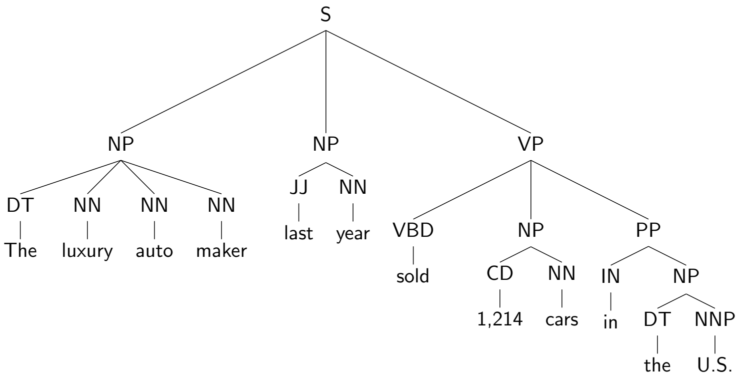
Natural Language Processing (CSE 447/547M): Dependency Syntax and Parsing

Noah A. Smith
Swabha Swayamdipta
Jungo Kasai

University of Washington
{nasmith,swabha,jkasai}@cs.washington.edu

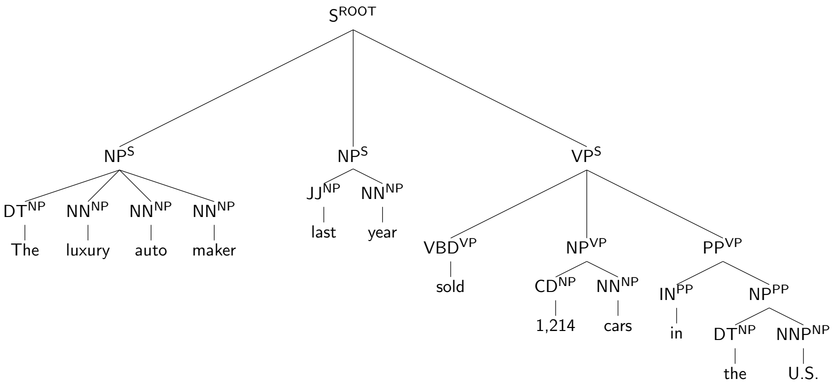
Feb 25, 2019

Recap: Phrase Structure



Parent Annotation

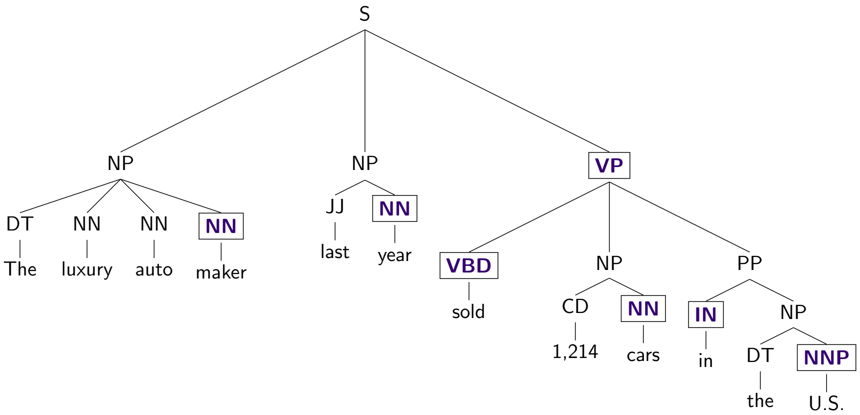
(Johnson, 1998)



