Natural Language Processing (CSEP 517): Dependency Syntax and Parsing

Noah Smith

© 2017

University of Washington
nasmith@cs.washington.edu

May 1, 2017

To-Do List

- Online quiz: due Sunday
- Read: Kübler et al. (2009, ch. 1, 2, 6)
- A3 due May 7 (Sunday)
- A4 due May 14 (Sunday)

Dependencies

Informally, you can think of **dependency** structures as a transformation of phrase-structures that

- maintains the word-to-word relationships induced by lexicalization,
- adds labels to them, and
- eliminates the phrase categories.

There are also linguistic theories built on dependencies (Tesnière, 1959; Mel'čuk, 1987), as well as treebanks corresponding to those.

- Free(r)-word order languages (e.g., Czech)

Dependency Tree: Definition

Let $\boldsymbol{x} = \langle x_1, \ldots, x_n \rangle$ be a sentence. Add a special ROOT symbol as "x_0."

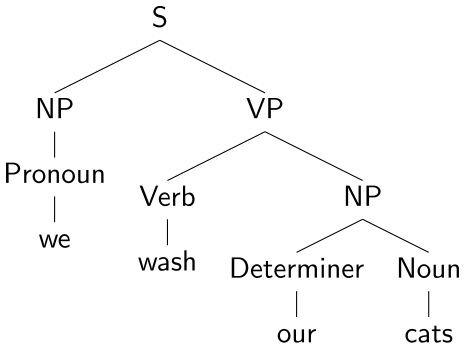
A dependency tree consists of a set of tuples $\langle p, c, \ell \rangle$, where

- $p \in \{0, \ldots, n\}$ is the index of a parent
- $c \in \{1, \ldots, n\}$ is the index of a child
- $\ell \in \mathcal{L}$ is a label

Different annotation schemes define different label sets $\mathcal{L}$, and different constraints on the set of tuples. Most commonly:

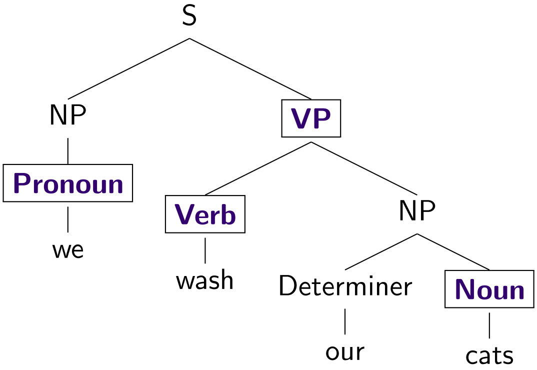
- The tuple is represented as a directed edge from x_p to x_c with label ℓ.
- The directed edges form an arborescence (directed tree) with x_0 as the root (sometimes denoted ROOT).

Example



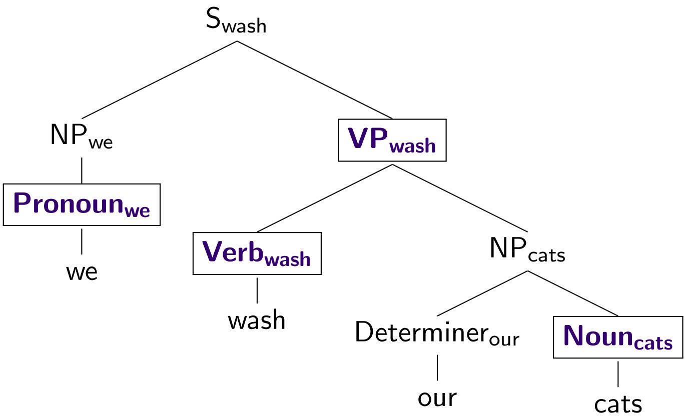
Phrase-structure tree.

Example



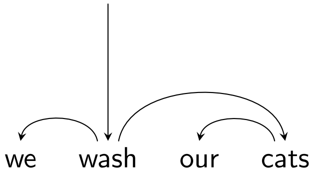
Phrase-structure tree with heads.

Example



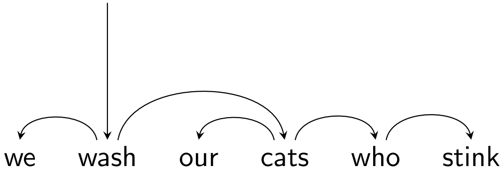
Phrase-structure tree with heads, lexicalized.

Example

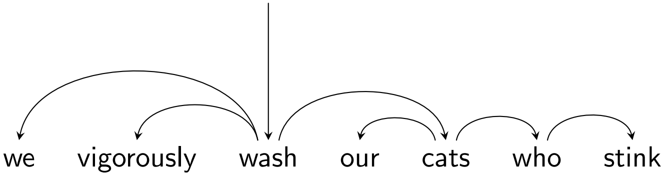


“Bare bones” dependency tree.

Example

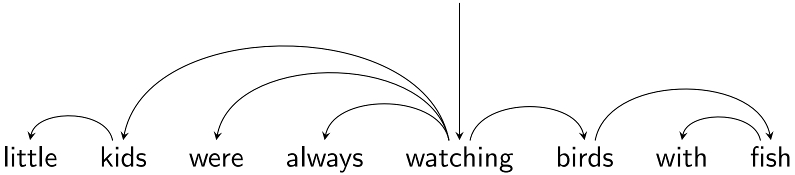


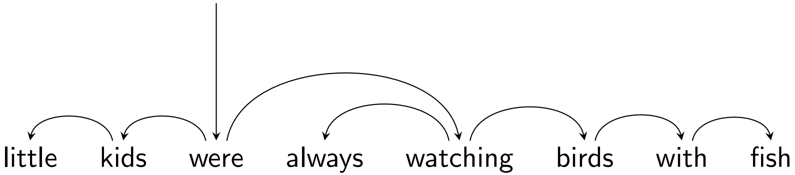
Example



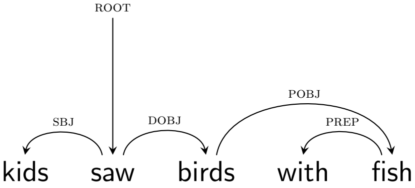
Content Heads vs. Function Heads

Credit: Nathan Schneider





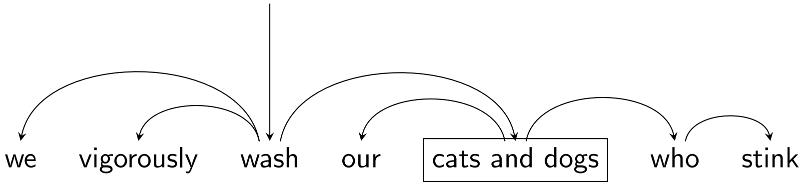
Labels



Key dependency relations captured in the labels include: subject, direct object, preposition object, adjectival modifier, adverbial modifier.

