Graph Matching and Graph Rewriting: GREW tools for corpus exploration, maintenance and conversion

Bruno Guillaume
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
Bruno.Guillaume@inria.fr

Abstract

This article presents a set of tools built around the Graph Rewriting computational framework which can be used to compute complex rule-based transformations on linguistic structures. Application of the graph matching mechanism for corpus exploration, error mining or quantitative typology are also given.

1 Introduction

The motivation of GREW is to have an effective tool to design rule-based transformations of linguistic structures. When designing GREW, our goal was to be able to manipulate at least syntactic and semantic representations of natural language (one of the first application of GREW was the modeling of a syntax-semantics interface). In a naive view, we can say that syntactic structures are trees and semantic ones are graphs. Then, if we want to work with both kinds of structures in a common framework, we can use the fact that a tree can be considered as a graph and hence consider that all structures are graphs.[1]

Now, if we consider all structures as graphs, how to describe rule-based transformation on these structures? In practice, these transformations can of course be computed with some programs but when it becomes complex and implies many rules, it is difficult to maintain and to debug. To deal with this, we propose to use the graph rewriting formalism to describe these transformations.

Graph rewriting is a well-defined mathematical formalism and we know that any computable transformation can be expressed by a graph rewriting system. In this approach, a global transformation is decomposed in a successive application of small and local transformations which are described by rules; linguistic transformations can be decomposed in a modular way in atomic steps which are easier to manage.

Several graph rewriting tools already exist but some specificities of NLP made it useful to build a system dedicated to this domain. In GREW tools, a built-in notion of feature structure is available and rules can be parametrised by lexical information. Moreover, transformations on dependency structures often requires to change head of substructures and a dedicated command ease this kind of operation (see Section 3.5).

In Section 2, we give a more precise definition of our graphs and graph rewriting framework and the next parts present examples about rewriting (Section 3) and about matching (Section 4).

2 Graphs and graph rewriting

The book (Bonfante et al., 2018) gives a complete description of the graphs and graph rewriting system used in GREW. We give here a short description on the main aspects.

In our framework, a graph is defined by a set of nodes labelled by non-recursive feature structure and a set of labelled edges (note that edges encode relations and hence, we do not consider multiple edges with the same label on the same pair of nodes). In addition to the usual graph mathematical definition of graphs, we also add a notion of order on nodes. For each graph, a sub-part of the nodes are ordered. The subset of ordered nodes can contains all the nodes (for instance in dependency structures like in Figure 1); it can be empty (for instance in semantic graphs like AMR structures shown in Section 4.2); but we can also have structures where a strict subpart is ordered, for instance with phrase structure trees where lexical nodes are ordered following the tokens order in the input sentence whereas non-lexical nodes are unordered.

[1] We may lose information if the order between the child nodes of a given node (see Section 2).

Global transformations of graphs are decomposed in small steps; each step is described as a rule. A rule encodes a local transformation and is composed in two parts: the left-hand side which expresses the conditions for the application of the rule and the right-hand part which describes the modifications to be done on the graph.

Formally, the conditions of application are described by a pattern which is itself a graph. Graph matching is used to decide if a pattern can be found in a graph. The pattern can be refined by a set of NAP (negative application patterns) which are used to filter out some occurrences given by the first pattern. The main pattern is introduced by the keyword `pattern` and NAPs are introduced with the keyword `without` (see examples in the next section).

To avoid complex mathematical definitions and to propose an operational way to modify graph, GREW describes the modifications of the graph through a sequence of atomic commands for edge deletion, edge creation, feature updating...

When the number of rules increases, it may become tricky to control the order in which they should be applied; a dedicated notion of rewriting strategies was design to let the user control these applications.

When using rewriting, confluence and termination are important aspects. These questions are discussed on examples in the next section.

3 Graph rewriting in practice

The goal of this section is to present through examples the usage of the rewriting part of GREW. Some important concepts like confluence and termination will be also discussed.

3.1 First rules

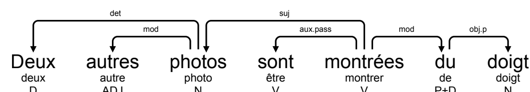
The conversion between different formats is one the common usage of GREW. We will use the example of the conversion from one dependency annotation format (used in the Sequoia project (Candito and Seddah, 2012)) to Universal Dependencies (UD) (Nivre et al., 2016). The Figure 1 shows the annotations of a French sentence in both formats.

