# Dependency Parsing

## Data structures and algorithms for Computational Linguistics III

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

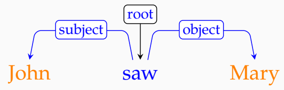
University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2019–2020

---

## Dependency grammars

a refresher



- No constituents, units of syntactic structure are words
- The structure of the sentence is represented by *asymmetric*, *binary* relations between syntactic units
- Each relation defines one of the words as the head and the other as dependent
- The arcs (relations) have labels (dependency types)
- Often an artificial *root* node is used for computational convenience

---

## Dependency grammars

common assumptions, variations

- *Single-headed*: most dependency formalisms require a word to have a single head
- *Acyclic*: most dependency formalism do not allow loops in the graph
- *Connected*: all nodes are reachable from the ‘root’ node
- *Projective*: no crossing dependencies

The above assumptions (except projectivity) are common in dependency parsing.

---

## Dependency parsing

an overview

- Dependency parsing has many similarities with context-free parsing (e.g., the result is a tree)
- They also have some different properties (e.g., number of edges and depth of trees are limited)
- The process involves discovering the relations between words in a sentence
  - Determine the head of each word
  - Determine the relation type
- Dependency parsing can be
  - grammar-driven (hand crafted rules or constraints)
  - data-driven (rules/model is learned from a treebank)

---

## Dependency parsing

common methods for data-driven parsers

There are two main approaches:

Graph-based search for the best tree structure, for example
- find minimum spanning tree (MST)
- adaptations of CF chart parser (e.g., CKY)

(in general, computationally more expensive)

Transition-based similar to shift-reduce parsing (used for programming language parsing)
- Single pass over the sentence, determine an operation (shift or reduce) at each step
- Linear time complexity
- We need an approximate method to determine the operation

---

## Shift-Reduce parsing

a refresher through an example

Grammar

S → P | S + P | S − P
P → Num | P × Num | P / Num

Parser states/actions

| Stack | Input buffer | Action |
|---|---|---|
| | 2 + 3 × 4 | shift |
| 2 | + 3 × 4 | reduce (P → Num) |
| P | + 3 × 4 | reduce (S → P) |
| S | + 3 × 4 | shift |
| S + | 3 × 4 | shift |
| S + 3 | × 4 | reduce (P → Num) |
| S + P | × 4 | shift |
| S + P × | 4 | shift |
| S + P × 4 | | reduce (P → P × Num) |
| S + P | | reduce (S → S + P) |
| S | | accept |

---

## Transition-based parsing

differences from shift-reduce parsing

- The shift-reduce parsers (for programming languages) are deterministic, actions are determined by a table lookup
- Natural language sentences are ambiguous, hence a dependency parser’s actions cannot be made deterministic
- Operations are (somewhat) different: instead of reduce (using phrase-structure rules) we use *arc* operations connecting two nodes with a label
- Further operations are often defined (e.g., to deal with non-projectivity)

---

## Transition based parsing

- Use a *stack* and a *buffer* of unprocessed words
- Parsing as predicting a sequence of transitions like

| | |
|---|---|
| LEFT-ARC: | mark current word as the head of the word on top of the stack |
| RIGHT-ARC: | mark current word as a dependent of the word on top of the stack |
| SHIFT: | push the current word on to the stack |

- Algorithm terminates when all words in the input are processed
- The transitions are not naturally deterministic, best transition is predicted using a machine learning method

(Yamada and Matsumoto 2003; Nivre, Hall, and Nilsson 2004)

# A typical transition system

$$(\underbrace{\sigma \mid \overset{\text{stack top}}{\overbrace{w_i}}}_{\text{stack}},\ \underbrace{\overset{\text{next word}}{\overbrace{w_j}} \mid \beta}_{\text{buffer}},\ \underbrace{A}_{\text{arcs}})$$

Left-Arc$_r$: $(\sigma \mid w_i, w_j \mid \beta, A) \Rightarrow (\sigma\ , w_j \mid \beta, A \cup \{(w_j, r, w_i)\})$

- pop $w_i$,
- add arc $(w_j, r, w_i)$ to $A$ (keep $w_j$ in the buffer)