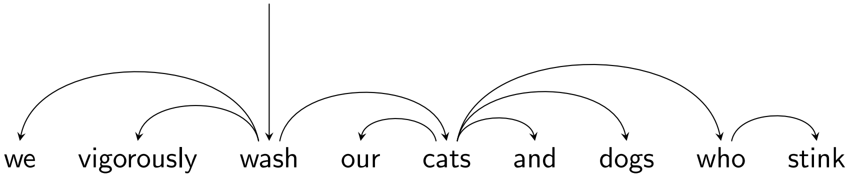
In this lecture, I will mostly not discuss labels, to keep the algorithms simpler.

Coordination Structures



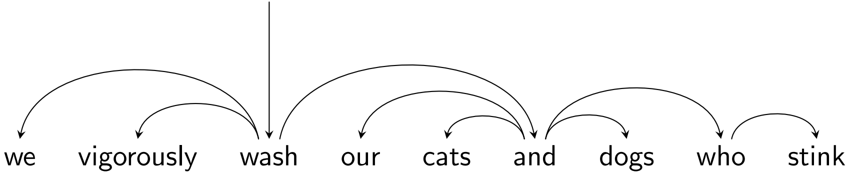
The bugbear of dependency syntax.

Example



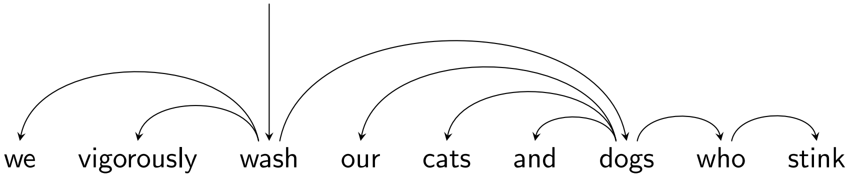
Make the first conjunct the head?

Example



Make the coordinating conjunction the head?

Example



Make the second conjunct the head?

Dependency Schemes

- Transform the treebank: define "head rules" that can select the head child of any node in a phrase-structure tree and label the dependencies.
- More powerful, less local rule sets, possibly collapsing some words into arc labels.
  - Stanford dependencies are a popular example (de Marneffe et al., 2006).
- Direct annotation.

Three Approaches to Dependency Parsing

1. Dynamic programming with the Eisner algorithm.
2. Transition-based parsing with a stack.
3. Chu-Liu-Edmonds algorithm for arborescences.

Dependencies and Grammar

Context-free grammars can be used to encode dependency structures.

For every head word and constellation of dependent children:

$$\mathsf{N}_{\text{head}} \quad \rightarrow \quad \mathsf{N}_{\text{leftmost-sibling}} \ldots \mathsf{N}_{\text{head}} \ldots \mathsf{N}_{\text{rightmost-sibling}}$$

And for every $v \in \mathcal{V}$: $\mathsf{N}_v \rightarrow v$ and $\mathsf{S} \rightarrow \mathsf{N}_v$.

Dependencies and Grammar

Context-free grammars can be used to encode dependency structures.

For every head word and constellation of dependent children:

$$\mathsf{N}_{\text{head}} \quad \rightarrow \quad \mathsf{N}_{\text{leftmost-sibling}} \ldots \mathsf{N}_{\text{head}} \ldots \mathsf{N}_{\text{rightmost-sibling}}$$

And for every $v \in \mathcal{V}$: $\mathsf{N}_v \rightarrow v$ and $\mathsf{S} \rightarrow \mathsf{N}_v$.

A **bilexical** dependency grammar binarizes the dependents, generating only one per rule.

Dependencies and Grammar

Context-free grammars can be used to encode dependency structures.

For every head word and constellation of dependent children:

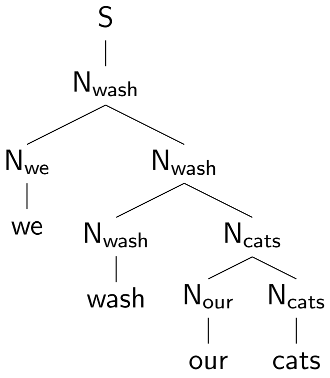
$$\mathsf{N}_{\text{head}} \quad \rightarrow \quad \mathsf{N}_{\text{leftmost-sibling}} \ldots \mathsf{N}_{\text{head}} \ldots \mathsf{N}_{\text{rightmost-sibling}}$$

And for every $v \in \mathcal{V}$: $\mathsf{N}_v \rightarrow v$ and $\mathsf{S} \rightarrow \mathsf{N}_v$.

A **bilexical** dependency grammar binarizes the dependents, generating only one per rule.

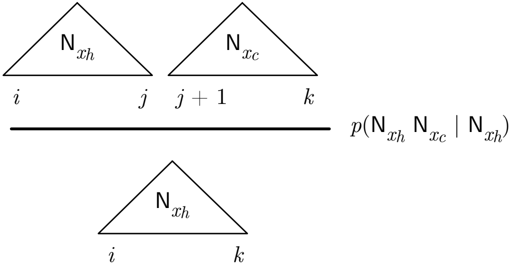
Such a grammar can produce only **projective** trees, which are (informally) trees in which the arcs don't cross.

