Parsing

Daniel Zeman

January 6, 2022



Charles University
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

Constituent Parsing

Observation: Phrases Are Related to Context-Free Grammars

- Phrase structure of sentence $\leftarrow$ derivation tree
- Example:
  - S $\rightarrow$ NP VP *(sentence has subject and predicate)*
  - NP $\rightarrow$ N *(noun phrase may be noun)*
  - VP $\rightarrow$ V NP *(verb phrase may consist of verb and object)*
- Lexicon part of the grammar:
  - N $\rightarrow$ dog | cat | man | John ...
  - V $\rightarrow$ see | sees | saw | bring | brings | brought | ...

Extended Example for Czech (7 Cases!)

- NP $\rightarrow$ N | AP N
- AP $\rightarrow$ A | AdvP A
- AdvP $\rightarrow$ Adv | AdvP Adv

- $\text{NP}_{nom} \rightarrow \text{N}_{nom}$
- $\text{NP}_{nom} \rightarrow \text{AP}_{nom}\ \text{N}_{nom}$
- $\text{NP}_{nom} \rightarrow \text{N}_{nom}\ \text{NP}_{gen}$

- $\text{NP}_{gen} \rightarrow \text{N}_{gen}$
- $\text{NP}_{gen} \rightarrow \text{AP}_{gen}\ \text{N}_{gen}$
- $\text{NP}_{gen} \rightarrow \text{N}_{gen}\ \text{NP}_{gen}$

- N $\rightarrow$ *pán* | *hrad* | *muž* | *stroj* ...
- A $\rightarrow$ *mladý* | *velký* | *zelený* ...
- Adv $\rightarrow$ *velmi* | *včera* | *zeleně* ...

- $\text{N}_{nom} \rightarrow$ *pán* | *hrad* | *muž* ...
- $\text{N}_{gen} \rightarrow$ *pána* | *hradu* | *muže* ...
- $\text{N}_{dat} \rightarrow$ *pánovi* | *hradu* | *muži* ...
- $\text{N}_{acc} \rightarrow$ *pána* | *hrad* | *muže* ...
- $\text{N}_{voc} \rightarrow$ *pane* | *hrade* | *muži* ...
- $\text{N}_{loc} \rightarrow$ *pánovi* | *hradu* | *muži* ...
- $\text{N}_{ins} \rightarrow$ *pánem* | *hradem* | *mužem* ...

Extended Example for Czech (Verbs)

- VP $\rightarrow$ $\text{VP}_{obligatory}$
- VP $\rightarrow$ $\text{VP}_{obligatory}$ OptMod

- $\text{VP}_{obligatory} \rightarrow \text{V}_{intr}$
- $\text{VP}_{obligatory} \rightarrow \text{V}_{trans}$ NP_{acc}
- $\text{VP}_{obligatory} \rightarrow \text{V}_{ditr}$ NP_{dat} NP_{acc}
- $\text{VP}_{obligatory} \rightarrow \text{V}_{mod}$ VINF

- OptMod $\rightarrow$ $\text{AdvP}_{location}$ | AdvP_{time} | AdvP_{manner} ...

- $\text{V}_{intr} \rightarrow$ *šedivět* | *brzdit* ...
- $\text{V}_{trans} \rightarrow$ *koupit* | *ukrást* ...
- $\text{V}_{ditr} \rightarrow$ *dát* | *půjčit* | *poslat* ...
- $\text{V}_{mod} \rightarrow$ *moci* | *smět* | *muset* ...

Unification Grammar

- Alternative to nonterminal splitting
- Instead of seven context-free rules:
  - $\text{NP}_{nom} \rightarrow \text{AP}_{nom}\ \text{N}_{nom}$
  - $\text{NP}_{gen} \rightarrow \text{AP}_{gen}\ \text{N}_{gen}$
  - $\text{NP}_{dat} \rightarrow \text{AP}_{dat}\ \text{N}_{dat}$
  - $\text{NP}_{acc} \rightarrow \text{AP}_{acc}\ \text{N}_{acc}$
  - $\text{NP}_{voc} \rightarrow \text{AP}_{voc}\ \text{N}_{voc}$
  - $\text{NP}_{loc} \rightarrow \text{AP}_{loc}\ \text{N}_{loc}$
  - $\text{NP}_{ins} \rightarrow \text{AP}_{ins}\ \text{N}_{ins}$

- One unification rule:
  - $\text{NP} \rightarrow \text{AP N} := [\text{case} = \text{AP.case} \# \text{N.case}]$

Parsing with a Context-Free Grammar

- Hierarchy of grammars:
  - Noam Chomsky (1957): *Syntactic Structures*
- Couple of classical algorithms
  - CYK (Cocke-Younger-Kasami) ... complexity $O(n^3)$
    - John Cocke ("inventor")
    - Tadao Kasami (1965), Bedford, MA, USA (another independent "inventor")
    - Daniel H. Younger (1967) (computational complexity analysis)

    - Constraint of CYK: grammar is in CNF (Chomsky Normal Form), i.e., right-hand side of every rule: either two nonterminals or one terminal. (CFGs can be easily transformed to CNF.)

Parsing with a Context-Free Grammar

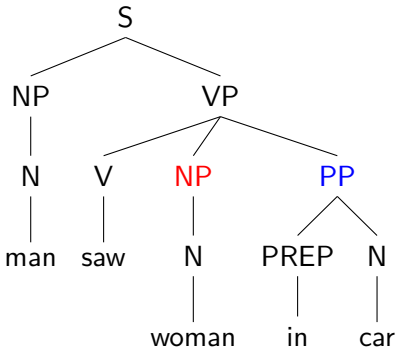
- **Chart parser:** CYK requires a data structure – *chart* (proposed in turn of 1960s and 1970s)
- Jay Earley (1968), PhD thesis, Pittsburgh, PA, USA
  - Somewhat different version of chart parsing ("Earley parser")
- For details on chart parsing, see earlier lecture on morphology

