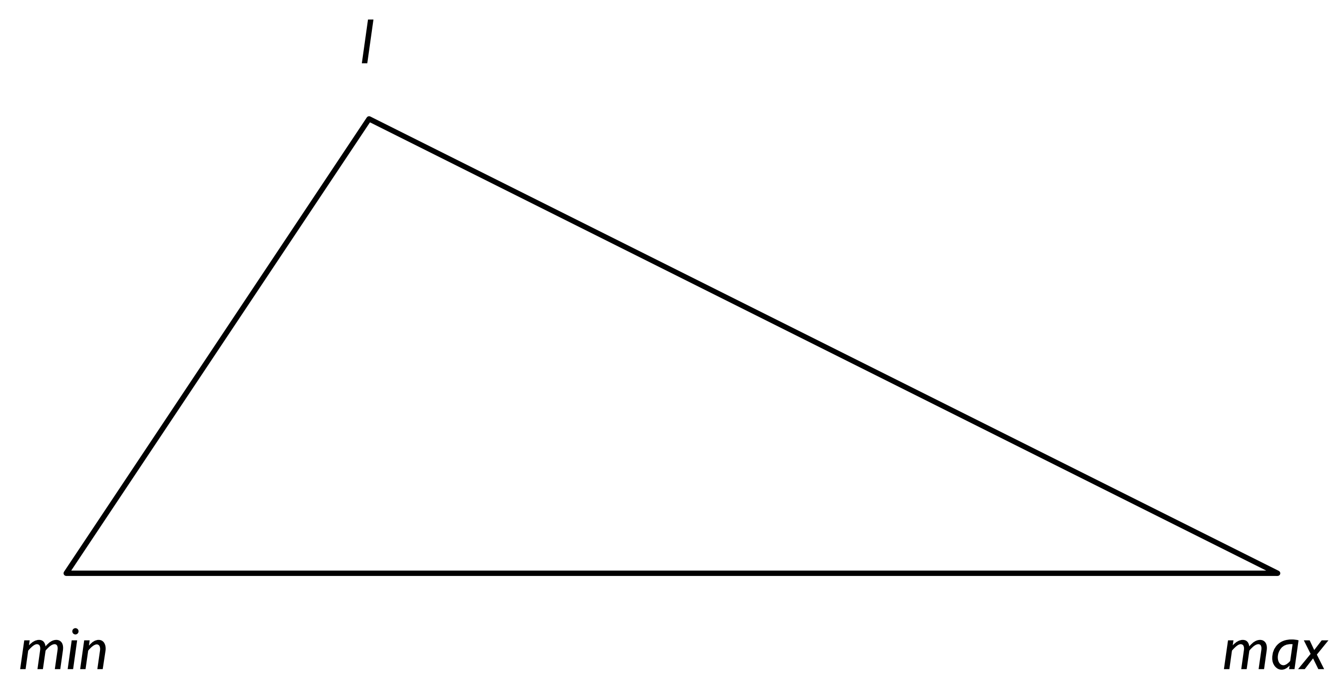
In Eisner’s algorithm, the same thing is done in three steps, each one specified by 3 positions.

Basic idea



In Eisner’s algorithm, the same thing is done in three steps, each one specified by 3 positions.

