Lecture 14

Parsing

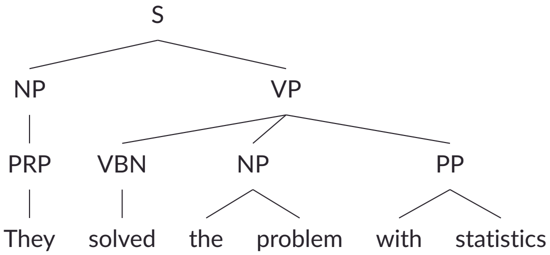
Part 1: Definition and Representation

Machine Learning for Structured Data
Vlad Niculae · LTL, UvA · `https://vene.ro/mlsd`

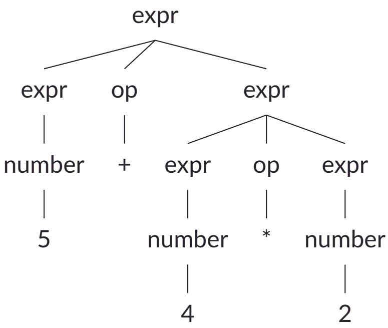
Syntactic Analysis

Syntax is an underlying structure of languages, often analyzed using parse trees.

human language (English):

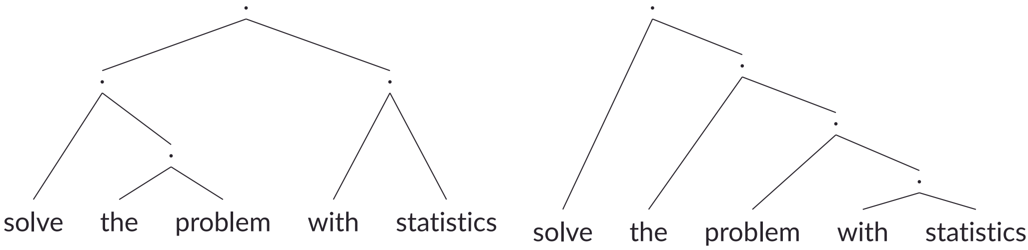


programming language (Python):



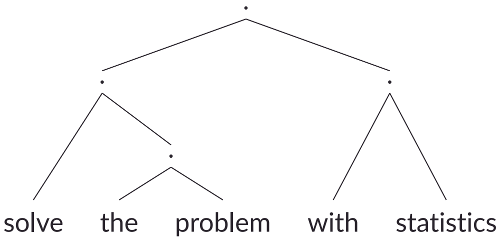
Binary Parsing

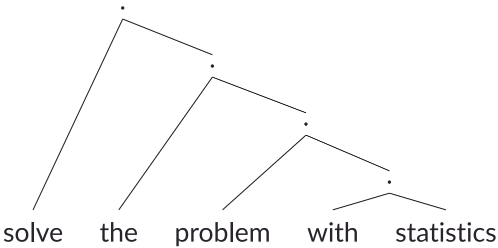
There are many kinds of complicated syntactic analysis formalisms. For simplicity, we focus on: binary trees. Let's start without labels too.



Binary Parsing

There are many kinds of complicated syntactic analysis formalisms. For simplicity, we focus on: binary trees. Let's start without labels too.





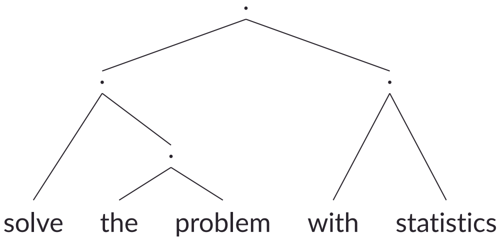
A binary parse tree with no labelling is the same thing as a bracketing:

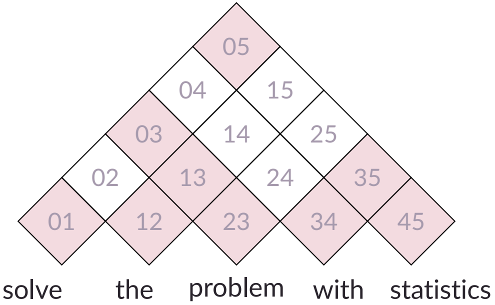
((solve (the problem)) (with statistics))   ((solve (the (problem (with statistics)))))

Bracketing: Representation

Assign a score a_{ij} to the span from i to j (fencepost).