Increases the "vertical" Markov order:

$$p(\text{children} \mid \text{parent}, \text{grandparent})$$

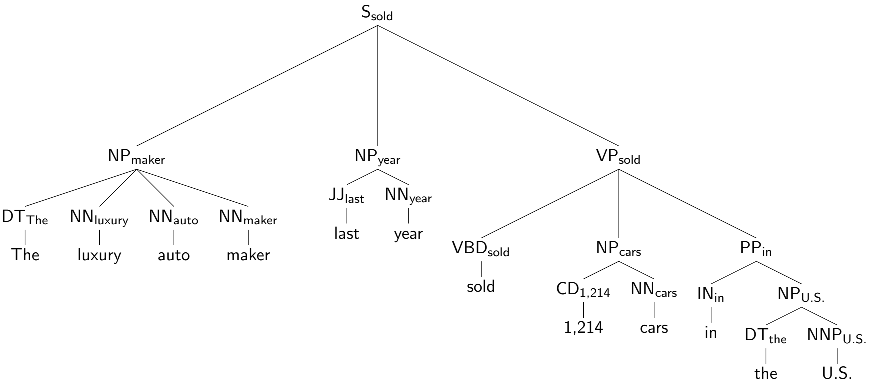
Headedness



Suggests "horizontal" markovization:

$$p(\text{children} \mid \text{parent}) = p(\text{head} \mid \text{parent}) \cdot \prod_i p(i\text{th sibling} \mid \text{head}, \text{parent})$$

Lexicalization



Each node shares a **lexical head** with its head child.

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."

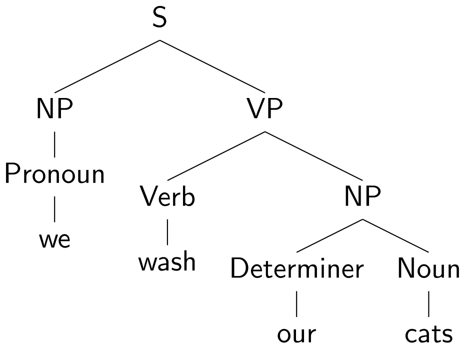
A dependency tree consists of a set of tuples $\langle p, c, \ell \rangle$, where

- $p \in \{0, \ldots, n\}$ 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:

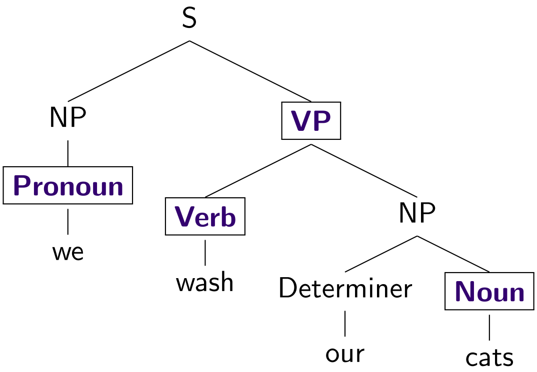
- The tuple is represented as a directed edge from x_p to x_c with label ℓ.
- The directed edges form an arborescence (directed tree) with x_0 as the root (sometimes denoted ROOT).

Example



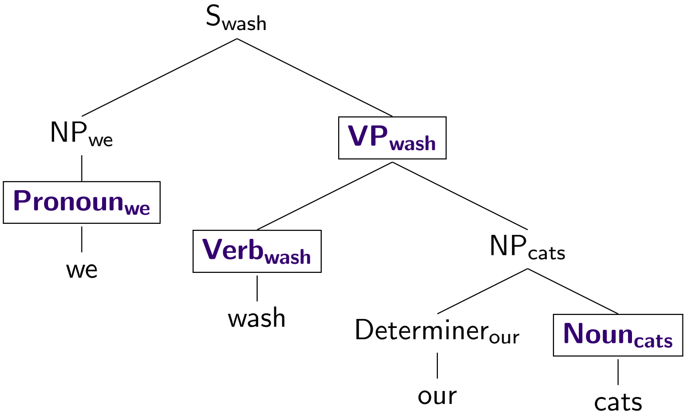
Phrase-structure tree.

Example



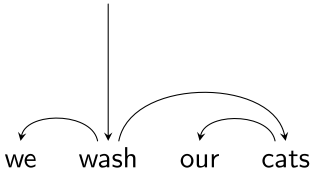
Phrase-structure tree with heads.

