Lecture 5: Dependency Parsing

Tianpei Xie

Jul. 4th., 2022

Contents

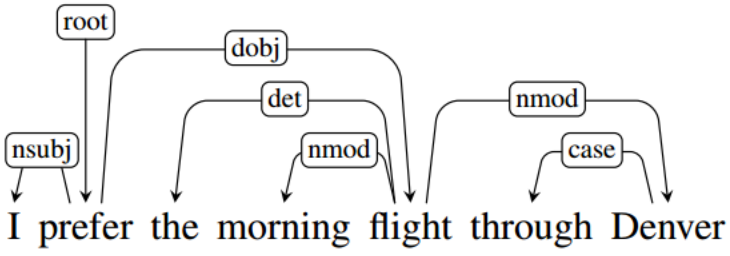

Figure 1: An example dependent parse tree.

1 Concepts

There are two major categories of grammar study:

- **Phrase-Structure Grammars**: discussed in previous chapters. It focuses on *constituency structures and relations*, groups of words can behave as single units. Part of developing a grammar involves building an inventory of the constituents in the language. The constituents can be placed in one place together but not individual word within it.
- **Dependency Grammars (DG)**: focuses on the *dependency* relation (as opposed to the *constituency relation* of *phrase structure*). In dependency formalisms, phrasal constituents and phrase-structure rules *do not play a direct role*. Instead, the syntactic structure of a sentence is described *solely* in terms of ***directed binary* grammatical *relations*** between the words

An example of dependency parse tree is illustrated in Figure 1. Relations among the words are illustrated above the sentence with *directed, labeled arcs* from heads to dependents. We call this a **typed dependency structure** because the labels are drawn from a *fixed inventory* of grammatical relations. A **root** node explicitly marks the root of the tree, the **head** of the entire structure.

Note the absence of nodes corresponding to phrasal constituents or lexical categories in the dependency parse; the internal structure of the dependency parse consists solely of directed relations between lexical items in the sentence. These **head-dependent relationships** directly encode important information that is often buried in the more complex phrase-structure parses.

A major **advantage** of dependency grammars is their ability to deal with languages that are *morphologically* rich and have a relatively **free word order**. The same type of dependent relation but with different word ordering may require different rules in phase structure grammar. Thus, a dependency grammar approach abstracts away from *word order information*, representing only the information that is necessary for the parse.

An additional practical motivation for a dependency-based approach is that the **head-dependent relations** provide an approximation to the ***semantic relationship*** between **predicates** and their **arguments** that makes them directly useful for many applications such as *coreference resolution*, *question answering* and ***information extraction***. Constituent-based approaches to parsing provide similar information, but it often has to be *distilled* from the trees via techniques such as the **head-finding rules**.

Figure 2 shows the difference between a dependency parse tree and a constituency parse tree for the same sentence. In constituency parse tree, only leaves represent the words, and the internal

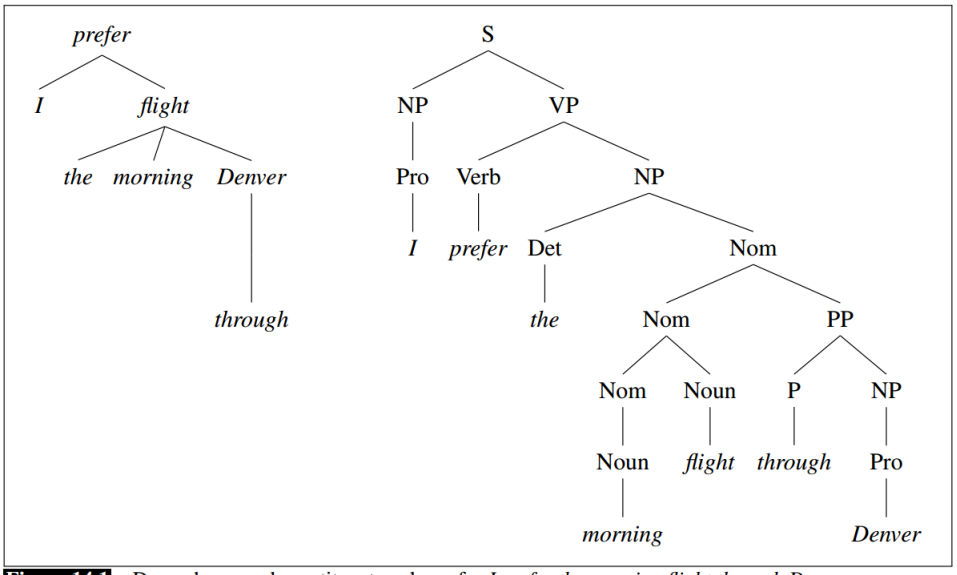

Figure 14.1 Dependency and constituent analyses for *I prefer the morning flight through Denver.*

Figure 2: A comparison between the dependency parse tree and constituency parse tree.

non-terminal nodes are phase structures. In dependency parse tree, the edges (dependency relation) are directed links from one word to another word. In graph perspective, the dependency tree is homogenous graph with nodes all representing words, while the constituency parse tree is heterogenous graph with terminal nodes representing words and non-terminal nodes representing abstract phrases.