The whole transformation is decomposed into small steps which are described by rules. When GREW is used to rewrite an input graph, a strategy describes how rules should be applied. In the first examples below, the strategy consists in just one rule.

In our conversion example, we need a rule to change the POS for adjectives: `A` is used in Sequoia and `ADJ` in UD. The GREW rule for this transformation is:

```
rule adj {
  pattern { N [upos=A] }
  commands { N.upos = ADJ }
}
```

The application of this rule on the input graph produces, as expected the graph below:



We can then imagine others similar rules for other POS tags: `P` is Sequoia becomes `ADP` in UD, `N` is Sequoia becomes `NOUN` in UD.

```
rule prep {
  pattern { N [upos=P] }
  commands { N.upos = ADP }
}
rule noun {
  pattern { N [upos=N] }
  commands { N.upos = NOUN }
}
```

But applying the rule `prep` to the input graph produces an empty set and the application of `noun` the input graph produced two different graphs (one with *photos* tagged as `NOUN`, the other with *doigt* tagged as `NOUN`)!

In fact, the result of the application of a rule on a graph is a set of graphs, one for each occurence of the pattern found in the input graph. This set is then empty if the pattern is not found (like `pattern {N [upos=P]}`) or contains two graphs if the pattern if found twice (like `pattern {N [upos=N]}`). To iterate the application of a rule, one has to use more complex strategies.

The strategy `Onf(noun)`[2] iterates the application of the strategy `noun` on the input graph. With the same input graph (of Figure 1), the application of GREW with the strategy `Onf(noun)` produces a graph where the two nouns have the new tag `NOUN`.

Note that `Onf(...)` always outputs exactly one graph. With the strategy `Onf(prep)` for instance, the rewriting process will output one graph, identical to the input graph, obtained after 0 application of the `prep` rule.

In previous examples, we considered rules separately, but in a global transformation all the previous rules must be used in the same global transformation. A solution to use several rules in the

[2] `Onf` stands for "one normal form"; it will be explained more in detail later with other strategies.

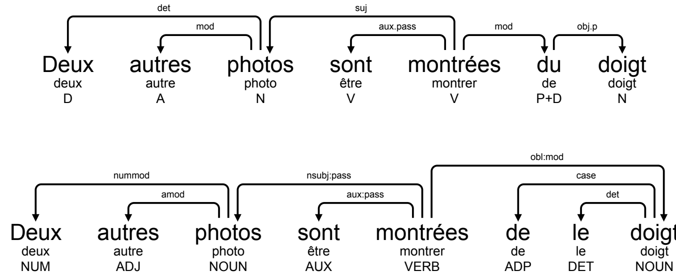


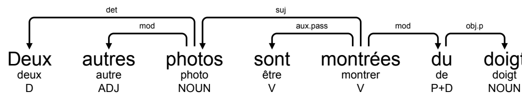
Figure 1: Annotation of the sentence *Deux autres photos sont montrées du doigt* [en: *Two other photos are pointed out*] in Sequoia (above) and in UD (below)

same rewriting process is to put them in the same `package` construction, for instance with the 3 rules above:

```
package POS {
  rule adj { ... }
  rule prep { ... }
  rule noun { ... }
}
```

The package name `POS` can be used as a strategy name for rewriting. Applying the package `POS` corresponds to the application of one of the rules of the package. With our input graph, it produces three different graphs, obtained either by the application of the rule `adj` or by the two possible applications of the rule `noun`.

In order to iterate the package, we need the strategy `Onf(POS)`. As before with `Onf`, exactly one graph is produced with three successive applications of the rules:



3.2 Termination

One key problem that may arise when using rewriting is the non-termination of the process. If we go on with the previous example about POS and consider verbs: the same tag `V` should be converted to `AUX` or to `VERB`. One way to decide that the new POS must be `AUX` is the presence of the relation `aux.pass`. We can propose the rule:

```
rule aux_1 {
  pattern { M -[aux.pass]-> N }
  commands { N.upos = AUX }
}
```

But the process of rewriting with strategy `Onf(aux_1)` is not terminating because nothing prevents the rule to be applied again and again, the pattern is still present after the application of the rule. In practice, a bound can be set on the number of rules applied[3] and an error is thrown when this bound is reached, in order to avoid non-terminating computation.

