

QUICK USER GUIDE TO THE LINEAR ARRANGEMENT LIBRARY

Development version 99.99

LLUÍS ALEMANY-PUIG & RAMON FERRER-I-CANCHO

Computational and Quantitative Linguistics Laboratory (CQLLab)
Department of Computer Science, Institute of Mathematics of UPC-Barcelona Tech (IMTech)
Universitat Politècnica de Catalunya
Barcelona, Catalonia

Last updated: August 3, 2023

Contents

1	Introduction	3
1.1	Why LAL	3
1.1.1	Because it is easy to use	3
1.1.2	Because it includes new and state-of-the-art methods and algorithms	5
1.1.3	Because it is thoroughly tested	5
1.1.4	Because it is efficient	5
1.2	Installation	5
1.2.1	Windows	6
1.2.2	Ubuntu – Anaconda	10
1.2.3	Ubuntu – no Anaconda	11
1.2.4	Mac OS	11
1.2.5	Ensuring everything works	12
1.2.6	Troubleshooting	12
1.3	Uninstalling	13
1.3.1	Windows	13
1.3.2	Ubuntu	14
1.4	Seeking help online	14
1.5	Citing the library	14
1.6	Staying up-to-date	15
1.7	Contents of this guide	15
2	Dependency structures	15
2.1	Represented as a list of edges	16
2.2	Represented as a head vector	16
2.3	Reading dependency structures from a file	17
2.4	Making dependency structures manually	18
2.4.1	Using an edge list	18
2.4.2	Using a head vector	19
2.4.3	Ensuring we are constructing trees correctly	19
2.5	Calculating metrics on dependency structures	20
2.5.1	The sum of syntactic dependency distances and its minimum	22
2.5.2	Proportion of head initial dependencies	24
2.5.3	Dependency flux	24
2.5.4	Mean Hierarchical distance	26
2.5.5	Omega (Ω)	26
3	Working with a single treebank	27
3.1	Before jumping to the analysis of treebanks	27
3.2	Analyzing a treebank file automatically	29
3.2.1	Calculating Ω in a treebank	31
3.3	Custom analysis of a treebank file	32
3.4	Notes on corner cases	34
3.5	Correctness outside treebanks	35
4	Working with a treebank collection	35
4.1	Before jumping to the analysis of a treebank collection	35
4.2	Analyzing a treebank collection automatically	36
4.3	Custom analysis of a treebank collection	38
4.4	Hints on efficiency	38

1 Introduction

The Linear Arrangement Library (LAL) is a collection of tools to work with syntactic dependency structures and compute their statistical features. As such, it serves to researchers of different communities: quantitative linguistics (quantitative dependency syntax), computational linguistics (dependency parsing), graph theory and computer science.

The Linear Arrangement Library owes its name to the term linear arrangement, namely an ordering of the vertices of a graph, that is borrowed from graph theory and computer science. The relationship with dependency syntax is straightforward: the syntactic dependency structure of a sentence is a three-fold structure: a graph that is defined by word pairwise syntactic dependencies, a linear arrangement that is usually defined by the natural order of words in a sentence and additional information (such as labels attached to vertices or edges of the graph).

LAL is written in C++ for efficiency reasons but it is also available in Python to work quickly and integrate systems more effectively, also to ease its use by those who are not acquainted with C++ or advanced programming in general. This document is intended as a quick guide for language researchers or students with a minimum background of Python. As we will see, even having no background of Python should not be a serious hindrance for starting to use LAL and benefiting from it.

1.1 Why LAL

There are many reasons why researchers working in Dependency Syntax or Quantitative Linguistics might be interested in the Linear Arrangement Library.

1.1.1 Because it is easy to use

LAL has been designed with simplicity in mind, and hence many tasks can be done with *one liners*, which are just individual lines of code that can perform many tasks at the same time. This means that no advanced programming skills, only elementary knowledge of Python, are needed to work with comfortably. More importantly, users who have never programmed before, neither in Python nor any other language, can learn Python as they learn how to use LAL.

The first example showing LAL's simplicity is that of the representation of a sentence with the library. Take, for example the sentence in Figure 1.

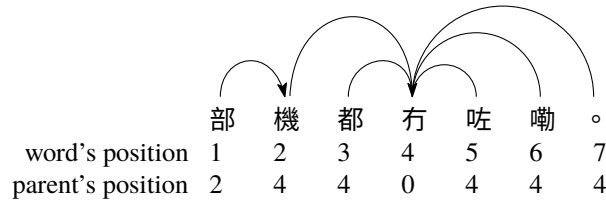


Figure 1: The 9th sentence example taken from UD 2.7 [28] Cantonese Github repository [27].

In Figure 1, we see the position number of every word, and below it we see the position of each word's parent (this is discussed in more detail in Sections 2.2 and 2.4.2). Such numbers make up the *head vector* of the sentence, and this can be used to represent that sentence very easily. Oftentimes, head vectors of sentences are stored in files, called *treebanks*. For example, the Cantonese treebank [27] (of the Universal Dependencies [28] dataset) starts with the 9 trees in Figure 2. Notice that the last line of Figure 2 contains the head vector of the sentence given in Figure 1.

```

3 3 0 3 3 3
2 0 2 5 3 2 10 10 10 6 10 2 2
2 3 5 5 0 5 5 5
3 1 6 6 6 0 6 6 6
4 4 4 0 4 4
0 1 6 6 6 1 8 6 8 6 6
6 6 6 6 4 0 6 7 12 12 12 6 12 12 6
4 4 4 0 4 4 4 4 4
2 4 4 0 4 4 4

```

Figure 2: The first 9 sentences in UD 2.7 [28] Cantonese treebank in [27].

Treebanks can be processed very easily using LAL. Processing the treebank in Figure 2, stored in a file called `Cantonese.txt`, can be done as in Code 1.1. These two lines of code¹ will produce a `.csv` file with columns listed and explained in Table 3. In many applications, that `.csv` suffices to visualize results in the form of figures or is directly the input of statistical analyses that will produce the main results.

```

import lal
err = lal.io.process_treebank("Cantonese.txt", "output_file.txt")
print(err)

```

Code 1.1: Processing treebanks with LAL is extremely straightforward. However, we should check for errors in case we misspell the name of the treebank.

Readers will find in Code 1.3 a summary of some of the tasks that can be performed with LAL in a single line; these are further discussed in later sections of this introductory manual. We can easily read a rooted tree from a file, containing either an edge list (Code 1.2#1) or a head vector (Code 1.2#2), and we can also create one manually from a list of edges (Code 1.2#3) or from a head vector (Code 1.2#4).

```

import lal
# (1) Read an edge list from a file
rt = lal.io.read_edge_list("rooted_tree", "path/to/edge_list.txt")
# (2) Read head vector from a file
rt = lal.io.read_head_vector("rooted_tree", "path/to/head_vector.txt")
# (3) Create a tree manually from a list of edges
rt = lal.graphs.rooted_tree(7)
rt.set_edges([ (3,1),(3,2),(3,4),(3,5),(3,6), (1,0) ])
# (4) Create a tree manually from a head vector
rt = lal.graphs.from_head_vector_to_rooted_tree([2,4,4,0,4,4,4])

```

Code 1.2: Creating a sentence structure with LAL in just a single line.

In computer science, the minimum linear arrangement problems consists of finding the minimum sum of dependency distances over all the possible orderings of the vertices of the tree structure [2,

¹Code 1.1 can be considered a one liner if we ignore the `import` and consider that no errors arise in the second line.