Right-Arc$_r$: $(\sigma \mid w_i, w_j \mid \beta, A) \Rightarrow (\sigma\ , w_i \mid \beta, A \cup \{(w_i, r, w_j)\})$

- pop $w_i$,
- add arc $(w_i, r, w_j)$ to $A$,
- move $w_i$ to the buffer

Shift: $(\sigma\ , w_j \mid \beta, A) \Rightarrow (\sigma \mid w_j, \ \beta, A)$

- push $w_j$ to the stack
- remove it from the buffer

(Kübler, McDonald, and Nivre 2009, p.23)

---

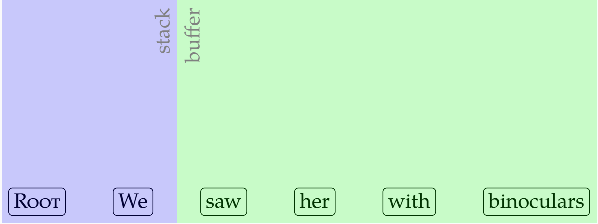
# Transition based parsing: example

Shift

---

# Transition based parsing: example

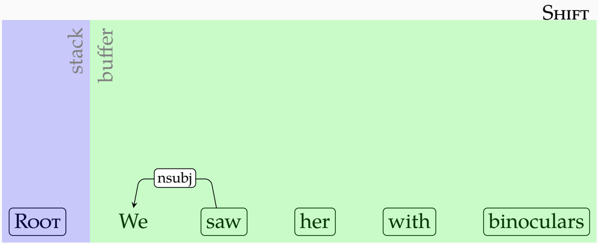
Left-Arc(nsubj)



---

# Transition based parsing: example

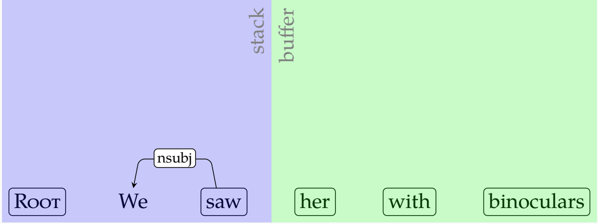
Shift



---

# Transition based parsing: example

Right-Arc(obj)

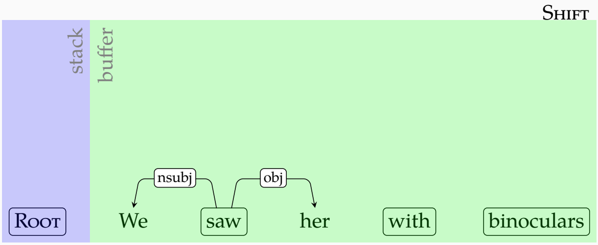


Note: We need Shift for NP attachment.

---

# Transition based parsing: example

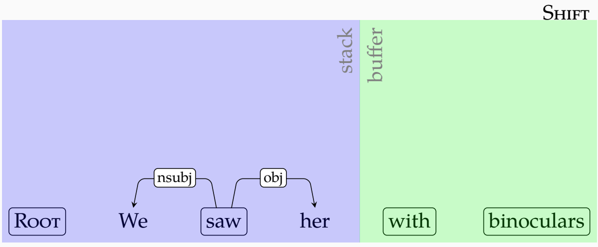
Shift



---

# Transition based parsing: example

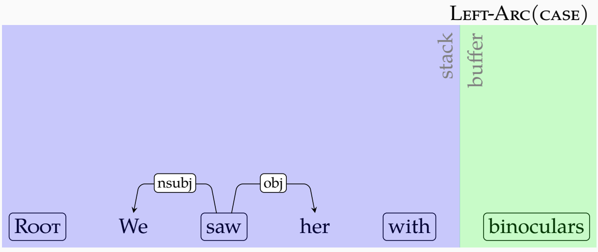
Shift



---

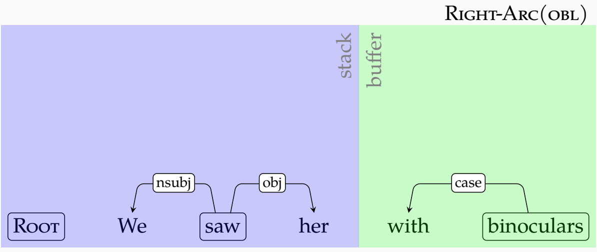
# Transition based parsing: example

Left-Arc(case)