2 Dependent relations

The traditional linguistic notion of **grammatical relation** provides the basis for the binary relations that comprise these dependency structures. The arguments to these relations consist of a **head** and a **dependent**.

- the **head** word of a constituent was the central organizing word of a larger constituent (e.g, the primary noun in a noun phrase, or verb in a verb phrase).
- The remaining words in the constituent are either direct, or indirect, **dependents** of their head.

In dependency-based approaches, the head-dependent relationship is made explicit by directly **linking heads** to the words that are **immediately dependent** on them, bypassing the need for constituent structures. It also classify different kinds of grammatical relations, or **grammatical function** that the dependent plays with respect to its head. These include familiar notions such as *subject*, *direct object* and *indirect object*. Note that in English, hese notions strongly correlate with, but by no means determine, both position in a sentence and constituent type and are therefore somewhat *redundant* with the kind of information found in phrase-structure trees.

The **Universal Dependencies (UD)** project [Nivre et al., 2016] provides an inventory of dependency relations that are linguistically motivated, computationally useful, and cross-linguistically

Clausal Argument Relations	Description
NSUBJ	Nominal subject
DOBJ	Direct object
IOBJ	Indirect object
CCOMP	Clausal complement
XCOMP	Open clausal complement
Nominal Modifier Relations	**Description**
NMOD	Nominal modifier
AMOD	Adjectival modifier
NUMMOD	Numeric modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions, postpositions and other case markers
Other Notable Relations	**Description**
CONJ	Conjunct
CC	Coordinating conjunction

Figure 14.2 Some of the Universal Dependency relations (de Marneffe et al., 2014).

Figure 3: Some depedent relation tags.

applicable. Figure 3 provides a subset of the UD relations. The core set of frequently used relations can be broken into two sets:

- **clausal relations** that *describe syntactic roles* with respect to a *predicate (often a verb)*; and
- **modifier relations** that categorize the ways that words can *modify* their heads.

2.1 Dependency Formalisms

A dependency structure can be represented as a directed graph $G = (V, A)$, consisting of a set of vertices V, and a set of ordered pairs of vertices A, which well call *arcs*. V is the set of words in a sentence and A captures the headdependent and grammatical function relationships between the elements in V.

A **dependency tree** is a *directed* graph that satisfies the following constraints:

1. There is a *single* designated **root** node that has no incoming arcs. (**single head**)
2. With the exception of the root node, each vertex has **exactly one incoming arc**. (**connected**)
3. There is a **unique path** from the root node to each vertex in V.

The notion of **projectivity** imposes an additional constraint that is derived from the order of the words in the input. An arc from a head to a dependent is said to be **projective** if there is a path from the head to every word that lies between the head and the dependent in the sentence. **A dependency tree** is then said to be **projective** if **all** the arcs that make it up are projective.

As we can see from the diagram Figure 1, projectivity (and non-projectivity) can be detected in the way weve been drawing our trees. A dependency tree is *projective* if it can be drawn with **no crossing edges**. The dependency tree generated from constituent tree in the TreeBank using *head-finding rule* is projective. Also some of commonly used algorithms can only generate projective dependency parse tree. The translation process from constituent to dependency structures has two *subtasks*: identifying all the ***head-dependent relations*** in the structure and identifying the **correct dependency relations** for these relations. The head-finding rules are used for the first

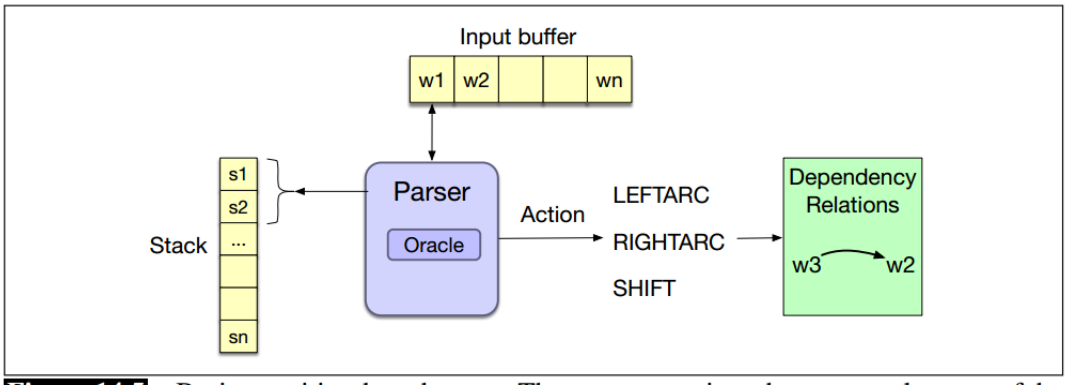

Figure 14.5 Basic transition-based parser. The parser examines the top two elements of the stack and selects an action by consulting an oracle that examines the current configuration.

Figure 4: The architecture of shift-reduce parser.

task. Heres a simple and effective algorithm

1. Mark the head child of each node in a phrase structure, using the appropriate head rules.
2. In the dependency structure, make the *head* of each *non-head child* depend on the *head of the head-child*.