3]. As its names indicates, the Linear Arrangement Library allows to deal with problems involving word order, such as the minimum linear arrangement problem (Code 1.3#1), and to calculate scores on linear arrangements that are essential in Quantitative Linguistics (Code 1.3#2).

```

1  import lal
2  # (1) Minimum Linear Arrangements under no constraint (1.1), under the
3  # planarity constraint (1.2) and under the projectivity constraint (1.3)
4  min_unc, arr_unc = lal.linarr.min_sum_edge_lengths(rt) # (1.1)
5  min_plan, arr_plan = lal.linarr.min_sum_edge_lengths_planar(rt) # (1.2)
6  min_proj, arr_proj = lal.linarr.min_sum_edge_lengths_projective(rt) # (1.3)
7  # (2) Computing scores essential in Quantitative Linguistics
8  D = lal.linarr.sum_edge_lengths(rt)
9  head_initial = lal.linarr.head_initial(rt)
10 dependency_fluxes = lal.linarr.compute_flux(rt)

```

Code 1.3: Calculating minimum linear arrangements and scores essential in Quantitative Linguistics. Line 4 calculates a minimum linear arrangement under no formal constraint [2, 3]; line 5 does so under the planarity constraint [5, 18] (which includes those arrangements that are not under the projectivity constraint); line 6 does so under the projectivity constraint [18]. Line 8 calculates the sum of dependency distances; line 9 the proportion of head initial dependencies [9]; and line 10 computes the dependency fluxes of a rooted tree [13].

1.1.2 Because it includes new and state-of-the-art methods and algorithms

LAL includes state-of-the-art methods and algorithms that are either new (and thus not found somewhere else) or that are difficult to implement and/or test and, more importantly, it includes our own research carried out in order to support current and further new research in Quantitative Dependency Syntax. Since LAL is thoroughly documented, users will find all the references in its documentation online [26].

1.1.3 Because it is thoroughly tested

We, the developers of LAL, have implemented a rigorous testing procedure, and crucial complex algorithms are tested with brute force exhaustive methods that result from transferring knowledge from previous research [11, 12]. During development, our test suite is executed regularly, ensuring that not even the smallest changes change the results, and that no program that uses LAL correctly will ever produce wrong results. Moreover, we ensure correctness at the lowest level of execution, so as to avoid failures in programs that use LAL.

1.1.4 Because it is efficient

Simplicity and correctness are not the only hallmarks of the library. At present, the library allows users to process all the treebanks in the Universal Dependencies 2.5 [16] in less than 1½ minutes while generating about 400 MB of data.

1.2 Installation

It is recommended that both Windows and Ubuntu users install an Integrated Development Environment (IDE) to write Python code to use the Linear Arrangement Library. Our suggestion is to install Spyder [24] via Anaconda [21]. Anaconda is a manager of scientific computing software comprising

many programming languages such as R and Python. Users should install Anaconda first and then install Spyder.

The advantages of using an IDE are many:

- Documentation in the IDE (Figure 3(right)). All methods come with an explanation of what that method does, what its parameters are, their type, what the method returns, requirements (preconditions) and expected results (postconditions), ... In other words, methods come with the full documentation every user needs. Such documentation is available online [26], but IDEs typically show that documentation within the program's window making the process of coding much faster. Spyder [24] provides such documentation both while typing and after the method's name has been typed, we can see the method's documentation by pressing **Ctrl+i** when our cursor is placed over the method's name.
- Auto-completions (Figure 3(left)). Due to the fact that this library is quite large, most methods are unknown to beginners in using LAL, therefore we recommend using an IDE that provides autocompletions that help users in both finding and typing the methods available to a class.
- The library is easier to install. Especially for Windows users, the library can be more easily installed if they use the installer provided (Section 1.2.1).

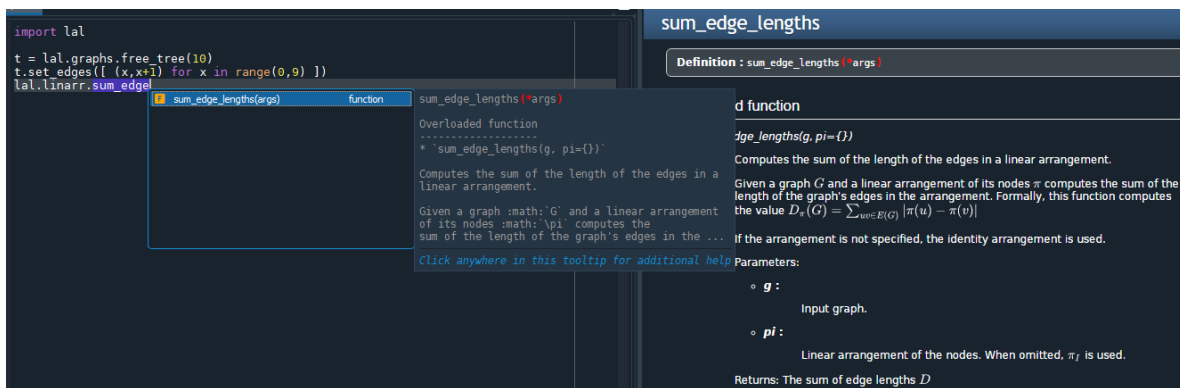


Figure 3: Left: Spyder's autocompletion in action. Right: the documentation of the function `sum_edge_lengths`.

1.2.1 Windows

In order to install Spyder, first download Anaconda from their site [21]; the installation should use the default values (install for just the current user only, and do not add Anaconda3 to Window's path environment variable).

Create a new environment We recommend creating a new environment with a python version of your choice. We support Python 3.9 and 3.10, so users should choose one of these two. In order to create an environment launch anaconda prompt and run the command shown in Figure 4. If a suitable environment already exists, skip to the next step. Figure 4 shows how to create an environment for Python 3.9. Users can change the environment's name by replacing the text `py39` in the command with a more preferable option.

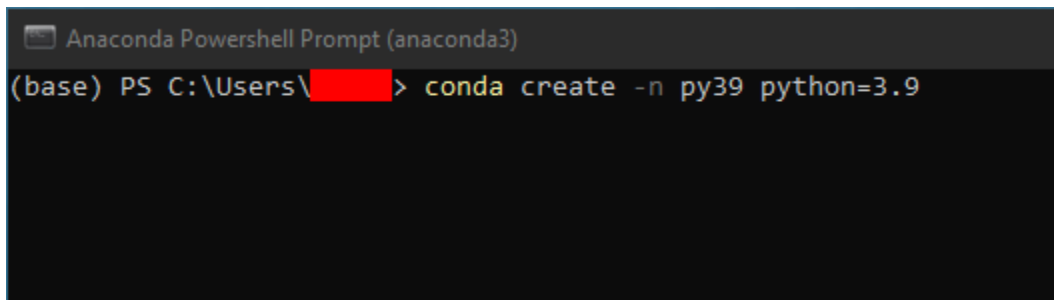


Figure 4: Creating a new environment using anaconda's prompt.

Installing Spyder First, choose the preferred environment via Anaconda navigator (Figure 5). Now, find Spyder's logo and install it. If it is already installed, the 'launch' button should be enabled (Figure 6).

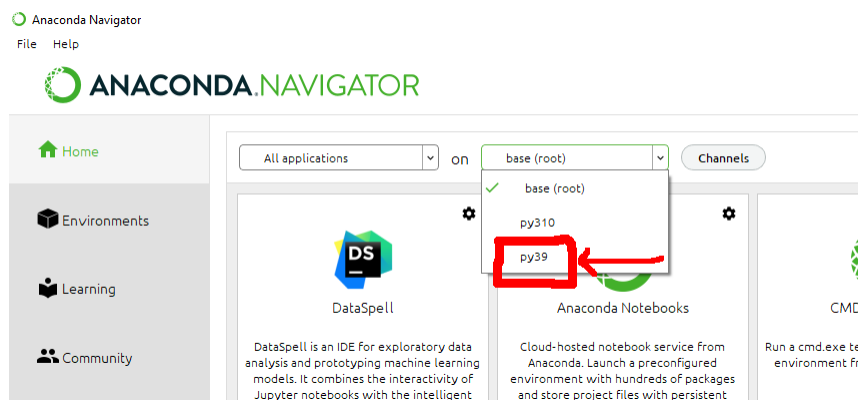


Figure 5: Choosing the environment inside Anaconda navigator. Here we want to choose the environment we have just created.