## Transition based parsing: example

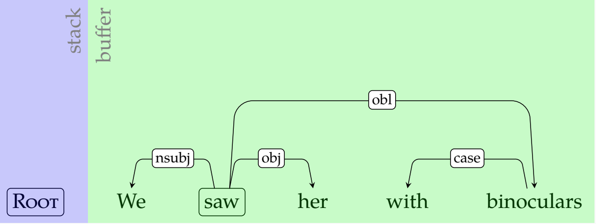
Right-Arc(obl)



---

## Transition based parsing: example

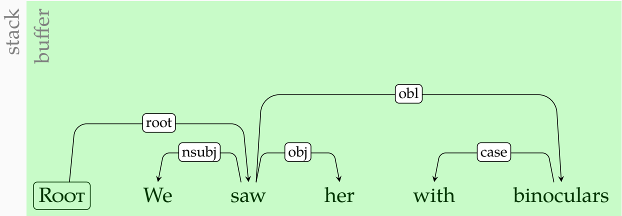
Left-Arc(root)



---

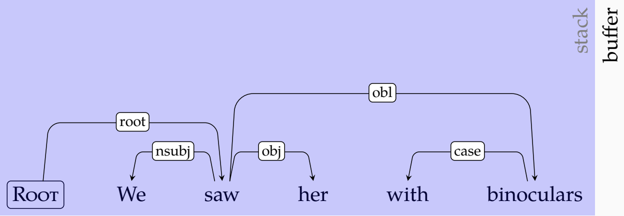
## Transition based parsing: example

Shift



---

## Transition based parsing: example



---

## Making transition decisions

- In classical shift-reduce parsing the actions are deterministic
- In transition-based dependency parsing, we need to choose among all possible transitions
- The typical method is to train a (discriminative) classifier on features extracted from gold-standard *transition sequences*
- Almost any machine learning method method is applicable. Common choices include
  - Memory-based learning
  - Support vector machines
  - (Deep) neural networks

---

## Features for transition-based parsing

- The features come from the parser configuration, for example
  - The word at the top of the stack, (peeking towards the bottom of the stack is also fine)
  - The first/second word on the buffer
  - Right/left dependents of the word on top of the stack/buffer
- For each possible 'address', we can make use of features like
  - Word form, lemma, POS tag, morphological features, word embeddings
  - Dependency relations – $(w_i, r, w_j)$ triples
- Note that for some 'address'–'feature' combinations may be missing

---

## The training data

- We want features like,
  - `lemma[Stack] = duck`
  - `POS[Stack] = NOUN`
  - …
- But treebank gives us:

```
1   Read  read  VERB  VB  Mood=Imp|VerbForm=Fin 0 root
2   on    on    ADV   RB  _                     1 advmod
3   to    to    PART  TO  _                     4 mark
4   learn learn VERB  VB  VerbForm=Inf          1 xcomp
5   the   the   DET   DT  Definite=Def          6 det
6   facts fact  NOUN  NNS Number=Plur           4 obj
7   .     .     PUNCT .   _                     1 punct
```

- The treebank has the outcome of the parser, but none of our features.

---

## The training data

- The features for transition-based parsing have to be from *parser configurations*
- The data (treebanks) need to be preprocessed for obtaining the training data
- Construct a transition sequence by parsing the sentences, and using treebank annotations (the set A) as an 'oracle'
- Decide for

  Left-Arc$_r$ if $(\beta[0], r, \sigma[0]) \in A$
  Right-Arc$_r$ if $(\sigma[0], r, \beta[0]) \in A$
  and all dependents of $\beta[0]$ are attached
  Shift otherwise
- There may be multiple sequences that yield the same dependency tree, the above defines a 'canonical' transition sequence

