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)
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 $x = \langle x_1, \ldots, x_n \rangle$ 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 $\ell$.
- The directed edges form an arborescence (directed tree) with $x_0$ as the root (sometimes denoted root).
Phrase-structure tree.
Example

Phrase-structure tree with heads.
Example

Phrase-structure tree with heads, lexicalized.
Example

“Bare bones” dependency tree.
Example

we wash our cats who stink
Example

we vigorously wash our cats who stink
little kids were always watching birds with fish

little kids were always watching birds with fish
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.

we vigorously wash our cats and dogs who stink
Example

we vigorously wash our cats and dogs who stink

Make the first conjunct the head?
Example

we vigorously wash our cats and dogs who stink

Make the coordinating conjunction the head?
Example

We vigorously wash our cats and dogs who stink.

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.
Dependencies and Grammar

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

For every head word and constellation of dependent children:

\[ N_{\text{head}} \rightarrow N_{\text{leftmost-sibling}} \cdots N_{\text{head}} \cdots N_{\text{rightmost-sibling}} \]

And for every \( v \in V \): \( N_v \rightarrow v \) and \( S \rightarrow N_v \).
Dependencies and Grammar

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

For every head word and constellation of dependent children:

\[ N_{\text{head}} \rightarrow N_{\text{leftmost-sibling}} \cdots N_{\text{head}} \cdots N_{\text{rightmost-sibling}} \]

And for every \( v \in \mathcal{V} \): \( N_v \rightarrow v \) and \( S \rightarrow N_v \).

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:

$$N_{\text{head}} \rightarrow N_{\text{leftmost-sibling}} \cdots N_{\text{head}} \cdots N_{\text{rightmost-sibling}}$$

And for every $v \in V$: $N_v \rightarrow v$ and $S \rightarrow N_v$.

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

\[ p(N_{xh} N_{xc} | N_{xh}) \]

\[ p(N_{xc} N_{xh} | N_{xh}) \]
CKY Example

The diagram illustrates a tree structure for the sentence "we wash our cats." The tree is built using the CKY algorithm, which involves finding the non-terminal nodes (N) and the terminal nodes (S). The non-terminal nodes include "S" (sentence), "N_{wash}" (noun phrase for wash), and "N_{cats}" (noun phrase for cats). The terminal nodes are "we," "wash," "our," and "cats." The tree structure shows how these elements are connected to form the sentence and its meaning.
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:**

\[ p(x_i \mid N_{x_i}) \]

\[
\begin{array}{cc}
i & i \\
\end{array}
\]

\[
\begin{array}{cc}
1 & 1 \\
\end{array}
\]

**Goal:**

\[ p(N_{x_i} \mid S) \]

\[
\begin{array}{cccc}
1 & i & i & n \\
\end{array}
\]

\[
\begin{array}{cc}
goal & \text{goal} \\
\end{array}
\]
Dependency Parsing with the Eisner Algorithm
(Eisner, 1996)

Attaching a left dependent:

Complete a left child:

\[ p(N_{xi} N_{xk} | N_{xk}) \]
Dependency Parsing with the Eisner Algorithm
(Eisner, 1996)

Attaching a right dependent:

i
j
j + 1
k

Complete a right child:

i
j
j
k

p(N_{xi} N_{xk} | N_{xi})
Eisner Algorithm Example

given we wash our cats

goal

we wash our cats
Three Approaches to Dependency Parsing

1. Dynamic programming with the Eisner algorithm.
2. Transition-based parsing with a stack.
Transition-Based Parsing

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

- Process $x$ 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 $x$ 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

- Process $x$ 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 $x$ and the stack to contain the root symbol.
Transition-Based Parsing

- Process $x$ 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 $x$ 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.)
Transition-Based Parsing

- Process $x$ 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 $x$ 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

Stack $S$:

Buffer $B$:

- we
- vigorously
- wash
- our
- cats
- who
- stink

Actions:
Transition-Based Parsing: Example

Stack $S$:

```
we
ROOT
```

Buffer $B$:

```
vigorously
wash
our
cats
who
stink
```

Actions: SHIFT
Transition-Based Parsing: Example

Stack $S$:  

- vigorously
- we
- ROOT

Buffer $B$:  

- wash
- our
- cats
- who
- stink

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 $S$:  

Buffer $B$:  

Actions: SHIFT SHIFT SHIFT LEFT-ARC
Transition-Based Parsing: Example

Stack $S$: 

Buffer $B$: 

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC
Transition-Based Parsing: Example

Stack $S$:

```
our

we  vigorously  wash

ROOT
```

Buffer $B$:

```
cats
who
stink
```

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT
Transition-Based Parsing: Example

Stack $S$:

cats
our

we
vigorously
wash

ROOT

Buffer $B$:

who
stink

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT SHIFT
Transition-Based Parsing: Example

Stack $S$:

Buffer $B$:

**Actions:** SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT SHIFT LEFT-ARC
Transition-Based Parsing: Example

Stack $S$:

who

our  cats

we  vigorously  wash

ROOT

Buffer $B$:

stink

Actions: SHIFT  SHIFT  SHIFT  LEFT-ARC  LEFT-ARC  SHIFT  SHIFT  LEFT-ARC  SHIFT
Transition-Based Parsing: Example

Stack $S$:

stink
who

 Buffer $B$:

our cats
we vigorously wash

ROOT

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT SHIFT LEFT-ARC SHIFT SHIFT
Transition-Based Parsing: Example

Stack $S$:

```
who  stink

our  cats

we  vigorously  wash
```

Buffer $B$:

Actions: SHIFT SHIFT SHIFT LEFT-ARC LEFT-ARC SHIFT SHIFT LEFT-ARC SHIFT SHIFT RIGHT-ARC
Transition-Based Parsing: Example

Stack $S$:

```
our  cats  who  stink
we   vigorously  wash
```

ROOT

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 $S$:

\[ \text{we} \rightarrow \text{vigorously} \rightarrow \text{wash} \rightarrow \text{our} \rightarrow \text{cats} \rightarrow \text{who} \rightarrow \text{stink} \]

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
At each iteration, choose among \{\text{SHIFT, RIGHT-ARC, LEFT-ARC}\}. (Actually, among all $\mathcal{L}$-labeled variants of \text{RIGHT-} and \text{LEFT-ARC}.)
The Core of Transition-Based Parsing: Classification

- At each iteration, choose among \{\textsc{shift}, \textsc{right-arc}, \textsc{left-arc}\}. (Actually, among all \(\mathcal{L}\)-labeled variants of \textsc{right-} and \textsc{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 \(\text{RIGHT-}\) and \(\text{LEFT-ARC}\).)
- 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: \(\langle\text{state, correct transition}\rangle\). Each word gets \text{SHIFTed} once and participates as a child in one \text{ARC}.
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.
Acknowledgment

Slides are mostly adapted from those by Swabha Swayamdipta and Sam Thomson.
For the Eisner algorithm, the score of an unlabeled parse $y$ was

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

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

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

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

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

\[ y^* = \arg\max_{y \subset E} s_{\text{global}}(y) = \arg\max_{y \subset E} \sum_{e \in y} s(e) \]

subject to the constraint that \( y \) 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.
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 $\text{bestInEdge}[v]$ 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 $e.\text{kicksOut}$ to $\text{bestInEdge}[v]$, and
      - Set $e.s$ to be $e.s - e.\text{kicksOut}.s$
    - 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

```python
def adjust(e, v_C):
e' ← copy(e)
e'.original ← e
if e.dest ∈ C:
e'.dest ← v_C
    e'.kicksOut ← bestInEdge[e.dest]
e'.s ← e.s - e'.kicksOut.s
elif e.src ∈ C:
e'.src ← v_C
return e'
```

66 / 96
Contraction Example

![Graph Diagram]

<table>
<thead>
<tr>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
</tr>
<tr>
<td>V2</td>
</tr>
<tr>
<td>V3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>i</td>
</tr>
</tbody>
</table>
Contraction Example

**bestInEdge**

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**kicksOut**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Contraction Example
Contraction Example

\begin{tabular}{|c|c|}
\hline
\textbf{bestInEdge} & \textbf{kicksOut} \\
\hline
V1 & g \\
V2 & d \\
V3 & \\
\hline
\end{tabular}
Contraction Example

\[
\begin{align*}
V4 & \xrightarrow{a:-5} \text{ROOT} \\
b & \xrightarrow{-10} & c & \xrightarrow{1} \\
V4 & \xrightarrow{f:5} & V3 \\
i & \xrightarrow{-3} & e & \xrightarrow{4} \\
h & \xrightarrow{-1} \\
\end{align*}
\]

\[
\begin{array}{|c|c|}
\hline
\text{bestInEdge} & \\
V1 & g \\
V2 & d \\
V3 & \\
V4 & \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{kicksOut} & \\
a & g \\
b & d \\
c & \\
d & \\
e & \\
f & \\
g & \\
h & g \\
i & d \\
\hline
\end{array}
\]
Contraction Example

<table>
<thead>
<tr>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
</tr>
<tr>
<td>V2</td>
</tr>
<tr>
<td>V3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>i</td>
</tr>
</tbody>
</table>
Contraction Example

![Diagram](image)

<table>
<thead>
<tr>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
</tr>
<tr>
<td>V2</td>
</tr>
<tr>
<td>V3</td>
</tr>
<tr>
<td>V4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>i</td>
</tr>
</tbody>
</table>
Contraction Example

### bestInEdge

<table>
<thead>
<tr>
<th></th>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>g</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>V3</td>
<td>f</td>
</tr>
<tr>
<td>V4</td>
<td>h</td>
</tr>
<tr>
<td>V5</td>
<td></td>
</tr>
</tbody>
</table>