The score of a parse tree is the sum of all scores of its (nested!) spans.



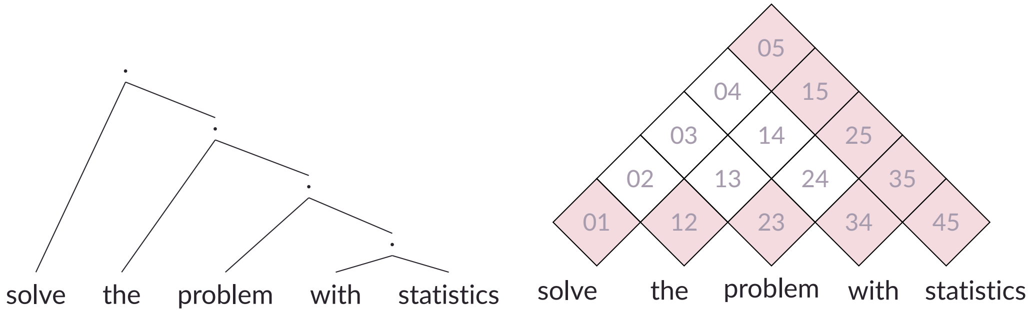


$$\mathrm{score}(y) = a_{01} + a_{12} + a_{23} + a_{34} + a_{45} + a_{13} + a_{35} + a_{03} + a_{05}$$

Bracketing: Representation

Assign a score a_{ij} to the span from i to j (fencepost).

The score of a parse tree is the sum of all scores of its (nested!) spans.



$$\text{score}(y) = a_{01} + a_{12} + a_{23} + a_{34} + a_{45} + a_{35} + a_{25} + a_{15} + a_{05}$$

Lecture 14

Parsing

Part 2: Bracketing: Algorithm

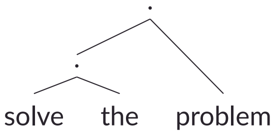
Machine Learning for Structured Data

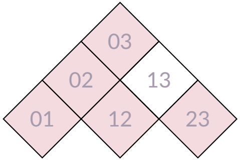
Vlad Niculae · LTL, UvA · `https://vene.ro/mlsd`

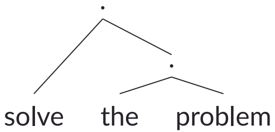
Parsing

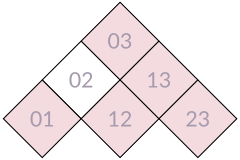
Algorithm

Possible parses of the subsequence (0, 3):

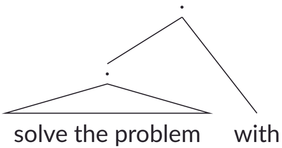


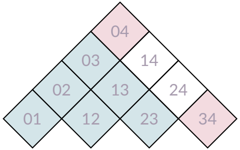


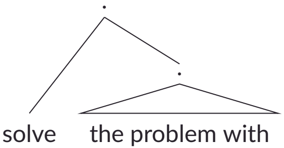


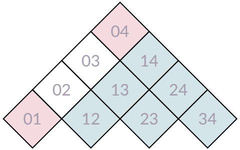


Possible parses of the subsequence (0, 4):

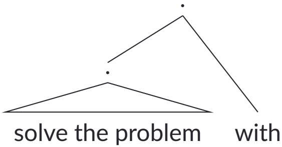


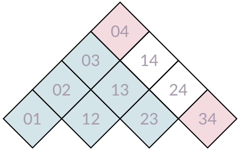


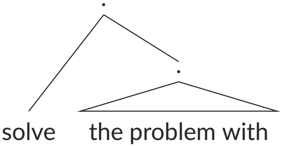


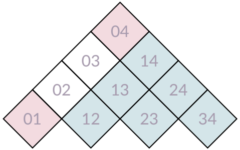


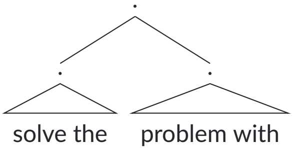
Possible parses of the subsequence (0, 4):