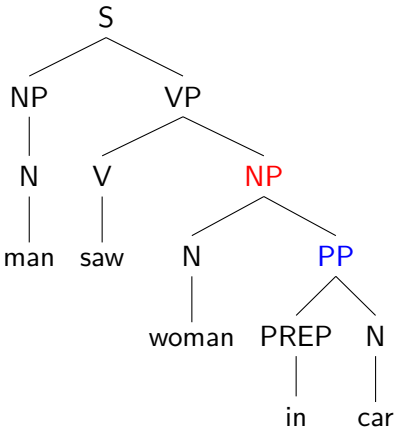
Probabilistic Context-Free Grammars

- PCFG *(probabilistc context-free grammars)*
- Several possible parses? $\Rightarrow$ Weigh them!
- Competing rules with same left-hand side: competing parses
- Idea: probabilistic distribution for rules with same left-hand side
  - Example: grammar has `VP` $\rightarrow$ `V NP` and `VP` $\rightarrow$ `V NP PP`.
  - Input sentence allows both these readings too.
  - But we know (e.g.) that the second structure is more frequent:
    - $p(\texttt{V NP}|\texttt{VP}) = 0.3$
    - $p(\texttt{V NP PP}|\texttt{VP}) = 0.7$

Ambiguous Parse

- S → NP VP
- VP → V NP PP
- VP → V NP
- NP → N
- NP → N PP
- PP → PREP N
- N → *man*
- N → *woman*
- N → *car*
- V → *saw*
- PREP → *in*





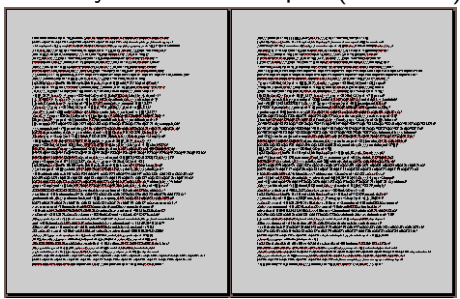
Phrase-Based Parsers (History)

- Rule-based parsers, e.g. Fidditch (Donald Hindle, 1983)
- Collins parser (Michael Collins, 1996–1999)
  - Probabilistic context-free grammars, lexical heads
  - Labeled precision & recall on Penn Treebank / Wall Street Journal / Section 23 = 85%
  - Reimplemented in Java by Dan Bikel ("Bikel parser"), downloadable
- Charniak parser (Eugene Charniak, NAACL 2000)
  - Maximum entropy inspired parser
  - P ∼ R ∼ 89.5%
  - Mark Johnson: reranker ⇒ over 90%
- Stanford parser (Chris Manning et al., 2002–2010)
  - Initial P ∼ R ∼ 86.4%
  - Produces dependencies, too

Dependency Parsing

Dependency Parsing with a Statistical Model

Manually annotated corpus (treebank)



$p(edge(\text{ve}, \text{dveřích})) = p_1$
$p(edge(\text{v}, \text{dveřích})) = p_2$
$p(edge(\text{ve}, \text{dveře})) = p_3$
$p(edge(\text{ve}, \text{dveřím})) = p_4$

where perhaps:

$p_1 > p_2 > p_3$ and $p_4 \to 0$

(*v/ve* “in”; *dveře/dveřím/dveřích* “door”)

Dependency Parsing

- **Chart** (Eisner, CKY)
  - $O(n^3)$
  - Produces only projective parses

Dependency Parsing

- **Chart** (Eisner, CKY)
  - $O(n^3)$
  - Produces only projective parses
- **Transition-based** (shift-reduce)
  - $O(n)$ (fast!)
  - Can be extended to capture nonprojectivity

Dependency Parsing

- **Chart** (Eisner, CKY)
  - $O(n^3)$
  - Produces only projective parses
- **Transition-based** (shift-reduce)
  - $O(n)$ (fast!)
  - Can be extended to capture nonprojectivity
- **Graph-based** (MST)
  - $O(n^2)$
  - Can produce projective and nonprojective parses

Dependency Parsing

Transition-Based Parsers

Transition-Based Parsers: Malt

- Nivre et al., *Natural Language Engineering* 2007
- `http://maltparser.org/`
- Based on *transitions* from one configuration to another
- Configuration:
  - Input buffer (words of the sentence, left-to-right)
  - Stack
  - Output tree (words, edges, labels)
- Transitions:
  - `Shift`: move word from buffer to stack
  - `Larc`: connect two topmost stack words (higher is parent)
  - `Rarc`: connect two topmost stack words (lower is parent)

Malt Parser

- Driven by oracle
  - Looks at current configuration
  - Selects next transition

Malt Parser

- Driven by oracle
  - Looks at current configuration
  - Selects next transition
- Training: decompose the training tree to a sequence of transitions
  - Sometimes more than one possibility
    - Various learning strategies: e.g. create dependencies eagerly, as soon as possible

Malt Parser

- Driven by oracle
  - Looks at current configuration
  - Selects next transition
- Training: decompose the training tree to a sequence of transitions
  - Sometimes more than one possibility
    - Various learning strategies: e.g. create dependencies eagerly, as soon as possible
- The oracle learns based on the features of the configuration
  - E.g. word, lemma, POS, case, number...
    - n^{th} word from the top of the stack
    - k^{th} word remaining in the buffer
    - particular node in output tree part created so far

Malt Parser

- Machine learning responsible for training, here the Support Vector Machines (SVM)
  - Classifier. Input vectors: values of all features of the current configuration
  - In addition, during training there is the output value, i.e. action identifier (`shift` / `larc` / `rarc`)
  - The trained oracle (SVM) tells the output value during parsing
- Training on the whole PDT may take weeks!
  - Complexity $O(n^2)$ where n is number of training examples
  - Over 3 million training examples can be extracted from PDT
- Parsing comparatively faster ($\sim$ 1 sentence / second) and can be parallelized

Example of Malt Parsing

STACK		BUFFER
ROOT		Pavel dal Petrovi dvě hrušky .

Pavel dal Petrovi dvě hrušky .
Pavel gave Petr two pears .

Example of Malt Parsing

STACK		BUFFER
ROOT		Pavel dal Petrovi dvě hrušky .