### kicksOut

<table>
<thead>
<tr>
<th></th>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g, h</td>
</tr>
<tr>
<td>b</td>
<td>d, h</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>
Contraction Example

### bestInEdge

<table>
<thead>
<tr>
<th>Vertex</th>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>g</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>V3</td>
<td>f</td>
</tr>
<tr>
<td>V4</td>
<td>h</td>
</tr>
<tr>
<td>V5</td>
<td></td>
</tr>
</tbody>
</table>

### kicksOut

<table>
<thead>
<tr>
<th>Vertex</th>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g, h</td>
</tr>
<tr>
<td>b</td>
<td>d, h</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
</tr>
<tr>
<td>d</td>
<td>f</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>h</td>
<td>d</td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>
Contraction Example

**bestInEdge**

<table>
<thead>
<tr>
<th></th>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>g</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>V3</td>
<td>f</td>
</tr>
<tr>
<td>V4</td>
<td>h</td>
</tr>
<tr>
<td>V5</td>
<td>a</td>
</tr>
</tbody>
</table>

**kicksOut**

<table>
<thead>
<tr>
<th></th>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g, h</td>
</tr>
<tr>
<td>b</td>
<td>d, h</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>g</td>
</tr>
<tr>
<td></td>
<td>d</td>
</tr>
</tbody>
</table>
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

```
V5
 ROOT
   / \  \
   b : -9  a : -4  c : -4
   /     /  \
  V5   V1   V2
  |     |     |
  e     d     f
  |     |     |
  V3   V4   V5
```

- bestInEdge:
  - V1: g
  - V2: d
  - V3: f
  - V4: h
  - V5: a

- kicksOut:
  - a: g, h
  - b: d, h
  - c: f
  - d: f
  - e: 
  - f: 
  - g: g
  - h: d
Expansion Example

**bestInEdge**

<table>
<thead>
<tr>
<th></th>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>a, g</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>V3</td>
<td>f</td>
</tr>
<tr>
<td>V4</td>
<td>a, h</td>
</tr>
<tr>
<td>V5</td>
<td>a</td>
</tr>
</tbody>
</table>

**kicksOut**

<table>
<thead>
<tr>
<th></th>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g, h</td>
</tr>
<tr>
<td>b</td>
<td>d, h</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
</tr>
<tr>
<td>d</td>
<td>f</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>h</td>
<td>d</td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>
Expansion Example

**bestInEdge**

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g</td>
<td>d</td>
<td>f</td>
<td>a</td>
<td>k</td>
</tr>
</tbody>
</table>

**kicksOut**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>g, h</td>
<td>d, h</td>
<td>f</td>
<td>f</td>
<td>g</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Edges:
- **a**: -5
- **b**: -10
- **c**: 1
- **f**: 5
- **i**: -3
- **e**: 4
- **h**: -1
Expansion Example

**Graph:**
- ** ROOT**
- **V4**
  - \(a: -5\)
  - \(b: -10\)
  - \(c: 1\)
- **V3**
  - \(f: 5\)
  - \(i: -3\)
  - \(e: 4\)
  - \(h: -1\)

**Tables:**

<table>
<thead>
<tr>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
</tr>
<tr>
<td>V2</td>
</tr>
<tr>
<td>V3</td>
</tr>
<tr>
<td>V4</td>
</tr>
<tr>
<td>V5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>i</td>
</tr>
</tbody>
</table>
Expansion Example

\[
\begin{array}{c|c|c|c|c}
\text{V1} & \text{V2} & \text{V3} & \text{V4} & \text{V5} \\
\hline
a & b & c & d & e \\
5 & 1 & 1 & 11 & 2 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{bestInEdge} & \text{kicksOut} \\
\hline
\text{V1} & a, g \\
\text{V2} & d \\
\text{V3} & f \\
\text{V4} & a, h \\
\text{V5} & a \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{V1} & \text{V2} \\
\hline
\text{d} & \text{g} \\
11 & 5 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{V2} & \text{V3} \\
\hline
\text{f} & \text{g} \\
5 & 10 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{V3} & \text{V4} \\
\hline
\text{f} & \text{h} \\
9 & 8 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{V4} & \text{V5} \\
\hline
\text{a} & \text{g} \\
\text{b} & \text{h} \\
\text{c} & \text{d} \\
\text{d} & \text{h} \\
\text{e} & \text{f} \\
\text{f} & \text{f} \\
\text{g} & \text{g} \\
\text{h} & \text{d} \\
\text{i} & \text{d} \\
\end{array}
\]
Expansion Example

![Graph Diagram]

**bestInEdge**

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**kicksOut**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g, h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>d, h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d</td>
</tr>
</tbody>
</table>
Observation

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

Is this a good thing or a bad thing?
A hearing is scheduled on the issue today.
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.**
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).
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.


