Dependency Grammar

Introduction
Dependency syntax postulates that syntactic structure consists of lexical items linked by binary asymmetric relations ("arrows") called dependencies.
Dependency Grammar and Dependency Structure

Dependency syntax postulates that syntactic structure consists of lexical items linked by binary asymmetric relations ("arrows") called dependencies.

The arrows are commonly typed with the name of grammatical relations (subject, prepositional object, apposition, etc.).
Dependency Grammar and Dependency Structure

Dependency syntax postulates that syntactic structure consists of lexical items linked by binary asymmetric relations ("arrows") called dependencies.

The arrow connects a head (governor, superior, regent) with a dependent (modifier, inferior, subordinate).

Usually, dependencies form a tree (connected, acyclic, single-head).

```
Bills
prep
on
pobj
ports
cp conj
and immigration
nsubjpass
auxpass
submitted
were
by
pobj
Brownback
nn appos
Republican
prep
do
pobj
Kansas
```
Dependency Grammar and Dependency Structure

ROOT Discussion of the outstanding issues was completed.

- Some people draw the arrows one way; some the other way!
  - Tesnière had them point from head to dependent...
- Usually add a fake ROOT so every word is a dependent of precisely 1 other node
Dependency Grammar/Parsing History

• The idea of dependency structure goes back a long way
  • To Pāṇini’s grammar (c. 5th century BCE)
  • Basic approach of 1st millennium Arabic grammarians
• Constituency is a new-fangled invention
  • 20th century invention (R.S. Wells, 1947)
• Modern dependency work often linked to work of L. Tesnière (1959)
  • Was dominant approach in “East” (Russia, China, …)
    • Good for free-er word order languages
• Among the earliest kinds of parsers in NLP, even in the US:
  • David Hays, one of the founders of U.S. computational linguistics, built early (first?) dependency parser (Hays 1962)
Relation between phrase structure and dependency structure

- A dependency grammar has a notion of a head. Officially, CFGs don’t.
- But modern linguistic theory and all modern statistical parsers (Charniak, Collins, Stanford, ...) do, via hand-written phrasal “head rules”:
  - The head of a Noun Phrase is a noun/number/adj/…
  - The head of a Verb Phrase is a verb/modal/…
- The head rules can be used to extract a dependency parse from a CFG parse

- The closure of dependencies give constituency from a dependency tree
- But the dependents of a word must be at the same level (i.e., “flat”) – there can be no VP!
Dependency Conditioning Preferences

What are the sources of information for dependency parsing?

1. Bilexical affinities  \([\text{issues} \rightarrow \text{the}]\) is plausible
2. Dependency distance  mostly with nearby words
3. Intervening material
   Dependencies rarely span intervening verbs or punctuation
4. Valency of heads
   How many dependents on which side are usual for a head?

ROOT Discussion of the outstanding issues was completed .
Dependency Parsing

• A sentence is parsed by choosing for each word what other word (including ROOT) that it is a dependent of.

• Usually some constraints:
  • Only one word is a dependent of ROOT
  • Don’t want cycles A → B, B → A

• This makes the dependencies a tree

• Final issue is whether arrows can cross (non-projective) or not
Methods of Dependency Parsing

1. Dynamic programming (like in the CKY algorithm)
   You can do it similarly to lexicalized PCFG parsing: an $O(n^5)$ algorithm
   Eisner (1996) gives a clever algorithm that reduces the complexity to $O(n^3)$, by producing parse items with heads at the ends rather than in the middle

2. Graph algorithms
   You create a Minimum Spanning Tree for a sentence
   McDonald et al.’s (2005) MSTParser scores dependencies independently using a ML classifier (he uses MIRA, for online learning, but it could be something else)

