Dependency parsing

Data Structures and Algorithms for Computational Linguistics III (ISCL-BA-07)

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

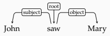
Winter Semester 2022/23

version: 6f31f40 @2023-01-30

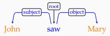
Dependency grammars

introduction

- Dependency grammars gained popularity in linguistics (particularly in CL) rather recently
- They are old: roots can be traced back to Pāṇini (approx. 5th century BCE)
- Modern dependency grammars are often attributed to **tesniere1959**
- The main idea is capturing the relations between words, rather than grouping them into (abstract) constituents

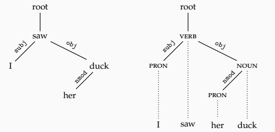


Dependency grammars



- 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
- Typically, the links (relations) have labels (dependency types)
- Often an artificial *root* node is used for computational convenience

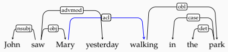
Dependency grammars: alternative notation(s)



Dependency grammars: common assumptions

- Every word has a single head
- The dependency graphs are acyclic
- The graph is connected
- With these assumptions, the representation is a tree
- Note that these assumptions are not universal but common for dependency parsing

Dependency grammars: projectivity



- If a dependency graph has no crossing edges, it is said to be *projective*, otherwise *non-projective*
- Non-projectivity stems from long-distance dependencies and free word order
- Projective dependency trees can be represented with context-free grammars
- In general, projective dependencies are parseable more efficiently

Dependency grammars

Advantages and disadvantages

+ Close relation to semantics
+ Easier for flexible/free word order
+ Lots, lots of (multi-lingual) computational work, resources
+ Often much useful in downstream tasks
+ More efficient parsing algorithms
– No distinction between modification of head or the whole 'constituent'
– Some structures are difficult to annotate, e.g., coordination

Dependency parsing

- Dependency parsing has many similarities with context-free parsing (e.g., trees)
- It also has some differences (e.g., number of edges and depth of trees are limited)
- Dependency parsing can be
  - grammar-driven (hand crafted rules or constraints)
  - data-driven (rules/model is learned from a treebank)

Grammar-driven dependency parsing

- Grammar-driven dependency parsers typically based on
  - lexicalized CF parsing
  - constraint satisfaction problem
    - start from fully connected graph, eliminate edges that do not satisfy the constraints
    - exact solution is intractable, often heuristics, approximate methods are employed
    - sometimes 'soft', or weighted, constraints are used
  - Practical implementations exist
- Our focus will be on data-driven methods

Data-driven dependency parsing

common methods for data-driven parsers

- Almost any modern/practical dependency parser is statistical
- The 'grammar', and the (soft) constraints are learned from a *treebank*
- 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 (LR(k)) 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 best 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 (LR) parsers for formal languages are deterministic, actions are determined by a table lookup
- Natural language sentences are ambiguous, 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 words with a labeled arc
- More operations may be 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

A typical transition system

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

LEFT-ARC$_r$: $(\sigma \mid w_i, w_j \mid \beta, A) \Rightarrow (\sigma \quad, w_j \mid \beta, A \cup \{(w_j, r, w_i)\})$

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

RIGHT-ARC$_r$: $(\sigma \mid w_i, w_j \mid \beta, A) \Rightarrow (\sigma \quad, w_i \mid \beta, A \cup \{(w_i, r, w_j)\})$

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

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

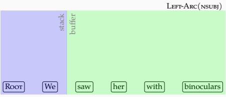
- push w_j to the stack
- remove it from the buffer

Transition based parsing: example

SHIFT

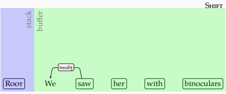
Transition based parsing: example

LEFT-ARC(NSUBJ)



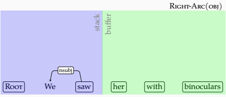
Transition based parsing: example

SHIFT



Transition based parsing: example

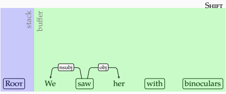
RIGHT-ARC(OBJ)



Note: We need SHIFT for NP attachment.

