

# Dependency Parsing: Parsing with the Eisner Algorithm

Please store your personal files created during this session so that you can continue working with them next week.

## 1 Derivation Trees

Correctness of the Eisner's parser (defined during the lecture) is implied by the inference rules. Namely, it can be shown that assertions of the chart items are preserved by the rules. Thus, by induction, if item  $(0, n, L)$  is derived – where  $n$  is the length of the sentence – we can be sure that a dependency tree spanning the entire sentence, rooted in the artificial node on position 0, exists.

However, this does not tell us much about the procedure of extracting the best-weight dependency trees from the resulting chart. Nor does it explain if the algorithm is *complete* – i.e., if *all* dependency trees can be actually derived using the rules.

Completeness relies on the property that any projective dependency tree can be represented as a *derivation tree*, in which:

- Nodes correspond to chart items
- The root node corresponds to item  $(0, n, L)$
- The individual nodes and their children nodes correspond to applications of the inference rules

Conversely, any derivation tree corresponds to a particular dependency tree. In this tutorial, we start by looking at the conversion functions which allow to translate between the two representations.

Download the source code accompanying the today's session. It should contain the following source files:

- `core.py` – core data types (trees, chart items)
- `deriv.py` – conversion between dependency trees and derivation trees
- `eisner.py` – parsing with the Eisner algorithm
- `conllu.py` – parsing CoNLL-U files
- `main.py` – parsing with features

as well as the `en_partut-ud-test.conllu` testing file, which you will use at the end of this tutorial to test your parsing model with real UD sentences in English.

Run a `python3` interpreter session in the directory with all the source files. You can also create a separate file with your private definitions and then use it to test your functions. Let's first have look at the simple dependency tree corresponding to the sentence *John eats pizza*.

```

from core import Node, Leaf, print_tree

pizza_tree = Node(0, [Node(2, [Leaf(1), Leaf(3)])])
print_tree(pizza_tree)

```

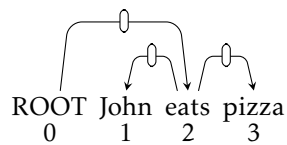
This should give the following output:

```

0
 2
  1
  3

```

In this tree, each word is represented by its position – 0 corresponds to the artificial root node, 2 to *eats*, 1 to *John*, and 3 to *pizza*. Node 0 is the head of 2, 2 is the head of 1 and 3. This dependency tree can be also graphically represented as:



Note that this tree ignores the labels assigned to the arcs (see Ex. 3). To encode this dependency tree as a derivation tree, use the following piece of code:

```

import deriv
pizza_deriv = deriv.encode(pizza_tree)
print_tree(pizza_deriv)

```

This should give:

```

Item(beg=0, end=3, typ=<Typ.L: 1>)
  Item(beg=0, end=2, typ=<Typ.B: 3>)
    Item(beg=0, end=0, typ=<Typ.L: 1>)
    Item(beg=1, end=2, typ=<Typ.R: 2>)
      Item(beg=1, end=1, typ=<Typ.R: 2>)
      Item(beg=1, end=2, typ=<Typ.B: 3>)
        Item(beg=1, end=1, typ=<Typ.L: 1>)
        Item(beg=2, end=2, typ=<Typ.R: 2>)
      Item(beg=2, end=3, typ=<Typ.L: 1>)
      Item(beg=2, end=3, typ=<Typ.B: 3>)
        Item(beg=2, end=2, typ=<Typ.L: 1>)
        Item(beg=3, end=3, typ=<Typ.R: 2>)
      Item(beg=3, end=3, typ=<Typ.L: 1>)

```

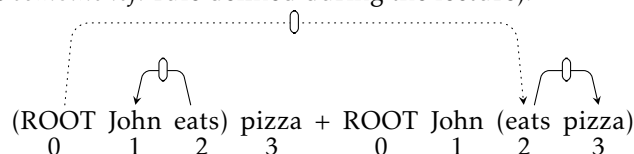
Note that this derivation tree describes the inferences steps needed to construct the above dependency tree. For instances, the top-level rule application:

```

Item(beg=0, end=3, typ=<Typ.L: 1>)
  Item(beg=0, end=2, typ=<Typ.B: 3>)
  ...
  Item(beg=2, end=3, typ=<Typ.L: 1>)
  ...

```

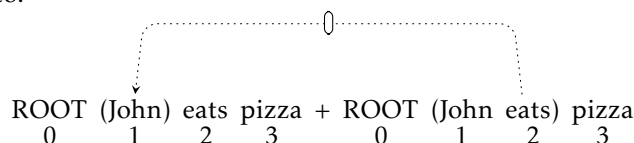
corresponds to the following operation on dependency trees, with a new arc between nodes 0 and 2 being added (cf. the *combine left* rule defined during the lecture).



Similarly, the following rule application:

```
Item(beg=1, end=2, typ=<Typ.R: 2>)
Item(beg=1, end=1, typ=<Typ.R: 2>)
Item(beg=1, end=2, typ=<Typ.B: 3>)
...
```

corresponds to:



Have a look at the definition of the encode function to see how the encoding procedure is implemented and how does it relate to the inference rules defined during the lecture. The code was designed with clarity in mind, so it is not necessarily the most efficient implementation.<sup>1</sup>

**Exercise 1.** Define a dependency tree for another, slightly more complex sentence (e.g. *The quick brown fox jumps over the lazy dog*). Try to manually determine its derivation tree and verify that the encode function gives the same result.