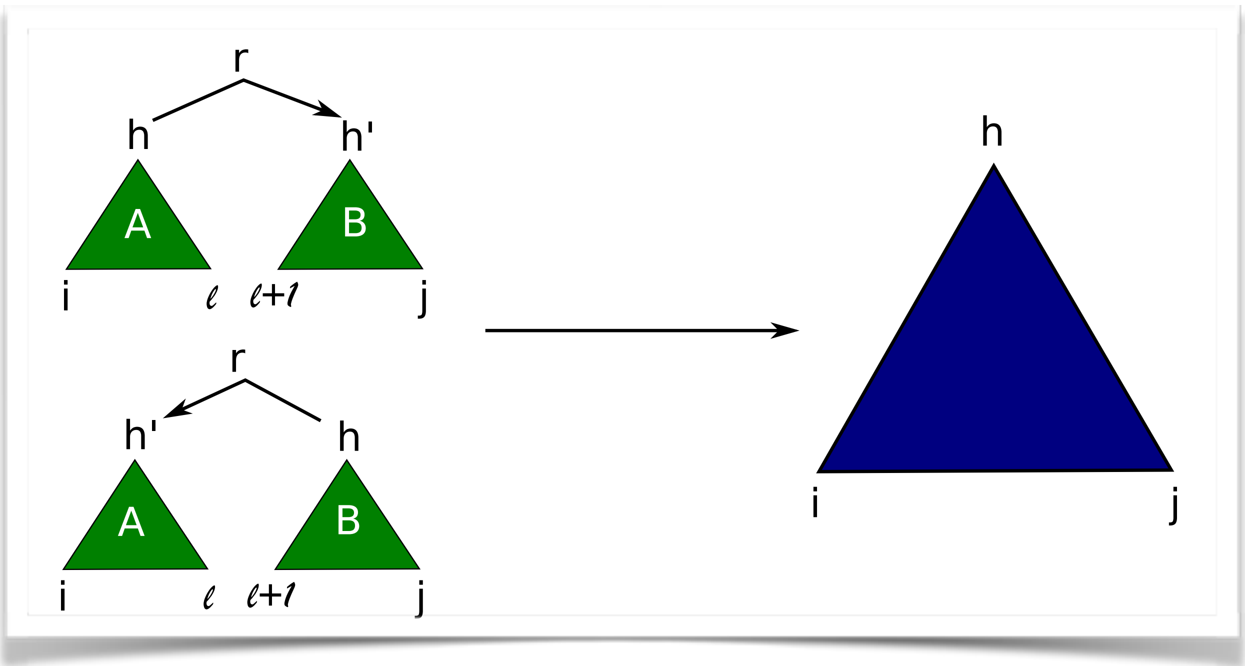
Basic idea

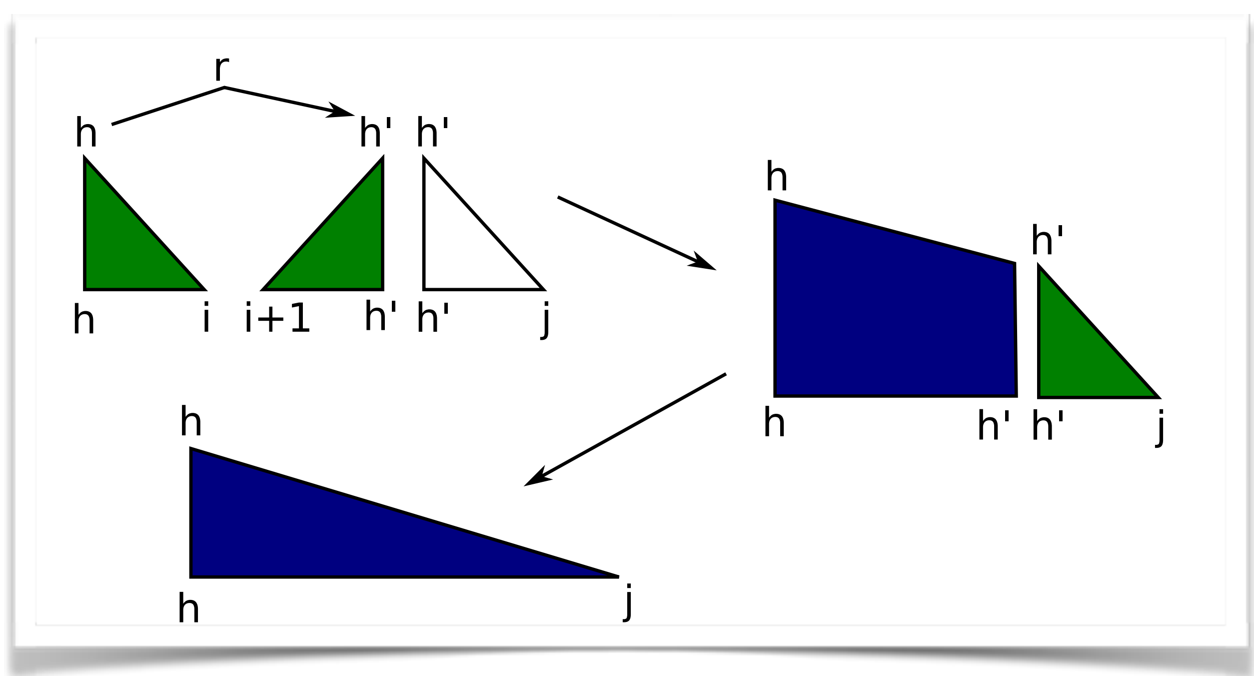


In Eisner’s algorithm, the same thing is done in three steps, each one specified by 3 positions.

UPPSALA UNIVERSITET

Comparison



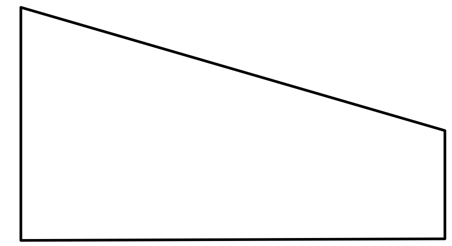


Dynamic programming tables

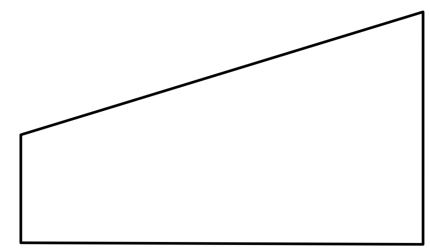
- Collins':
  - [min,max,head]
- Eisner's