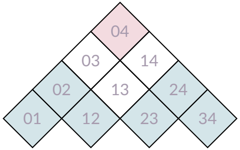




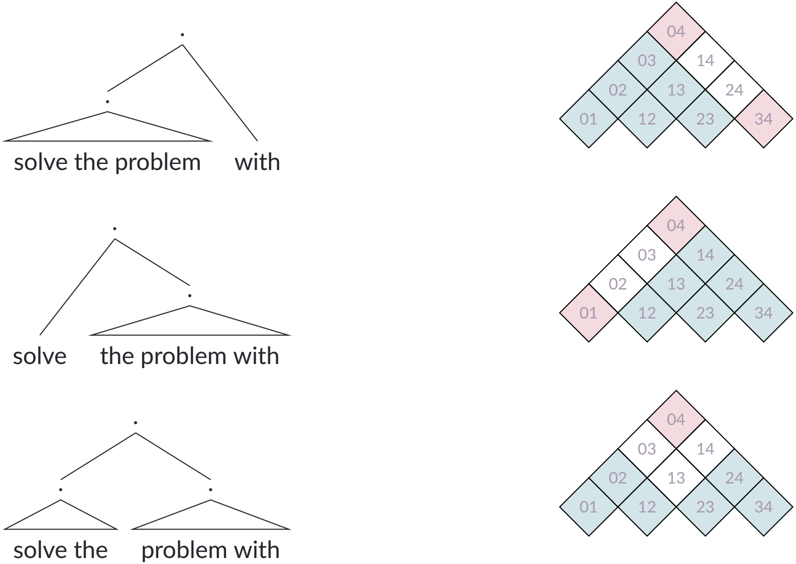






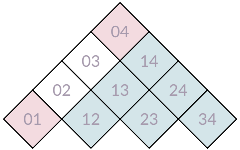


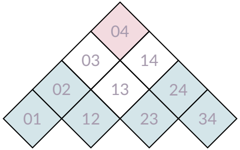
Possible parses of the subsequence (0, 4): see the pattern?

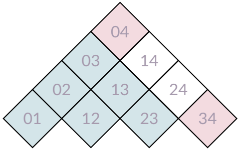


Dynamic Programming for Bracketing

In general: a partial parse that covers subsequence (i, j) must consist of two partial parses: one covering (i, k) and one covering (k, j) for some $i < k < j$.





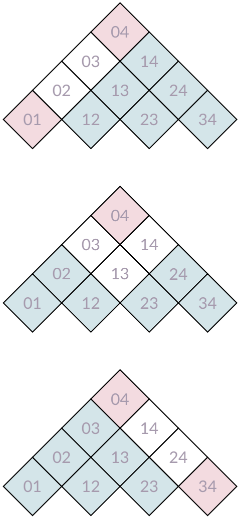


Dynamic Programming for Bracketing

In general: a partial parse that covers subsequence (i, j) must consist of two partial parses: one covering (i, k) and one covering (k, j) for some $i < k < j$.

Define M_{ij} as the maximum-scoring parse of subtree from i to j. Then:

$$
\begin{aligned}
M_{i,i+1} &= a_{i,i+1} \\
M_{i,j} &= \max_{i<k<j} a_{i,j} + M_{i,k} + M_{k,j}
\end{aligned}
$$



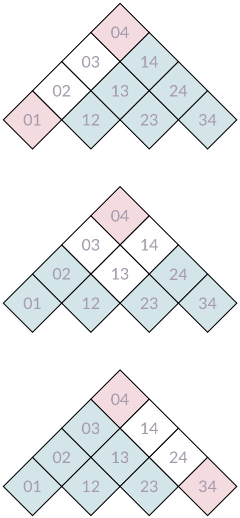
Dynamic Programming for Bracketing

In general: a partial parse that covers subsequence (i, j) must consist of two partial parses: one covering (i, k) and one covering (k, j) for some $i < k < j$.

