Dependency parsing

LING83600

1 Introduction

Simple context-free grammars do not encode the notion of *headedness*, informally defined as follows:

> X is the *head* of a constituent XP if and only if X is an immediate daughter of XP, and X "determines the category of XP".

But headedness is a core topic in recent syntactic theory, and modern CFG parsing makes extensive use of *head rules*, hand-written generalizations about headedness, e.g., "the head of a verb phrase (`VP`) is a verb (`V*`) or modal (`MD`)", to resolve grammatical ambiguities.

Dependency grammar (Tesnière 1959) began its life as a fringe theory of syntax, largely ignored until it was rediscovered in by specialists in natural language processing in search of faster parsing algorithms. Dependency parsing is now a core NLP task, and the subject of numerous shared tasks over the last decade or so (e.g., Buchholz and Maris 2006, Nivre et al. 2007, Seddah et al. 2014, Zeman et al. 2017, 2018). Thanks to the Universal Dependencies project,[1] dependency-parsed data is freely-available for over seventy languages.

Bibliographic note

Eisenstein (2019: ch. 11) discusses dependency parsing in detail. Kübler et al. 2009 is the only book-length treatment of dependency parsing, and some definitions here have been loosely adapted from chap. 2 of that resource. The shift-reduce parsing example is adapted from unpublished slides by Brian Roark.

Software note

UDPipe (Straka et al. 2016) is an excellent tokenizer-tagger-dependency parser, and comes with pre-trained models for Universal Dependencies languages.[2] SpaCy is another popular option, with good documentation.[3] New BERT-based parsers are slowly replacing all of the above.

[1] `https://universaldependencies.org/`
[2] `http://ufal.mff.cuni.cz/udpipe`
[3] `https://spacy.io/`

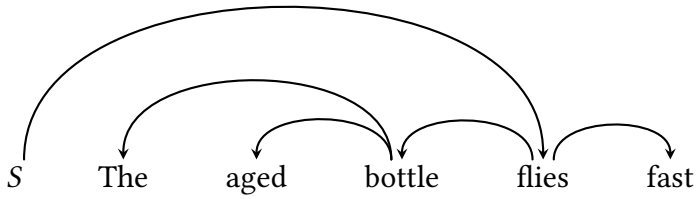

Figure 1: Dependency graph for the sentence *The aged bottle flies fast.*

2 Definitions

2.1 Dependency grammar

Dependency grammar has proved difficult to formalize, but the following provides a general formalism expressed in similar terms to the formalization of CFGs earlier. A dependency grammar G is a triple where:

- S is the designated start symbol (i.e., the highest projection in a sentence).
- Σ is an array of *terminal symbols*, corresponding to words (i.e., X_0s) in a syntactic tree.
- R is a set of *production rules*. These rules are of the form $A \to \beta$ where $A \in \Sigma \cup \{S\}$ and $\beta \in \Sigma$.

2.2 Derivation

A derivation in dependency grammar is expressed as a *dependency graph*, a directed graph represented by states $\Sigma \cup \{S\}$ and arcs R. A dependency graph is *well-formed* when for all $s, s' \in \Sigma \cup \{S\}$ if $(s, d) \in R$ then $(s', d) \notin R$ where $s' \neq s$. That is, a dependency graph is well-formed if each terminal state $\beta \in \Sigma$ has exactly one incoming arc from $A \in \Sigma \cup \{S\}$ such that $A \to \beta$ is a production rule in R. Because of this last restriction, we can represent a dependency tree compactly by storing the terminal array Σ and an array $\mathcal{H}$ where $\mathcal{H}(i)$ is the index of the incoming arc (i.e., the head of) the ith state in Σ. For instance, Figure 1 is characterized by:

- $\Sigma = [\text{The}, \text{aged}, \text{bottle}, \text{flies}, \text{fast}]$.
- $\mathcal{H} = [3, 3, 4, 0, 4]$.