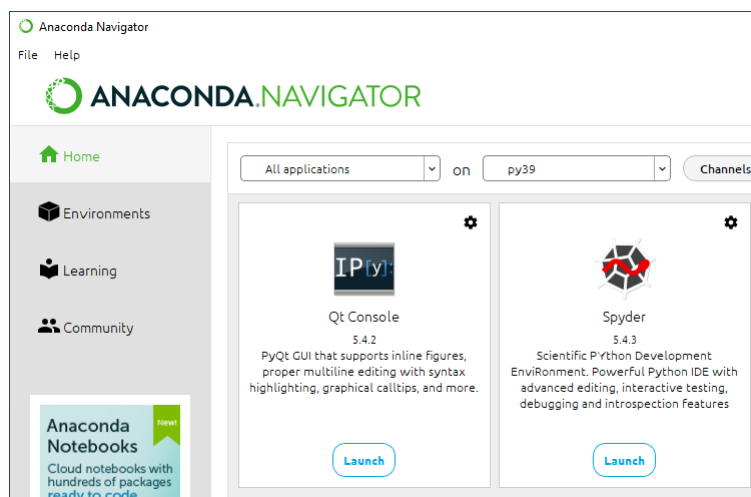


Figure 6: Launch Spyder.

Installing LAL The steps to be taken for LAL’s installation are listed in Figures 7 to 11. Each figure’s caption explains the task to be done at each step. These steps illustrate the installation of LAL for Python 3.9.

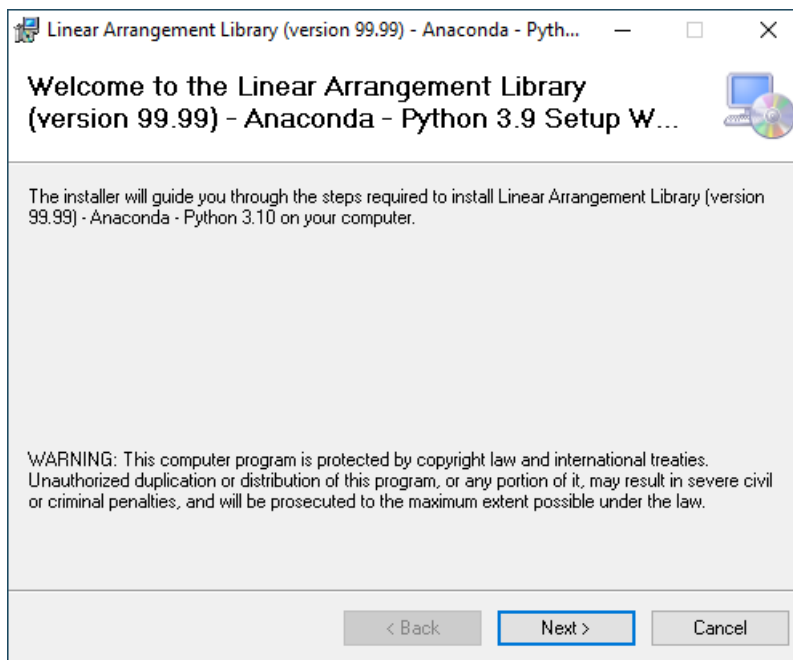


Figure 7: Step 1. The welcome message of the installer wizard.

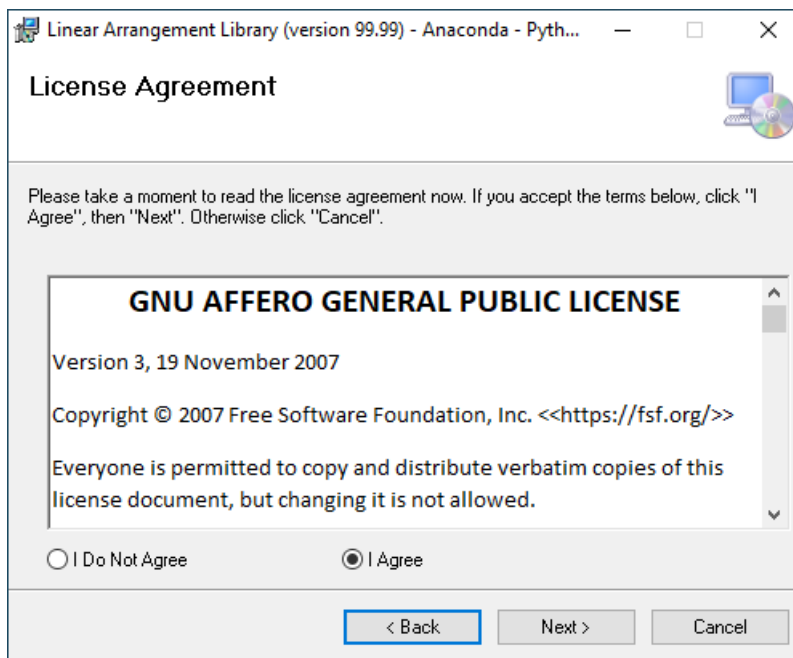


Figure 8: Step 2. Users must accept the License agreement.

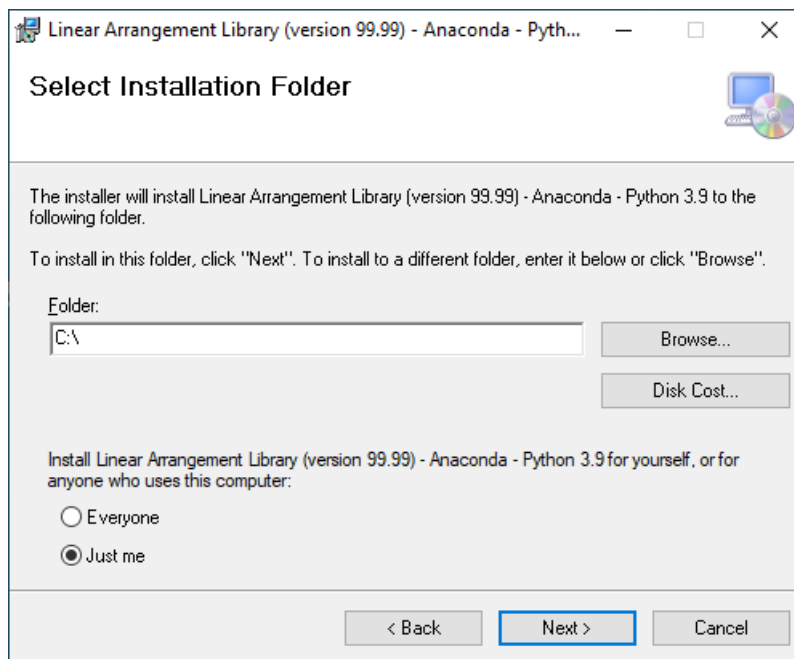


Figure 9: Step 3. The installer will not look for your Anaconda installation. Users must manually find Anaconda's root directory as depicted in Figure 10.

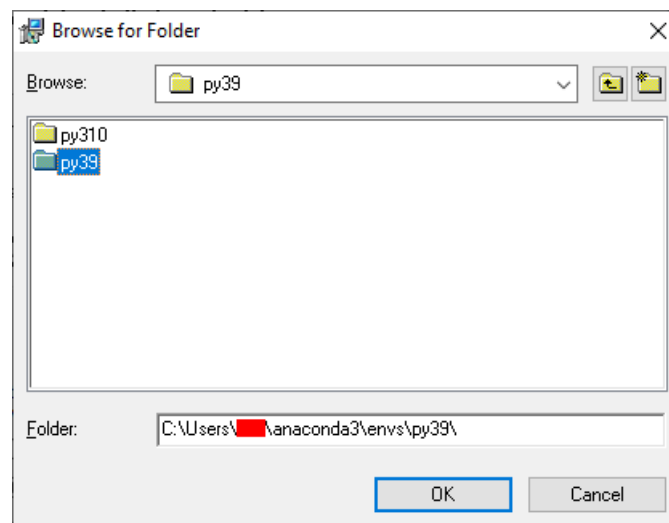


Figure 10: Choosing the environment's directory. Since we named our environment `py39` we should look for a equally-named directory inside `envs` folder within anaconda's installation folder.

