CS562/CS662 (Natural Language Processing): Dependency parsing

Kyle Gorman
Center for Spoken Language Understanding
Oregon Health & Science University

1 Introduction

The context-free grammar (CFG) formalism does not directly encode the notion of *headedness*, 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 modern linguistic theory, and as we shall see, modern CFG parsing makes extensive use of *head rules*—hand-written generalizations about headedness, e.g., "the head of a VP is a verb or a modal"—to condition PCFG rule probabilities and as features for tree (re)scoring.

Dependency grammar (Tesnière 1959) began its life as a fringe theory of syntax. The focus of this theory is dependency (read: headedness): there is no phrase structure and no constituents, merely head-dependent relationships. Tesnière's sole book on the theory was published posthumously, and it was largely ignored. Or it was, until approximately 15 years ago when natural language processing experts in search of efficiently parseable syntactic representations revived the theory. Dependency parsing is now one of the core NLP tasks.

2 Formalism

Dependency grammar has proved difficult to formalize, but it can be defined as follows:

- A dependency grammar is a 3-tuple (Σ, R, S) where:
  - Σ is the set of terminal symbols (i.e., words)
  - R is the set of rules (to be defined)
  - S is the designated start symbol
- All rules in R are all of the form $\alpha \rightarrow \beta$, where $\alpha \in (\Sigma \cup S)$ and $\beta \in \Sigma$

A derivation in dependency grammar is a directed acyclic graph, defined as follows:

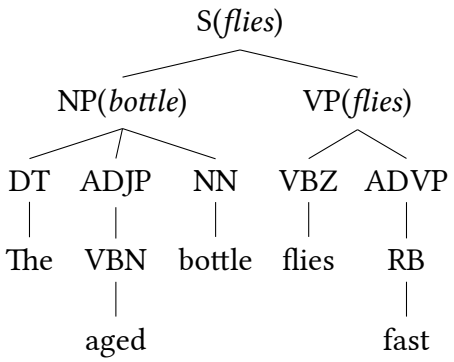

Figure 1: Penn Treebank-style constituency parse of the sentence *The aged bottle flies fast*, with propagated heads in parentheses

- A dependency tree is a 2-tuple $(\mathcal{B}, \mathcal{A})$ where:
  - $\mathcal{B}$ is the list of terminal symbols in the sentence (i.e., words)
  - $\mathcal{A}$ is the set of all arcs of the form $\alpha \rightarrow \beta$ for all $\beta \in \mathcal{B}$, where $\alpha \in (\mathcal{B} \cup S)$ and $\alpha \neq \beta$

With this formalism, a dependency tree can be represented compactly by the arrays $\mathcal{B}$ and $\mathcal{H}$ where $\mathcal{H}_i$ is the index of the head of $\mathcal{B}_i$.

3 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 1)[1]
- Create a dependency from the start symbol S of the dependency tree (here, ROOT) to the head of the start symbol of the constituency tree
- Traverse the tree, and for each non-terminal node n, find all nodes n' it immediately dominates; if n and the dominated node n' do not share a head label, create a dependency from the head of n to the head of n'

Note that, by convention, the main verb is the head of a sentence. The resulting dependency tree is shown in Figure 2. While dependency is not an inherently typed relation, it is also possible to label dependencies (e.g., `advmod`: "adverbial modifier of"). It is also possible to convert a dependency tree to a constituency tree, but the resulting tree will in many cases be "flatter" than the original.

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

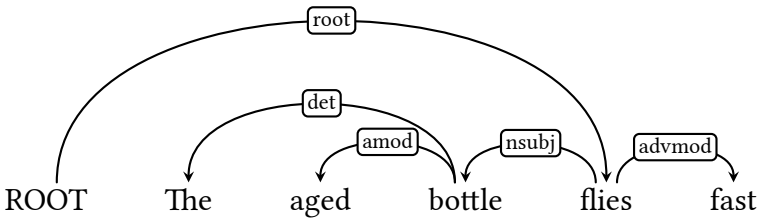

Figure 2: Projective labeled dependency tree of the sentence *The aged bottle flies fast*

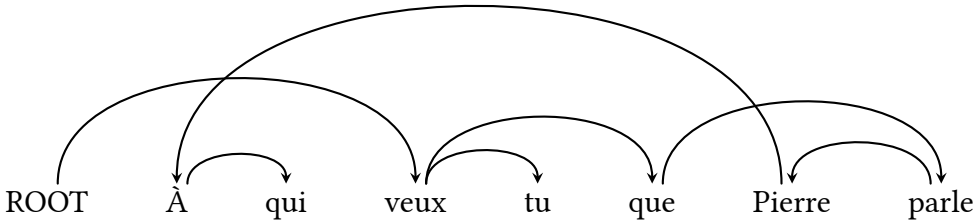

Figure 3: Non-projective dependency tree of the French sentence *À qui veux-tu que Pierre parle?* (lit. 'To whom want you that Pierre talks?')