3 Projectivity

We say a dependency graph is *projective* if when the words are put in linear order, none of the dependency edges cross/intersect. One way to formalize this is to require that if some $A \in \Sigma$ is a head then all heads and their direct dependencies form a contiguous substring; that is, if A is a head, all of its *left dependents* must be immediately to the left and all of its *right dependents*

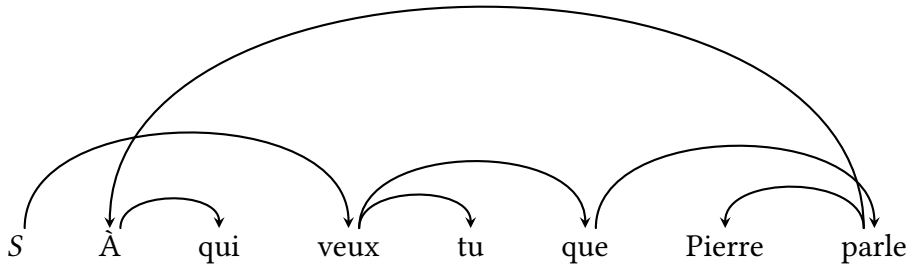

Figure 2: Non-projective dependency tree of the French sentence *À qui veux-tu que Pierre parle* 'To whom do you want Pierre to talk?'.

must be immediately to the right. This property is often assumed by methods for converting constituency to dependency, as well as by some parsing algorithms. However, non-projectivity is useful, or even essential, in many languages (e.g., Figure 2; cf. McDonald et al. 2005).

4 Dependency-constituency conversion

With a complete set of head rules, it is possible to convert a constituency tree to a dependency tree, according to the following algorithm:

- Decorate each non-terminal node in the constituency tree with its head (e.g., Figure 3).[4]
- Initialize the graph with an arc from the start symbol S to the head of the constituency tree.
- Traverse the tree, and for each non-terminal node $s \in S$, find all nodes s' it immediately dominates; if s and the dominated node s' do not share a head label, add an arc $(\mathcal{H}(s), \mathcal{H}(s'))$;

While dependency is not an inherently typed relation, it is also possible to label dependencies (e.g., `advmod`: "adverbial modifier of"). A constituency tree "decorated" with head rules is shown in Figure 3, and Figure 4 shows a labeled dependency graph for the same sentence.

5 Parsing

5.1 Maximum spanning tree parsing

The earliest work on automatic dependency parsing employs a highly inefficient generative model. McDonald et al. (2005) propose an influential graph-theoretic algorithm, as follows:

- Initialize the parse as a fully connected directed graph with states $\Sigma \cup \{S\}$.
- Weight all arcs according to some arbitrary fitness function.

[4] We assume that terminals are their own heads and ignore unary productions (including preterminals).

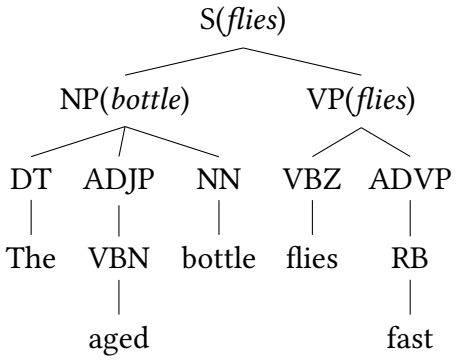

Figure 3: Penn Treebank-style constituency parse of the sentence *The aged bottle flies fast*, with propagated heads in parentheses

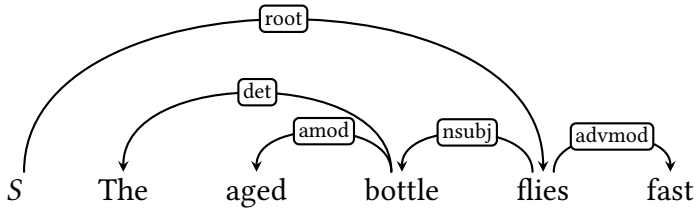

Figure 4: Labeled dependency graph for the sentence *The aged bottle flies fast.*

- Compute the *maximum spanning tree* of this graph.[5]

5.2 Shift-reduce parsing

However, if we assume projective graphs, *shift-reduce* parsing offers a linear-time algorithm for dependency parsing. In this framework, parses are generated incrementally by factoring them into a series of shift (state changing) and reduce (arc-adding) operations. The fitness of a shift or reduce operation is determined by an arbitrary fitness function over the incomplete parse.