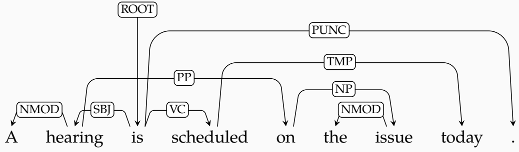
# Non-projective parsing

- The transition-based parsing we defined so far works only for projective dependencies
- One way to achieve (limited) non-projective parsing is to add special operations:
  - Swap operation that swaps tokens in swap and buffer
  - Left-Arc and Right-Arc transitions to/from non-top words from the stack
- Another method is pseudo-projective parsing:
  - preprocessing to ‘projectivize’ the trees before training
    - The idea is to attach the dependents to a higher level head that preserves projectivity, while marking it on the new dependency label
  - post-processing for restoring the projectivity after parsing
    - Re-introduce projectivity for the marked dependencies

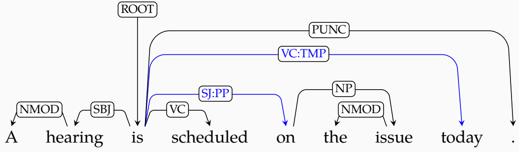
---

# Pseudo-projective parsing

*Non-projective tree:*



*Pseudo-projective tree:*



---

# Transition based parsing: summary/notes

- Linear time, greedy parsing
- Can be extended to non-projective dependencies
- One can use arbitrary features,
- We need some extra work for generating gold-standard transition sequences from treebanks
- Early errors propagate, transition-based parsers make more mistakes on long-distance dependencies
- The greedy algorithm can be extended to beam search for better accuracy (still linear time complexity)

---

# Classification

## the use in dependency parsing

- In transition-based parsing, transition decisions come from a classifier
- At each step during parsing, we have features like
  - `form[Stack] = saw`
  - `lemma[Stack] = see`
  - `POS[Stack] = VERB`
  - `form[Buff] = her`
  - `lemma[Buff] = she`
  - `POS[Buf] = PRON`
- We need to make a transition decision such as
  - Shift
  - Right-Arc(obj)
  - Right-Arc(obl)
  - Left-Arc(acl)
- We can use any multi-class classifier, examples in the literature include
  - SVMs
  - Decision Trees
  - Neural networks
  - ...

---

# Graph-based parsing: preliminaries

- Enumerate all possible dependency trees
- Pick the best scoring tree
- Features are based on limited parse history (like CFG parsing)
- Two well-known flavors:
  - Maximum (weight) spanning tree (MST)
  - Chart-parsing based methods

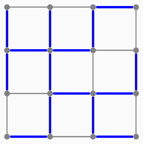
**eisner1996**; McDonald, Pereira, Ribarov, and Hajič 2005

---

# MST parsing: preliminaries

## Spanning tree of a graph

- Spanning tree of a connected graph is a sub-graph which is a tree and traverses all the nodes
- For fully-connected graphs, the number of spanning trees are exponential in the size of the graph
- The problem is well studied
- There are efficient algorithms for enumerating and finding the optimum spanning tree on weighted graphs



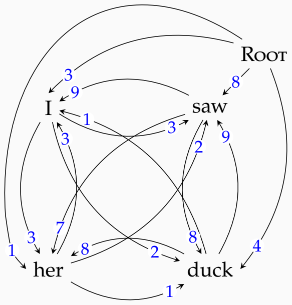
---

# MST algorithm for dependency parsing

- For directed graphs, there is a polynomial time algorithm that finds the minimum/maximum spanning tree (MST) of a fully connected graph (Chu-Liu-Edmonds algorithm)
- The algorithm starts with a dense/fully connected graph
- Removes edges until the resulting graph is a tree