Example



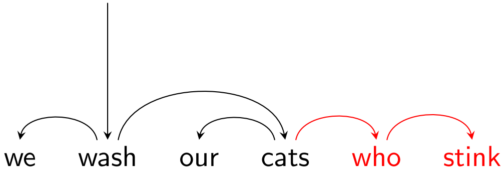
Phrase-structure tree with heads, lexicalized.

Example

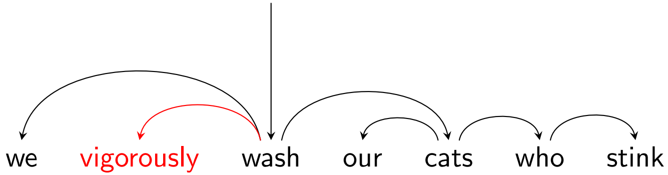


"Bare bones" dependency tree.

Example

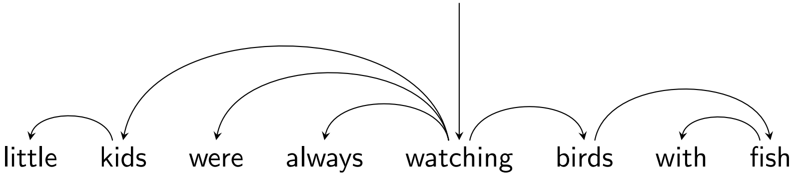


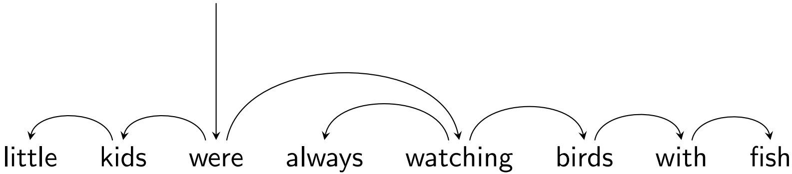
Example



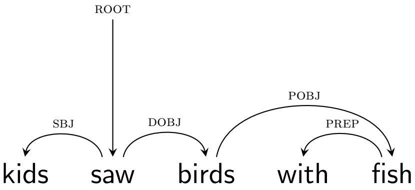
Content Heads vs. Function Heads

Credit: Nathan Schneider





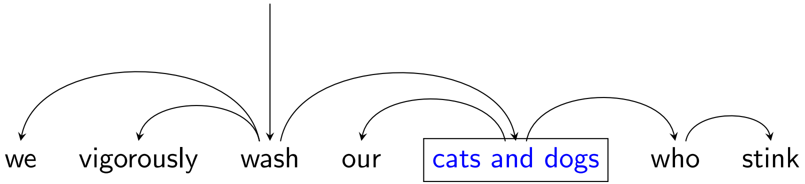
Labels



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

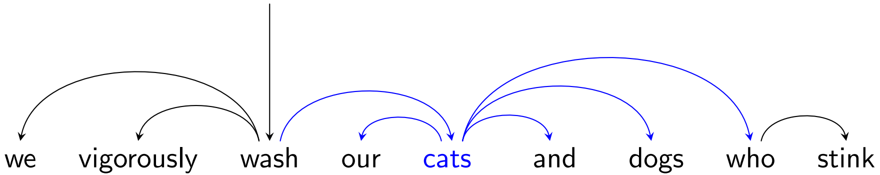
In this lecture, I will mostly not discuss labels, to keep the algorithms simpler.

Coordination Structures



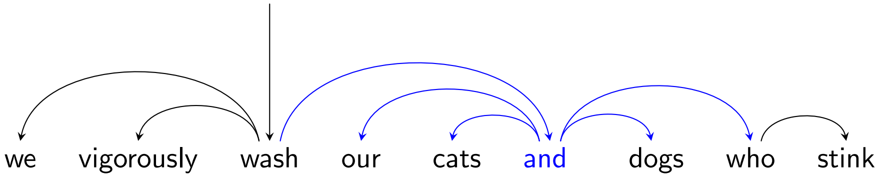
The bugbear of dependency syntax.