The shift-reduce parser state is represented by a stack $\mathcal{S}$, a buffer $\mathcal{B}$, as well as a partial parse. At initialization, $\mathcal{S}$ contains only the designated start symbol S, and the buffer $\mathcal{B}$ is simply Σ. Parsing halts when $\mathcal{B}$ is empty and $\mathcal{S}$ contains only S. The parse is built up incrementally by a series of shift and reduce *transitions*. There are several ways to define these transitions. One of the best known is the *arc-standard* system, which defines the following operations:

- The *shift* operation removes the first element from the buffer and pushes it onto the stack.
- The *left-reduce* operation pops the top two elements of the stack (s_i and s_j, where s_j was previously immediately above s_i), creates a leftward dependency $s_i \leftarrow s_j$, and then pushes the head s_j back onto the stack.

[5] This is the subset of the graph that connects all states without cycles and maximizes the total weight of the remaining arcs. The standard algorithm runs in cubic time.

	Stack	Buffer	Move
0.	S	The aged bottle flies fast	`shift`
1.	S The	aged bottle flies fast	`shift`
2.	S The aged	bottle flies fast	`shift`
3.	S The aged bottle	flies fast	`left-reduce` (aged ← bottle)
4.	S The bottle	flies fast	`left-reduce` (The ← bottle)
5.	S bottle	flies fast	`shift`
6.	S bottle flies	fast	`left-reduce` (bottle ← flies)
7.	S flies	fast	`shift`
8.	S flies fast		`right-reduce` (flies → fast)
9.	S flies		`right-reduce` (S → flies)
10.	S		`halt`

Table 1: A sample arc-standard derivation of the sentence *The aged bottle flies fast.*

	Stack	Buffer	Move
0.	S	The aged bottle flies fast	`shift`
1.	S The	aged bottle flies fast	`shift`
2.	S The aged	bottle flies fast	`left-reduce` (aged ← bottle)
3.	S The	bottle flies fast	`left-reduce` (The ← bottle)
4.	S	bottle flies fast	`shift`
5.	S bottle	flies fast	`left-reduce` (bottle ← flies)
6.	S	flies fast	`shift`
7.	S flies	fast	`shift`
8.	S flies fast		`right-reduce` (flies → fast)
9.	S flies		`right-reduce` (S → flies)
10.	S		`halt`

Table 2: A sample arc-hybrid derivation of the sentence *The aged bottle flies fast.*

- The *right-reduce* operation pops the top two elements of the stack (s_i and s_j, defined as before), creates a rightward dependency $s_i \rightarrow s_j$, and pushes the head s_i onto the stack.

Note that the reduce operations are only defined when there are at least two symbols on the stack; otherwise, shift is the only option. A sample arc-standard derivation is shown in Table 1.

The *arc-hybrid* transition system is a variant of the arc-standard system. The invariant property of the arc-standard system is that reduce operations apply when the top two elements on the stack are in a head-dependent relationship. In contrast, the invariant property of the arc-hybrid system is that reduce operations apply when the top of the stack is a dependent. The arc-hybrid system uses the same definition of the shift and right-reduce operations as the arc-standard system. Thus, when head of the top of the stack is immediately below it on the stack, right-reduce applies as before. However, arc-hybrid left-reduce applies now when the head of the top of the stack is at the front of the buffer. A sample arc-hybrid derivation is shown in Table 2.