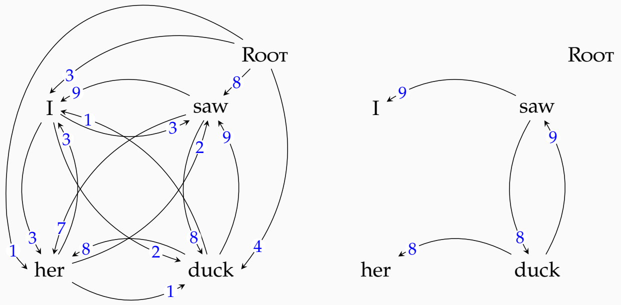
---

# MST example



For each node select the incoming arc with highest weight

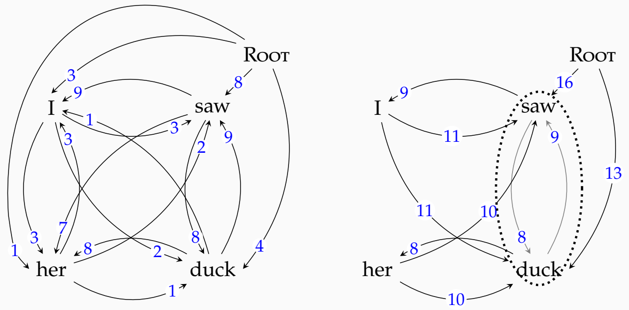
## MST example



Detect the cycles, contract them to a 'single node'

---

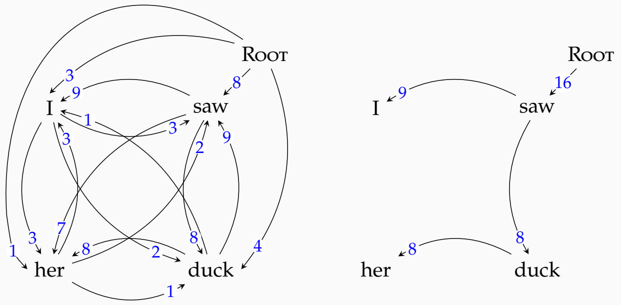
## MST example



Pick the best arc into the combined node, break the cycle

---

## MST example



Once all cycles are eliminated, the result is the MST

---

## Properties of the MST parser

- The MST parser is non-projective
- There is an algorithm with $O(n^2)$ time complexity (Tarjan 1977)
- The time complexity increases with typed dependencies (but still close to quadratic)
- The weights/parameters are associated with edges (often called 'arc-factored')
- We can learn the arc weights directly from a treebank
- However, it is difficult to incorporate non-local features

---

## CKY for dependency parsing

- The CKY algorithm can be adapted to projective dependency parsing
- For a naive implementation the complexity increases drastically $O(n^6)$
  - Any of the words within the span can be the head
  - Inner loop has to consider all possible splits
- For projective parsing, the observation that the left and right dependents of a head are independently generated reduces the complexity to $O(n^3)$

(Eisner 1997)

---

## Non-local features

- The graph-based dependency parsers use edge-based features
- This limits the use of more global features
- Some extensions for using 'more' global features are possible
- This often leads non-projective parsing to become intractable
- Another option is using beam search, and re-ranking based on different/global features

---

## External features

- For both type of parsers, one can obtain features that are based on unsupervised methods such as
  - clustering
  - dense vector representations (embeddings)
  - alignment/transfer from bilingual corpora/treebanks

---

## Errors from different parsers

- Different parsers make different errors
  - Transition based parsers do well on local arcs, worse on long-distance arcs
  - Graph based parsers tend to do better on long-distance dependencies
- Parser combination is a good way to combine the powers of different models. Two common methods
  - Majority voting: train parsers separately, use the weighted combination of their results
  - Stacking: use the output of a parser as features for another

(McDonald and Satta 2007; Sagae and Lavie 2006; Nivre and McDonald 2008)

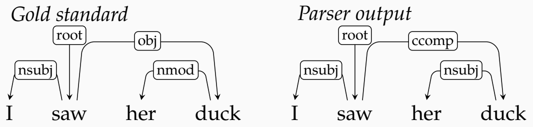
# Evaluation metrics for dependency parsers

- Like CF parsing, exact match is often too strict
- *Attachment score* is the ratio of words whose heads are identified correctly.
    - *Labeled attachment score* (LAS) requires the dependency type to match
    - *Unlabeled attachment score* (UAS) disregards the dependency type
- *Precision/recall/F-measure* often used for quantifying success on identifying a particular dependency type

| | |
|---|---|
| precision | is the ratio of correctly identified dependencies (of a certain type) |
| recall | is the ratio of dependencies in the gold standard that parser predicted correctly |
| f-measure | is the harmonic mean of precision and recall $\left(\frac{2\times\text{precision}\times\text{recall}}{\text{precision}+\text{recall}}\right)$ |