A way to solve this problem is to make the pattern stricter. With the rule below and the strategy `Onf(aux_2)`, the expected output is obtained after one application of the rule.

```
rule aux_2 {
 pattern { M -[aux.pass]-> N; N[upos=V] }
 commands { N.upos = AUX }
}
```

Of course, in a more general setting, we can have loops which imply more than one rule and which are more difficult to manage. Unfortunately, it is not possible to decide algorithmically if some rewriting system is terminating or not.

Anyway, in NLP applications like conversions from format A to format B, it is often easy to ensure termination be defining measure which stands for the fact that we are "closer" to the B format after each rule application. For instance, in all the non-looping rules above, if we count the number of Sequoia POS tag in the graph, it is strictly decreasing at each rule application.

3.3 Confluence

Another well-known issue with rewriting is the problem of confluence. As said earlier, the Sequoia tag V may be converted to AUX or VERB. A naive way to encode this in rules is to write the package:

```
package v_1 {
  rule aux {
    pattern { N [upos = V] }
    commands { N.upos = AUX }
  }
  rule verb {
    pattern { N [upos = V] }
    commands { N.upos = VERB }
  }
}
```

[3] 10,000 by default

The two rules overlap: each time a POS `V` is found, both rules can be used and produces a different output! We call this kind of system non-confluent. Anyway, the strategy `Onf(v_1)` still produced exactly one graph by choosing (in a way which cannot be controlled) one of the possible ways to rewrite.

What should we do with non-confluent system? There are two possible situations: (1) The two rules are correct and there is a real (linguistic) ambiguity and all solutions must be considered or (2) There is no ambiguity, the rules must be corrected.

In our example, we are clearly in the second case, but we consider briefly the other case for the explanation on how to deal with really non-confluent setting. Let us suppose that we are interested in all possible solutions. GREW provides a strategy `Iter(v_1)` to do this: this strategy applied to the same input graph produces 4 different graphs with different combinations of either `AUX` or `VERB` for the two words *sont* and *montrées*.

Of course, in our POS tags conversion example, the correct solution is to design more carefully our two rules, in order to produce the correct output:

```
package v_2 {
 rule aux {
  pattern {N[upos=V]; M -[aux.pass]-> N}
  commands { N.upos=AUX } }
 rule verb {
  pattern { N [upos=V] }
  without { M -[aux.pass]-> N }
  commands { N.upos=VERB } }
}
```

Here, the two rules are clearly exclusive: the same clause `M -[aux.pass]-> N` is used first in the `pattern` part of rule `aux` and in the `without` part of rule `verb`. With these two new rules, the system is confluent, and there is only one possible output. This can be tested with the `Iter(v_2)` strategy which produces all possible graphs, exactly one in this case.

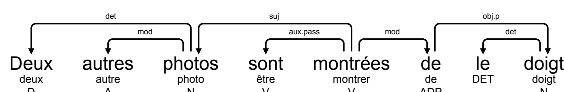
Of course, the strategy `Onf(v_2)` produces the same output in this setting. When a package `p` is confluent, the two strategies `Onf(p)` and `Iter(p)` give the same result. In practice, the strategy `Onf(p)` must be preferred because it is much more efficient to compute.

3.4 More commands

In Figure 1, we can observe that in addition to a different POS tagset, the UD format also uses a different tokenisation. The word *du* of the input sentence is a token with a POS `P+D` in Sequoia but this is in fact an amalgam of two lexical units: a preposition and a determiner[4]. In UD, such combined tag are not allowed and the sentence is annotated with two tokens *de* and *le* for the word *du*. Hence, we have to design a rule to make this new tokenisation. The rule below computes this transformation:

```
rule amalgam {
  pattern {
    N [form = "du", upos = "P+D"];
    N -[obj.p]-> M }
  commands {
    add_node D :> N;
    N.form = "de"; N.upos = ADP;
    D.form = "le"; D.upos = DET;
    add_edge M -[det]-> D }
}
```

This is our first rule with more than one commands. In general, the transformation is described by a sequence of commands which are applied successively to the current graph. The application of this rule to our input graph builds:



Note that `N -[obj.p]-> M` is not required to find a place where the rule must be applied, but we need it to get access to the node with identifier `M` and to define properly the command `add_edge`.