3 Transition-Based Dependency Parsing

Our first approach to dependency parsing is called **transition-based parsing**. This architecture draws on ***shift-reduce parsing***, a paradigm originally developed for analyzing programming languages. In transition-based parsing well have a **stack** on which we build the parse, a **buffer** of tokens to be parsed, and a **parser** which takes actions on the parse via a predictor called an **oracle**. Figure 4 illustrates the architecture of the shif-reduce parser.

The parser walks through the sentence left-to-right, successively *shifting* items from the buffer onto the stack. At each time point we examine the **top two elements** on the stack, and the oracle makes a decision about what **transition** to apply to build the parse. The possible transitions correspond to the *intuitive actions* one might take in creating a dependency tree by examining the words in *a single pass* over the input from left to right:

- Assign the current word as the *head* of some *previously seen* word;
- Assign some previously seen word as the *head* of the *current* word;
- *Postpone* dealing with the current word, *storing* it for later processing

Well formalize this intuition with the following three transition operators that will operate on the top two elements of the stack:

- **LEFTARC**: **Assert** a head-dependent relation between the word at the ***top*** of the **stack** and the *second* word; ***remove*** the **second** word from the stack.
- **RIGHTARC**: **Assert** a head-dependent relation between the **second** word on the **stack** and the word at the *top*; ***remove*** the **top** word from the stack;

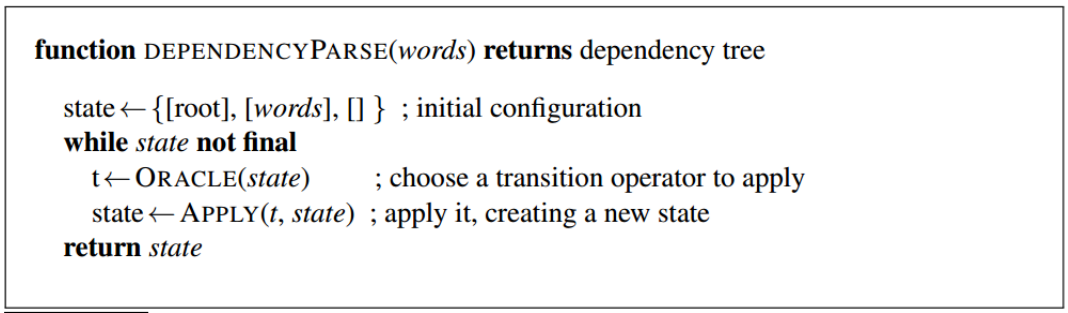

Figure 14.6 A generic transition-based dependency parser

Figure 5: The algorithm for transit-based parser.

- **SHIFT**: **Remove** the word from the front of the input buffer and **push** it onto the stack.

Well sometimes call operations like LEFTARC and RIGHTARC **reduce operations**, based on a metaphor from shift-reduce parsing, in which reducing means combining elements on the stack. There are some *preconditions* for using operators. The LEFTARC operator cannot be applied when **ROOT** is the *second* element of the stack (since by definition the ROOT node cannot have any incoming arcs). And both the LEFTARC and RIGHTARC operators require **two elements to be on the stack** to be applied.

This particular set of operators implements what is known as the **arc standard approach** to transition-based parsing. In arc standard parsing the transition operators only assert relations between elements at the *top of the stack*, and once an element has been assigned its *head* it is *removed* from the stack and is not available for further processing.

The specification of a transition-based parser is quite simple, based on representing the current **state** of the parse as a **configuration**: the ***stack***, an ***input buffer*** of words or tokens, and a set of ***relations*** representing a dependency tree. *Parsing* means making a **sequence of transitions through the space of possible configurations**. We start with an **initial configuration** in which the stack contains the *ROOT* node, the buffer has the tokens in the sentence, and an *empty set of relations* represents the parse. In the **final goal state**, the *stack* and the *word list* should be ***empty***, and the set of relations will represent the **final parse**. Figure 5 provides the algorithm description. The process **ends** when all the words in the sentence have been consumed and **the ROOT node is the only element remaining on the stack**.

The efficiency of transition-based parsers should be apparent from the algorithm. The complexity is **linear** in the length of the sentence since it is based on a single left to right pass through the words in the sentence. (Each word must first be shifted onto the stack and then later reduced.) Unlike the dynamic programming algorithm for constituency parsing, the transition-based dependency parser is **greedy algorithm**. The oracle provides a *single choice* at each step and the parser proceeds with that choice, no other options are explored, no backtracking is employed, and a single parse is returned in the end.

There are several important things to note when examining sequences during the parsing. First, the sequence given is **not the only one** that might lead to a reasonable parse. In general, there may be **more than one path** that leads to the same result, and due to *ambiguity*, there may be other transition sequences that lead to different equally valid parses. Second, we are assuming that the **oracle** always provides the correct operator at each point in the parsean assumption that is

unlikely to be true in practice. As a result, given the greedy nature of this algorithm, incorrect choices will lead to incorrect parses since the parser has no opportunity to go back and pursue alternative choices.

3.1 Creating an Oracle

The oracle for greedily selecting the appropriate transition is trained by supervised machine learning. The oracle from the algorithm in Figure 5 takes as input a *configuration* and returns a *transition operator*. Therefore, to train a classifier, we will need configurations paired with transition operators (i.e., *LEFTARC*, *RIGHTARC*, or *SHIFT*). Unfortunately, treebanks pair entire sentences with their corresponding trees, not configurations with transitions.