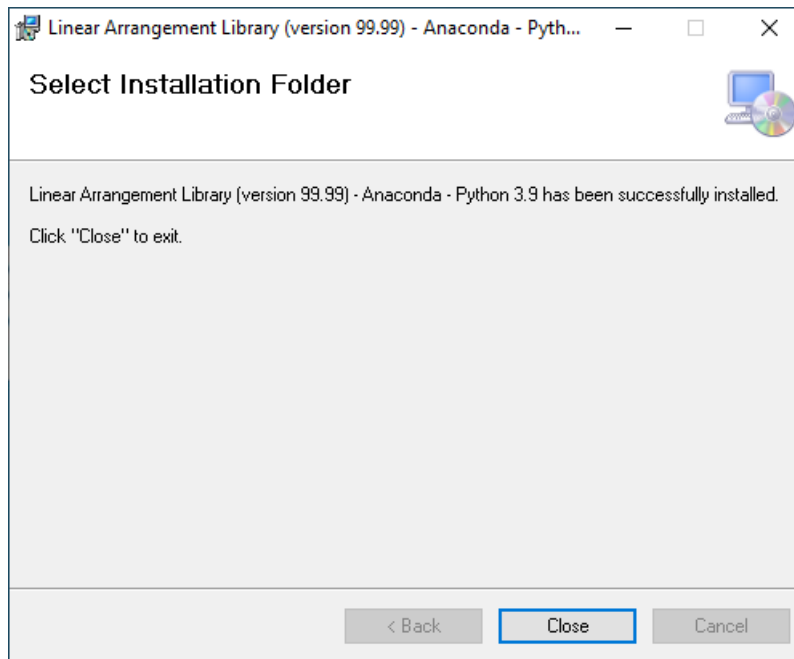


Figure 11: Step 4. The installation is now complete.

1.2.2 Ubuntu – Anaconda

In order to install Anaconda, first download the installer from their site [21].

Installing Spyder Those Ubuntu users that choose to use Spyder should first download Anaconda’s installer from its webpage. Save the downloaded file anywhere in your system, say, your desktop. Once the download has finished open a command line terminal (the shortcut **Ctrl + Alt + T** should work), and navigate to the folder containing the downloaded file, like so

```
$ cd ~/Desktop
```

Then, execute the downloaded file; for this, users must know the name of the downloaded file. At the time of writing this guide, the name was ‘**Anaconda3-2023.05-Linux-x86.64.sh**’. This might require granting the file execution permissions.

```
$ chmod +x Anaconda3-2023.05-Linux-x86_64.sh # grant execution permissions
$ ./Anaconda3-2023.05-Linux-x86_64.sh          # install anaconda 3
```

The second command will prompt users to something similar to this

```
Welcome to Anaconda3 2023.03-1

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>>
```

Follow the instructions that appear on the screen. They should be straightforward enough for everyone, so we do not detail them here. We recommend to not change the default values given by anaconda. For example, when the prompt asks for the installation location

```
Anaconda3 will now be installed into this location:
/home/$USER/anaconda3
```

- Press ENTER to confirm the location
- Press CTRL-C to abort the installation
- Or specify a different location below

```
[/home/$USER/anaconda3] >>>
```

we recommend choosing the default value by simply pressing ENTER.

Creating a new environment Users who have just installed anaconda are encouraged to create a new environment. For this, open a terminal and run the following two commands

```
$ conda create --name py39 python=3.9 # create a new environment
$ conda activate py39
```

Installing Spyder Spyder should now be easy to install with a single command

```
$ conda install -c anaconda spyder
```

To run Spyder simply execute the following command

```
$ ~/anaconda3/envs/py39/bin/spyder
```

Installing LAL Finally, download the python files for LAL. Uncompress them in a directory, say in the home directory. To install LAL, navigate to the directory that has just been uncompressed

```
$ cd ~/python-libs
```

and simply run the following command

```
$ ./anaconda_install.sh ~/anaconda3/envs/py39 3.9
```

Replace the environment name (first parameter) and the python version (second parameter) accordingly.

1.2.3 Ubuntu – no Anaconda

If, on the contrary, users choose not to use anaconda, then the installation of LAL changes slightly. In this case, LAL's C++ distribution files are to be placed in the system `/usr/lib` directory. To install Python files simply copy the folders `lal/` and `laloptimized/` into the system directory `/usr/lib/python3.x` (where `x` is to be replaced by the minor version of Python). This requires superuser permissions. To do this, run the following commands:

```
$ sudo cp -r lal/ laloptimized/ /usr/lib/python3.x
$ sudo cp liblal* /usr/lib
```

where `x` should be replaced with the minor version of Python.

1.2.4 Mac OS

Unfortunately, we are not able to provide our users with a MacOS package. However, we hope that we can do so in the near future.

1.2.5 Ensuring everything works

Launch Spyder and run the line of code given in Code 1.4 to make sure that the library has been installed properly and that your system recognizes it properly. To run that line, make sure your cursor is placed on that line and then press F9.

```
import lal
```

Code 1.4: Importing the library.

Users who have successfully installed the library should see no warning or error message whatsoever in the lower right corner of Spyder (i.e., the `ipython` console), as in Figure 12.

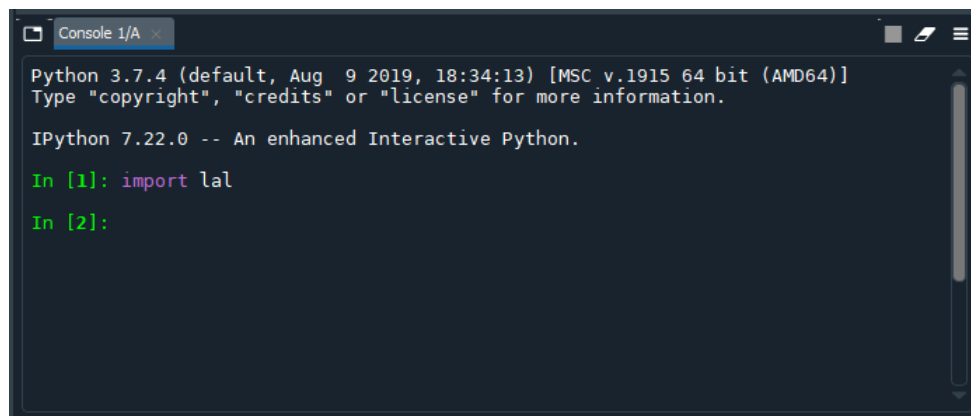
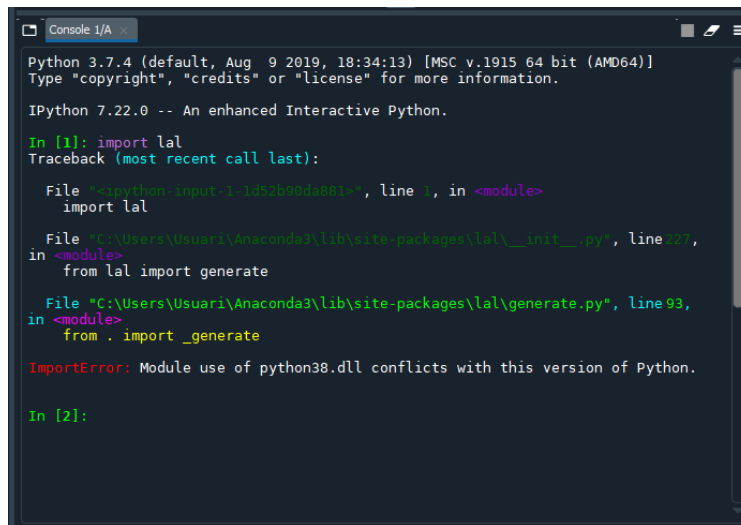


Figure 12: A successful load of the Linear Arrangement Library should look like this: no error messages below the `import lal`.