4 Projectivity

We say a dependency tree is *projective* if when the words are put in in linear order, none of the dependency edges cross/intersect (Hays 1964). More formally:

> A tree is projective if all heads and their direct dependencies form a contiguous substring; that is, if n is a head, all of its left dependents must be immediately to the left and all of its right dependents 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 but projective trees are not sufficient to analyze all sentences in all languages (e.g., Figure 3), or even English in some cases.

5 Parsing

- The earliest work on automatic dependency parsing employs a generative model; the naïve algorithm is $O(n^5)$
- McDonald et al. (2005) propose a graph-theoretic algorithm:
  - Initialize the parse as a fully connected directed graph where the vertices are symbols
  - Score all arcs according to some fitness function
  - Compute the *maximum spanning tree* of this graph

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

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

However, the most popular methods are based on shift-reduce 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 (possibly incomplete) parse is a function of the scores of the operations used to reach the current parse state.

All shift-reduce parser consist of a stack, a buffer, as well as a partial parse ($\mathcal{B}$, $\mathcal{A}$). At initialization, the stack contains only the designated start symbol S, and the buffer is the ordered list of words in $\mathcal{B}$; $\mathcal{A}$, the list of arcs, is empty. Parsing terminates when the buffer is empty and the stack 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:

- *Shift* operation removes the first element from the buffer and pushes it onto the stack
- *Left-reduce* operation pops the top two elements of the stack (w_i and w_j, where w_j was previously immediately above w_i), creates a leftward dependency ($w_i \leftarrow w_j$), and then pushes the head w_j back onto the stack
- *Right-reduce* operation pops the top two elements of the stack (w_i and w_j, defined as before), creates a rightward dependency ($w_i \rightarrow w_j$), and then pushes the head w_i back onto the stack

Note that both 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

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

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

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 linear classifier with features based on any of the following:

- Direction of dependency (e.g., how likely is token or tag X to have a leftward dependents?)
- Distance of dependency (e.g., how likely is token or tag X to have a dependent n to the left?)
- Valency of heads (e.g., how likely is token or tag X to have n dependents?)
- Bilexical relations (e.g., how likely is token or tag X to have token or tag Y as a dependent?)
- Tokens or tags of nearby dependencies in the partial parse
- Last few tokens or tags in the stack
- Next few tokens or tags in the buffer
- Various combinations of the above

It is necessary to generate an *oracle* which derives optimal transition sequences from gold dependency parse trees used to train the transition classifier. A *static oracle* produces a single transition sequence, but this is suboptimal as most transition systems 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 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.[2] The dynamic oracle predicts the highest-scoring no-cost transition. In the case that there is no such transition, we may either terminate parsing for this sentence (Collins and Roark 2004), or randomly select a valid move.

6 Beam search

A shift-reduce dependency parser backed by a linear model can also be augmented with beam search (Zhang and Clark 2011), as follows:

- Each of the k states in the beam consists of a stack, partial parse, and buffer
- The score of each state is given by the sum of the scores of all moves that produced the current stack/parse/buffer configuration.
- For each iteration, generate the successors for all states in the beam by applying shift, left-reduce, and right-reduce to each state,
- Score the resulting states using the linear model; only the k-best scoring states are retained in the beam
- Let a state be *final* if its stack contains only the ROOT symbol and its buffer is empty; search terminates once all states in the beam are final, and the parse of the top-scored state is returned

This generally produces higher-quality parses, but is considerably slower.

7 Applications

In addition to recovering head-dependent relationships, dependency parsing is an efficient way to generate syntactic features for downstream NLP tasks, such as:

- "string-to-dependency" machine translation (Shen et al. 2008)
- Vector-space semantic models (Padó and Lapata 2007)
- Grammatical error detection (Morley et al. 2014)
- Disfluency detection (Honnibal and Johnson 2014)

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

References

Collins, Michael, and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In *ACL*, 111–118.

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

Hays, David G. 1964. Dependency theory: A formalism and some observations. *Language* 40:511–525.

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

McDonald, Ryan, Fernando Pereira, Kiril Ribarow, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *EMNLP*, 523–530.

Morley, Eric, Anna Eva Hallin, and Brian Roark. 2014. Data-driven grammatical error detection in transcripts of children's speech. In *EMNLP*, 980–989.

Padó, Sebastian, and Mirella Lapata. 2007. Dependency-based construction of semantic space models. *Computational Linguistics* 33:161–199.

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

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

Zhang, Yue, and Steven Clark. 2011. Syntactic processing using the generalized perceptron and beam search. volume 37, 105–151.