To generate the required training data, we supply our oracle with the **training sentences** to be parsed *along with* their **corresponding reference parses** from the treebank. To produce training instances, we then **simulate** the operation of the parser by running the algorithm and relying on a new **training oracle** to give us correct transition operators for each successive configuration.

To be more precise, define V as a set of vertices in the dependency tree (words), R_p as a set of dependency relations. The **reference parse** consists of (V, R_p). The current **configuration** is defined as (S, R_c) where S represents the stack and R_c represents a set of current dependency relations. Given a **reference parse** (V, R_p) and a **configuration** (S, R_c), the training oracle proceeds as follows:

- Choose ***LEFTARC*** if it produces a *correct head-dependent relation* given the reference parse and the current configuration,

$$S_1 \;\; r \;\; S_2 \in R_p$$

  where $S_1, S_2 \in S$.

- Otherwise, choose ***RIGHTARC*** if (1) it produces a correct head-dependent relation given the reference parse **and** (2) all of the *dependents* of the word at the **top** of the stack *have already been assigned*,

$$S_2 \;\; r \;\; S_1 \in R_p$$
$$\text{and } \forall r', w, \quad \text{if } S_1 \;\; r' \;\; w \in R_p, \quad \text{then } S_1 \;\; r' \;\; w \in R_c$$

  where $S_1, S_2 \in S$.

- Otherwise, choose ***SHIFT***.

The restriction on RIGHTARC is to make sure that a word is not popped from the stack, and thus lost to further processing, *before* all its dependents have been assigned to it.

3.2 Feature-based classifier

Featured-based classifiers generally use the same features weve seen with part-of-speech tagging and partial parsing: *Word forms*, *lemmas*, *parts of speech*, the *head*, and the *dependency relation to the head*. The features are extracted from the *training configurations*, which consist of the *stack*, the *buffer* and the *current set of relations*. We can use the **feature template** that we introduced

Step	Stack	Word List	Predicted Action
0	[root]	[book, the, flight, through, houston]	SHIFT
1	[root, book]	[the, flight, through, houston]	SHIFT
2	[root, book, the]	[flight, through, houston]	SHIFT
3	[root, book, the, flight]	[through, houston]	LEFTARC
4	[root, book, flight]	[through, houston]	SHIFT
5	[root, book, flight, through]	[houston]	SHIFT
6	[root, book, flight, through, houston]	[]	LEFTARC
7	[root, book, flight, houston]	[]	RIGHTARC
8	[root, book, flight]	[]	RIGHTARC
9	[root, book]	[]	RIGHTARC
10	[root]	[]	Done

Figure 14.8 Generating training items consisting of configuration/predicted action pairs by simulating a parse with a given reference parse.

Figure 6: The training samples are generated from reference parse.

for sentiment analysis and part-of-speech tagging. Feature templates allow us to automatically *generate* large numbers of specific features from a training set.

$$< s_1.w, op >, < s_2.w, op >< s_1.t, op >, < s_2.t, op >< b_1.w, op >, < b_1.t, op >< s_1.wt, op >$$

Here features are denoted as location.property, where s = stack, b = the word buffer, w = word forms, l = lemmas, t = part-of-speech, and op = operator. $s_1.w$ is the word form at top of stack. $b_1.w$ is the word form at front of buffer. $b_1.t$ is the POS tag for the word at front of buffer.

3.3 Neural classifier

The oracle can also be implemented by a neural classifier. A standard architecture is simply to pass the sentence through an encoder, then take the presentation of the top 2 words on the stack and the first word of the buffer, concatenate them, and present to a feedforward network that predicts the transition to take. Figure 7 shows the architecture for the neural classifier.

3.4 Advanced Methods in Transition-Based Parsing

- The arc-standard transition system described above is only one of many possible systems. A frequently used alternative is the **arc eager transition system**. The arc eager approach gets its name from its ability to *assert rightward relations much sooner* than in the arc standard approach.
- The computational efficiency of the transition-based approach discussed earlier derives from the fact that it makes a single pass through the sentence, greedily making decisions without considering alternatives. Of course, this is also a weakness *once a decision has been made it can not be undone*, even in the face of overwhelming evidence arriving later in a sentence. We can use **beam search** to explore alternative decision sequences.

  Beam search uses a **breadth-first search strategy** with a ***heuristic filter*** that prunes the *search frontier* to stay within a fixed-size **beam width**.