Example



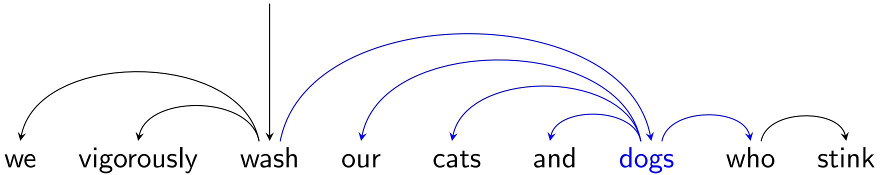
Make the first conjunct the head?

Example



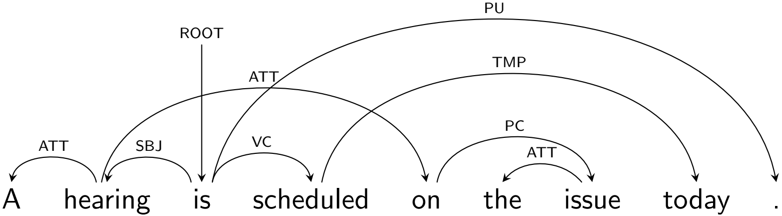
Make the coordinating conjunction the head?

Example



Make the second conjunct the head?

Nonprojective Example



Dependency Schemes

- Direct annotation.
- 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).
  - Only results in projective trees.
- Rule based dependencies, followed by manual correction.

Approaches to Dependency Parsing

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

Acknowledgment

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

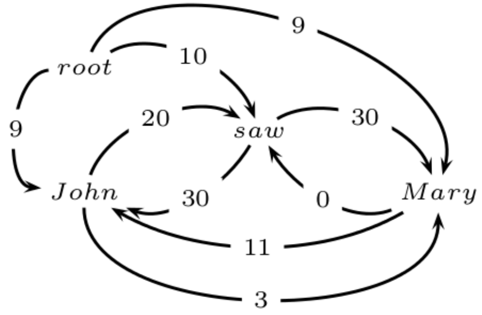
Graph-Based Dependency Parsing

Selects structures which are globally optimal.

Graph-Based Dependency Parsing

Selects structures which are globally optimal.

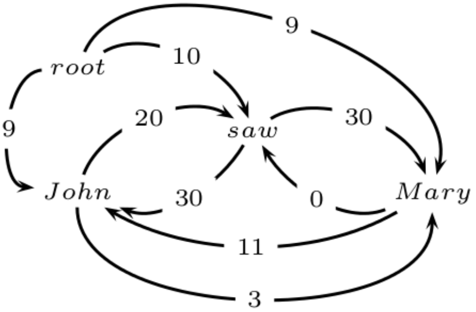
Start with a fully connected graph. Set of $O(n^2)$ edges, E.



Graph-Based Dependency Parsing

Selects structures which are globally optimal.

Start with a fully connected graph. Set of $O(n^2)$ edges, E.



No incoming edges to x_0, ensuring that it will be the root.

First-Order Graph-Based (FOG) Dependency Parsing

(McDonald et al., 2005)

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

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

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.
- Maximum spanning tree.

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.
- Maximum spanning tree.