3.5 Changing head

For transformation between different syntactic annotation frameworks, we often have to deal with the fact that heads of constituents may change. For instance, with the sentence *je vois que tu es malade* [en: *I see that you are sick*]. The head of the clause *que tu es malade* is *es* in Sequoia and *malade* in UD. In practice, we have to realise the transformation between the two graphs described by Figure 2.

We can use what was presented before to remove the edge `ats`, to add a new edge `cop` and to change the POS of *es*; but we need something more: moving all other edges incident to the old head *es* towards the new head *malade*. GREW provides a dedicated command `shift` to compute this. In the rule below, the command `shift V ==> ATS` means: change all edges starting (resp. ending) on the node `V` to make them start (resp. end) on `ATS`.

```
rule ats {
  pattern {
    V[upos=VERB];
    e: V -[ats]-> ATS }
```

[4] This is exactly what the tag `P+D` means.

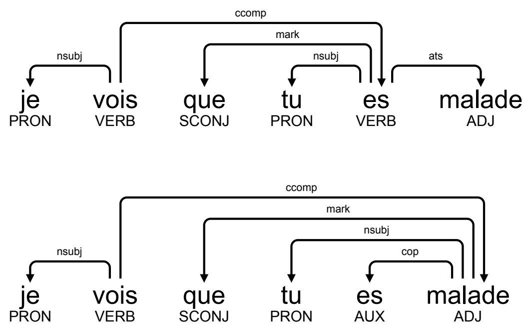


Figure 2: Graph transformation for head changes

```
  commands {
    del_edge e;
    shift V ==> ATS;
    add_edge ATS -[cop]-> V;
    V.upos = AUX }
}
```

3.6 More strategies

Above, we have seen how to handle atomic transformations through rules. But, in order to define a complete transformation system, some larger set of rules are needed. It is important to be able to control the order in which subset of rules should applied. In practice, large transformation system are divided in several steps and sub-systems are applied successively. In our example (Sequoia to UD), the global transformation can be divided into: 1) change POS and tokenisation, 2) change relation labels, 3) make needed head changes. The can be expressed in Grew by a strategy `Seq(POS, relations, heads)`, where `POS`, `relations` and `heads` correspond to dedicated subset of rules.

4 Application of graph matching

Graph matching is a subpart of the system used to describe left part of rewriting rules, but it is also useful alone as a way to make requests on a graph or a set of graphs. In practice, it can be used for searching examples of a given construction, for checking consistencies of annotations or for error mining. This subpart of Grew is now proposed as a separate tool, named Grew-match and freely available as a web service[5]. This graph matching system is also available in the ArboratorGrew tool (Guibon et al., 2020)[6].

A screenshot of the Grew-match interface is shown in Figure 3. With the top bar and the list on the left, the user can chose a corpus (all 183 UD and SUD 2.7 corpora and a few other freely available corpora can be requested). A Request is entered and the user can visualise the occurrences found in the corpora with elements of the pattern highlighted in the sentence.

4.1 Error mining

It is difficult in general to ensure consistent annotations in large corpora. Grew-match can be used to detect this kind of inconsistencies by making linguistic observation on some corpus. The Figure 3 illustrates the first step of such usage with the request: find `nsubj` relations where there is a `Number` disagreement (the head and the dependant of the relation both have a `Number` feature but with different values). In version 2.7 of UD_English-GUM, 120 occurrences of the pattern are found, but there are not all errors, as the example of the figure shows. We can then refine the request by adding some negative patterns (with the `without` keyword), for instance to exclude occurrences with a copula linked to the head:

```
pattern {
  M -[nsubj]-> N;
  M.Number <> N.Number; }
without { M -[cop]-> C }
```

The new request returns 25 occurrences which can be manually inspected: we have found a mix of annotation errors, irregularities (institution plural name used as a singular *the United Nations rates...*) or misspelled sentences. The same approach can be used for many aspect: searching for verbs without subjects, for unwanted multiple relation (more than one `obj` on the same node).