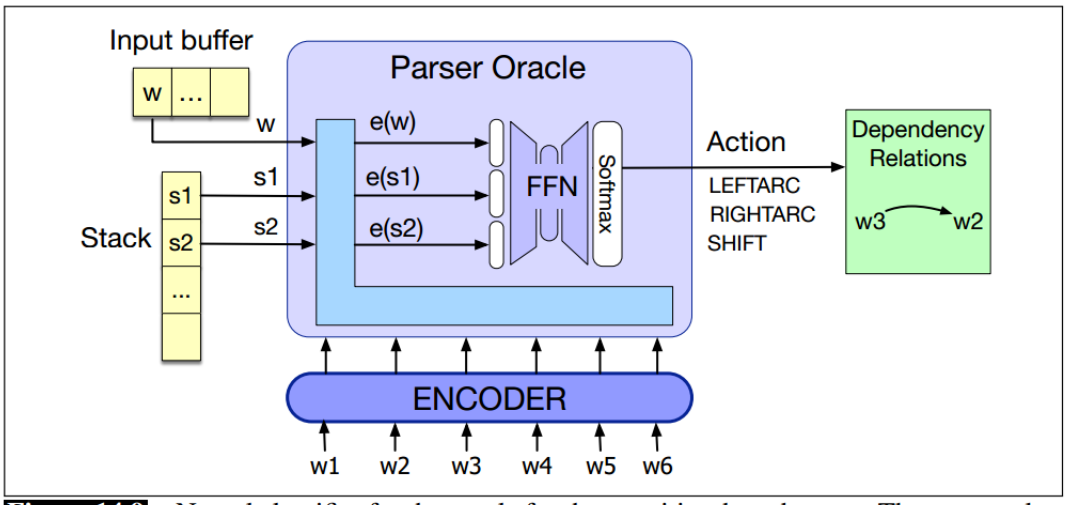

Figure 14.9 Neural classifier for the oracle for the transition-based parser. The parser takes the top 2 words on the stack and the first word of the buffer, represents them by their encodings (from running the whole sentence through the encoder), concatenates the embeddings and passing through a softmax to choose a parser action (transition).

Figure 7: The neural network as oracle in transit-based dependency parsing.

- Instead of choosing the single best transition operator at each iteration, well apply **all applicable operators** to each state on an agenda and then **score** the resulting configurations.
- We then **add each** of these new configurations to the frontier, subject to the constraint that there has to be room within the beam. As long as the size of the agenda is within the specified beam width, we can add new configurations to the agenda.
- Once the agenda reaches the limit, we only add new configurations that are **better** than *the worst configuration* on the agenda (**removing** the worst element so that we stay within the limit).
- Finally, to insure that we retrieve the best possible state on the agenda, the while loop **continues** *as long as there are non-final states* on the agenda

The beam search approach requires a more elaborate notion of scoring than we used with the greedy algorithm. The oracle's choice can be viewed as assigning a score to all the possible transitions and picking the best one. Let Score(t, c) is the score assigned by Oracle based on features from current configuration c and operator t. In order to track the score for the entire history, we can define the score for a new configuration as the score of its predecessor plus the score of the operator used to produce it.

$$\text{ConfigScore}(c_0) = 0.0$$
$$\text{ConfigScore}(c_i) = \text{ConfigScore}(c_{i-1}) + \text{Score}(t_i, c_{i-1})$$

Figure 8 describes the algorithm for beam search on dependency parsing.

```
function DEPENDENCYBEAMPARSE(words, width) returns dependency tree

  state ← {[root], [words], [], 0.0}     ;initial configuration
  agenda ← ⟨state⟩                       ;initial agenda

  while agenda contains non-final states
    newagenda ← ⟨⟩
    for each state ∈ agenda do
      for all {t | t ∈ VALIDOPERATORS(state)} do
        child ← APPLY(t, state)
        newagenda ← ADDTOBEAM(child, newagenda, width)
    agenda ← newagenda
  return BESTOF(agenda)

function ADDTOBEAM(state, agenda, width) returns updated agenda

  if LENGTH(agenda) < width then
    agenda ← INSERT(state, agenda)
  else if SCORE(state) > SCORE(WORSTOF(agenda))
    agenda ← REMOVE(WORSTOF(agenda))
    agenda ← INSERT(state, agenda)
  return agenda
```

Figure 14.11 Beam search applied to transition-based dependency parsing.

Figure 8: The algorithm for beam search on dependency parsing.

4 Graph-Based Dependency Parsing

- **Graph-based parsers** are *more* ***accurate*** than transition-based parsers, especially on long sentences;
- Transition-based methods have trouble when the heads are **very far from the dependents**. Graph-based methods **avoid this difficulty by scoring entire trees**, rather than relying on greedy local decisions.
- Furthermore, unlike transition-based approaches, graph-based parsers can produce **non-projective trees**. Although projectivity is not a significant issue for English, it is definitely a problem for many of the worlds languages.

Graph-based dependency parsers search through the space of possible trees for a given sentence for a tree (or trees) that **maximize some score**. These methods encode the search space as directed graphs and employ methods drawn from *graph theory* to *search* the space for optimal solutions.

More formally, given a sentence S were looking for the best dependency tree in $\mathcal{G}(S)$, the space of all possible trees for that sentence, that maximizes some score:

$$\hat{T}(S) = \text{argmax}_{t \in \mathcal{G}(S)} \text{Score}(t, S).$$

This score function is **edge-factored**, i.e. it can be decomposed into sum of scores on edges.

$$\text{Score}(t, S) = \sum_{e \in t} \text{Score}(e, S)$$

Graph-based algorithms have to solve two problems:

1. assigning a score to each edge,
2. finding the best parse tree given the scores of all potential edges.