  - [min,max,head-side,complete]
    - head-side (binary): is head to the left or right?
    - complete (binary:) is the non-head side still looking for dependents?

Graphic representation

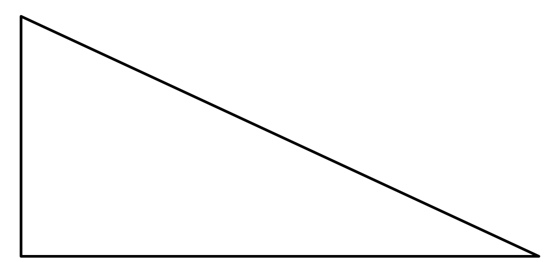
- [min,max,left,yes]



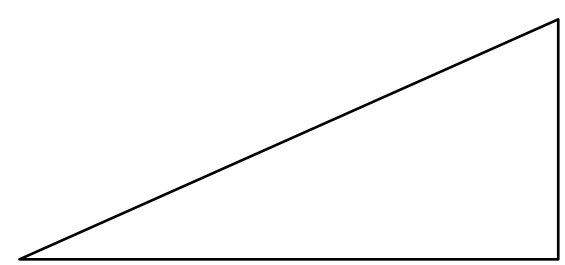
- [min,max,right,yes]



- [min,max,left,no]

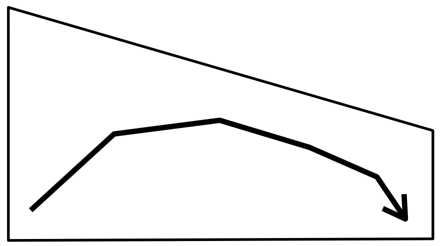


- [min,max,right,no]