3. Constraint Satisfaction
   Edges are eliminated that don’t satisfy hard constraints. Karlsson (1990), etc.

4. “Deterministic parsing”
   Greedy choice of attachments guided by good machine learning classifiers
   MaltParser (Nivre et al. 2008)
DP for generative dependency grammars
1. Start with left wall 
2. Generate root $w_0$
3. Generate left children $w_{-1}$, $w_{-2}$, ..., $w_{-\ell}$ from the FSA $\lambda_{w_0}$
4. Generate right children $w_1$, $w_2$, ..., $w_r$ from the FSA $\rho_{w_0}$
5. Recurse on each $w_i$ for $i$ in {-\ell, ..., -1, 1, ..., r}, sampling $\alpha$ (steps 2-4)
6. Return $\alpha_{-\ell} \ldots \alpha_{-1} w_0 \alpha_1 \ldots \alpha_r$

These 5 slides are based on slides by Jason Eisner and Noah Smith
Naïve Recognition/Parsing

\[ O(n^5) \]

combinations

takes

goal

takes

takes

takes
to
tango

takes
two
to
tango

It

It

It

It

It
Dependency Grammar Cubic Recognition/Parsing (Eisner & Satta, 1999)

- **Triangles**: span over words, where tall side of triangle is the head, other side is dependent, and no non-head words expecting more dependents.

- **Trapezoids**: span over words, where larger side is head, smaller side is dependent, and smaller side is still looking for dependents on its side of the trapezoid.
Dependency Grammar Cubic Recognition/Parsing (Eisner & Satta, 1999)

A triangle is a head with some left (or right) subtrees.

One trapezoid per dependency.

It takes two to tango
Cubic Recognition/Parsing (Eisner & Satta, 1999)

- $O(n)$ combinations
- $O(n^3)$ combinations
- $O(n^3)$ combinations

Gives $O(n^3)$ dependency grammar parsing
Graph Algorithms: MSTs
McDonald et al. (2005 ACL)
Online Large-Margin Training of Dependency Parsers

- One of two best-known recent dependency parsers
- Score of a dependency tree = sum of scores of dependencies
- Scores are independent of other dependencies
- If scores are available, parsing can be solved as a minimum spanning tree problem
  - Chiu-Liu-Edmonds algorithm
  - One then needs a score for dependencies
McDonald et al. (2005 ACL): Online Large-Margin Training of Dependency Parsers

- Edge scoring is via a discriminative classifier
  - Can condition on rich features in that context
  - Each dependency is a linear function of features times weights
- Feature weights were learned by MIRA, an online large-margin algorithm
  - But you could use an SVM, maxent, or a perceptron
- Features cover:
  - Head and dependent word and POS separately
  - Head and dependent word and POS bigram features
  - Words between head and dependent
  - Length and direction of dependency
Greedy Transition-Based Parsing

MaltParser
MaltParser
[Nivre et al. 2008]

- A simple form of greedy discriminative dependency parser
- The parser does a sequence of bottom up actions
  - Roughly like “shift” or “reduce” in a shift-reduce parser, but the “reduce” actions are specialized to create dependencies with head on left or right
- The parser has:
  - a stack $\sigma$, written with top to the right
    - which starts with the ROOT symbol
  - a buffer $\beta$, written with top to the left
    - which starts with the input sentence
  - a set of dependency arcs $A$
    - which starts off empty
  - a set of actions
Basic transition-based dependency parser

Start: \( \sigma = [\text{ROOT}], \beta = w_1, \ldots, w_n, A = \emptyset \)
1. Shift \( \sigma, w_i | \beta, A \Rightarrow \sigma | w_i, \beta, A \)
2. Left-Arc\(_r\) \( \sigma | w_i, w_j | \beta, A \Rightarrow \sigma, w_j | \beta, A \cup \{r(w_j, w_i)\} \)
3. Right-Arc\(_r\) \( \sigma | w_i, w_j | \beta, A \Rightarrow \sigma, w_i | \beta, A \cup \{r(w_i, w_j)\} \)
Finish: \( \beta = \emptyset \)

Notes:
- Unlike the regular presentation of the CFG reduce step, dependencies combine one thing from each of stack and buffer
Actions ("arc-eager" dependency parser)