## 2 Eisner Parser

The Eisner parser is implemented in the `eisner.py` module by the `parse` function. The implementation provides a relatively direct translation of the inference rules. For instance, the *combine left* and the *init left* rules are implemented by the embedded `left` function (see the code for more details):

```
# We use a standard memoization trick (hence the @memoize decoration)
# to compute the result of left(i, j) for a given (i, j) pair at most once
# (see the code for more details).
@memoize
def left(i, j):
    # If i == j, the set of premises (items from which item (i, j, L) is
    # derived) is empty and the weight is 0 (neutral element). This
    # corresponds to the "init left" inference rule.
    if i == j:
        return ([], 0)
    # best[0] -- optimal set of premise items
    # best[1] -- the resulting optimal weight
    best = [], float("-inf")
    # k is such that i+1 <= k <= j
    for k in range(i+1, j+1):
```

<sup>1</sup>Nevertheless, asymptotically it should be  $\mathcal{O}(n^3)$ .

```

    # the set of premise items for the given choice of k
    tail = [Item(i, k, B), Item(k, j, L)]
    # the resulting weight is the total weight of the premise items
    # (basically the sum of their optimal weights) and the weight of
    # the newly created arc
    weight = total_weight(tail) + arc_weight(i, k)
    if weight > best[1]:
        best = tail, weight
return best

```

Similarly, the embedded function `right` implements the rules *init right* and *combine right*, while both implements *combine both*.

The `left` function computes not only the best weight of a left-rooted dependency tree spanning  $(i, j)$ , but also the optimal set of premise items from which  $(i, j, L)$  can be derived. This allows to later extract the optimal derivation tree corresponding to the entire sentence.

Note that the weights are summed up rather than multiplied – we assume that the weight of a dependency tree is a sum of the weights of its arcs.

To parse a sentence of length 3, run the follow code:

```

import eisner
def arc_weight(i, j):
    return 0
best_deriv = eisner.parse(arc_weight, sent_len=3)
print_tree(deriv.decode(best_deriv))

```

Note that in the above code listing we set the weight of each arc to 0, hence the resulting tree is not particularly relevant. Note also that the parser is independent from the exact form of the input sentence – only the length of the sentence is passed as argument. In practice, however, the weights of the arcs depend on the words they link and the corresponding model features.

**Exercise 2.** Consider the sentence *The quick brown fox jumps over the lazy dog*.

- Define the `arc_weight` function in such a way that the dependency tree provided by the parser is consistent with the Universal Dependencies-based analysis.
- Is it possible to achieve the previous point in such a way that all the values of the `arc_weight` function are non-positive?

**Exercise 3.** The implemented parser does not handle arc labels at all. What parts of the parser would have to be modified in order to account for the labels? (Note that the rules defined during the lecture do take labels into account.)

### 3 Features

In practice, the weights of the arcs depend on the words they link and the corresponding model features. Have a look the module `main.py`, which shows how model features can be defined. For instance, the following code:

```

# Word form of the head + word form of the dependent
Form2 = namedtuple('Form2', ['head_form', 'dep_form'])

```

tells that any pair (word form of the head, word form of the dependent) constitutes a potential feature in our model. Another piece of code:

```
# Part-of-speech of the head + part-of-speech of the dependent
Pos2 = namedtuple('Pos2', ['head_pos', 'dep_pos'])
```

tells that any pair (POS of the head, POS of the dependent) also constitutes a potential feature in our model. The features relevant for a particular arc are then retrieved using the `features` function, while the features (and their counts) occurring in a particular dependency can be retrieved using the `tree_features` function.

### 3.1 Model

A particular model, which determines the weights of the individual arcs and thus allows to parse a given tokenized sentence, can be simply understood as a function which assigns weights to features. It can be defined as in the following code listing:

```
import main
from main import Pos1, Pos2
def model(feat):
    # dictionary which manually assigns weights to selected features
    feat_dict = {
        Pos1("VERB"): 1,
        Pos2("VERB", "NOUN"): 1,
        Pos2("NOUN", "DET"): 1,
        Pos2("DET", "NOUN"): -1
    }
    # all features not present in the map get the value '0' by default
    return feat_dict.get(feat, 0)
```

This function basically states that:

- Verbs are likely candidates for tree roots
- Nouns are often dependents of verbs
- Determiners are likely to be dependents of nouns, but
- Not the other way around

You can now load some UD sentences and try to parse them using our model:

```
import core
import conllu
conllu_sents = conllu.parse(open("en_partut-ud-test.conllu").read())
sents = [list(map(conllu.create_token, sent)) for sent in conllu_sents]
out_tree = main.parse(model, sents[3])
core.print_tree(out_tree)
```

You can now repeat Exercise 2: focus on a particular UD sentence and try to define a model which will maximize the weight of the corresponding UD dependency tree.

Manually assigning the weights to the individual model features is be close to impossible in practice, since a model can rely on millions of features. We will look at how these weights can be estimated automatically from treebanks during subsequent sessions.