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

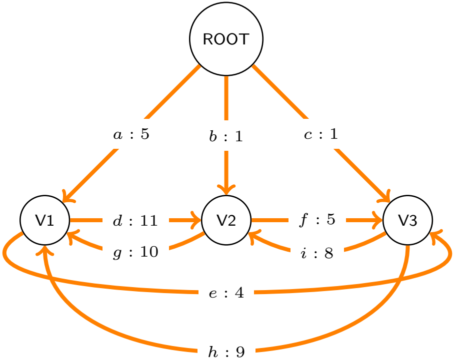
```
def maxArborescence(V, E, ROOT):
    # returns best arborescence as a map from each node to its parent
    for c in V \ ROOT:
        bestInEdge[c] ← argmax_{e∈E:e=⟨p,c⟩} e.s   # i.e., s(e)

        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)

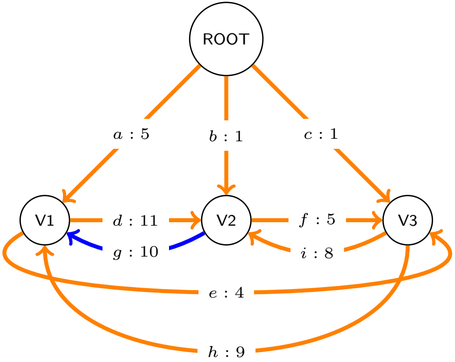
Contraction Example



	`bestInEdge`
V1	
V2	
V3	

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

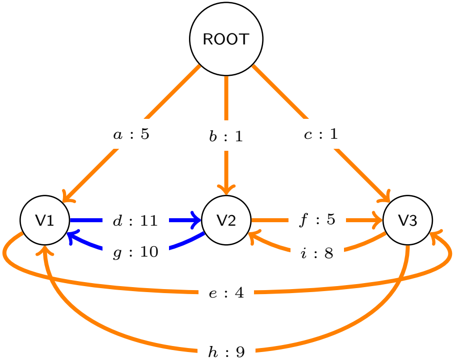
Contraction Example



	bestInEdge
V1	g
V2	
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

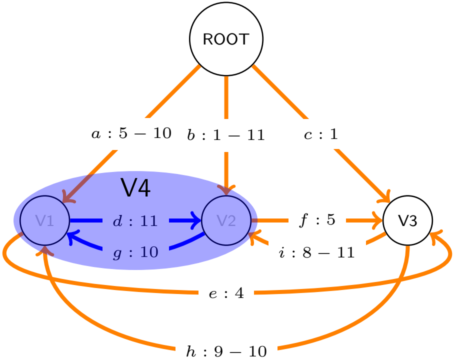
Contraction Example



	bestInEdge
V1	g
V2	d
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

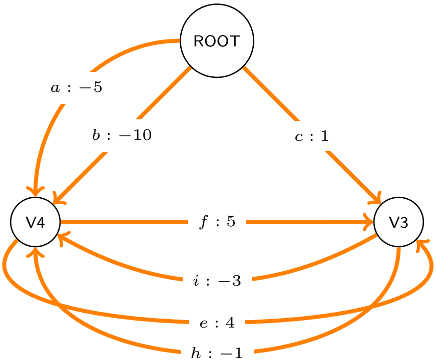
Contraction Example



	`bestInEdge`
V1	g
V2	d
V3	

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

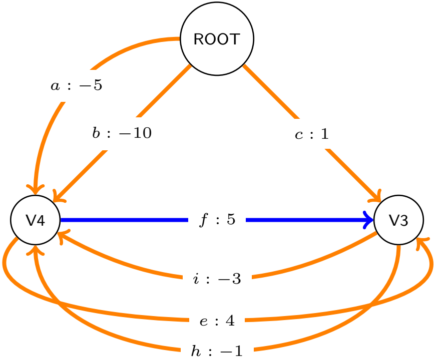
Contraction Example



	bestInEdge
V1	g
V2	d
V3	
V4	

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

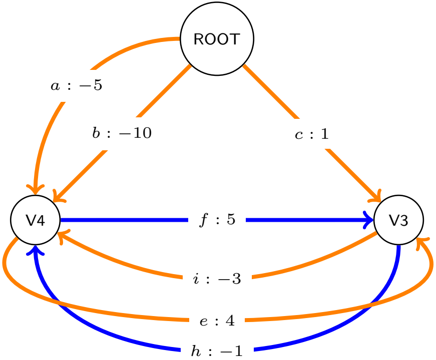
Contraction Example



	bestInEdge
V1	g
V2	d
V3	f
V4	

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

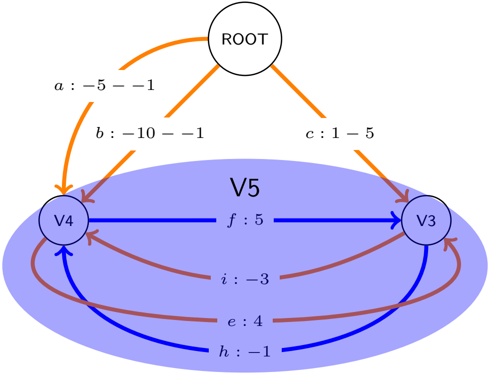
Contraction Example



	bestInEdge
V1	g
V2	d
V3	f
V4	h

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

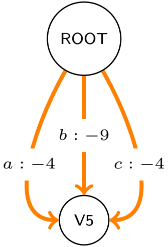
Contraction Example



	`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

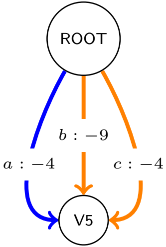
Contraction Example



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

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