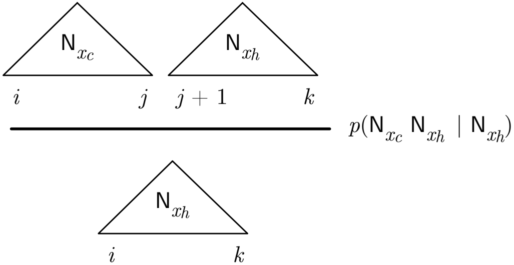
Bilexical Dependency Grammar: Derivation



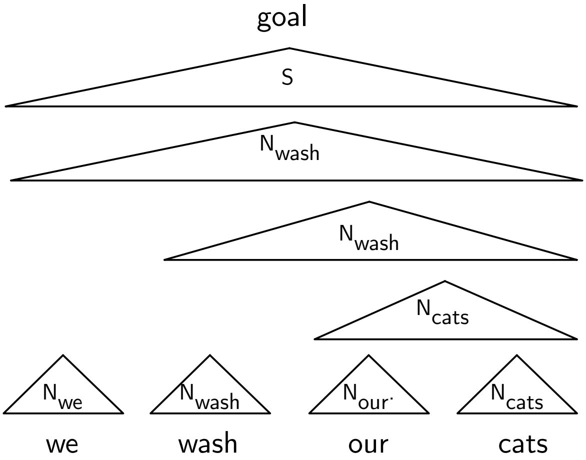
Naïvely, the CKY algorithm will require $O(n^5)$ runtime. Why?

CKY for Bilexical Context-Free Grammars





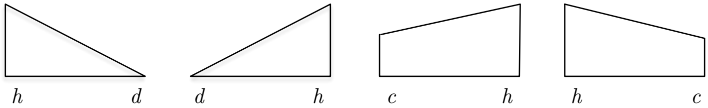
CKY Example


goal
S
N_{wash}
N_{wash}
N_{cats}
N_{we}
N_{wash}
N_{our}
N_{cats}
we
wash
our
cats

Dependency Parsing with the Eisner Algorithm

(Eisner, 1996)

Items:

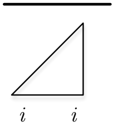


- Both triangles indicate that x_d is a descendant of x_h.
- Both trapezoids indicate that x_c can be attached as the child of x_h.
- In all cases, the words "in between" are descendants of x_h.

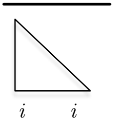
Dependency Parsing with the Eisner Algorithm

(Eisner, 1996)

Initialization:

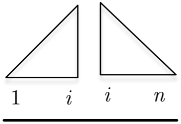


$p(x_i \mid \mathsf{N}_{x_i})$



1

Goal:



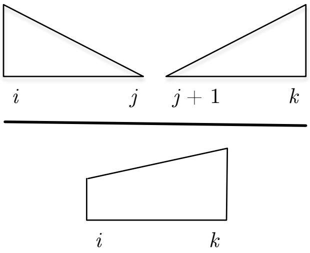
$p(\mathsf{N}_{x_i} \mid \mathsf{S})$

goal

Dependency Parsing with the Eisner Algorithm

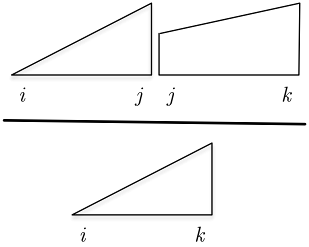
(Eisner, 1996)

Attaching a left dependent:



$p(\mathsf{N}_{xi}\ \mathsf{N}_{xk} \mid \mathsf{N}_{xk})$

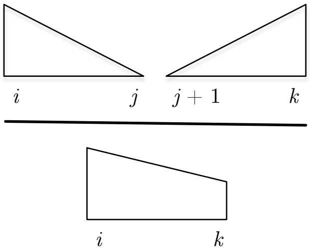
Complete a left child:



Dependency Parsing with the Eisner Algorithm

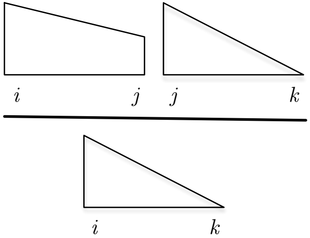
(Eisner, 1996)

Attaching a right dependent:

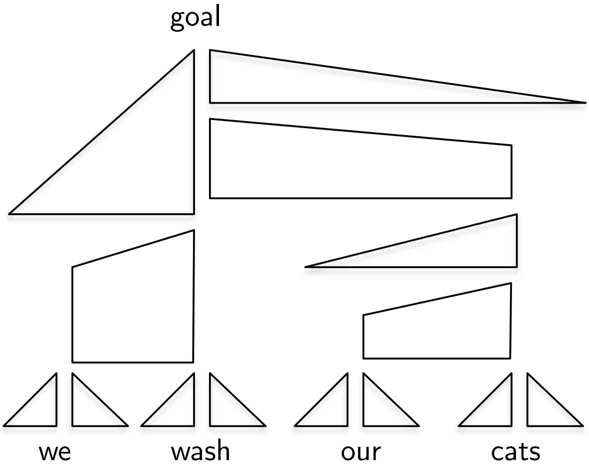


$p(\mathsf{N}_{xi}\ \mathsf{N}_{xk} \mid \mathsf{N}_{xi})$

Complete a right child:



Eisner Algorithm Example



Three Approaches to Dependency Parsing

1. Dynamic programming with the Eisner algorithm.
2. Transition-based parsing with a stack.
3. Chu-Liu-Edmonds algorithm for arborescences.

Transition-Based Parsing

- Process x once, from left to right, making a sequence of greedy parsing decisions.

Transition-Based Parsing

- Process $\boldsymbol{x}$ once, from left to right, making a sequence of greedy parsing decisions.
- Formally, the parser is a **state machine** (*not* a finite-state machine) whose state is represented by a stack S and a buffer B.

Transition-Based Parsing

- Process $\boldsymbol{x}$ once, from left to right, making a sequence of greedy parsing decisions.
- Formally, the parser is a **state machine** (*not* a finite-state machine) whose state is represented by a stack S and a buffer B.
- Initialize the buffer to contain $\boldsymbol{x}$ and the stack to contain the root symbol.

Transition-Based Parsing

- Process $\boldsymbol{x}$ once, from left to right, making a sequence of greedy parsing decisions.
- Formally, the parser is a **state machine** (*not* a finite-state machine) whose state is represented by a stack S and a buffer B.
- Initialize the buffer to contain $\boldsymbol{x}$ and the stack to contain the root symbol.
- The "arc standard" transition set (Nivre, 2004):
  - SHIFT the word at the front of the buffer B onto the stack S.
  - RIGHT-ARC: $u =$ pop(S); $v =$ pop(S); push$(S, v \rightarrow u)$.
  - LEFT-ARC: $u =$ pop(S); $v =$ pop(S); push$(S, v \leftarrow u)$.