SHIFT

Pavel dal Petrovi dvě hrušky .
Pavel gave Petr two pears .

Example of Malt Parsing

STACK		BUFFER
ROOT Pavel		dal Petrovi dvě hrušky .

Pavel dal Petrovi dvě hrušky .
Pavel gave Petr two pears .

Example of Malt Parsing

STACK	BUFFER
ROOT Pavel	dal Petrovi dvě hrušky .

SHIFT

Pavel dal Petrovi dvě hrušky .
Pavel gave Petr two pears .

Example of Malt Parsing

STACK		BUFFER
ROOT Pavel dal		Petrovi dvě hrušky .

Pavel dal Petrovi dvě hrušky .
Pavel gave Petr two pears .

Example of Malt Parsing

STACK		BUFFER
ROOT Pavel **dal**		Petrovi dvě hrušky .

LARC

Pavel dal Petrovi dvě hrušky .
Pavel gave Petr two pears .

Example of Malt Parsing

STACK	BUFFER
ROOT dal	Petrovi dvě hrušky .

Pavel dal Petrovi dvě hrušky .
Pavel gave Petr two pears .

Example of Malt Parsing

STACK	BUFFER
ROOT dal	Petrovi dvě hrušky .

SHIFT

Pavel dal Petrovi dvě hrušky .
Pavel gave Petr two pears .

Example of Malt Parsing

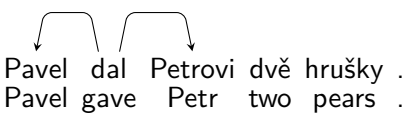
STACK	BUFFER
ROOT dal Petrovi	dvě hrušky .

Pavel dal Petrovi dvě hrušky .
Pavel gave Petr two pears .

Example of Malt Parsing

STACK	BUFFER
ROOT **dal** Petrovi	dvě hrušky .

RARC



Example of Malt Parsing

STACK		BUFFER
ROOT dal		dvě hrušky .

Pavel	dal	Petrovi	dvě	hrušky	.
Pavel	gave	Petr	two	pears	.

Example of Malt Parsing

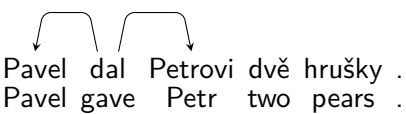
STACK	BUFFER
ROOT dal	dvě hrušky .

SHIFT

Pavel	dal	Petrovi	dvě	hrušky	.	
Pavel	gave	Petr	two	pears	.	

Example of Malt Parsing

STACK	BUFFER
ROOT dal dvě	hrušky .



Example of Malt Parsing

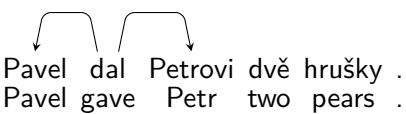
STACK	BUFFER
ROOT dal dvě	hrušky .

SHIFT

Pavel	dal	Petrovi	dvě	hrušky	.	
Pavel	gave	Petr	two	pears	.	

Example of Malt Parsing

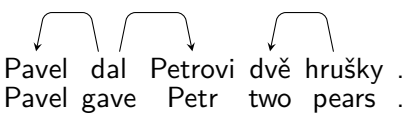
STACK		BUFFER
ROOT dal dvě hrušky		.



Example of Malt Parsing

STACK	BUFFER
ROOT dal dvě **hrušky**	.

LARC



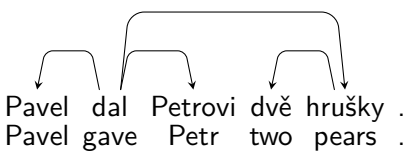
Example of Malt Parsing

STACK		BUFFER
ROOT dal hrušky		.

Example of Malt Parsing

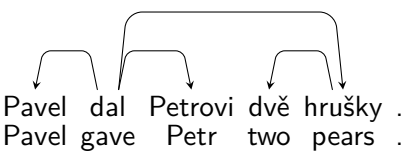
STACK	BUFFER
ROOT **dal** hrušky	.

RARC



Example of Malt Parsing

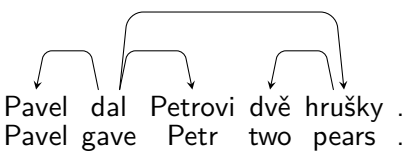
STACK		BUFFER
ROOT dal		.



Example of Malt Parsing

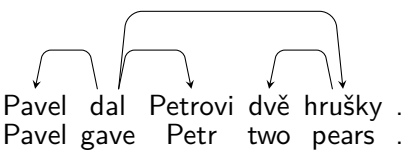
STACK		BUFFER
ROOT dal		.

SHIFT



Example of Malt Parsing

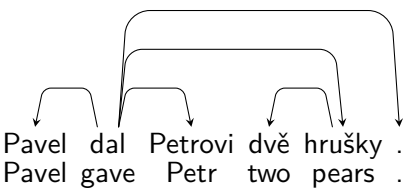
STACK		BUFFER
ROOT dal .		



Example of Malt Parsing

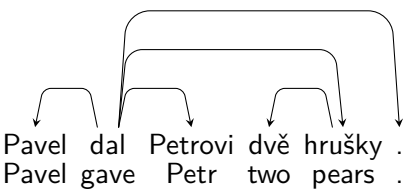
STACK	BUFFER
ROOT **dal** .	