---

# Evaluation example



| | | |
|---|---|---|
| UAS | 100% | |
| LAS | 50% | |
| $\text{Precision}_{\mathtt{nsubj}}$ | 50% | |
| $\text{Recall}_{\mathtt{nsubj}}$ | 100% | |
| $\text{Precision}_{\mathtt{obj}}$ | 0% | (assumed) |
| $\text{Recall}_{\mathtt{obj}}$ | 0% | |

---

# Averaging evaluation scores

- Average scores can be

| | |
|---|---|
| macro-averaged | over sentences |
| micro-averaged | over words |

- Consider a two-sentence test set with

| | words | correct |
|---|---|---|
| sentence 1 | 30 | 10 |
| sentence 2 | 10 | 10 |

    - word-based average attachment score: 50% (20/40)
    - sentence-based average attachment score: 66% ((1 + 1/3)/2)

---

# Dependency parsing: summary

- Dependency relations are often semantically easier to interpret
- It is also claimed that dependency parsers are more suitable for parsing free-word-order languages
- Dependency relations are between words, no phrases or other abstract nodes are postulated
- Two general methods:

| | |
|---|---|
| transition based | greedy search, non-local features, fast, less accurate |
| graph based | exact search, local features, slower, accurate (within model limitations) |

- Combination of different methods often result in better performance
- Non-projective parsing is more difficult
- Most of the recent parsing research has focused on better machine learning methods (mainly using neural networks)

---

# References / additional reading material

- Kübler, McDonald, and Nivre (2009) is an accessible book on to dependency parsing
- The new version of Jurafsky and Martin (2009) also includes a draft chapter on dependency grammars and dependency parsing

---

# References / additional reading material (cont.)

Eisner, Jason (1997). “Bilexical grammars and a cubic-time probabilistic parser”. In: *Proceedings of the Fifth International Conference on Parsing Technologies (IWPT)*.

Jurafsky, Daniel and James H. Martin (2009). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. second. Pearson Prentice Hall. ISBN: 978-0-13-504196-3.

Kübler, Sandra, Ryan McDonald, and Joakim Nivre (2009). *Dependency Parsing*. Synthesis lectures on human language technologies. Morgan & Claypool. ISBN: 9781598295962.

McDonald, Ryan, Fernando Pereira, Kiril Ribarov, and Jan Hajič (2005). “Non-projective Dependency Parsing Using Spanning Tree Algorithms”. In: *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*. HLT ’05. Vancouver, British Columbia, Canada: Association for Computational Linguistics, pp. 523–530. DOI: `10.3115/1220575.1220641`. URL: `http://dx.doi.org/10.3115/1220575.1220641`.

McDonald, Ryan and Giorgio Satta (2007). “On the complexity of non-projective data-driven dependency parsing”. In: *Proceedings of the 10th International Conference on Parsing Technologies*. Association for Computational Linguistics, pp. 121–132.

Nivre, Joakim, Johan Hall, and Jens Nilsson (2004). “Memory-based dependency parsing”. In: *Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL)*. Ed. by Hwee Tou Ng and Ellen Riloff, pp. 49–56.

---

# References / additional reading material (cont.)

Nivre, Joakim and Ryan McDonald (June 2008). “Integrating Graph-Based and Transition-Based Dependency Parsers”. In: *Proceedings of ACL-08: HLT*. Columbus, Ohio: Association for Computational Linguistics, pp. 950–958. URL: `http://www.aclweb.org/anthology/P/P08/P08-1108`.

Sagae, Kenji and Alon Lavie (June 2006). “Parser Combination by Reparsing”. In: *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*. New York City, USA: Association for Computational Linguistics, pp. 129–132. URL: `http://www.aclweb.org/anthology/N/N06/N06-2033`.

Tarjan, R. E. (1977). “Finding optimum branchings”. In: *Networks* 7.1, pp. 25–35. ISSN: 1097-0037. DOI: `10.1002/net.3230070103`.

Yamada, Hiroyasu and Yuji Matsumoto (2003). “Statistical dependency analysis with support vector machines”. In: *Proceedings of 8th international workshop on parsing technologies (IWPT)*. Ed. by Gertjan Van Noord, pp. 195–206.