(For labeled parsing, add labels to the RIGHT-ARC and LEFT-ARC transitions.)

Transition-Based Parsing

- Process $\boldsymbol{x}$ once, from left to right, making a sequence of greedy parsing decisions.
- Formally, the parser is a **state machine** (*not* a finite-state machine) whose state is represented by a stack S and a buffer B.
- Initialize the buffer to contain $\boldsymbol{x}$ and the stack to contain the root symbol.
- The "arc standard" transition set (Nivre, 2004):
  - SHIFT the word at the front of the buffer B onto the stack S.
  - RIGHT-ARC: $u = \text{pop}(S)$; $v = \text{pop}(S)$; $\text{push}(S, v \rightarrow u)$.
  - LEFT-ARC: $u = \text{pop}(S)$; $v = \text{pop}(S)$; $\text{push}(S, v \leftarrow u)$.

  (For labeled parsing, add labels to the RIGHT-ARC and LEFT-ARC transitions.)
- During parsing, apply a **classifier** to decide which transition to take next, greedily. No backtracking.

Transition-Based Parsing: Example

Buffer B:

we
vigorously
wash
our
cats
who
stink

Stack S:

ROOT

Actions:

Transition-Based Parsing: Example

Buffer B:

vigorously
wash
our
cats
who
stink

Stack S:

we
ROOT

Actions: SHIFT

Transition-Based Parsing: Example

Buffer B:

wash
our
cats
who
stink

Stack S:

vigorously
we
ROOT

Actions: SHIFT SHIFT

Transition-Based Parsing: Example

Stack S:

wash
vigorously
we
ROOT

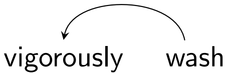
Buffer B:

our
cats
who
stink

Actions: SHIFT SHIFT SHIFT

Transition-Based Parsing: Example

Stack S:



we

ROOT

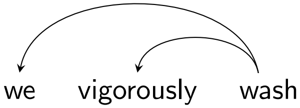
Buffer B:

our
cats
who
stink

Actions: SHIFT SHIFT SHIFT LEFT-ARC

Transition-Based Parsing: Example

Stack S:



ROOT

Buffer B:

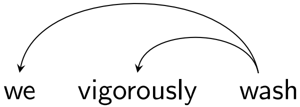
our
cats
who
stink

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC

Transition-Based Parsing: Example

Stack S:

our



ROOT

Buffer B:

cats
who
stink

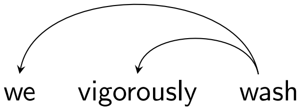
Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT

Transition-Based Parsing: Example

Stack S:

cats

our



ROOT

Buffer B:

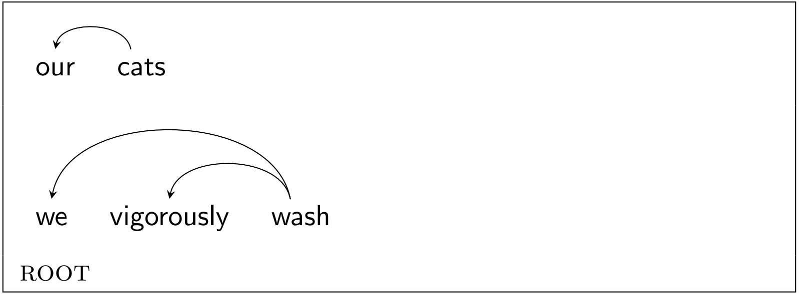
who

stink

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT SHIFT

Transition-Based Parsing: Example

Stack S:



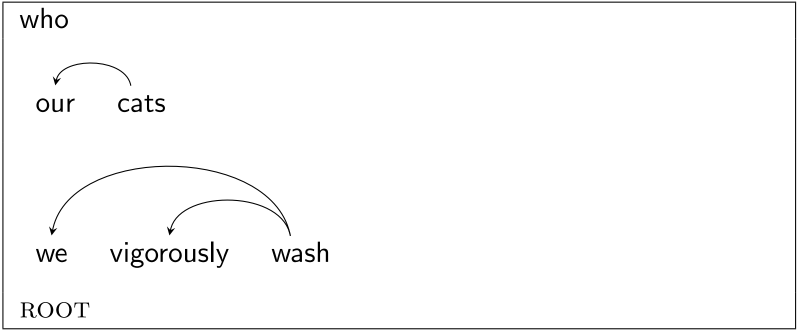
Buffer B:

who
stink

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT SHIFT LEFT-ARC

Transition-Based Parsing: Example

Stack S:



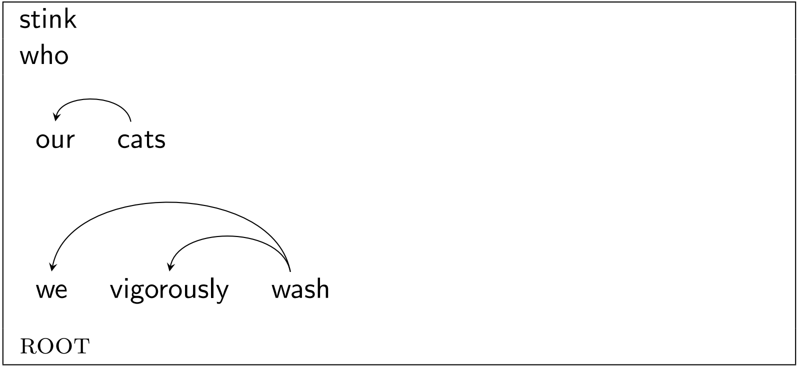
Buffer B:

stink

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT SHIFT LEFT-ARC SHIFT

Transition-Based Parsing: Example

Stack S:

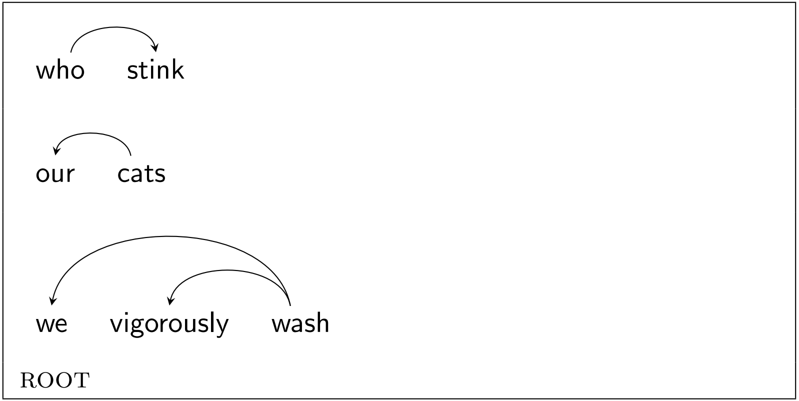


Buffer B:

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT SHIFT LEFT-ARC SHIFT SHIFT

Transition-Based Parsing: Example

Stack S:

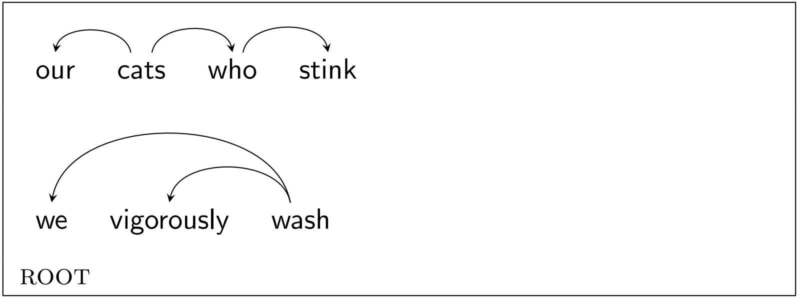


Buffer B:

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT SHIFT LEFT-ARC SHIFT SHIFT RIGHT-ARC

Transition-Based Parsing: Example

Stack S:

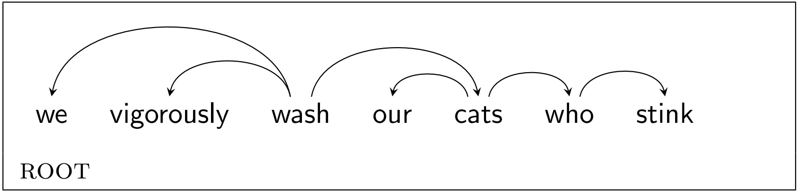


Buffer B:

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT SHIFT LEFT-ARC SHIFT SHIFT RIGHT-ARC RIGHT-ARC

Transition-Based Parsing: Example

Stack S:

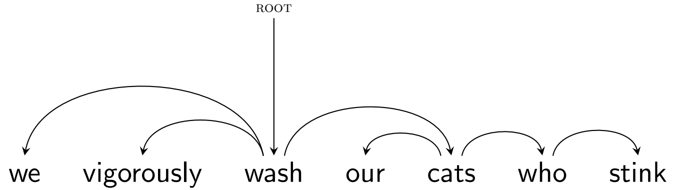


Buffer B:

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT SHIFT LEFT-ARC SHIFT SHIFT RIGHT-ARC RIGHT-ARC RIGHT-ARC

Transition-Based Parsing: Example

Stack S:



Buffer B:

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT SHIFT LEFT-ARC SHIFT SHIFT RIGHT-ARC RIGHT-ARC RIGHT-ARC RIGHT-ARC

The Core of Transition-Based Parsing: Classification

- At each iteration, choose among {SHIFT, RIGHT-ARC, LEFT-ARC}.
  (Actually, among all $\mathcal{L}$-labeled variants of RIGHT- and LEFT-ARC.)

The Core of Transition-Based Parsing: Classification

- At each iteration, choose among {SHIFT, RIGHT-ARC, LEFT-ARC}. (Actually, among all $\mathcal{L}$-labeled variants of RIGHT- and LEFT-ARC.)
- Features can look S, B, and the history of past actions—usually there is no decomposition into local structures.

The Core of Transition-Based Parsing: Classification

- At each iteration, choose among {SHIFT, RIGHT-ARC, LEFT-ARC}. (Actually, among all $\mathcal{L}$-labeled variants of RIGHT- and LEFT-ARC.)
- Features can look S, B, and the history of past actions—usually there is no decomposition into local structures.
- Training data: "oracle" transition sequence that gives the right tree converts into $2 \cdot n$ pairs: ⟨state, correct transition⟩. Each word gets SHIFTed once and participates as a child in one ARC.

Transition-Based Parsing: Remarks

- Can also be applied to phrase-structure parsing (e.g., Sagae and Lavie, 2006). Keyword: "shift-reduce" parsing.
- The algorithm for making decisions doesn't need to be greedy; can maintain multiple hypotheses.
  - E.g., **beam search**, which we'll discuss in the context of machine translation later.
- Potential flaw: the classifier is typically trained under the assumption that previous classification decisions were all *correct*.
  - As yet, no principled solution to this problem, but see "dynamic oracles" (Goldberg and Nivre, 2012).

Three Approaches to Dependency Parsing

1. Dynamic programming with the Eisner algorithm.
2. Transition-based parsing with a stack.
3. Chu-Liu-Edmonds algorithm for arborescences.

Acknowledgment

Slides are mostly adapted from those by Swabha Swayamdipta and Sam Thomson.

Features in Dependency Parsing

For the Eisner algorithm, the score of an unlabeled parse $\boldsymbol{y}$ was

$$s_{\text{global}}(\boldsymbol{y}) = \sum_{c=1}^{n} \log p(x_c \mid \mathsf{N}_{x_c}) + \log \begin{cases} p(\mathsf{N}_{x_c}\ \mathsf{N}_{x_p} \mid \mathsf{N}_{x_p}) & \text{if } \langle p, c\rangle \in \boldsymbol{y}\ \wedge\ c < p\ \wedge\ p > 0 \\ p(\mathsf{N}_{x_p}\ \mathsf{N}_{x_c} \mid \mathsf{N}_{x_p}) & \text{if } \langle p, c\rangle \in \boldsymbol{y}\ \wedge\ c > p\ \wedge\ p > 0 \\ p(\mathsf{N}_{x_c} \mid \mathsf{S}) & \text{if } \langle 0, c\rangle \in \boldsymbol{y} \end{cases}$$

For transition-based parsing, we could use any past decisions to score the current decision:

$$s_{\text{global}}(\boldsymbol{y}) = s(\boldsymbol{a}) = \sum_{i=1}^{|\boldsymbol{a}|} s(a_i \mid \boldsymbol{a}_{0:i-1})$$

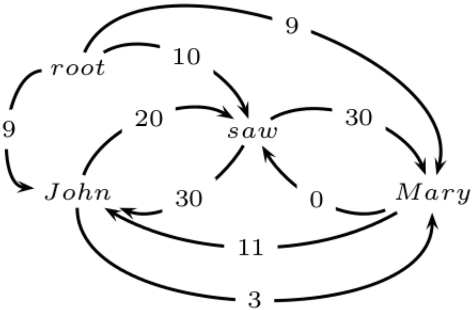
We gave up on any guarantee of finding the best possible $\boldsymbol{y}$ in favor of arbitrary features.