RARC



Example of Malt Parsing

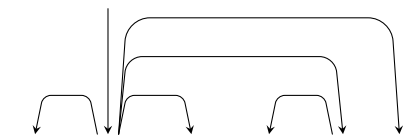
STACK		BUFFER
ROOT dal		



Example of Malt Parsing

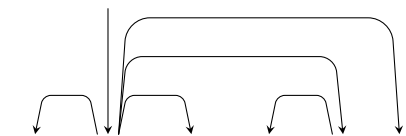
STACK		BUFFER
ROOT dal		

RARC



Example of Malt Parsing

STACK		BUFFER
ROOT		



Nonprojective Mode of Malt

- It can be proven that the above transition system is
  - **correct**
    - resulting graph is always a tree (connected, cycle-free)

Nonprojective Mode of Malt

- It can be proven that the above transition system is
  - **correct**
    - resulting graph is always a tree (connected, cycle-free)
  - **complete** for the set of projective trees
    - every projective tree can be expressed as a sequence of transitions

Nonprojective Mode of Malt

- It can be proven that the above transition system is
  - **correct**
    - resulting graph is always a tree (connected, cycle-free)
  - **complete** for the set of projective trees
    - every projective tree can be expressed as a sequence of transitions
- How to add nonprojective dependencies?
  - Pseudo-projective parsing (Nivre and Nilsson, 2005):
  - Projectivize training tree by re-attaching nodes
  - Encode the transition in extended dependency labels
  - Let the parser predict the projective tree including the labels
  - Read the transformations off the labels, reverse them

Nonprojective Mode of Malt

- It can be proven that the above transition system is
  - **correct**
    - resulting graph is always a tree (connected, cycle-free)
  - **complete** for the set of projective trees
    - every projective tree can be expressed as a sequence of transitions
- How to add nonprojective dependencies?
  - Pseudo-projective parsing (Nivre and Nilsson, 2005):
  - Projectivize training tree by re-attaching nodes
  - Encode the transition in extended dependency labels
  - Let the parser predict the projective tree including the labels
  - Read the transformations off the labels, reverse them

  - New transition **SWAP**:
  - Return second topmost stack word to buffer $\Rightarrow$ swap their order
  - Permitted only for words not swapped before (prevent endless loop)

Nonprojective Parsing

STACK		BUFFER
ROOT		I found the best example ever

I found the best example ever

Nonprojective Parsing

STACK	BUFFER
ROOT	I found the best example ever

SHIFT

I found the best example ever

Nonprojective Parsing

STACK	BUFFER
ROOT I	found the best example ever

I found the best example ever

Nonprojective Parsing

STACK		BUFFER
ROOT I		found the best example ever

SHIFT

I found the best example ever

Nonprojective Parsing

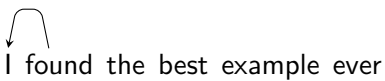
STACK	BUFFER
ROOT I found	the best example ever

I found the best example ever

Nonprojective Parsing

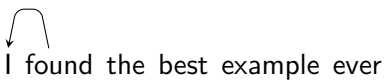
STACK	BUFFER
ROOT | **found**	the best example ever

LARC



Nonprojective Parsing

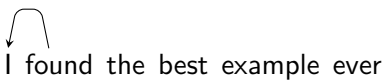
STACK	BUFFER
ROOT found	the best example ever



Nonprojective Parsing

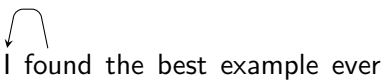
STACK	BUFFER
ROOT found	the best example ever

SHIFT



Nonprojective Parsing

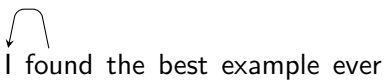
STACK	BUFFER
ROOT found the	best example ever



Nonprojective Parsing

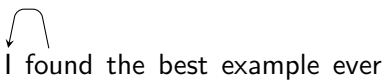
STACK	BUFFER
ROOT found the	best example ever

SHIFT



Nonprojective Parsing

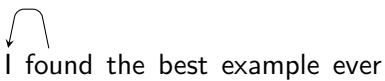
STACK	BUFFER
ROOT found the best	example ever



Nonprojective Parsing

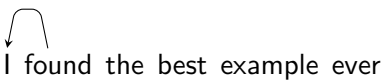
STACK	BUFFER
ROOT found the best	example ever

SHIFT



Nonprojective Parsing

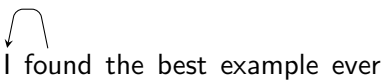
STACK	BUFFER
ROOT found the best example	ever



Nonprojective Parsing

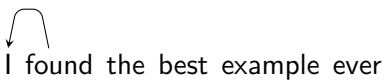
STACK	BUFFER
ROOT found the best example	ever

SWAP



Nonprojective Parsing

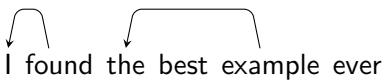
STACK	BUFFER
ROOT found the example	best ever



Nonprojective Parsing

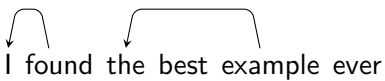
STACK	BUFFER
ROOT found the **example**	best ever

LARC



Nonprojective Parsing

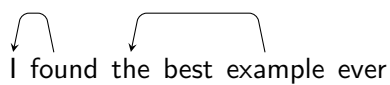
STACK	BUFFER
ROOT found example	best ever



Nonprojective Parsing

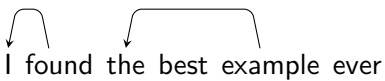
STACK	BUFFER
ROOT found example	best ever

SHIFT



Nonprojective Parsing

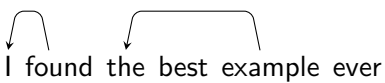
STACK	BUFFER
ROOT found example best	ever



Nonprojective Parsing

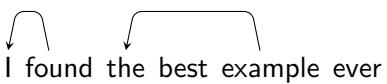
STACK	BUFFER
ROOT found example best	ever

SHIFT



Nonprojective Parsing

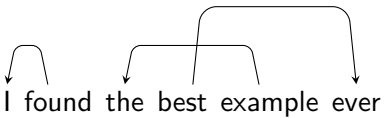
STACK	BUFFER
ROOT found example best ever	



Nonprojective Parsing

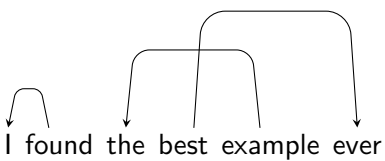
STACK	BUFFER
ROOT found example **best** ever	

RARC



Nonprojective Parsing

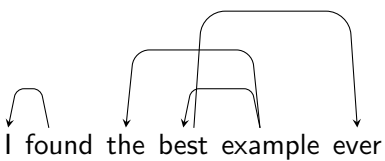
STACK		BUFFER
ROOT found example best		



Nonprojective Parsing

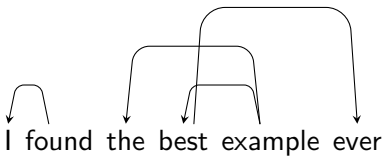
STACK	BUFFER
ROOT found **example** best	