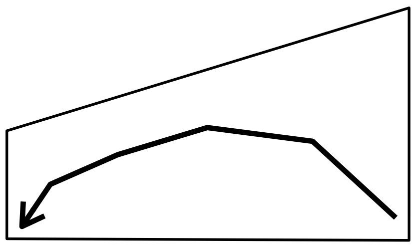


Graphic representation

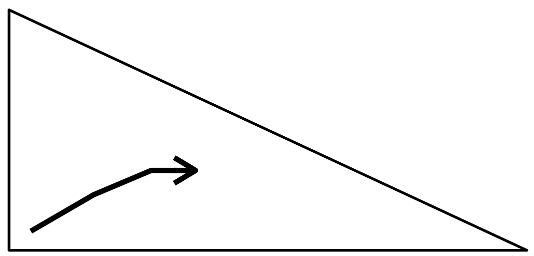
- [min,max,left,yes]



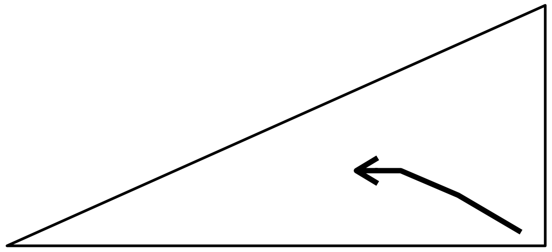
- [min,max,right,yes]



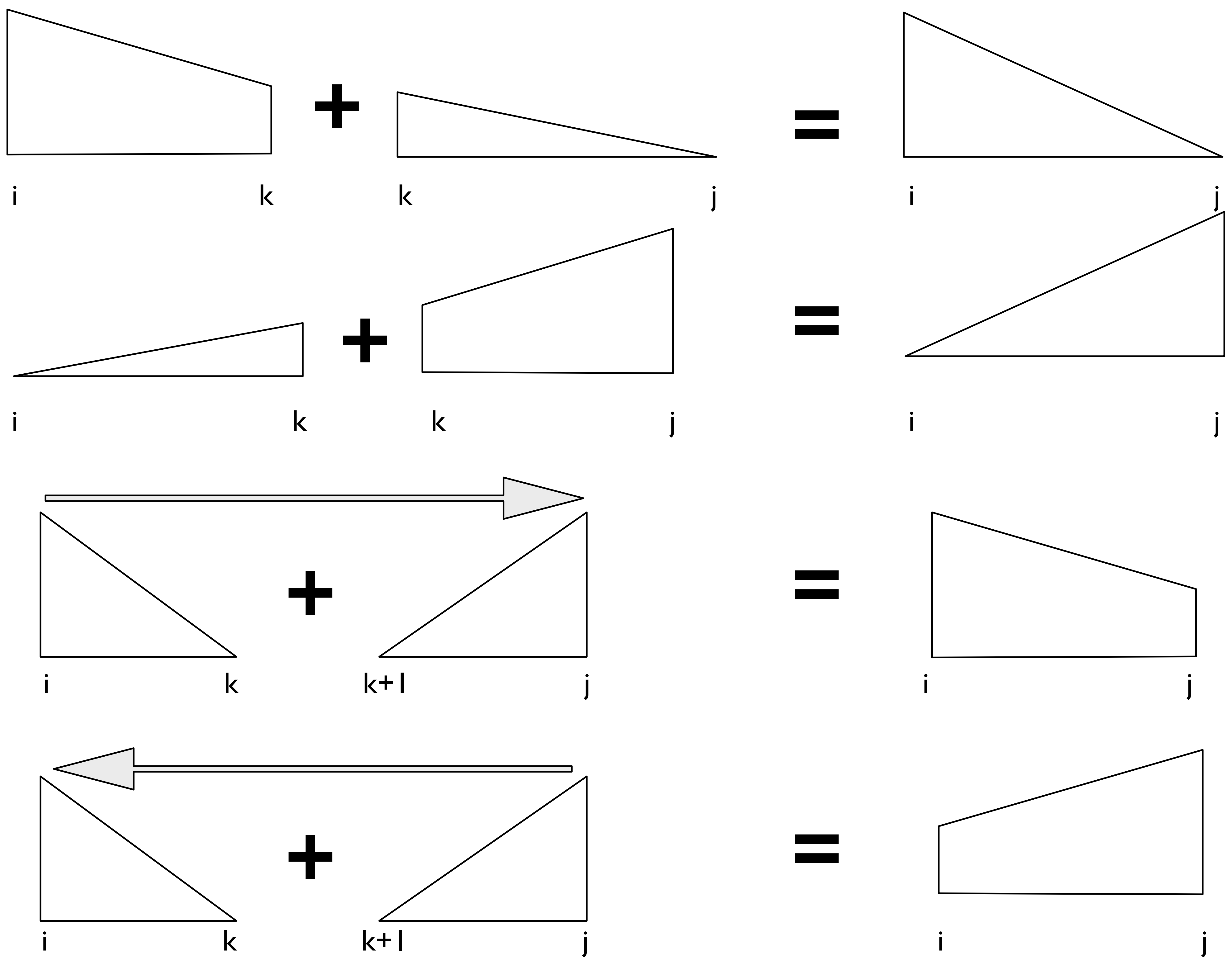
- [min,max,left,no]