1.2.6 Troubleshooting

There are many reasons for which importing can go wrong. We describe two of them related to a mismatch between the minor versions of Python, i.e., between the version used to generate LAL for Python, and the one installed in the user's computer. These usually arise in Windows due to having used the wrong installer. For example, if the minor version of Python installed is '7', but the installer for '8' is used, then we will see an error like the one in Figure 13. In this case, notice that the error message 'ImportError: Module use of python38.dll conflicts with this version of Python' gives users a really good clue on what is wrong. Errors can also happen when choosing the installer for Python 3.6 for a computer where version 3.7 is installed. In this case, the error message may look like the one in Figure 14. Users who encounter any of these errors must uninstall LAL (by double clicking on the installer and choosing the option to uninstall LAL) and then installing LAL using the appropriate installer.



```

Python 3.7.4 (default, Aug  9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.22.0 -- An enhanced Interactive Python.

In [1]: import lal
Traceback (most recent call last):

  File "<ipython-input-1-105b04d4001>", line 1, in <module>
    import lal

  File "C:\Users\Usuari\Anaconda3\lib\site-packages\lal\__init__.py", line 27,
in <module>
    from lal import generate

  File "C:\Users\Usuari\Anaconda3\lib\site-packages\lal\generate.py", line 93,
in <module>
    from . import _generate

ImportError: Module use of python38.dll conflicts with this version of Python.

In [2]:

```

Figure 13: An unsuccessful load of the Linear Arrangement Library due to a mismatch in the minor versions of Python: the minor version used to generate LAL is in conflict with the minor version used by Spyder. Notice that the error message says that LAL was generated with the minor version ‘8’ and that atop the window we can read the minor version that is used by Spyder (version ‘7’).



```

Python 3.7.4 (default, Aug  9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.22.0 -- An enhanced Interactive Python.

In [1]: import lal
Traceback (most recent call last):

  File "<ipython-input-1-105b04d4001>", line 1, in <module>
    import lal

  File "C:\Users\Usuari\Anaconda3\lib\site-packages\lal\__init__.py", line 27,
in <module>
    from lal import generate

  File "C:\Users\Usuari\Anaconda3\lib\site-packages\lal\generate.py", line 93,
in <module>
    from . import _generate

ImportError: DLL load failed: The specified module could not be found.

In [2]:

```

Figure 14: An unsuccessful load of the Linear Arrangement Library due Python not being able to find a module. In this case it is more difficult to know the actual reasons why this happened, but it can arise because of Python minor versions not being equal.

1.3 Uninstalling

1.3.1 Windows

In order to install a newer version of LAL, the older (installed) version of the library has to be uninstalled first. This is done in different ways depending on the user’s OS: in Windows, users can use Control Panel’s **Programs & Features** app to uninstall the library. Look for LAL’s entry in the list, right-click on it and choose **Uninstall**.

1.3.2 Ubuntu

In Ubuntu, simply remove the files that you copied in your system (Section 1.2.3).

1.4 Seeking help online

Users might find it useful to receive help from other users of the library, or from the developers of LAL themselves. When it comes to making inquiries about LAL's usage, one possibility is to use the online platform Stack Overflow [25] whose community is dedicated to helping programmers, including beginners, in understanding how to use a certain programming language or a given library. We, the developers of LAL, often visit the site and suggest that users of LAL post their usage-related questions with the tag 'lal-python' or 'lal-c++' depending on what language the user uses LAL with. We will be watchful for posts containing that tag and hope that we can help our users the best we can through that site. For bug-related issues, we strongly recommend and encourage users to open an issue at LAL's Github page [20]. When opening such an issue, it is paramount to clearly state what the problem is, what result should be expected, and what result is actually received. Also, users should add a few notes on why they think the result should be different. Furthermore, the issue tracker on LAL's Github page also admits feature requests; users interested in some well-established score/algorithm/method being implemented in the library should make a serious proposal through the webpage by providing references to papers, and a comprehensive yet concise paragraph explaining why that feature should be included in the library. Alternatively, if the means above do not fit you, you can reach the LAL team for support (like installation issues on any OS, probable bugs, usage enquiries, ...) sending an email to lal-support@mylist.upc.edu. We will answer your email as soon as we can.

Exceptionally, users can contact the team directly via email.

Person	Contact information	Role
Lluís Alemany Puig Ph. D. Candidate	lluis.alemany.puig@upc.edu	Main developer and designer
Juan Luis Esteban-Ángeles Associate professor	juan.luis.esteban@upc.edu	Collaborator
Ramon Ferrer-i-Cancho Associate professor	ramon.ferrer@upc.edu	PI and designer

1.5 Citing the library

In this library users will find calculations (e.g., scores) or implementations of algorithms from papers written by the developers of LAL themselves, and calculations and implementations of algorithms devised by other researchers. This library is the result of a great effort put directly by the developers of LAL, and indirectly by many other researchers whose algorithms have been implemented by the developers of LAL. Thus, if you use the library in your publications and/or research other products, you should cite two kinds of sources. First, the relevant publications of LAL, and the publications of the scores/algorithms calculated by LAL that are used in your research. It goes without saying that users *must* acknowledge the effort made by other researchers by citing the papers where they present scores, algorithms,... implemented in this library. Such papers are cited in the C++ documentation and can be found easily in the Doxygen [22]. In case a reference is lacking, there is some mistake, or any other issue that needs to be addressed, please do not hesitate to contact the developers of LAL to get that fixed. Second, to acknowledge the developers of LAL, you should cite the webpage [26] until that citation can be replaced by an article presenting LAL to the scientific community. On top of all this, users must also cite the Zenodo repository of the exact version of LAL they are using. The development version has no Zenodo repository.

When citing the library², users *must* acknowledge the effort made by other researchers by citing their papers in which they present the algorithms implemented in this library. Such papers are cited in the C++ documentation and can be found easily in the Doxygen [22] documentation. In case a reference is lacking, there is some mistake, or there is any other inquiry of your interest that needs to be resolved, please do not hesitate to contact the authors to get that fixed.

1.6 Staying up-to-date

If you wish to receive announcements about the library (new versions, bug fixes, new documentation, courses,...) please register to this mailing list <https://mylist.upc.edu/wws/subscribe/lal-announcements>.

The developers of LAL have multiple obligations. New versions of the library are not likely to be frequent, with probably several months apart from each new version. Nevertheless, older versions will be updated in case bugs (or other issues) are found. These fixes will be uploaded to the library's webpage [26] until a version stops being maintained, namely, versions of the library will only be maintained for a certain period of time.

1.7 Contents of this guide

With this introductory manual, users will learn how to do the most basic operations with LAL in Section 2, namely, creating single dependency structures by either reading one from a file (Section 2.3) or by creating it manually (Section 2.4); it is further explained how to use them to calculate statistical features in Section 2.5. Moreover, readers will also learn how to process single treebanks easily in Section 3 and collections of treebanks in Section 4.

2 Dependency structures

Dependency structures are a representation of sentences and the syntactic relationships between pairs of words which are represented with links. Figure 15 shows an example of such a dependency structure. In that figure, words are related by a *directed* link, pointing from the *head* word to the *dependent* word. We can see that for most words there are ingoing edges and also outgoing edges; words like *John* and *Yorkshire* have no outgoing edges, and the only word that has no ingoing edges is the verb *was*. To the right of the sentence we see the tree structure of the sentence, where the verb *was* is placed atop this tree since it has no ingoing edges.