RARC



Nonprojective Parsing

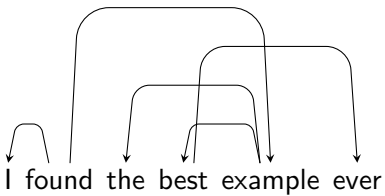
STACK	BUFFER
ROOT found example	



Nonprojective Parsing

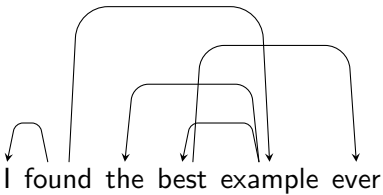
STACK	BUFFER
ROOT **found** example	

RARC



Nonprojective Parsing

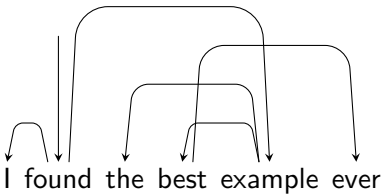
STACK		BUFFER
ROOT found		



Nonprojective Parsing

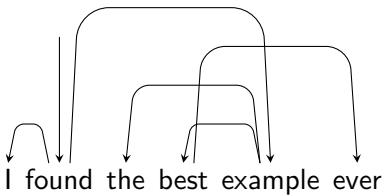
STACK	BUFFER
ROOT found	

RARC



Nonprojective Parsing

STACK		BUFFER
ROOT		



Dependency Parsing

Graph-Based Parsers

Graph-Based Parsers: MST

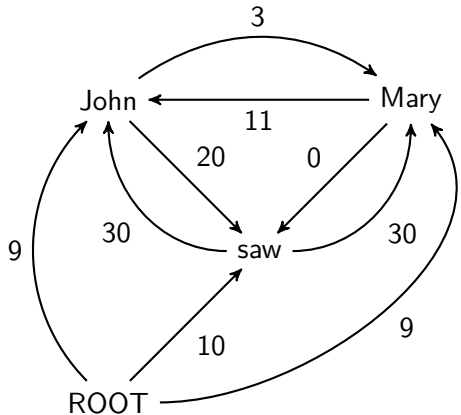
- McDonald et al., HLT-EMNLP 2005
- `http://sourceforge.net/projects/mstparser/`
- MST = maximum spanning tree = [cs] *nejlépe ohodnocená kostra (orientovaného) grafu*
- Start with a total graph
  - We assume any two words can be connected with a dependency
- Gradually remove poorly valued edges
- Machine learning takes care of the weights
  - Train on edge features
  - E.g.: lemma, POS, case… of parent / child node

MST Parser

- Feature engineering (tell the parser what features to track) by modifying the source code (Java)
- Not easy to incorporate 2^{nd} *order features*
  - E.g.: edge weight may depend on the grandparent
- Parser can run in nonprojective mode
- Training on PDT reportedly took about 30 hours
  - Iterate over all feature combinations
  - Look for the most useful ones
- Parsing proper is comparatively fast

Chu-Liu-Edmonds Algorithm (based on slides by Sam Thomson)

Every possible labeled directed edge e between every pair of nodes gets a score, $score(e)$.



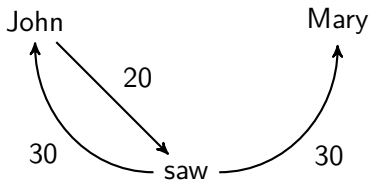
$G = \langle V, E \rangle =$ ($O(n^2)$ edges)

Ex. from *Non-projective Dependency Parsing using Spanning Tree Algorithms,* McDonald et al., EMNLP 2005

Maximum Spanning Tree

Best parse is:

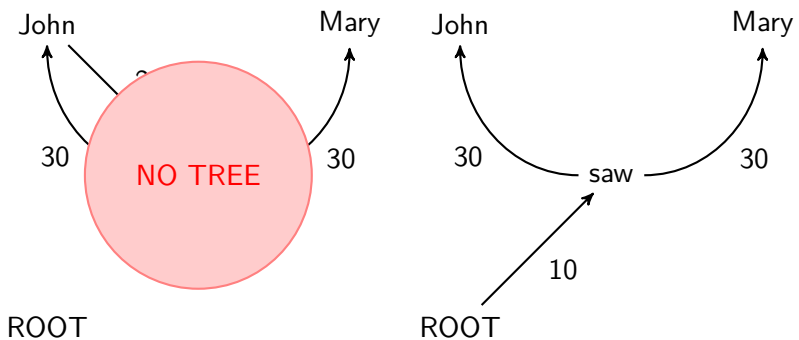
$$A^* = \underset{A \subseteq G}{\text{argmax}} \sum_{e \in A} score(e)$$



Maximum Spanning Tree

Best parse is:

$$A^* = \underset{\substack{A \subseteq G \\ A \text{ is tree}}}{\text{argmax}} \sum_{e \in A} score(e)$$



Chu-Liu-Edmonds Algorithm

- The Chu-Liu-Edmonds algorithm finds the argmax.
  - Chu and Liu, 1965. On the Shortest Arborescence of a Directed Graph. In: *Science Sinica*
  - Edmonds, 1967. Optimum Branchings. In: *JRNBS*

Intuition

- Every non-ROOT node needs exactly 1 incoming edge

Intuition

- Every non-ROOT node needs exactly 1 incoming edge
- Every connected component that doesn't contain ROOT needs exactly 1 incoming edge

Intuition

- Every non-ROOT node needs exactly 1 incoming edge
- Every connected component that doesn't contain ROOT needs exactly 1 incoming edge
  - Greedily pick an incoming edge for each node
  - Do we have a tree? Great!