But how do we predict which transition to use at any point? Transitions are usually predicted

using a classifier with features based on any of the following:

- Direction of dependency (e.g., how likely is a token/tag to have a leftward dependent?)
- Distance of dependency (e.g., how likely is a token/tag to have a dependent n symbols to the left?)
- Valency of heads (e.g., how likely is a token/tag to have n dependents?)
- Bilexical relations (e.g., how likely is a token/tag to have some other hypothesized token/tag as a dependent?)
- Tokens or tags of nearby dependencies in the partial parse
- The last few tokens or tags in the stack
- The next few tokens or tags in the buffer
- Various conjunctions of the above

5.3 Dynamic oracles

To train the fitness function, it is also necessary to construct an *oracle*, a function which derives optimal transition sequences from gold dependency parse trees. A *static oracle* produces a single transition sequence, but this is suboptimal as most transition systems (including the ones above) exhibit *spurious ambiguity*—many transition sequences may map onto the same gold tree. Thus state-of-the-art transition-based dependency parsers use a *dynamic oracle* (Goldberg and Nivre 2012). At each parser configuration, the dynamic oracle computes the *cost* for all valid transitions. A transition t is optimal if it does not commit the parser to a parsing error, meaning that the number of gold arcs reachable after applying t to the current configuration is not less than those reachable before applying it.

For instance, if we pop the top of the stack (as in both arc-hybrid reduce operations), it is no longer possible to create arcs between the element at the top of the stack and any elements in the buffer, so this has non-zero cost if there are any such dependencies in the gold parse. The *cost* of a transition t is defined as the number of gold arcs made unreachable by applying it the current configuration.[6] The dynamic oracle predicts the highest-scoring no-cost transition. In the case that there is no such transition, one may halt training for this sentence, randomly select a valid move, or select the lowest-cost move.

6 Applications

In addition to recovering head-dependent relationships—itself a useful service—dependency parsing is an efficient way to generate syntactic features for downstream NLP tasks including:

- "string-to-dependency" machine translation (e.g., Shen et al. 2008),

[6] For a full explication of cost functions for various transition systems, see Goldberg and Nivre 2013.

- vector-space semantic models (e.g., Levy and Goldberg 2014),
- grammatical error detection (e.g., Morley et al. 2014), and
- disfluency detection (e.g., Honnibal and Johnson 2014).

References

Buchholz, Sabine, and Erwin Maris. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, 149–164.

Eisenstein, Jacob. 2019. *Introduction to Natural Language Processing*. MIT Press.

Goldberg, Yoav, and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of COLING 2012*, 959–976.

Goldberg, Yoav, and Joakim Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *Transactions of the Association for Computational Linguistics* 1:403–414.

Honnibal, Matthew, and Mark Johnson. 2014. Joint incremental disfluency detection and dependency parsing. *Transactions of the Association of Computational Linguistics* 2:131–142.

Kübler, Sandra, Ryan McDonald, and Joakim Nivre. 2009. *Dependency Parsing*. Morgan & Claypool.

Levy, Omer, and Yoav Goldberg. 2014. Dependency-based word embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 302–308.

McDonald, Ryan, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, 523–530.

Morley, Eric, Anna Eva Hallin, and Brian Roark. 2014. Data-driven grammatical error detection in transcripts of children's speech. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 980–989.

Nivre, Joakim, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. 2007. The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, 915–932.

Seddah, Djamé, Reut Tsarfaty, and Sandra Kübler. 2014. Introducing the SPRML 2014 shared task on parsing morphologically-rich languages. In *1st Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages*, 103–109.

Shen, Libin, Jinxi Xu, and Ralph Weischedel. 2008. A new string-to-dependency machine translation algorithm with a target dependency language model. In *Proceedings of ACL-08: HLT*, 577–585.

Straka, Milan, Jan Hajič, and Jana Straková. 2016. UDPipe: trainable pipeline for processing CoNLL-U files performing tokenization, morphological analysis, POS tagging and parsing. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation*, 4290–4297.

Tesnière, Lucien. 1959. *Éléments de syntaxe structurale*. Klincksieck.

Zeman, Daniel, Jan Hajič, Martin Popel, Martin Potthast, Milan Straka, Filip Ginter, Joakim Nivre, and Slav Petrov. 2018. CoNLL 2018 shared task: multilingual parsing from raw text to universal dependencies. In *CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, 1–21.

Zeman, Daniel, Martin Popel, Milan Straka, Jan Hajič, Joakim Nivre, Filip Ginter, …, and Josie Li. 2017. CoNLL 2017 shared task: multilingual parsing from raw text to universal dependencies. In *CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, 1–19.