Start: \( \sigma = [\text{ROOT}], \beta = w_1, ..., w_n, A = \emptyset \)

1. Left-Arc_{r} \( \sigma | w_i, w_j | \beta, A \Rightarrow \sigma, w_j | \beta, A \cup \{r(w_j, w_i)\} \)
   Precondition: \( r'(w_k, w_i) \notin A, w_i \neq \text{ROOT} \)

2. Right-Arc_{r} \( \sigma | w_i, w_j | \beta, A \Rightarrow \sigma | w_i | w_j, \beta, A \cup \{r(w_i, w_j)\} \)

3. Reduce \( \sigma | w_i, \beta, A \Rightarrow \sigma, \beta, A \)
   Precondition: \( r'(w_k, w_i) \in A \)

4. Shift \( \sigma, w_i | \beta, A \Rightarrow \sigma | w_i, \beta, A \)

Finish: \( \beta = \emptyset \)

This is the common "arc-eager" variant: a head can immediately take a right dependent, before its dependents are found.
Happy children like to play with their friends.
Happy children like to play with their friends.

You terminate as soon as the buffer is empty. Dependencies = $A_9$
Example

1. \text{Left-Arc}_r \quad \sigma | w_i, w_j | \beta, A \rightarrow \sigma, w_j | \beta, A \cup \{ r(w_i, w_j) \}
   \text{Precondition:} (w_k, r', w_i) \notin A, w_i \neq \text{ROOT}

2. \text{Right-Arc}_r \quad \sigma | w_i, w_j | \beta, A \rightarrow \sigma | w_i | w_j, \beta, A \cup \{ r(w_i, w_j) \}

3. \text{Reduce} \quad \sigma | w_i, \beta, A \rightarrow \sigma, \beta, A
   \text{Precondition:} (w_k, r', w_i) \in A

4. \text{Shift} \quad \sigma, w_i | \beta, A \rightarrow \sigma | w_i, \beta, A

\text{S} \left[ \text{Red figures on the screen indicated falling stocks} \right] \text{Q}
Example

\[
\begin{align*}
\text{S} & \quad \text{Red} \\
\text{Q} & \quad \text{figures on the screen indicated falling stocks}
\end{align*}
\]

Shift

1. **Left-Arc** \( r | w_i, w_j | \beta, A \rightarrow \sigma, w_j | \beta, A \cup \{r(w_i,w_j)\} \)
   Precondition: \((w_k, r', w_i) \notin A, w_i \neq \text{ROOT}\)

2. **Right-Arc** \( r | w_i, w_j | \beta, A \rightarrow \sigma | w_i | w_j | \beta, A \cup \{r(w_i,w_j)\} \)

3. **Reduce** \( r | w_i, \beta, A \rightarrow \sigma, \beta, A \)
   Precondition: \((w_k, r', w_i) \in A\)

4. **Shift** \( r | w_i, \beta, A \rightarrow \sigma | w_i, \beta, A \)
Example

\[
\begin{align*}
\text{S} & \quad \text{Red} \\
\{\text{figures on the screen indicated falling stocks}\} & \quad \text{Q}
\end{align*}
\]

Left-arc

1. **Left-Arc**$_r$ \( \sigma | w_p, w_j | \beta, A \rightarrow \sigma, w_j | \beta, A \cup \{r(w_p, w_j)\} \)
   
   Precondition: \((w_k, r', w_i) \notin A, w_i \neq \text{ROOT}\)

2. **Right-Arc**$_r$ \( \sigma | w_p, w_j | \beta, A \rightarrow \sigma | w_i | w_j, \beta, A \cup \{r(w_i, w_j)\} \)

3. **Reduce** \( \sigma | w_p, \beta, A \rightarrow \sigma, \beta, A \)
   
   Precondition: \((w_k, r', w_i) \in A\)

4. **Shift** \( \sigma, w_i | \beta, A \rightarrow \sigma | w_i, \beta, A \)
Example

\[ \text{Red figures on the screen indicated falling stocks} \]