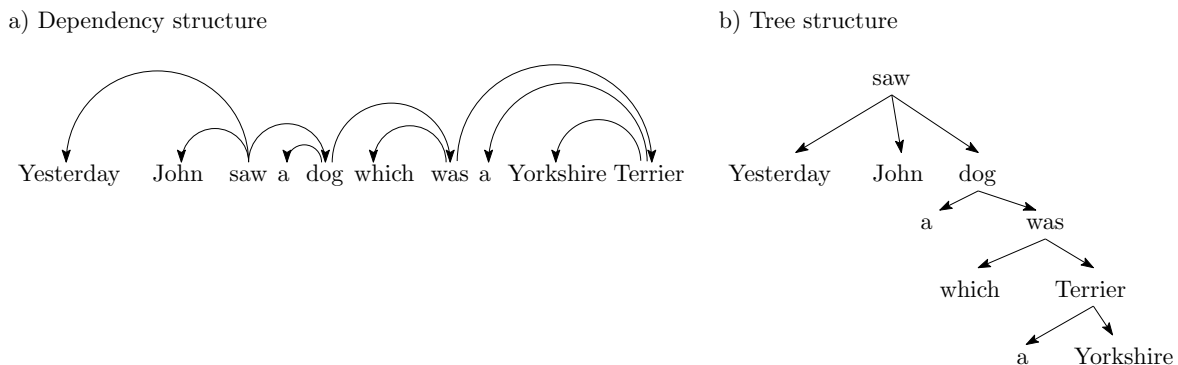


Figure 15: The sentence “Yesterday John saw a dog which was a Yorkshire Terrier” and a possible dependency structure and its tree-like structure. Example adapted from [7, Figure 2].

²As of June 2021 there is no paper exposing the library to the scientific community that LAL users can cite in their academic papers. Please, cite the webpage [26] instead until there is a better option to do so.

Dependency structures have very remarkable properties. Links, or *edges*, are directed: they point from a word (the *head* word) to another word (the *dependent* word). Every dependency structure has a head word that is not the dependent word of any dependency, this is the *root* word of the structure. Notice that links always point away from the root word of the sentence. From now on, we will represent these structures as trees, where instead of words they contain *vertices*; it goes without saying that these are in one-to-one correspondence with the words in the sentence, but are represented in the library with numbers, the lowest being '0' and the largest being one less than the number of words in the sentence. In Figure 16, vertices are labeled with consecutive integers starting at zero from left to right, hence the first word of the sentence is labeled with a 0, the second word with a 1, the third with a 2 and so on and so forth.

In the next sections we will see various ways of creating a tree, which depend mainly on how the edges are defined or provided.

2.1 Represented as a list of edges

An *edge list* is the *collection of dependencies* between every pair of words, or, what is the same, between the vertices of a tree. In order for us to be able to construct such a list, we must begin by labeling the words of the sentence with whole numbers between 0 and $n - 1$, being n the number of words of the sentence. This numbering can be completely arbitrary with the only condition that each word must have a unique index. For simplicity in this quick guide, we will label vertices as in Figure 16. In order to work out the pair of values for an edge, we look at a single edge and locate the pair of vertices that it relates. Then, we write the indices of those two vertices so that the first value in the pair is the index corresponding to the head word, and the other value is the index corresponding to the dependent word. Therefore, when writing the pairs of values we must bear in mind that the pairs are *not* directionless; quite the opposite, pairs are directed and the left value represents the parent vertex, and the right value represents the child vertex.

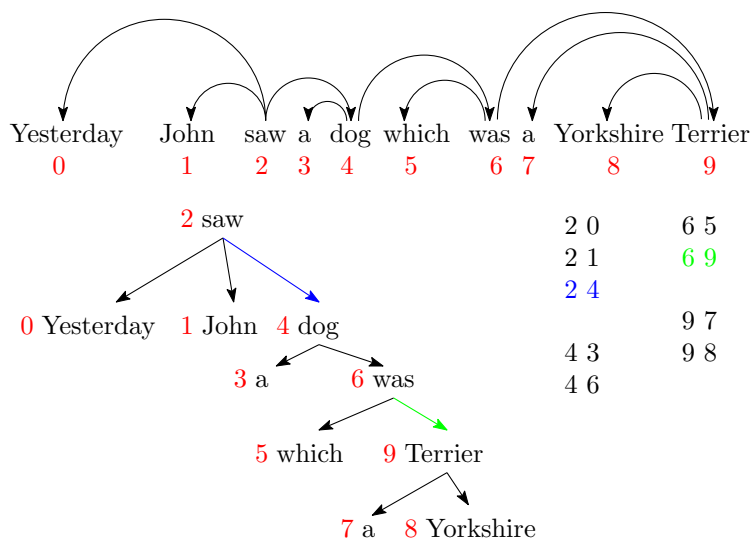


Figure 16: The tree structure of the dependency structure in Figure 15 with every word labeled and the edge list to the right of said tree structure. The blue link corresponds to the blue pair of indices, and the orange link corresponds to the orange pair of indices.

2.2 Represented as a head vector

A head vector is simply an array of as many numbers as words in the sentence, where every number represents the position of the parent word of the word in the position occupied by that number.

Constructing a head vector is slightly more complex than constructing an edge list, however it is a natural representation of the structure embedded in a linear sequence, since it captures the order of the words, or, what is the same, of the vertices of the tree. In addition, a head vector has the advantage of taking less space in disk than a list of edges. Our recipe and best advice from us to construct a head vector has two steps:

1. Label every word with a number according to the position that the word occupies within the sentence, starting at position 1 for the leftmost word. From now on, we refer to these numbers as the “position numbers” (Figure 17).

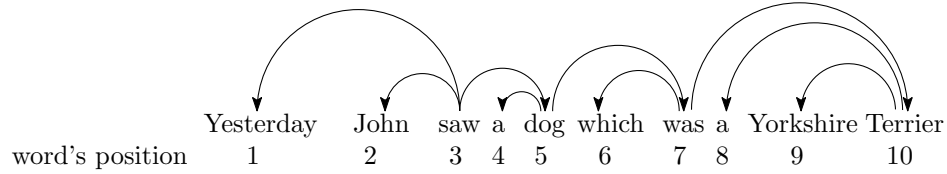


Figure 17: The dependency structure in Figure 15 with the position numbers written below the sentence.

2. Now we write the position numbers of the parent words. We illustrate this with an example. Take word **John** from the example in Figure 15. We look at the word that its only *ingoing* dependency is pointing *from*, i.e., word **saw**, and write its position number below the position number of word **John**. We repeat this for every word in the sentence, except for the word with no outgoing dependencies, i.e., word **saw**, for which we must write a zero ('0').

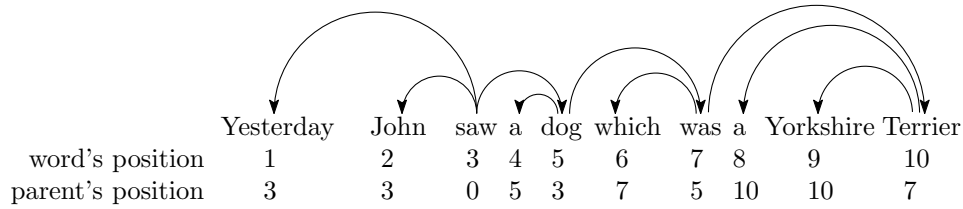


Figure 18: The dependency structure in Figure 15 with its head vector written below the position numbers.

Constructing a tree with a head vector with LAL will produce a rooted tree such that the word at position $i = 1, 2, 3, \dots$ corresponds to vertex with index $i - 1 = 0, 1, 2, \dots$. This remark is important to be able to understand more advanced concepts and uses within LAL.

2.3 Reading dependency structures from a file

Now follow a few examples of reading a tree structure from a file. The Linear Arrangement Library can parse a sentence's structure from a file in two different formats: as an edge list, or as a head vector of the sentence.

For a file, say `edge_list.txt`, that contains an edge list of a tree structure, we use the `read_edge_list` function in the `io` module (Code 2.1).

```
import lal
rt = lal.io.read_edge_list("rooted_tree", "path/to/edge_list.txt")
```

```
2 0
2 1
2 4
4 3
4 6
6 5
6 9
9 7
9 8
```

Code 2.1: Top: reading an edge list from a file. Bottom: contents of `path/to/edge_list.txt`.