4.1 Parsing via finding the maximum spanning tree

We **initialize** the graph G as a fully connected, directed, weighted graph where the vertices are the input words and the directed edges represent *all possible head-dependent assignments*. We also add a node $ROOT$ with outgoing edges directed at all of the other vertices. The edge weight reflects the score for each possible head-dependent relation assigned by some algorithm.

The optimal dependent tree can be found via **maximum spanning tree** over G. A spanning tree over a graph G is a subset of G that is a tree and covers all the vertices in G; a spanning tree over G that *starts* from the ROOT is a *valid parse* of S. A maximum spanning tree is the spanning tree with the **highest score**. Thus a maximum spanning tree of G emanating from the ROOT is the optimal dependency parse for the sentence.

Some intuitions on the spanning tree and directed graph:

- Every vertex in the spanning tree has **exactly one** incoming edge. This is because every connected components of a spanning tree will only have one incoming edge.

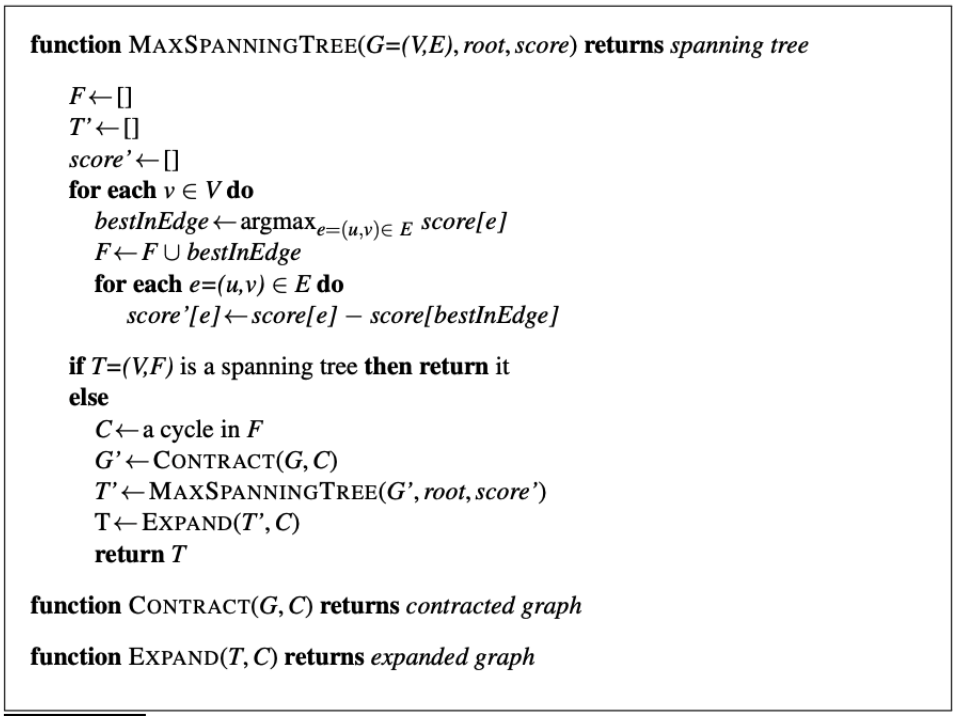

Figure 14.13 The Chu-Liu Edmonds algorithm for finding a maximum spanning tree in a weighted directed graph.

Figure 9: The algorithm for finding the maximum spanning tree on dependency parsing.

- The absolute values of the edge scores are not critical in determining the maximum spanning tree. It is the relative weights of the edges entering each vertex matters. Therefore, the tree structure will be maintained if all edges entering a vertex have subtracted a constant.

The algorithm for searching the maximum spanning tree is described in Figure 9 [Chu, 1965] and [Edmonds, 1968].

1. **The greedy edge selection phase**: The first step is to choose the incoming edge with the highest score. (i.e. greedy search finding the best head assignment for given dependent.) If the resulting set of edges are spanning tree, then we have obtained the maximum spanning tree.

2. **The weight adjustment/re-scoring phase**: If the resulting edges contain circles, we need to eliminate some of these edges to break the circle. The cleanup phase begins by adjusting all the weights in the graph by **subtracting the score of the maximum edge entering each vertex** from the *score of all the edges entering that vertex*. For the weight of edges in the circle, they have no bearing on the weight of *any of the possible spanning trees*, since it will result in a weight of zero for all edges selected during the greedy selection phase, including all edges in the circle.

3. **The node collapsing phase**: Having adjusted the weights, the algorithm creates a new graph by selecting a cycle and **collapsing** it into a single new node. Edges within the circle will be dropped and only edges connect to the circle remains. The other edges also are unchanged. This results in a new graph.