**Shift**

1. **Left-Arc** \( \sigma | w_i, w_j | \beta , A \rightarrow \sigma , w_j | \beta , A \cup \{r(w_i, w_j)\} \)
   
   **Precondition:** \( (w_k, r', w_i) \notin A, w_i \neq \text{ROOT} \)

2. **Right-Arc** \( \sigma | w_i, w_j | \beta , A \rightarrow \sigma | w_i, w_j | \beta , A \cup \{r(w_i, w_j)\} \)

3. **Reduce** \( \sigma | w_i, \beta , A \rightarrow \sigma , \beta , A \)
   
   **Precondition:** \( (w_k, r', w_i) \in A \)

4. **Shift** \( \sigma , w_i | \beta , A \rightarrow \sigma | w_i , \beta , A \)
Example

1. Left-Arc, $\sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_j|\beta, A \cup \{r(w_i, w_j)\}$
   Precondition: $(w_k, r', w_i) \not\in A, w_i \neq \text{ROOT}$
2. Right-Arc, $\sigma|w_i, w_j|\beta, A \rightarrow \sigma|w_i, w_j, \beta, A \cup \{r(w_i, w_j)\}$
3. Reduce, $\sigma|w_i, \beta, A \rightarrow \sigma, \beta, A$
   Precondition: $(w_k, r', w_i) \in A$
4. Shift, $\sigma, w_i|\beta, A \rightarrow \sigma|w_i, \beta, A$

$$\text{S} \quad \left\{ \begin{array}{l}
\text{Red figures on the screen indicated falling stocks}
\end{array} \right\} \quad \text{Q}$$

Right-arc
Example

1. **Left-Arc**, $\sigma | w_i, w_j | \beta, A \rightarrow \sigma, w_j | \beta, A \cup \{r(w_i, w_j)\}$
   Precondition: $(w_k, r', w_i) \notin A, w_i \neq \text{ROOT}$
2. **Right-Arc**, $\sigma | w_i, w_j | \beta, A \rightarrow \sigma | w_i | w_j | \beta, A \cup \{r(w_i, w_j)\}$
3. **Reduce**, $\sigma | w_i, \beta, A \rightarrow \sigma, \beta, A$
   Precondition: $(w_k, r', w_i) \in A$
4. **Shift**, $\sigma, w_i | \beta, A \rightarrow \sigma | w_i, \beta, A$

\[\begin{array}{c}
\text{\texttt{\_ROOT\_}} \quad \text{Red} \\
\text{figures} \quad \text{on} \quad \text{the} \\
\text{screen} \quad \text{indicated} \quad \text{falling} \quad \text{stocks} \\
\end{array}\]
Example

1. Left-Arc$_r$ \( \sigma | w_i, w_j | \beta, A \rightarrow \sigma, \beta, A \cup \{ r(w_i,w_j) \} \)
   Precondition: \((w_k, r', w_i) \notin A, w_i \neq \text{ROOT}\)
2. Right-Arc$_r$ \( \sigma | w_i, w_j | \beta, A \rightarrow \sigma | w_i | \beta, A \cup \{ r(w_i,w_j) \} \)
3. Reduce \( \sigma | w_i, \beta, A \rightarrow \sigma, \beta, A \)
   Precondition: \((w_k, r', w_i) \in A\)
4. Shift \( \sigma, w_i | \beta, A \rightarrow \sigma | w_i, \beta, A \)

\[
\begin{array}{c}
\text{\_\_ROOT\_} \quad \text{Red} \quad \text{figures} \quad \text{on} \quad S \quad \text{the} \quad \text{\{screen} \quad \text{indicated} \quad \text{falling} \quad \text{stocks} \quad \text{\}Q}
\end{array}
\]

Left-arc
Example

1. Left-Arc, \( \sigma |w_i, w_j| \beta, A \rightarrow \sigma, w_j| \beta, A \cup \{r(w_i, w_j)\} \)
   Precondition: \((w_k, r', w_i) \notin A, w_i \neq \text{ROOT}\)