Intuition

- Every non-ROOT node needs exactly 1 incoming edge
- Every connected component that doesn't contain ROOT needs exactly 1 incoming edge
  - Greedily pick an incoming edge for each node
  - Do we have a tree? Great!
  - No? $\Rightarrow$ There must be a cycle C

Intuition

- Every non-ROOT node needs exactly 1 incoming edge
- Every connected component that doesn't contain ROOT needs exactly 1 incoming edge
  - Greedily pick an incoming edge for each node
  - Do we have a tree? Great!
  - No? $\Rightarrow$ There must be a cycle C
  - One edge in C must get kicked out
  - Also, C needs an incoming edge

Intuition

- Every non-ROOT node needs exactly 1 incoming edge
- Every connected component that doesn't contain ROOT needs exactly 1 incoming edge
  - Greedily pick an incoming edge for each node
  - Do we have a tree? Great!
  - No? $\Rightarrow$ There must be a cycle C
  - One edge in C must get kicked out
  - Also, C needs an incoming edge
  - Choosing edge incoming to C determines which edge to kick out

Chu-Liu-Edmonds Recursive (Inefficient) Definition

```
def maxTree(V, E, ROOT):
    """ returns best tree as a set of edges
        (for each node just one incoming edge) """
    for v in V - ROOT:
        bestInEdge[v] = argmax [e.score for e in inEdges[v]]
        if bestInEdge contains a cycle C:
            # build a new graph where C is contracted into a single node
            vC = new Node()
            V1 = V - C + {vC}
            E1 = {adjust(e) for e in E - C}
            A = maxTree(V1, E1, ROOT)
            return {e.original for e in A} + C - A[vC].kicksOut
    # each node got a parent without creating any cycles
    return bestInEdge
```

Chu-Liu-Edmonds Recursive (Inefficient) Definition

```
def adjust(e):
    e1 = copy(e)
    e1.original = e
    if e.dest in C:
        e1.dest = vC
        e1.kicksOut = bestInEdge[e.dest]
        e1.score = e.score - e1.kicksOut.score
    elif e.src in C:
        e1.src = vC
    return e1
```

Chu-Liu-Edmonds

Consists of two stages:

- **Contracting** (everything *before* the recursive call)
- **Extracting** (everything *after* the recursive call)

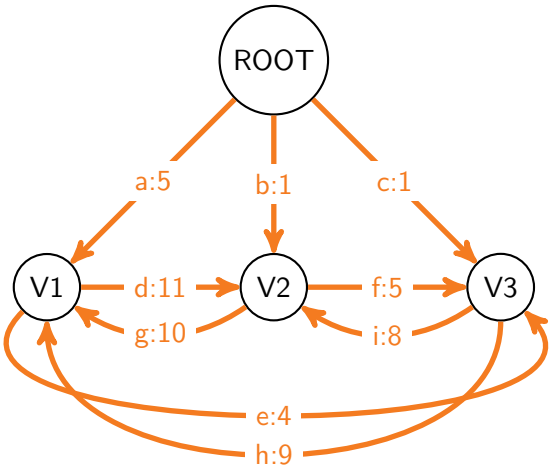
Preprocessing

- Remove every edge incoming to ROOT
  - $\Rightarrow$ ensure that ROOT is the root of any solution
- For every ordered pair of nodes, v_i, v_j, remove all but the highest-scoring edge from v_i to v_j

Contracting Stage

- For each non-root node v, set `bestInEdge[v]` to be its highest scoring incoming edge
- If a cycle C is formed:
  - Contract the nodes in C into a new node v_C
  - Edges outgoing from any node in C now go out of v_C
  - Edges incoming to any node in C now go to v_C
  - For each node u in C, and for each edge e incoming to u from outside of C:
    - Set e`.kicksOut` to `bestInEdge[u]`
    - Set e`.score` to be e`.score - `e`.kicksOut.score`
- Repeat until every non-root node has an incoming edge and no cycles are formed