- [min,max,right,no]



Possible operations



Pseudo code

```
for each i from 0 to n and all d,c do
   C[i][i][d][c] = 0.0
for each m from 1 to n do
  for each i from 0 to n-m do
     j = i+m
     C[i][j][←][1] = max_{i≤q<j}(C[i][q][→][0] + C[q+1][j][←][0]+score(w_j,w_i)
     C[i][j][→][1] = max_{i≤q<j}(C[i][q][→][0] + C[q+1][j][←][0]+score(w_i,w_j)
     C[i][j][←][0] = max_{i≤q<j}(C[i][q][←][0] + C[q][j][←][1])
     C[i][j][→][0] = max_{i≤q<j}(C[i][q][→][1] + C[q][j][→][0])
return [0][n][→][0]
```

Summary

- Eisner’s algorithm is an improvement over Collin’s algorithm that runs in time $O(|w|^3)$.
- The same scoring model can be used.
- The same technique for extending the parser to labeled parsing can be used, adding $O(|L||w|^2)$ to the run time.
- Eisner’s algorithm is the basis of current arc-factored dependency parsers.

Projectivity

- Eisner's algorithm, as well as Collin's algorithm, builds the tree bottom-up
- They only produce projective trees
- What about non-projective graph-based parsing?
  - Based on minimum-spanning tree algorithms

UPPSALA UNIVERSITET

Minimum-spanning tree parsing

- Based on graph algorithms to find the minimum spanning tree
  - Often: Chu-Liu-Edmonds algorithm (CLU)
- Directly produces non-projective trees
- First suggested in the MSTparser
- One of the most popular algorithms today

Minimum-spanning tree parsing

- **Intuition:**
- Score all word pairs in both directions
- Create a fully connected graph with these scores
- Remove all edges going into ROOT
- For each node, greedily keep only the highest-scoring incoming arc
    - If this produces a tree: done!
    - Otherwise: handle each cycle in the graph:
    - Recursively contract cycles, and recalculate incoming weights

Minimum-spanning tree parsing

- **Complexity:**
- Naive implementation:
  - O(n^3):
  - At most n recursive calls to contract graph, in each call find highest incoming edge: n^2
- Efficient implementation:
  - O(n^2)
  - Tarjan (1977)
- Naturally can produce non-projective trees

UPPSALA UNIVERSITET

Coming up

- March 4: literature seminar 2
  - Groups on the web page (note: new groups)
- Supervision in Chomsky+Turing :
  - March 6 and March 13
- Final seminar:
  - March 25 (NOTE: moved)
- Assignment 3, deadline March 11
- Project report, deadline March 22