2. Right-Arc, \( \sigma |w_i, w_j| \beta, A \rightarrow \sigma |w_i, \beta, A \cup \{r(w_i, w_j)\} \)

3. Reduce, \( \sigma |w_i, \beta, A \rightarrow \sigma, \beta, A \)
   Precondition: \((w_k, r', w_i) \in A\)

4. Shift, \( \sigma, w_i| \beta, A \rightarrow \sigma |w_i, \beta, A \)

\[
\begin{array}{c}
\text{\_ROOT\_} \\
\text{Red} \\
\text{figures} \\
\text{on} \\
\text{the} \\
\text{screen}
\end{array}
\rightarrow
\begin{array}{c}
\text{indicated} \\
\text{falling} \\
\text{stocks}
\end{array}
\]

Right-arc
Example

1. \text{Left-Arc}, \quad \sigma |w_i, w_j| \beta, A \Rightarrow \sigma, w_j| \beta, A \cup \{r(w_i, w_j)\}
   \text{Precondition: } (w_k, r', w_i) \notin A, w_i \neq \text{ROOT}

2. \text{Right-Arc}, \quad \sigma |w_i, w_j| \beta, A \Rightarrow \sigma |w_i| \beta, A \cup \{r(w_i, w_j)\}

3. \text{Reduce}, \quad \sigma |w_i, \beta, A \Rightarrow \sigma, \beta, A
   \text{Precondition: } (w_k, r', w_i) \in A

4. \text{Shift}, \quad \sigma , w_i| \beta , A \Rightarrow \sigma |w_i , \beta , A

\begin{align*}
\text{\_ROOT\_} & \quad \text{Red} \quad \text{figures} \quad \text{on} \quad S \quad \text{the} \quad \text{screen} \quad \{\text{indicated} \quad \text{falling} \quad \text{stocks}\} \quad Q
\end{align*}

Reduce
Example

\[ \text{Red figures on the screen \{ indicated falling stocks \} Q} \]

Reduce

1. Left-Arc \( \sigma | w_i, w_j | \beta , A \rightarrow \sigma , w_j | \beta , A \cup \{ r(w_i, w_j) \} \)
   Precondition: \((w_k, r, w_i) \notin A, w_i \neq \text{ROOT}\)

2. Right-Arc \( \sigma | w_i, w_j | \beta , A \rightarrow \sigma | w_i | w_j , \beta , A \cup \{ r(w_i, w_j) \} \)

3. Reduce \( \sigma | w_i, \beta , A \rightarrow \sigma , \beta , A \)
   Precondition: \((w_k, r, w_i) \in A\)

4. Shift \( \sigma , w_i | \beta , A \rightarrow \sigma | w_i , \beta , A \)
Example

1. Left-Arc
   \[ \sigma \left| w_i, w_j \right| \beta \in A \Rightarrow \sigma, w_i \beta \in A \cup \{r(w_i, w_j)\} \]
   Precondition: \((w_k, r', w_i) \notin A, w_i \neq \text{ROOT}\)

2. Right-Arc
   \[ \sigma \left| w_i, w_j \right| \beta \in A \Rightarrow \sigma \left| w_i \right| \beta, A \cup \{r(w_i, w_j)\} \]

3. Reduce
   \[ \sigma \left| w_i, \beta \in A \Rightarrow \sigma, \beta \in A \right] \]
   Precondition: \((w_k, r', w_i) \in A\)

4. Shift
   \[ \sigma, w_i \beta \in A \Rightarrow \sigma \left| w_i, \beta \in A \right] \]

\[ \text{Red} \quad \text{figures} \quad \text{on} \quad \text{the} \quad \text{screen} \quad \text{indicated} \quad \text{falling} \quad \text{stocks} \]

Left-arc
Red figures on the screen indicated falling stocks.