- For a neural network-based model that fully exploits this, see Dyer et al. (2015).

Graph-Based Dependency Parsing

(McDonald et al., 2005)

Every possible directed edge e between a parent p and a child c gets a local score, $s(e)$.



This set, E, contains $O(n^2)$ edges.
No incoming edges to x_0, ensuring that it will be the root.

First-Order Graph-Based (FOG) Dependency Parsing
(McDonald et al., 2005)

$$\boldsymbol{y}^* = \operatorname*{argmax}_{\boldsymbol{y} \subset E} s_{\mathsf{global}}(\boldsymbol{y}) = \operatorname*{argmax}_{\boldsymbol{y} \subset E} \sum_{e \in \boldsymbol{y}} s(e)$$

subject to the constraint that $\boldsymbol{y}$ is an *arborescence*

Classical algorithm to efficiently solve this problem: Chu and Liu (1965), Edmonds (1967)

Chu-Liu-Edmonds Intuitions

- Every non-root node needs exactly one incoming edge.

Chu-Liu-Edmonds Intuitions

- Every non-root node needs exactly one incoming edge.
- In fact, every connected component that doesn't contain x_0 needs exactly one incoming edge.

Chu-Liu-Edmonds Intuitions

- Every non-root node needs exactly one incoming edge.
- In fact, every connected component that doesn't contain x_0 needs exactly one incoming edge.

High-level view of the algorithm:

1. For every c, pick an incoming edge (i.e., pick a parent)—greedily.
2. If this forms an arborescence, you are done!
3. Otherwise, it's because there's a cycle, C.
   - Arborescences can't have cycles, so some edge in C needs to be kicked out.
   - We also need to find an incoming edge for C.
   - Choosing the incoming edge for C determines which edge to kick out.

Chu-Liu-Edmonds: Recursive (Inefficient) Definition

```
def maxArborescence(V, E, ROOT):
    # returns best arborescence as a map from each node to its parent
    for c in V \ ROOT:
        bestInEdge[c] ← argmax_{e∈E:e=⟨p,c⟩} e.s   # i.e., s(e)

        if bestInEdge contains a cycle C:
            # build a new graph where C is contracted into a single node
            v_C ← new Node()
            V′ ← V ∪ {v_C} \ C
            E′ ← {adjust(e, v_C) for e ∈ E \ C}
            A ← maxArborescence(V′, E′, ROOT)
            return {e.original for e ∈ A} ∪ C \ {A[v_C].kicksOut}
    # each node got a parent without creating any cycles
    return bestInEdge
```

Understanding Chu-Liu-Edmonds

There are two stages:

- **Contraction** (the stuff before the recursive call)
- **Expansion** (the stuff after the recursive call)

Chu-Liu-Edmonds: Contraction

- 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

  `adjust` subroutine on next slide performs the following:
  - Edges incoming to any node in C now get destination v_C
  - For each node v in C, and for each edge e incoming to v from outside of C:
    - Set e.`kicksOut` to `bestInEdge`[v], and
    - Set $e.s$ to be $e.s - e$.`kicksOut`$.s$
  - Edges outgoing from any node in C now get source v_C
- Repeat until every non-ROOT node has an incoming edge and no cycles are formed

Chu-Liu-Edmonds: Edge Adjustment Subroutine

def `adjust`(e, v_C):

$e' \leftarrow \text{copy}(e)$

$e'.\texttt{original} \leftarrow e$

if $e.\texttt{dest} \in C$:

$e'.\texttt{dest} \leftarrow v_C$

$e'.\texttt{kicksOut} \leftarrow \texttt{bestInEdge}[e.\texttt{dest}]$

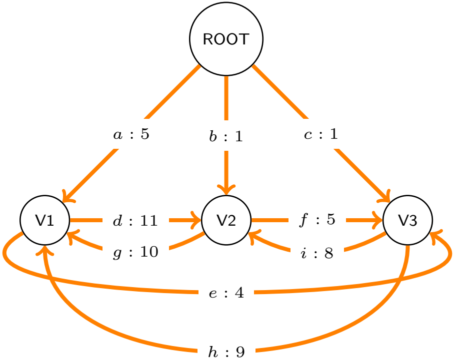
$e'.s \leftarrow e.s - e'.\texttt{kicksOut}.s$

elif $e.\texttt{src} \in C$:

$e'.\texttt{src} \leftarrow v_C$

return e'