For a file, say `head_vector.txt`, that contains a head vector of a dependency structure, we use the `read_head_vector` function in the `io` module (Code 2.2).

```
import lal
rt = lal.io.read_head_vector("rooted_tree", "path/to/head_vector.txt")
```

```
3 3 0 5 3 7 5 10 10 7
```

Code 2.2: Top: reading a head vector from a file. Bottom: contents of `path/to/head_vector.txt`.

2.4 Making dependency structures manually

Creating a tree manually can be done using an edge list or using a head vector.

2.4.1 Using an edge list

This consists of two steps. First, we must know the amount of words (or vertices) of the tree. First, we declare the variable that stores our tree; for this we need to use the number of vertices the tree will have (Code 2.3).

```
# initialise a 10-vertex rooted tree
rt = lal.graphs.rooted_tree(10)
```

Code 2.3: Declaring and initializing a rooted tree.

We then set the edges to the tree. This can be done in several ways; the easiest of them is by constructing a Python list with the edges as pairs in it. It is paramount that for an n -vertex tree we create a list of $n - 1$ edges, i.e., users must make a list that has one edge less than the number of words the tree has. We first declare a list `list_of_edges` with the edges of the tree in Figure 16, and then we set the tree's edges (Code 2.4).

```
edge_list = [ (2,0),(2,1),(2,4), (4,3),(4,6), (6,5),(6,9), (9,7),(9,8) ]
rt.set_edges(edge_list)
```

Code 2.4: Setting the edges to a rooted tree.

2.4.2 Using a head vector

Constructing a tree using a head vector (Code 2.5) is slightly easier than constructing it with an edge list. A rooted tree can be constructed from a head vector in a single line. We can use the head vector for the sentence in Figure 15 (recall it is given in Figure 18) to construct the sentence's tree.

```
rt = lal.graphs.from_head_vector_to_rooted_tree([3,3,0,5,3,7,5,10,10,7])
```

Code 2.5: Constructing a rooted tree from a head vector.

When using a head vector to construct a tree the words will be labeled with the position they are in the sentence. Therefore, the third number in the head vector, '0', corresponds to the word *was*.

Unsure users of the nature of head vectors may be afraid to misconstrue them. To check for errors, users can make use of two handy functions (Code 2.6).

```
errorlist = lal.io.check_correctness_head_vector([3,8,5,8,7,8,0])
print(errorlist)
errorlist = lal.io.check_correctness_head_vector("3 8 5 8 7 8 0")
print(errorlist)
```

Code 2.6: Checking correctness of a head vector.

2.4.3 Ensuring we are constructing trees correctly

We can check the correctness of the result by viewing the contents of our tree (Code 2.7). We can either print the list of edges (#1) or print the tree in a more obvious format (#2)

```
print(rt.get_edges()) #1
print(rt) #2
```

Code 2.7: Printing the contents of a tree.

The output of (#1) will be simply a python list whose contents are pairs of integer values; each pair represents an edge of the tree and for rooted trees the direction of the edge always goes from left to right. The output of (#2) has two sections: in the first section, for every vertex u , we will see the list of vertices v to which vertex u points, i.e., the edges (u, v) . In the second section we will see, for every vertex u , the vertex v vertex u is pointed by. We will also see an asterisk '*' which indicates the root vertex. For the example in Figure 15 constructed with its head vector (as in Code 2.5) the output of (#2) is as shown in Code 2.8.

```

out:
  0:
  1:
*2: 0 1 4
  3:
  4: 3 6
  5:
  6: 5 9
  7:
  8:
  9: 7 8

```

```

in:
  0: 2
  1: 2
*2:
  3: 4
  4: 2
  5: 6
  6: 4
  7: 9
  8: 9
  9: 6

```

Code 2.8: The output of ‘#2’ in Code 2.7 for the tree constructed in Code 2.5.

2.5 Calculating metrics on dependency structures

The Linear Arrangement Library offers many possibilities regarding calculations on dependency structures of sentences. All but one of the calculations explained in this section are heavily dependent on the concept of *linear arrangement*. A linear arrangement relates vertices of a tree and the positions which these words should occupy in the sentence. If we label the words using the indices in Figure 16, the linear arrangement that maps those words to the positions they have in the sentence in Figure 15 is the one given in Code 2.9. See Table 1.

Word	Position (+1)	Position (starting at 0)
Yesterday	1	0
John	2	1
saw	3	2
a	4	3
dog	5	4
which	6	5
was	7	6
a	8	7
Yorkshire	9	8
Terrier	10	9

Table 1: The position of each word in the sentence in Figure 15.

```

# index of word:                                0 1 2 3 4 5 6 7 8 9
linear_arrangement = lal.types.linear_arrangement([0,1,2,3,4,5,6,7,8,9])
#
# This number represents the position of the word corresponding to index 2,
# i.e., word 'was'. Word 'was' is mapped to position '2' in the sentence
# (recall that the first position is 0 -- Damn Computer Scientists!)
print(linear_arrangement)

```

```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9) | (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Code 2.9: An example of linear arrangement. It maps the words indexed as in Figure 16 to the positions indicated in the list so that the vertices are placed in the sentence as in Figure 15. The output consists of (linear arrangement | inverse linear arrangement).

Now, let's say we want to scramble the words of the sentence in Figure 15 as specified in Table 2.

Index of word	Word	Position (+1)	Position (starting at 0)
0	Yesterday	9	8
1	John	4	3
2	saw	8	7
3	a	1	0
4	dog	3	2
5	which	2	1
6	was	7	6
7	a	5	4
8	Yorkshire	10	9
9	Terrier	6	5

Table 2: The words of the sentence in Figure 15 scrambled randomly.

For this we change the specification of the linear arrangement as shown in Code 2.10. Said code shows two different ways of instantiating a linear arrangement. The first shows how to construct the arrangement by specifying the position each word is mapped to. The second uses the distribution of the words, that is, an array that is interpreted as the sequence of words in the linear ordering.

```
# index of word:                                0 1 2 3 4 5 6 7 8 9
arrangement = lal.types.linear_arrangement.from_direct([8,3,7,0,2,1,6,4,9,5])
print(arrangement)
#
arrangement = lal.types.linear_arrangement.from_inverse([3,5,4,1,7,9,6,2,0,8])
print(arrangement)
```

```
(8, 3, 7, 0, 2, 1, 6, 4, 9, 5) | (3, 5, 4, 1, 7, 9, 6, 2, 0, 8)
(8, 3, 7, 0, 2, 1, 6, 4, 9, 5) | (3, 5, 4, 1, 7, 9, 6, 2, 0, 8)
```

Code 2.10: Two examples of linear arrangement instantiation. Both instantiations construct the same linear arrangement, hence producing the same output in the `print` command.

2.5.1 The sum of syntactic dependency distances and its minimum

A popular metric in Cognitive Science and Quantitative Linguistics is the sum of dependency distances D (also known as sum of dependency lengths). Once a tree is constructed, we can calculate the sum of dependency distances in a single line. For example, we can construct the tree in Figure 15 using its head vector in Figure 18 (recall Code 2.5), and calculate the sum of edge lengths in that sentence as follows.

```
rt = lal.graphs.from_head_vector_to_rooted_tree([3,3,0,5,3,7,5,10,10,7])
D = lal.linarr.sum_edge_lengths(rt)
```

Code 2.11: Calculating D , the sum of dependency distances with the implicit linear arrangement.

Notice that in Code 2.11 we have not specified explicitly the positions of the words of the tree `rt` with a linear arrangement. That is because, in that case, the function we are calling uses the indices of the words as positions in the sentence. But we could also construct the same tree using the indices for the words in Table 1. Code 2.12 shows how to calculate D supplying different linear arrangements. These arrangements are interpreted to be direct arrangements (see Code 2.10).