Right-arc
Example

1. **Left-Arc**,
   \[\sigma | w_i, w_j | \beta, A \rightarrow \sigma, w_j | \beta, A \cup \{r(w_i, w_j)\}\]
   Precondition: \((w_k, r', w_i) \notin A, w_i \neq \text{ROOT}\)

2. **Right-Arc**,
   \[\sigma | w_i, w_j | \beta, A \rightarrow \sigma | w_i | w_j, \beta, A \cup \{r(w_i, w_j)\}\]

3. **Reduce**,
   \[\sigma | w_i, \beta, A \rightarrow \sigma, \beta, A\]
   Precondition: \((w_k, r', w_i) \in A\)

4. **Shift**,
   \[\sigma, w_i | \beta, A \rightarrow \sigma | w_i, \beta, A\]

---

**Shift**

- Red figures on the screen indicated falling stocks
1. **Left-Arc** \( \sigma |w_i, w_j| \beta, A \rightarrow \sigma, w_j| \beta, A \cup \{r(w_i, w_j)\} \)
   
   **Precondition:** \((w_k, r', w_i) \notin A, w_i \neq \text{ROOT}\)

2. **Right-Arc** \( \sigma |w_i, w_j| \beta, A \rightarrow \sigma |w_i, w_j, \beta, A \cup \{r(w_i, w_j)\} \)

3. **Reduce** \( \sigma |w_i, \beta, A \rightarrow \sigma, \beta, A \)
   
   **Precondition:** \((w_k, r', w_i) \in A\)

4. **Shift** \( \sigma, w_i| \beta, A \rightarrow \sigma |w_i, \beta, A \)

---

**Example**

\(\text{Red figures on the screen indicated falling stocks}\)

**Left-arc**
Example

1. Left-Arc $\sigma |w_i, w_j| \beta , A \rightarrow \sigma , w_j| \beta , A \cup \{r(w_i, w_j)\}
   Precondition: $(w_k, r', w_i) \notin A, w_i \neq \text{ROOT}$

2. Right-Arc $\sigma |w_i, w_j| \beta , A \rightarrow \sigma |w_i| w_j, \beta , A \cup \{r(w_i, w_j)\}$

3. Reduce $\sigma |w_i, \beta , A \rightarrow \sigma , \beta , A$
   Precondition: $(w_k, r', w_i) \in A$

4. Shift $\sigma , w_i| \beta , A \rightarrow \sigma |w_i, \beta , A$

Right-arc

Red figures on the screen indicated falling stocks
1. Left-Arc, $\sigma|w_i, w_j|\beta, A \Rightarrow \sigma, w_j|\beta, A \cup \{r(w_i, w_j)\}$
   Precondition: $(w_k, r', w_i) \notin A, w_i \neq \text{ROOT}$

2. Right-Arc, $\sigma|w_i, w_j|\beta, A \Rightarrow \sigma|w_i|w_j, \beta, A \cup \{r(w_i, w_j)\}$

3. Reduce, $\sigma|w_i, \beta, A \Rightarrow \sigma, \beta, A$
   Precondition: $(w_k, r', w_i) \in A$

4. Shift, $\sigma, w_i|\beta, A \Rightarrow \sigma|w_i, \beta, A$

Example

Red figures on the screen indicated falling stocks
Example

1. Left-Arc\(_\text{r}\) \(\sigma |w_i, w_j| \beta, A \rightarrow \sigma, w_i| \beta, A \cup \{r(w_i, w_j)\}\)
   Precondition: \((w_k, r', w_i) \notin A, w_i \neq \text{ROOT}\)

2. Right-Arc\(_\text{r}\) \(\sigma |w_i, w_j| \beta, A \rightarrow \sigma |w_i| \beta, A \cup \{r(w_i, w_j)\}\)

3. Reduce \(\sigma |w_i, \beta, A \rightarrow \sigma, \beta, A\)
   Precondition: \((w_k, r', w_i) \in A\)

4. Shift \(\sigma, w_i| \beta, A \rightarrow \sigma |w_i, \beta, A\)

Reduce

Red figures on the screen indicated falling stocks
MaltParser
[Nivre et al. 2008]