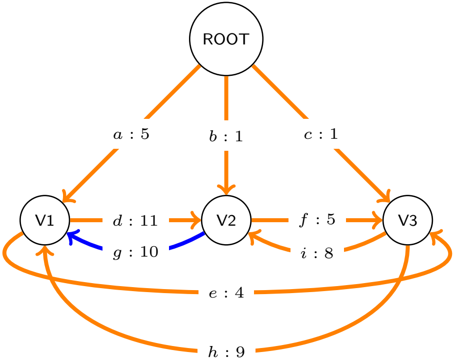
Contraction Example



	bestInEdge
V1	
V2	
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

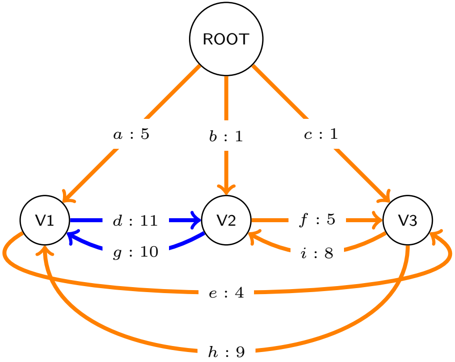
Contraction Example



	bestInEdge
V1	g
V2	
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

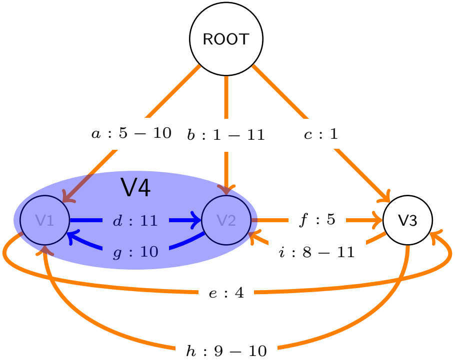
Contraction Example



	bestInEdge
V1	g
V2	d
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

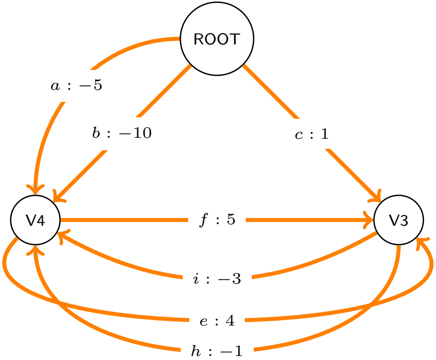
Contraction Example



	bestInEdge
V1	g
V2	d
V3	

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

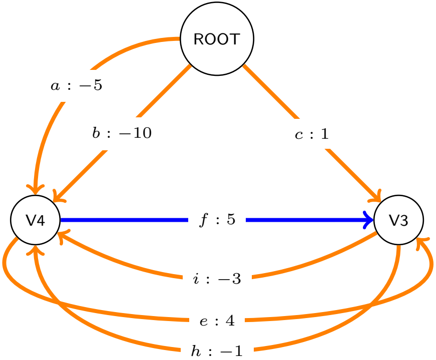
Contraction Example



	bestInEdge
V1	g
V2	d
V3	
V4	

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

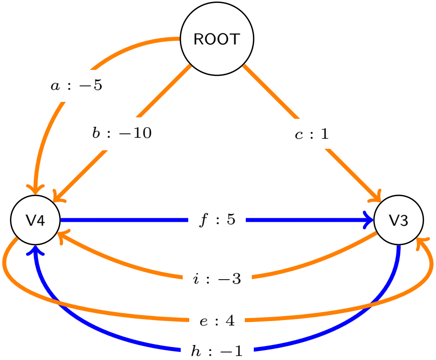
Contraction Example



	bestInEdge
V1	g
V2	d
V3	f
V4	

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

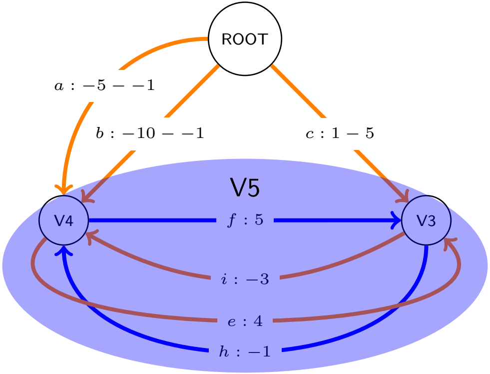
Contraction Example



	bestInEdge
V1	g
V2	d
V3	f
V4	h

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

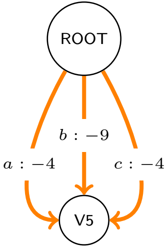
Contraction Example



	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	

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

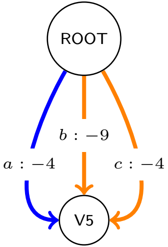
Contraction Example



	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

Contraction Example



	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	a

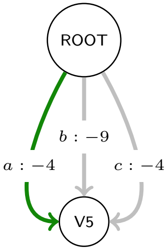
	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

Chu-Liu-Edmonds: Expansion

After the contraction stage, every contracted node will have exactly one `bestInEdge`. This edge will kick out one edge inside the contracted node, breaking the cycle.

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

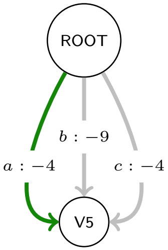
Expansion Example



	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

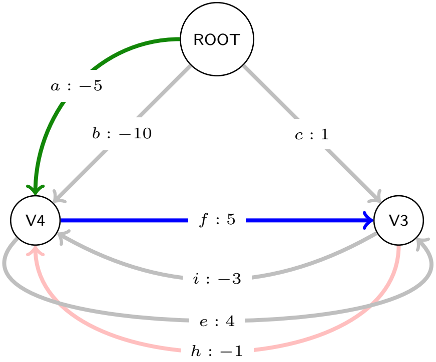
Expansion Example



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

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

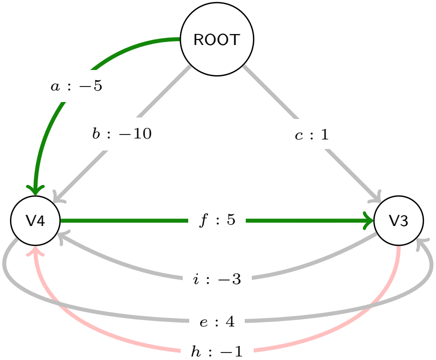
Expansion Example



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

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

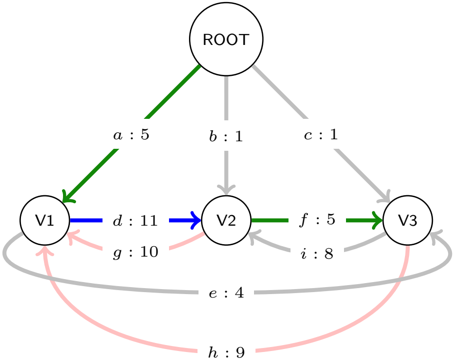
Expansion Example



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

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

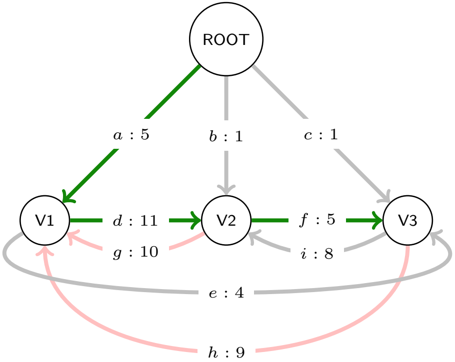
Expansion Example



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

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

Expansion Example



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

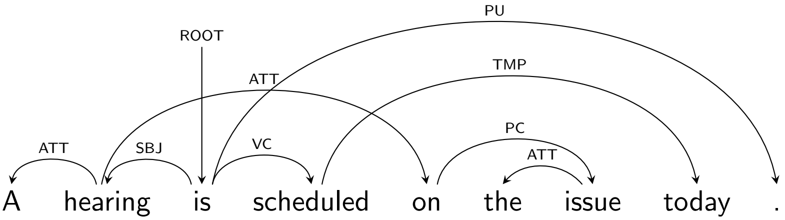
	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

Observation

The set of arborescences strictly includes the set of projective dependency trees.