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

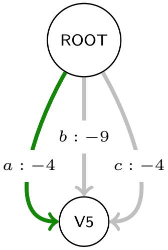
        $e'.s \leftarrow e.s - e'.\texttt{kicksOut}.s$

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

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

    return e'

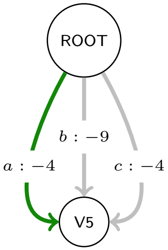
Expansion Example



	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

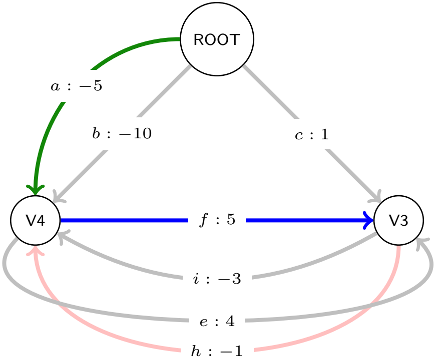
Expansion Example



	`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

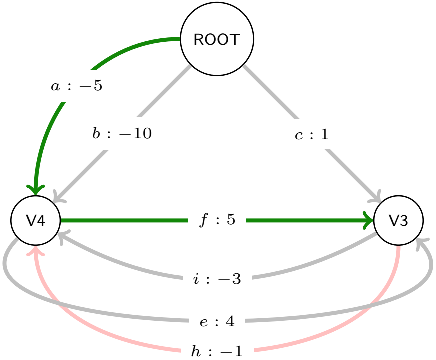
Expansion Example



	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

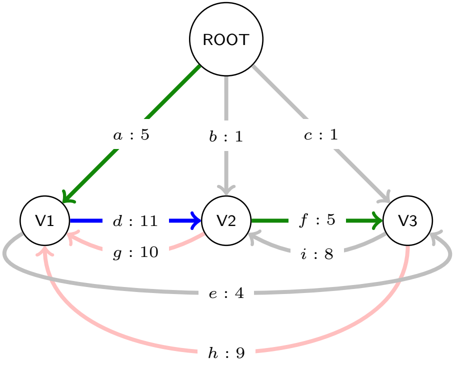
Expansion Example



	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

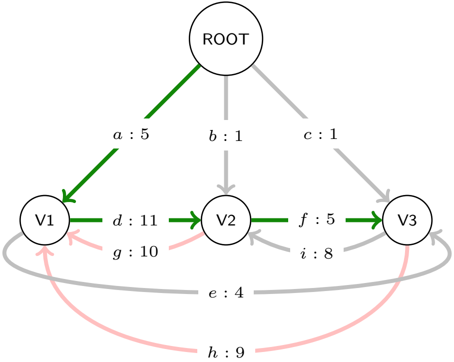
Expansion Example



	`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

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`.

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
```

Observation

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

CLE can handle both projective and non-projective dependency parsing.

Is this a good thing or a bad thing?

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, $O(n^2 + n \log n)$ **runtime.**

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.

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.

Mark Johnson. PCFG models of linguistic tree representations. *Computational Linguistics*, 24(4):613–32, 1998.

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.

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.

References II

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.