Define M_{ij} as the maximum-scoring parse of subtree from i to j. Then:

$$\begin{aligned} M_{i,i+1} &= a_{i,i+1} \\ M_{i,j} &= \max_{i<k<j} a_{i,j} + M_{i,k} + M_{k,j} \end{aligned}$$

Fill in the table bottom-up: **dynamic programming**.



Dynamic Programming for Bracketing

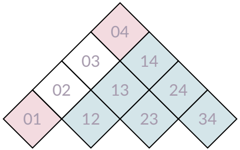
In general: a partial parse that covers subsequence (i, j) must consist of two partial parses: one covering (i, k) and one covering (k, j) for some $i < k < j$.

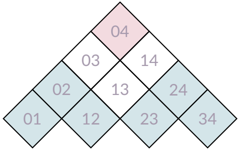
Define M_{ij} as the maximum-scoring parse of subtree from i to j. Then:

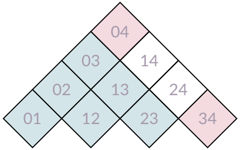
$$\begin{aligned} M_{i,i+1} &= a_{i,i+1} \\ M_{i,j} &= \max_{i<k<j} a_{i,j} + M_{i,k} + M_{k,j} \end{aligned}$$

Fill in the table bottom-up: **dynamic programming.**

CYK algorithm: Cocke, Younger, Kasami independently discovered it in the 1960s.







The CYK Algorithm

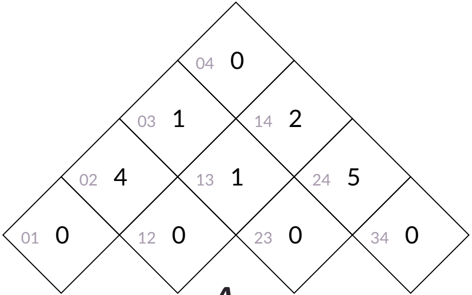
input: Scores $a_{i,j}$ for $0 \le i < j \le n$
$M_{i,j} = 0, \quad \pi_{i,j} = -1$, for $0 \le i < j \le n$.
$M_{i,i+1} = a_{i,i+1}$ for $0 \le i < n$.

Forward: compute max. scores for each span recursively
for $s = 2$ to n **do**
  for $i = 0$ to $n - s$ **do**
    $j = i + s$
    $M_{i,j} = \max_{i<k<j} a_{i,j} + M_{i,k} + M_{k,j}$
    $\pi_{i,j} = \arg\max_{i<k<j} a_{i,j} + M_{i,k} + M_{k,j}$

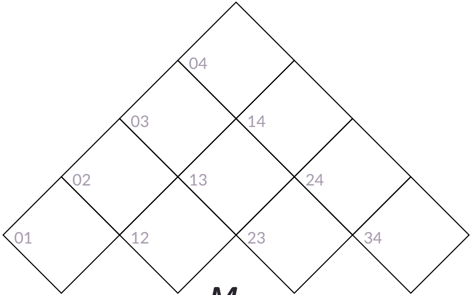
Backward: follow backpointers
$\boldsymbol{y}^\star = (), Q = \{(0, n)\}$.
while Q not empty **do**
  pop (i, j) from Q
  $\boldsymbol{y}^\star = \boldsymbol{y}^\star + (i, j)$
  $k = \pi_{i,j}$
  push (i, k) and (k, j) to Q **if** $k > 0$.

output: The highest-scoring bracketing $\boldsymbol{y}^\star$, and its total score $f^\star = M_{0,n}$.

The CYK Algorithm: Example



$\boldsymbol{A} =$



$\boldsymbol{M} =$

The *Inside* Algorithm for $\log Z$

input: Scores $a_{i,j}$ for $0 \le i < j \le n$
$Q_{i,j} = 0$, for $0 \le i < j \le n$.
$Q_{i,i+1} = a_{i,i+1}$ for $0 \le i < n$.