Is this a good thing or a bad thing?

Nonprojective Example



Chu-Liu-Edmonds: Notes

- This is a greedy algorithm with a clever form of delayed backtracking to recover from inconsistent decisions (cycles).

Chu-Liu-Edmonds: Notes

- This is a greedy algorithm with a clever form of delayed backtracking to recover from inconsistent decisions (cycles).
- CLE is exact: it always recovers an optimal arborescence.

Chu-Liu-Edmonds: Notes

- This is a greedy algorithm with a clever form of delayed backtracking to recover from inconsistent decisions (cycles).
- CLE is exact: it always recovers an optimal arborescence.
- What about labeled dependencies?
  - As a matter of preprocessing, for each $\langle p, c \rangle$, keep only the top-scoring labeled edge.

Chu-Liu-Edmonds: Notes

- This is a greedy algorithm with a clever form of delayed backtracking to recover from inconsistent decisions (cycles).
- CLE is exact: it always recovers an optimal arborescence.
- What about labeled dependencies?
  - As a matter of preprocessing, for each $\langle p, c \rangle$, keep only the top-scoring labeled edge.
- Tarjan (1977) offered a more efficient, but unfortunately incorrect, implementation.
  Camerini et al. (1979) corrected it.
  The approach is not recursive; instead using a disjoint set data structure to keep track of collapsed nodes.
  Even better: Gabow et al. (1986) used a Fibonacci heap to keep incoming edges sorted, and finds cycles in a more sensible way. Also constrains root to have only one outgoing edge.
  With these tricks, $O(n^2)$ runtime.

More Details on Statistical Dependency Parsing

- What about the scores? McDonald et al. (2005) used carefully-designed features and (something close to) the structured perceptron; Kiperwasser and Goldberg (2016) used bidirectional recurrent neural networks.

More Details on Statistical Dependency Parsing

- What about the scores? McDonald et al. (2005) used carefully-designed features and (something close to) the structured perceptron; Kiperwasser and Goldberg (2016) used bidirectional recurrent neural networks.
- What about higher-order parsing? Requires approximate inference, e.g., dual decomposition (Martins et al., 2013).

Important Tradeoffs (and Not Just in NLP)

1. Two extremes:
   - Specialized algorithm that efficiently solves your problem, under your assumptions. E.g., Chu-Liu-Edmonds for FOG dependency parsing.
   - General-purpose method that solves many problems, allowing you to test the effect of different assumptions. E.g., dynamic programming, transition-based methods, some forms of approximate inference.

Important Tradeoffs (and Not Just in NLP)

1. Two extremes:
    - Specialized algorithm that efficiently solves your problem, under your assumptions. E.g., Chu-Liu-Edmonds for FOG dependency parsing.
    - General-purpose method that solves many problems, allowing you to test the effect of different assumptions. E.g., dynamic programming, transition-based methods, some forms of approximate inference.
2. Two extremes:
    - Fast (linear-time) but greedy
    - Model-optimal but slow

Important Tradeoffs (and Not Just in NLP)

1. Two extremes:
   - Specialized algorithm that efficiently solves your problem, under your assumptions. E.g., Chu-Liu-Edmonds for FOG dependency parsing.
   - General-purpose method that solves many problems, allowing you to test the effect of different assumptions. E.g., dynamic programming, transition-based methods, some forms of approximate inference.
2. Two extremes:
   - Fast (linear-time) but greedy
   - Model-optimal but slow
     - Dirty secret: the best way to get (English) dependency trees is to run phrase-structure parsing, then convert.

References I

Paolo M. Camerini, Luigi Fratta, and Francesco Maffioli. A note on finding optimum branchings. *Networks*, 9 (4):309–312, 1979.

Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965.

Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *Proc. of LREC*, 2006.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. Transition-based dependency parsing with stack long short-term memory. In *Proc. of ACL*, 2015.

Jack Edmonds. Optimum branchings. *Journal of Research of the National Bureau of Standards*, 71B:233–240, 1967.

Jason M. Eisner. Three new probabilistic models for dependency parsing: An exploration. In *Proc. of COLING*, 1996.

Harold N. Gabow, Zvi Galil, Thomas Spencer, and Robert E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

Yoav Goldberg and Joakim Nivre. A dynamic oracle for arc-eager dependency parsing. In *Proc. of COLING*, 2012.

Eliyahu Kiperwasser and Yoav Goldberg. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *Transactions of the Association for Computational Linguistics*, 4:313–327, 2016.

References II

Sandra Kübler, Ryan McDonald, and Joakim Nivre. *Dependency Parsing*. Synthesis Lectures on Human Language Technologies. Morgan and Claypool, 2009. URL http://www.morganclaypool.com/doi/pdf/10.2200/S00169ED1V01Y200901HLT002.

André F. T. Martins, Miguel Almeida, and Noah A. Smith. Turning on the turbo: Fast third-order non-projective turbo parsers. In *Proc. of ACL*, 2013.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajic. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of HLT-EMNLP*, 2005. URL http://www.aclweb.org/anthology/H/H05/H05-1066.pdf.

Igor A. Mel'čuk. *Dependency Syntax: Theory and Practice*. State University Press of New York, 1987.

Joakim Nivre. Incrementality in deterministic dependency parsing. In *Proc. of ACL*, 2004.

Kenji Sagae and Alon Lavie. A best-first probabilistic shift-reduce parser. In *Proc. of COLING-ACL*, 2006.

Robert E. Tarjan. Finding optimum branchings. *Networks*, 7:25–35, 1977.

L. Tesnière. *Éléments de Syntaxe Structurale*. Klincksieck, 1959.