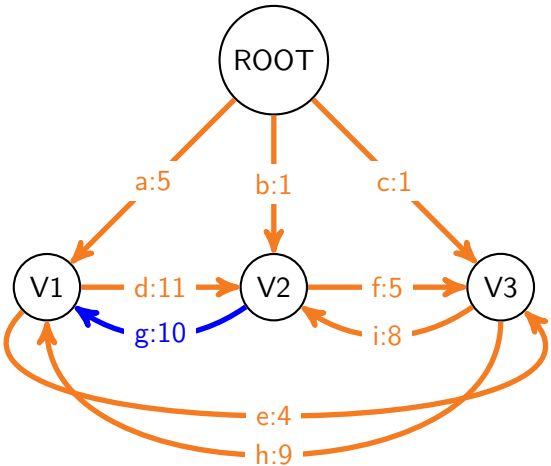
Contracting Stage



	bestInEdge
V1	
V2	
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

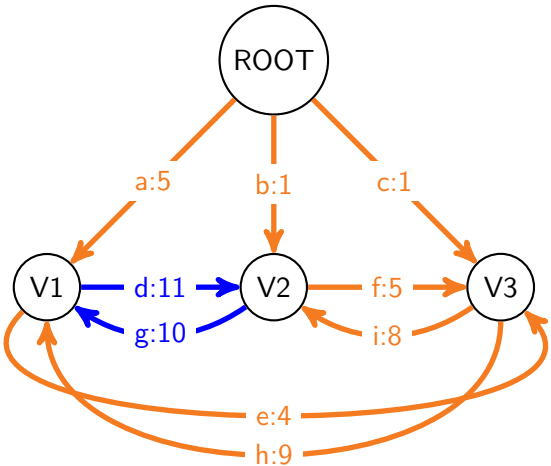
Contracting Stage



	bestInEdge
V1	g
V2	
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

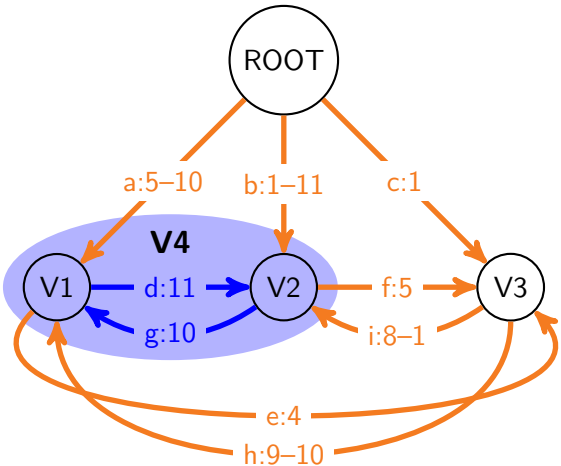
Contracting Stage



	bestInEdge
V1	g
V2	d
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

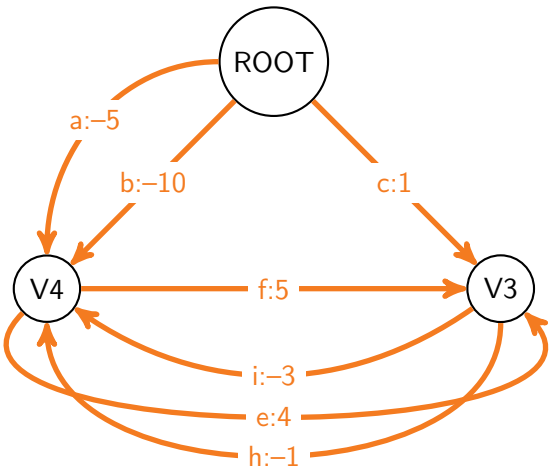
Contracting Stage



	bestInEdge
V1	g
V2	d
V3	

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

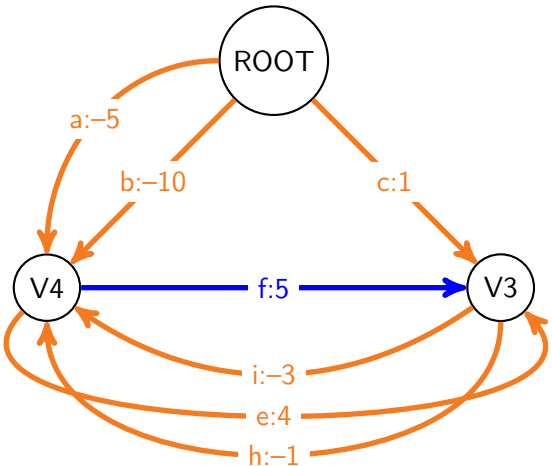
Contracting Stage



	bestInEdge
V1	g
V2	d
V3	
V4	

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

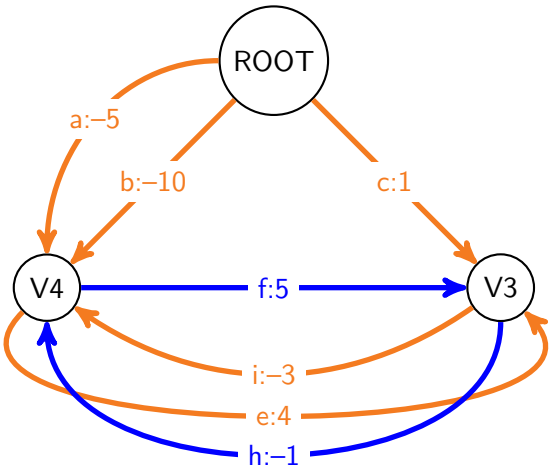
Contracting Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

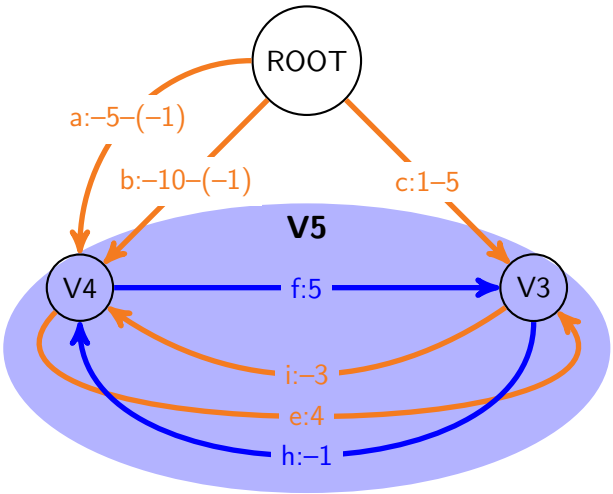
Contracting Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	h

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

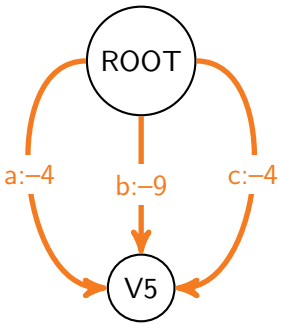
Contracting Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	h

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

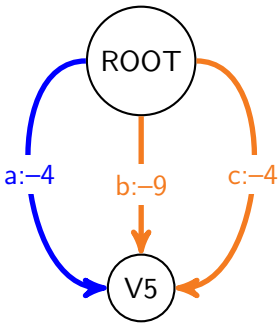
Contracting Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

Contracting Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	a

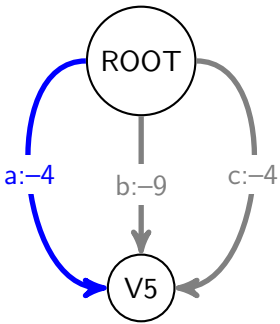
	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

Expanding Stage

- Now every contracted node has exactly one `bestInEdge`.
- This edge kicks out one edge inside, breaking the cycle.

- Go through each `bestInEdge` e in the reverse order that we added them.
- Lock down e, and remove every edge in e.`kicksOut` from `bestInEdge`.