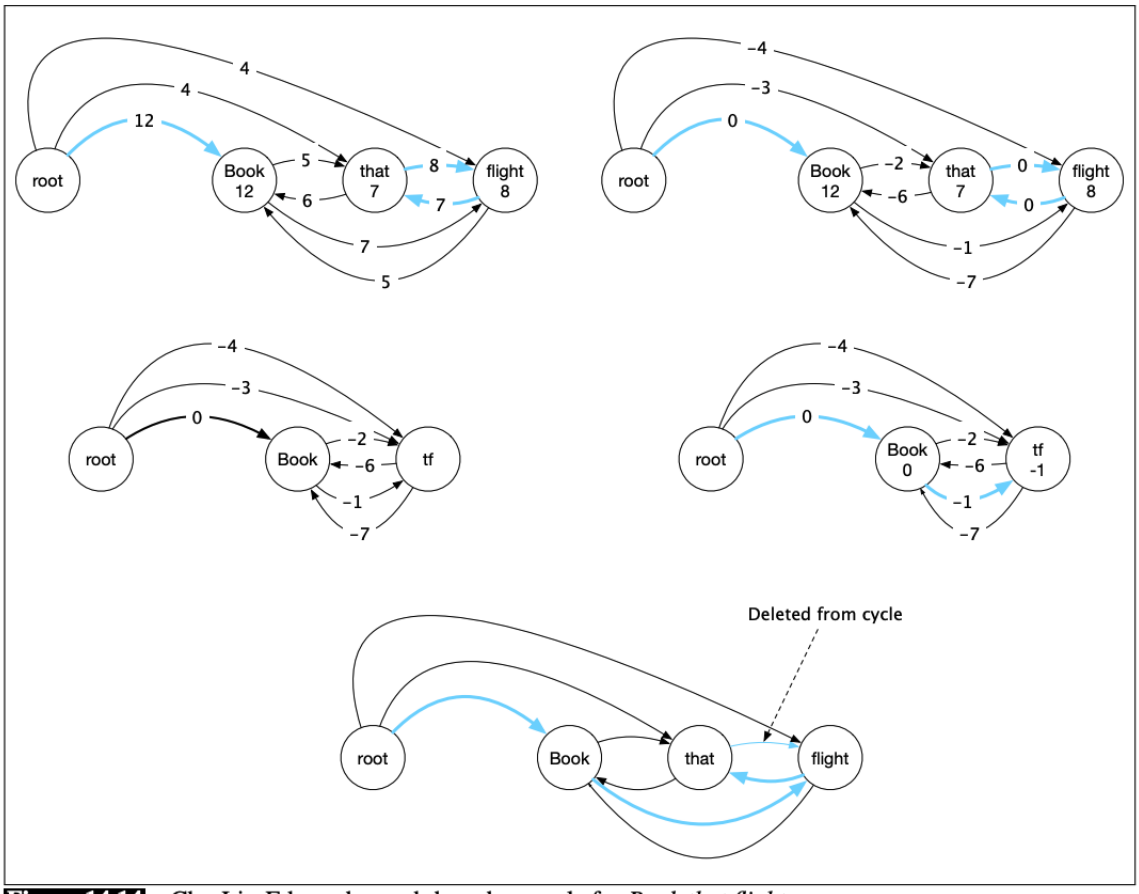

Figure 14.14 Chu-Liu-Edmonds graph-based example for *Book that flight*

Figure 10: The example for maximum spanning tree search. The blue edges are selected using greedy search for each vertex and the score are re-normalized based on the selected edge score.

4. **The recursive cleanup phase**: Now we recursively find the maximum spanning tree on the new graph. The edge of the maximum spanning tree directed towards the vertex representing the collapsed cycle tells us which edge to delete to eliminate the cycle. When each recursion completes we **expand the collapsed vertex**, restoring all the vertices and edges from the cycle with the exception of the single edge to be deleted.

We can see the example running this algorithm in Figure 10. The **time complexity** of the **Chu-Liu Edmonds (CLE) algorithm** ([Chu, 1965] and [Edmonds, 1968]) is $\mathcal{O}(mn)$ where m is number of edges and n is the number of nodes. Since we begins with fully connected graph $m = n^2$, the total time complexity is $\mathcal{O}(n^3)$. A more efficient algorithm from [Gabow et al., 1986] could reach to $\mathcal{O}(m + n \log n)$.

4.2 A feature-based algorithm for assigning scores

Note that the score for the tree is the sum of scores of the edges within the tree.

$$\text{Score}(t, S) = \sum_{e \in t} \text{Score}(e, S)$$

In a feature-based algorithm we compute the edge score as a weighted sum of features extracted from it:

$$\text{Score}(e, S) = \sum_{j=1}^{N} w_j f_j(e, S)$$

where $[f_j]_{1,\ldots,N}$ are a set of features based on

- *Word forms*, *lemmas*, *parts of speech* of head, its dependent,
- *Word embeddings*.
- the Corresponding features from the *contexts* before, after and between the words.,
- the *dependency relation to the head*.
- The *direction of the relation* (to the right or left)
- The *distance* from the head to the dependent

They are the same features used to define the training data for transit-based parser in Section 3.2. These features can be pre-selected or generated using feature templates.

To learn a set of weights for these features, we train a model that assigns higher scores to correct trees than to incorrect ones. (unlike in Section 3.1 where we need the class labels as well as parse actions in training set.) An effective framework for problems like this is to use **inference-based learning** combined with the perceptron learning rule. In this setting, we initialize the weight randomly and use it to parse the training sentences. If the resulting parse matches the corresponding tree in the training data, we do nothing to the weights. Otherwise, we find those features in the **incorrect parse** that are *not present in the reference parse* and we **lower their weights** by a small amount based on the learning rate. (**online learning**) We do this incrementally for each sentence in our training data until the weights converge.