Forward: compute logsumexp for each span recursively
for $s = 2$ to n **do**
  for $i = 0$ to $n - s$ **do**
    $j = i + s$
    $Q_{i,j} = \log \sum_{i<k<j} \exp a_{i,j} + Q_{i,k} + Q_{k,j}$

CYK vs Segmentation

- The two algorithms have the same inputs: a table of scores for every possible segment.
- The segmentation problem seeks the best low-level chunking.
- CYK seeks an entire tree of chunk “splits”.
- Segmentation is the simplest possible DAG. CYK cannot be represented as a DAG at all!

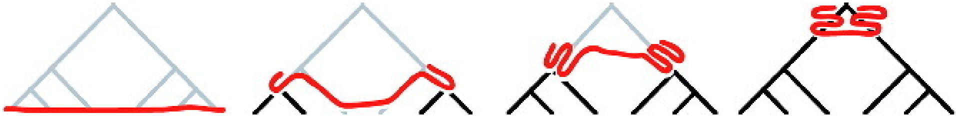
Lecture 14

Parsing

Part 3: Extensions and Evaluation

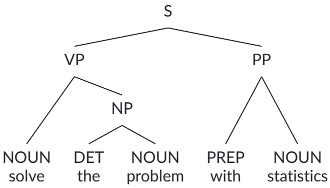
Machine Learning for Structured Data
Vlad Niculae · LTL, UvA · `https://vene.ro/mlsd`

Protein Folding as Binary Parsing



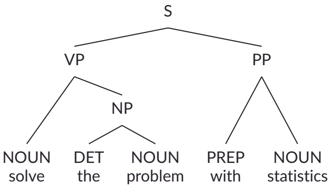
Julia Hockenmaier, Aravind K. Joshi, Ken A. Dill,
Routes are trees: The parsing perspective on protein folding. Proteins, 66–1, 2007.

Labelled Parsing



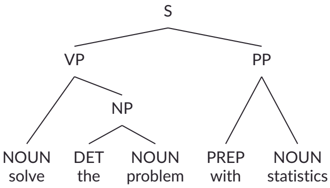
- Simple case: replace all segments with labeled segments (i, j, c).

Labelled Parsing



- Simple case: replace all segments with labeled segments (i, j, c).
- In this case, like for segmentation, we can pick the best label for each segment before starting Viterbi, and ignore the rest.

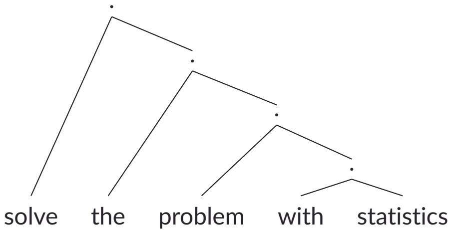
Labelled Parsing



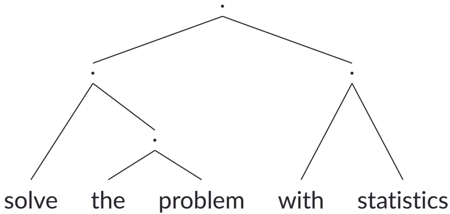
- Simple case: replace all segments with labeled segments (i, j, c).
- In this case, like for segmentation, we can pick the best label for each segment before starting Viterbi, and ignore the rest.
- We may want "transition scores"
  e.g., prefer S out of NP VP, dislike S out of VP PP.
  - related to *probabilistic context-free grammars*
  - handled by a similar DP algorithm,
    higher complexity
    (loop also over all combinations of labels).