• We have left to explain how we choose the next action
• Each action is predicted by a discriminative classifier (often SVM, can be perceptron, maxent classifier) over each legal move
  • Max of 4 untyped choices, max of $|R| \times 2 + 2$ when typed
  • Features: top of stack word, POS; first in buffer word, POS; etc.
• There is NO search (in the simplest and usual form)
  • But you could do some kind of beam search if you wish
• It provides VERY fast linear time parsing
• The model’s accuracy is slightly below the best Lexicalized PCFGs (evaluated on dependencies), but
• It provides close to state of the art parsing performance
Evaluation of Dependency Parsing: (labeled) dependency accuracy

Gold
1  2  She  nsubj
2  0  saw  root
3  5  the  det
4  5  video  nn
5  2  lecture  dobj

Parsed
1  2  She  nsubj
2  0  saw  root
3  4  the  det
4  5  video  nsubj
5  2  lecture  ccomp

Acc = \frac{\# \text{ correct deps}}{\# \text{ of deps}}

UAS = \frac{4}{5} = 80\%
LAS = \frac{2}{5} = 40\%
Representative performance numbers

- The CoNLL-X (2006) shared task provides evaluation numbers for various dependency parsing approaches over 13 languages
  - MALT: LAS scores from 65–92%, depending greatly on language/treebank
- Here we give a few UAS numbers for English to allow some comparison to constituency parsing

<table>
<thead>
<tr>
<th>Parser</th>
<th>UAS%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sagae and Lavie (2006) ensemble of dependency parsers</td>
<td>92.7</td>
</tr>
<tr>
<td>Charniak (2000) generative, constituency, as dependencies</td>
<td>92.2</td>
</tr>
<tr>
<td>Collins (1999) generative, constituency, as dependencies</td>
<td>91.7</td>
</tr>
<tr>
<td>McDonald and Pereira (2005) – MST graph-based dependency</td>
<td>91.5</td>
</tr>
<tr>
<td>Yamada and Matsumoto (2003) – transition-based dependency</td>
<td>90.4</td>
</tr>
</tbody>
</table>
Projectivity

- Dependencies from a CFG tree using heads, must be **projective**
  - There must not be any crossing dependency arcs when the words are laid out in their linear order, with all arcs above the words.
- But dependency theory normally does allow non-projective structures to account for displaced constituents
  - You can’t easily get the semantics of certain constructions right without these nonprojective dependencies

Who did Bill buy the coffee from yesterday?
Handling non-projectivity

• The arc-eager algorithm we presented only builds projective dependency trees

• Possible directions to head:
  1. Just declare defeat on nonprojective arcs
  2. Use a dependency formalism which only admits projective representations (a CFG doesn’t represent such structures...)
  3. Use a postprocessor to a projective dependency parsing algorithm to identify and resolve nonprojective links
  4. Add extra types of transitions that can model at least most non-projective structures
  5. Move to a parsing mechanism that does not use or require any constraints on projectivity (e.g., the graph-based MSTParser)
Dependencies

encode relational structure

Relation Extraction with Stanford Dependencies
Dependency paths identify relations like protein interaction

[Erkan et al. EMNLP 07, Fundel et al. 2007]

KaiC \(\text{nsubj}\) interacts \(\text{prep_with}\) SasA
KaiC \(\text{nsubj}\) interacts \(\text{prep_with}\) SasA \(\text{conj_and}\) KaiA
KaiC \(\text{nsubj}\) interacts \(\text{prep_with}\) SasA \(\text{conj_and}\) KaiB
Stanford Dependencies

[de Marneffe et al. LREC 2006]

• The basic dependency representation is projective
• It can be generated by postprocessing headed phrase structure parses (Penn Treebank syntax)
• It can also be generated directly by dependency parsers, such as MaltParser, or the Easy-First Parser
Bell, based in LA, makes and distributes electronic and computer products.
Graph modification to facilitate semantic analysis

Bell, based in LA, makes and distributes electronic and computer products.

Many relationships become short distance!