4.3 Using neural network for assigning scores

The state-of-the-art algorithm uses the neural network to assign scores. Instead of extracting hand-designed features to represent each edge between words w_i and w_j, these parsers run the sentence through an *encoder*, and then pass the encoded representation of the two words w_i and w_j through a network that estimates a score for the edge $i \rightarrow j$. Figure 11 shows one type of architectures.

In the network, there are two feed-forward networks for each word: one to produce representation as a head, the other one to produce representation as a dependent. When we assign score from $i \rightarrow j$, we use the head representation for w_i and the dependent representation for w_j and passing them through an *biaffine function* $\text{Biaff}(\boldsymbol{x}, \boldsymbol{y}) := \boldsymbol{x}^T \boldsymbol{U} \boldsymbol{y} + \boldsymbol{W}(\boldsymbol{x} \oplus \boldsymbol{y}) + b$. The idea of using a **biaffine scoring function** is allow the system to learn multiplicative interactions between the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. Finally we can pass the score function to a softmax to obtain probabilities. $p(i \rightarrow j) = \text{softmax}([\text{Score}(k \rightarrow j); , \forall k \neq j])$. This probability can then be passed to the maximum spanning tree algorithm.

Note that the algorithm as weve described it is *unlabeled*. To make this into a *labeled* algorithm, [Dozat and Manning, 2016] proposes algorithm actually trains two classifiers. The first classifier, **the edge-scorer**, the one we described above, assigns a probability $p(i \rightarrow j)$ to each word w_i and

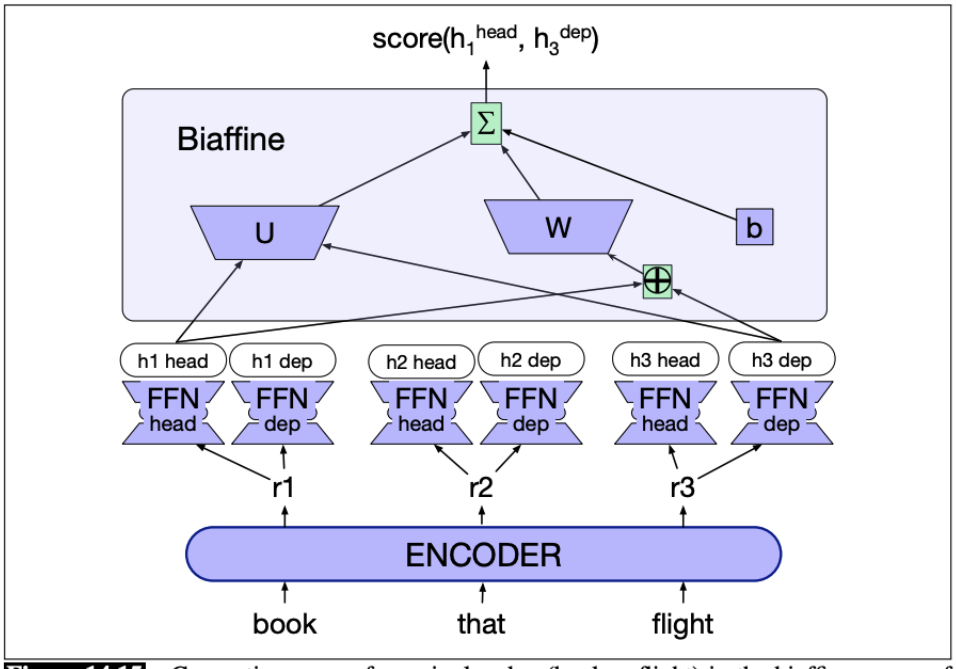

Figure 14.15 Computing scores for a single edge (book→ flight) in the biaffine parser of Dozat and Manning (2017); Dozat et al. (2017). The parser uses distinct feedforward networks to turn the encoder output for each word into a head and dependent representation for the word. The biaffine function turns the head embedding of the head and the dependent embedding of the dependent into a score for the dependency edge.

Figure 11: The neural network based score assignment. The neural network uses encoder to learn weight between words based on the the word embedding of two vertices.

w_j. Then the Maximum Spanning Tree algorithm is run to get a single best dependency parse tree for the second. We then apply a second classifier, the **label-scorer**, whose job is to find the maximum probability label for each edge in this parse. This second classifier has the same form as (Figure 11), but instead of being trained to predict with binary softmax the probability of an edge *existing* between two words, it is trained with a softmax over dependency labels to **predict the dependency label** between the words.

References

Yoeng-Jin Chu. On the shortest arborescence of a directed graph. *Scientia Sinica*, 14:1396–1400, 1965.

Timothy Dozat and Christopher D Manning. Deep biaffine attention for neural dependency parsing. *arXiv preprint arXiv:1611.01734*, 2016.

Jack Edmonds. Optimum branchings. *Mathematics and the Decision Sciences, Part*, 1(335-345): 25, 1968.

Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

Joakim Nivre, Marie-Catherine De Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, et al. Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 1659–1666, 2016.