Evaluation

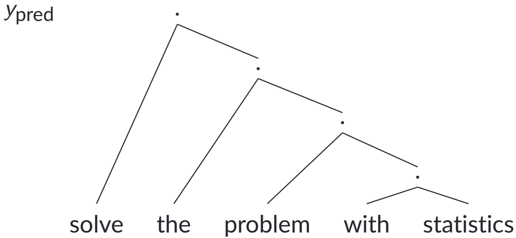
y_{pred}

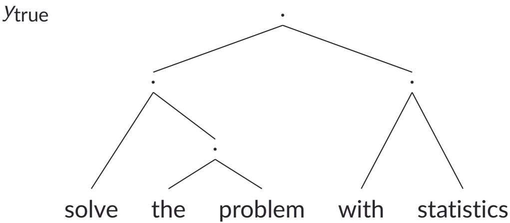


y_{true}



Evaluation

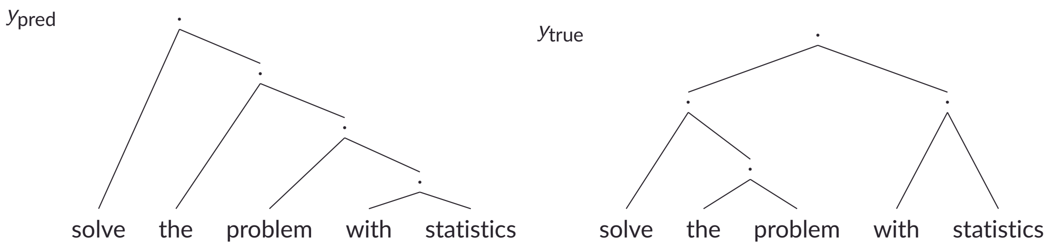




Predicted spans:
(0, 1), (0, 5) (1, 2), (1, 5), (2, 3), (2, 5), (3, 4) (3, 5), (4, 5)

True spans:
(0, 1), (0, 3), (0, 5), (1, 2), (1, 3), (2, 3), (3, 4), (3, 5), (4, 5)

Evaluation



Predicted spans:
(0, 1), (0, 5) (1, 2), (1, 5), (2, 3), (2, 5), (3, 4) (3, 5), (4, 5)

True spans:
(0, 1), (0, 3), (0, 5), (1, 2), (1, 3), (2, 3), (3, 4), (3, 5), (4, 5)

$P = \frac{\text{n. correct}}{\text{n. predicted}}$ $\qquad R = \frac{\text{n. correct}}{\text{n. true}}$ $\qquad F_1 = \frac{2PR}{P+R}$

Note: in the unlabelled case, P=R, since the number of segments in a bracketing is always the same.

In the labelled case: usually common to compute per-label P/R/F, averaged over the entire dataset.

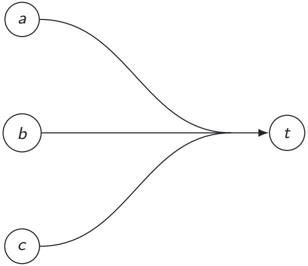
In linguistic applications, “real” parsing evaluation is more complicated, since trees are not binary.

Hyperedges and Hypergraphs

There is a formalism that generalizes DAGs and can express the CYK parsing problem, but its details are too complicated for our scope. Nevertheless, here is a glimpse.

Given nodes $V = \{1, 2, \ldots, n\}$

- instead of **edges**: $(s, t) : s \in V, t \in V.$
- define **hyperedges**: $\big((s_1, \ldots, s_k), t\big) : s_i \in V, t \in V.$



Hyperedges and Hypergraphs

There is a formalism that generalizes DAGs and can express the CYK parsing problem, but its details are too complicated for our scope. Nevertheless, here is a glimpse.

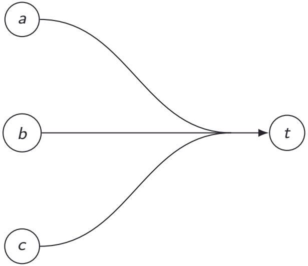
Given nodes $V = \{1, 2, \ldots, n\}$

- instead of **edges**: $(s, t) : s \in V, t \in V$.
- define **hyperedges**: $\big((s_1, \ldots, s_k), t\big) : s_i \in V, t \in V$.

Any directed graph can be represented as a directed hypergraph: if (s, t) is an edge in G, then make $((s), t)$ a hyperedge in HG.

Generalizations of DAG and topological sort exist; and Viterbi & Forward algorithms work.

Read more: Liang Huang, Advanced Dynamic Programming in Semiring and Hypergraph Frameworks, COLING 2008 tutorial.



Summary

- Binary parsing / bracketing can be solved with dynamic programming (even if it can't be represented as a DAG)
- Applications in computational linguistics:
  related to *grammars*.
- Can generalize the algorithms seen to compute logsumexp and sampling with DP, using a *hypergraphs* formalism.