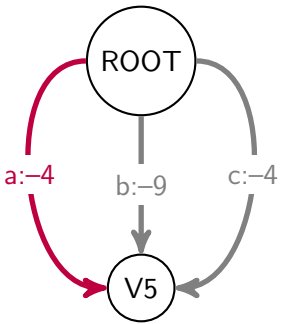
Expanding Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

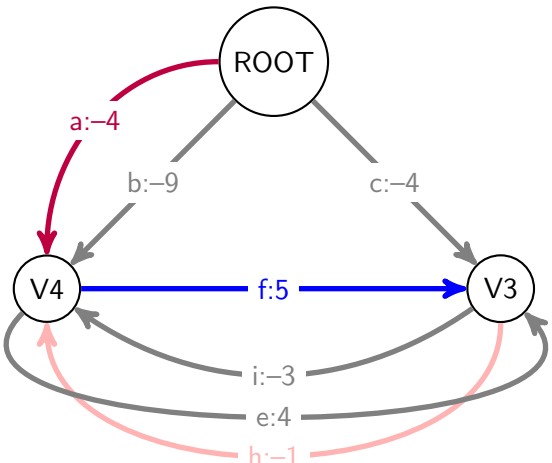
Expanding Stage



	bestInEdge
V1	~~g~~ a
V2	d
V3	f
V4	~~h~~ a
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

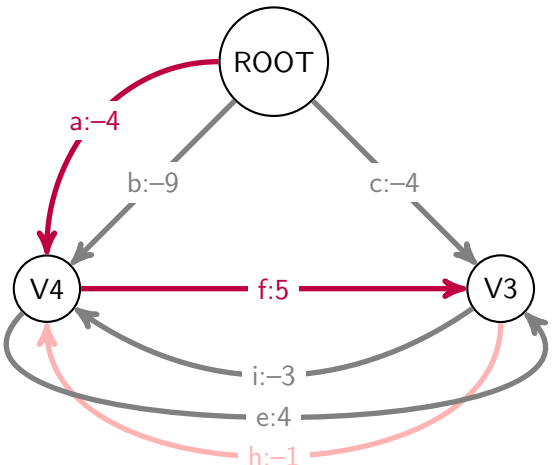
Expanding Stage



	bestInEdge
V1	~~g~~ a
V2	d
V3	f
V4	~~h~~ a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

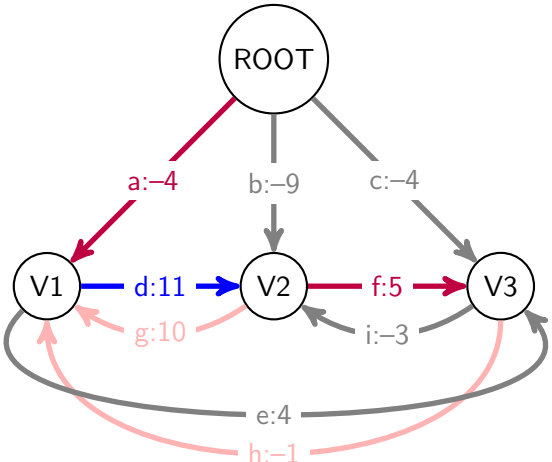
Expanding Stage



	bestInEdge
V1	~~g~~ a
V2	d
V3	f
V4	~~h~~ a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

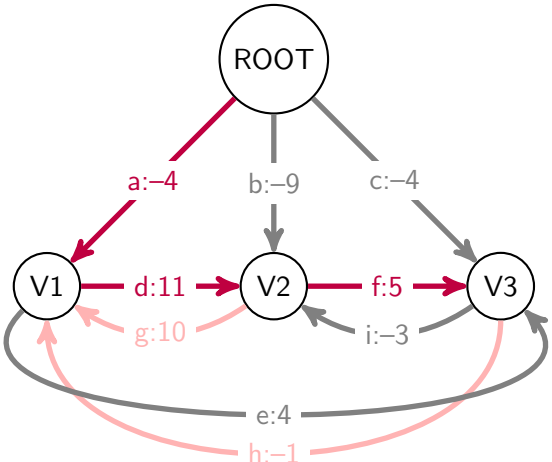
Expanding Stage



	bestInEdge
V1	g a
V2	d
V3	f

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

Expanding Stage



	bestInEdge
V1	g a
V2	d
V3	f

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

Notes

- A greedy algorithm with delayed backtracking to recover from cycles
- Always recovers (one of) the optimal tree(s)

- Efficient implementation: Tarjan, 1977: Finding Optimum Branchings. In: *Networks*
  - Not recursive. Uses a union-find (a.k.a. disjoint-set) data structure to keep track of collapsed nodes
  - An error in Tarjan 1977 fixed by Camerini et al., 1979: A note on finding optimum branchings. In: *Networks*
- Even more efficient: Gabow et al., 1986: Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs. In: *Combinatorica*
  - Uses a Fibonacci heap to keep incoming edges sorted.
  - Finds cycles by following `bestInEdge` instead of randomly visiting nodes.
  - Describes how to constrain ROOT to have only one outgoing edge.

Edge Scoring

First-order features can look at parent node, child node and edge label. For example:

- Number of words between parent and child
- Sequence of POS tags between parent and child
- Is parent to the left or to the right of the child?
- Vector state of a recurrent neural network at parent and child
- Vector embedding of the label
- etc.

Malt and MST Accuracy (before UD)

- Czech (PDT):
  - MST parser over 85%
  - Malt parser over 86%
    - Sentence accuracy ("complete match") 35%, that was high!
  - The two parsers use different strategies $\Rightarrow$ can be combined (either by voting (third parser needed) or one preparing features for the other)
- Other languages (CoNLL shared tasks)
  - MST was slightly better on most languages
  - Accuracies not comparable cross-linguistically

Features Were the Key to Success

- Both Malt and MST can use large number of input text features
- Nontrivial machine learning makes sure that important features get higher weight
- ML algorithms are general classifiers
  - Typically there is a library ready to download
  - The concrete problem (here tree building) must be converted to a sequence of classification decisions, e.g. vectors (feature values + answer)

Features Were the Key to Success

- Both Malt and MST can use large number of input text features
- Nontrivial machine learning makes sure that important features get higher weight
- ML algorithms are general classifiers
  - Typically there is a library ready to download
  - The concrete problem (here tree building) must be converted to a sequence of classification decisions, e.g. vectors (feature values + answer)

- Newer parsers
  - Neural networks (BiLSTM…)
  - Pre-trained word embeddings
  - Joint prediction of morphology and syntax

  - **UDPipe** (`http://ufal.mff.cuni.cz/udpipe`)