4.2 data exploration

More generally, Grew-match can be used for any kind of data exploration. Here, we use the example of AMR (Banarescu et al., 2013) annotations, this will allow us to show examples where the graph matching used cannot be reduced as a tree matching. Two corpora are available from the AMR website[7]: the English translation of the Saint-Exupéry's novel *The Little Prince* and some PubMed articles. With the pattern below, we search for a node which is the `ARG0` argument of two different related concepts.

```
pattern { P1 -> P2;
 P1 -[ARG0]-> N; P2 -[ARG0]-> N; }
```

[5] http://match.grew.fr
[6] https://arborator.github.io
[7] https://amr.isi.edu/

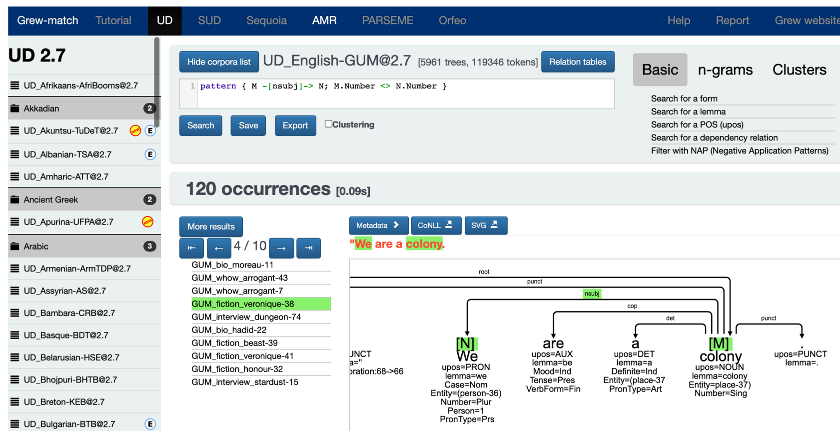
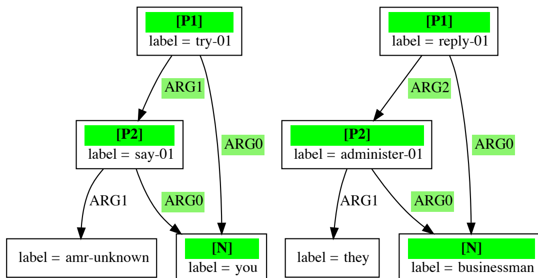


Figure 3: GREW-MATCH main interface

211 occurrences of this pattern are found in *The Little Prince*. Two of them are showed below, for the two sentences: *"What are you trying to say?"* and *"I administer them," replied the businessman.*



4.3 Typology

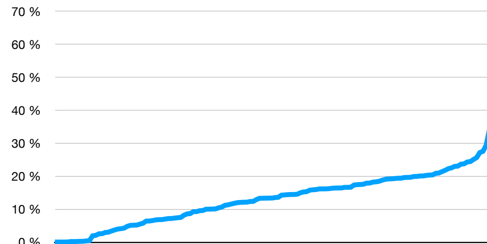
The pattern matching mechanism is also available in a `count` subcommand for GREW. Given a set of corpora and a set of requests, a table with the number of occurrences of each pattern on each corpora is returned. For instance, with the two patterns below, we can compute the ratio of `nsubj` relations which are use with or without a copula construction.

```
pattern { M -[nsubj]-> N; M -[cop]-> *}
```

and

```
pattern { M -[nsubj]-> N }
without { M -[cop]-> * }
```

The chart below shows these ratios, sorted by increasing values on the 141 corpora of UD 2.7 with more than 1000 sentences. Most corpora have a ratio between 0% and 25% with all value represented and a few corpora have a significantly higher proportion. 3 are above 30%: FAROESE-OFT with 67%, FRENCH-FQB with 43% and PERSIAN-SERAJI with 34%.



5 Related works

5.1 Rule-based transformations of linguistic structures

Many implementations of graph rewriting or graph transformation exist in other research areas. But the massive usage of feature structures in linguistic unit description, the usage of dedicated technical formats like CoNLL-U or the need for specific kinds of transformations (like the `shift` operation described above) make general graph transformation system difficult to use in NLP applications. Such applications would require several encodings of the data and they will not allow for a straightforward expression of linguistic transformations. Among existing rule-based software for transformations of linguistic structures, we can cite OGRE[8](Ribeyre, 2016) and Depedit[9].