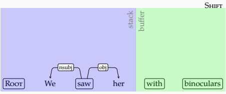
Transition based parsing: example

SHIFT



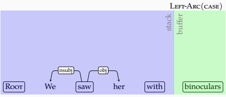
Transition based parsing: example

SHIFT



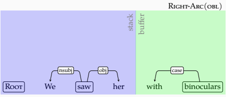
Transition based parsing: example

LEFT-ARC(CASE)



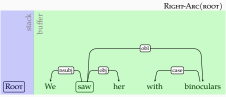
Transition based parsing: example

RIGHT-ARC(OBL)



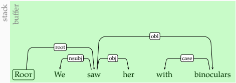
Transition based parsing: example

RIGHT-ARC(ROOT)

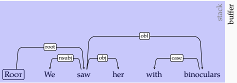


Transition based parsing: example

SHIFT



Transition based parsing: example



Making transition decisions

- Unlike deterministic parsing (for formal languages), we cannot build a table to determinize the parser actions
- The typical method is to train a (discriminative) classifier
- Almost any machine learning (classification) method is applicable
- The features used for prediction is extracted from the states of the parser:
  - Top-k words on the stack
  - Next-m words in the buffer
  - Transition decisions made so far (the arcs)
- Given these objects, one can extract and use arbitrary features:
  - Words as categorical variables
  - POS tags
  - Embeddings
  - …

The training data

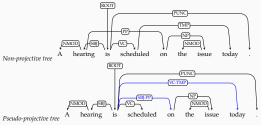
- The features for transition-based parsing have to be from *parser configurations*
- The data (treebanks) need to be preprocessed for obtaining the training data
- The general idea is to construct a transition sequence by performing a 'mock' parsing using treebank annotations as an 'oracle'
- There may be multiple sequences that yield the same dependency tree, this procedure defines a 'canonical' transition sequence
- For example,

  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

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 the stack and the 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 the operation on the new dependency label
  - post-processing for restoring the projectivity after parsing
    - Re-introduce projectivity for the marked dependencies

Pseudo-projective parsing



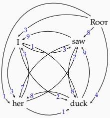
Transition based parsing: summary/notes

- Linear time, greedy, projective parsing
- Can be extended to non-projective dependencies
- 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)

MST algorithm for dependency parsing

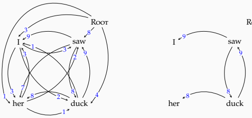
- For directed graphs, there is a polynomial time algorithm that finds the minimum/maximum spanning tree (MST) of a fully connected graph (Chu-Liu-Edmonds algorithm)
- The algorithm starts with a dense/fully connected graph
- Removes edges until the resulting graph is a tree

MST example



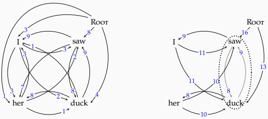
For each node select the incoming arc with highest weight

MST example



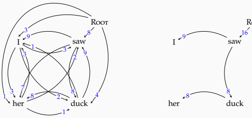
Detect the cycles, contract them to a 'single node'

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
- 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

External features

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

Evaluation metrics for dependency parsers

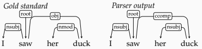
- Like CF parsing, exact match is often too strict
- *Attachment score* is the ratio of words whose heads are identified correctly.
  - *Labeled attachment score* (LAS) requires the dependency type to match
  - *Unlabeled attachment score* (UAS) disregards the dependency type
- *Precision*/*recall*/*F-measure* 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%	
$\text{Precision}_{\text{nsubj}}$	50%	
$\text{Recall}_{\text{nsubj}}$	100%	
$\text{Precision}_{\text{obj}}$	0%	(assumed)
$\text{Recall}_{\text{obj}}$	0%	

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)
- Reading suggestion: **jurafsky2009 kubler2009**

Acknowledgments, references, additional reading material