[8] https://gitlab.etermind.com/cribeyre/OGRE

[9] https://corpling.uis.georgetown.edu/depedit

OGRE uses a notion of rules which is very closed to the ones used in GREW, but it does not provide interface with lexicons and there is no notion of strategies for the description of complex graph transformations which imply a large number of rules.

Depedit can be used as a separate tool or as a Python library. It is specifically designed to manipulate only dependency trees. Contrary to GREW, it does not proposed a built-in notion of strategies and does not handle not confluent rewriting processing. Moreover, the notion of rules is also more restricted: there are no NAP and it is not possible to express additional contraints on morphological features like the one we used in Section 4.1: `M.Number <> N.Number`.

5.2 Tools for corpora querying

A large number of online query tools are available online. Some of them have a more restrictive query language like SETS[10] or Kontext[11]. In these two tool, there is no notion of NAP and the kind of contraints that can be expressed is limited.

The PML Tree Query[12] and INESS[13] offers a query language with the same expressive power as the one proposed in GREW. An advantage of GREW is that it is interfaced in the larger annotation tool ARBORATORGREW[14]. With ARBORATOR-GREW, the user may query on his own treebank and then have access to a manual editing mode on the query output or to automatic updating through GREW rules.

6 Conclusion

GREW was used in many tasks of corpus conversion. It is used for instance for conversion between UD and SUD (Gerdes et al., 2018, 2019): all UD corpora are converted into SUD with it. GREW is implemented in Ocaml and is quite efficient: for instance the conversion of UD 2.7 (1.48M sentences, 26.5M tokens) into SUD uses 100 rules and takes 5,500 seconds on a labtop (around 267 graphs rewritten by second).

GREW is available as a command line program or through a Python library. Installation procedures and usage documentation are given on the GREW website: https://grew.fr. A web-based interface for the usage of the rewriting part of the software will be provided soon.

In this article, examples are given on dependency syntax and on semantic representations like AMR. A more complete set of examples is given in (Bonfante et al., 2018). Many other linguistic structures can be encoded as graphs and we plan to extend the experiments to other kind of semantic representations.

Acknowledgments

Thanks to all GREW users for their feedback and their requests which helps improve the software.

[10] http://depsearch-depsearch.rahtiapp.fi/ds_demo

[11] http://lindat.mff.cuni.cz/services/kontext

[12] http://lindat.mff.cuni.cz/services/pmltq

[13] http://clarino.uib.no/iness

[14] https://arborator.github.io

References

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186.

Guillaume Bonfante, Bruno Guillaume, and Guy Perrier. 2018. *Application of Graph Rewriting to Natural Language Processing*, volume 1 of *Logic, Linguistics and Computer Science Set*. ISTE Wiley.

Marie Candito and Djamé Seddah. 2012. Le corpus Sequoia : annotation syntaxique et exploitation pour l'adaptation d'analyseur par pont lexical. In *TALN 2012 - 19e conférence sur le Traitement Automatique des Langues Naturelles*, Grenoble, France.

Kim Gerdes, Bruno Guillaume, Sylvain Kahane, and Guy Perrier. 2018. SUD or Surface-Syntactic Universal Dependencies: An annotation scheme near-isomorphic to UD. In *Universal Dependencies Workshop 2018*, Brussels, Belgium.

Kim Gerdes, Bruno Guillaume, Sylvain Kahane, and Guy Perrier. 2019. Improving Surface-syntactic Universal Dependencies (SUD): surface-syntactic relations and deep syntactic features. In *TLT 2019 - 18th International Workshop on Treebanks and Linguistic Theories*, Paris, France.

Gaël Guibon, Marine Courtin, Kim Gerdes, and Bruno Guillaume. 2020. When Collaborative Treebank Curation Meets Graph Grammars. In *LREC 2020 - 12th Language Resources and Evaluation Conference*, Marseille, France.

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. 2016. Universal dependencies

v1: A multilingual treebank collection. In *Proceedings of LREC 2016*, pages 1659–1666.

Corentin Ribeyre. 2016. *Méthodes d'analyse supervisée pour l'interface syntaxe-sémantique : De la réécriture de graphes à l'analyse par transitions*. Ph.D. thesis, Université Paris 7 Diderot & Inria.