```
# build the tree using the head vector
rt = lal.graphs.from_head_vector_to_rooted_tree([3,3,0,5,3,7,5,10,10,7])
# calculate D using explicitly the original linear arrangement
D1 = lal.linarr.sum_edge_lengths(rt, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# here we calculate D using a linear arrangement that puts the word
# 'Yesterday' at the end of the sentence
D2 = lal.linarr.sum_edge_lengths(rt, [9, 0, 1, 2, 3, 4, 5, 6, 7, 8])
assert(D1 <= D2)
# now using the inverse linear arrangement
D3 = lal.linarr.sum_edge_lengths(rt, [9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
# the sums of edge lengths are the same in the first and third examples are equal
assert(D1 == D3)
```

Code 2.12: Calculating D , the sum of dependency distances, with *explicit* linear arrangements.

The examples above (Codes 2.11 and 2.12) illustrate two kinds of linear arrangements. First, *implicit* arrangements, that are defined by the numbers used to label the vertices of the tree as in Code 2.11. These numbers are interpreted as vertex position. Second, *explicit* arrangements, that are defined by vectors that indicate the position of every vertex as in Code 2.12.

We can also calculate the minimum sum of dependency distances of a rooted tree `rt` when we do not consider any constraint possible on the positions of the words. That is, to say, we can answer the haunting questions:

1. “If we could arrange the words of a sentence *in any way possible*, what would be a way to do so such that it produces the minimum value of D for said sentence?”
2. “If we could arrange the words of a sentence not in any way possible, but constrained to *planarity*, what would be a way to do so such that it produces the minimum value of D of said sentence under such constraint?”
3. “If we could arrange the words of a sentence not in any way possible, but constrained to *projectivity*, what would be a way to do so such that it produces the minimum value of D of said sentence under such constraint?”

Mathematicians and Computer Scientists before us answered this question, and thus we implemented algorithms to answer these questions (for question (1) see [2, 3], for question (2) see [18, 5, 4], and for question (3) see [18, 8]). We can obtain answers for these questions using the Linear Arrangement Library by writing one single line of code for each of them. The minimum arrangements calculated in Code 2.13 are depicted in Figure 20.

```
# Question 1
min_unc, arr_unc = lal.linarr.min_sum_edge_lengths(rt)
print("Minimum arrangement:      ", arr_unc)
# Question 2
min_plan, arr_plan = lal.linarr.min_sum_edge_lengths_planar(rt)
print("Minimum planar arrangement:  ", arr_plan)
# Question 3
min_proj, arr_proj = lal.linarr.min_sum_edge_lengths_projective(rt)
print("Minimum projective arrangement:", arr_proj)
# by definition we have...
assert(min_unc <= min_plan and min_plan <= min_proj)
```

```
Minimum arrangement:      (8,9,7,6,5,4,3,2,0,1) | (8,9,7,6,5,4,3,2,0,1)
Minimum planar arrangement: (9,7,8,5,6,4,3,2,0,1) | (8,9,7,6,5,3,4,1,2,0)
Minimum projective arrangement: (7,9,8,6,5,4,3,2,0,1) | (8,9,7,6,5,4,3,0,2,1)
```

Code 2.13: Calculating D_{min} under different constraints: (1) No constraint, (2) Planarity constraint, (3) Projectivity constraint. Output has been slightly modified to fit it in the box.

The variables `arr_*` store a linear arrangement that attains the minimum value of D stored in their respective variables `min_*`. Notice this is just one of the arrangements that minimize D : there can be many arrangements minimizing D . If we wanted to ensure that the arrangements `arr_*` indeed yield the sum of edge lengths that is *promised* by the values stored in variables `min_*` we can write the last two lines in Code 2.13. It goes without saying that the arrangements calculated by the two algorithms need not be the same. In either case the minimum value of the sum of edge lengths returned is $D = 13$, and the arrangements are depicted in Figure 20.

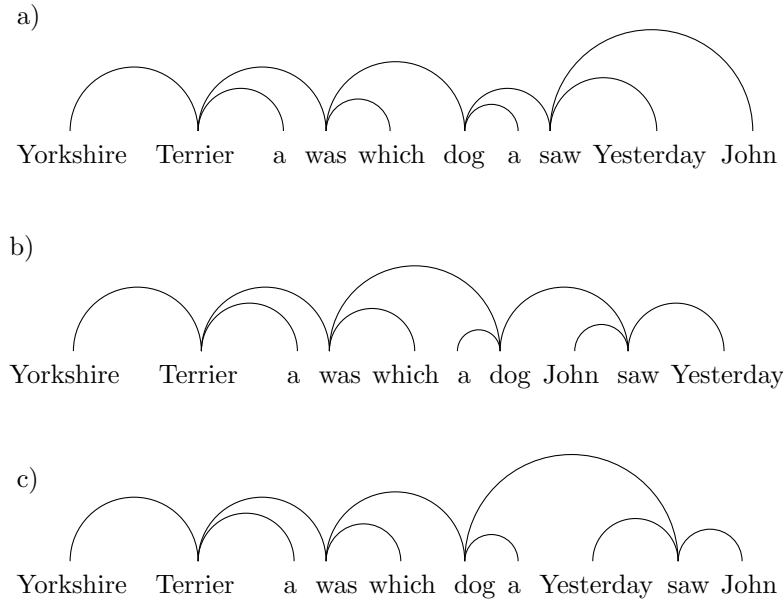


Figure 20: Arrangements that yield a sum of edge lengths $D = 13$. a) Arrangement returned by ‘#1’ in Code 2.13. b) Arrangement returned by ‘#2’ in Code 2.13. c) Arrangement returned by ‘#3’ in Code 2.13.

2.5.2 Proportion of head initial dependencies

The head initial of a sentence is the ratio of links in the dependency structure that point rightwards over the total number of links [9]. In the sentence in Figure 15, such value is exactly $3/9 = 1/3 = 0.333\dots$ and is calculated with the library in the single line shown in Code 2.14.

```
head_initial = lal.linarr.head_initial(rt)
```

Code 2.14: Calculating the head initial ratio of a tree.

2.5.3 Dependency flux

Dependency flux was devised by Kahane *et. al.* [13] with the aim to capture the cognitive processing cost of a sentence. A sentence has several dependency fluxes, each located between every pair of consecutive words. Therefore, in a sentence of n words we find $n - 1$ different fluxes. A flux is made up of the following concepts:

- Set of dependencies: each flux is made up of a set of dependencies crossing over the position of the flux.
- Left/Right span: this is the number of words to the left/right of this flux which are a word of a dependency in the flux.
- Weight: this is defined as the size of the largest subset of disjoint dependencies in the flux’s set of dependencies.

This is best explained with an example. In Figure 21 we have drawn three vertical lines to denote 3 different fluxes in the sentence. It is easy to see that the first flux, located between **Yesterday** and **John**,

- Has only one dependency in its set, {**saw**, **Yesterday**},
- Its left span is 1 (**Yesterday**), and so is its right span (**saw**),
- The weight is 1.

The 6th and 9th fluxes are slightly different. The 6th flux

- Has two dependencies {**dog**, **was**}, {**was**, **which**},
- Has left span size 2, and right span 1,
- Has weight 1.

Finally, the 9th flux

- Has three dependencies {**was**, **Terrier**}, {**a**, **Terrier**}, {**Yorkshire**, **Terrier**},
- Has left span size 3, and right span 1,
- Has weight 1.

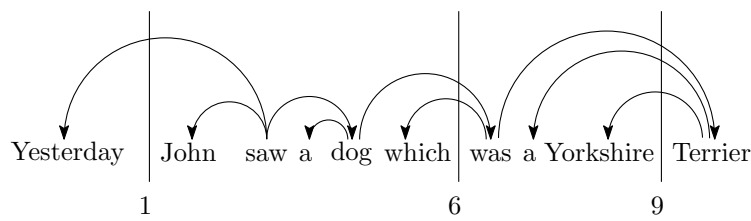


Figure 21: This sentence has 9 dependency fluxes; we have marked the first, the sixth and the ninth of them.

The information listed above are properties that are derived directly from the structure of the sentence. We can also derive other values such as the R/L and the W/S ratios [13]. The first ratio is the right span divided by the left span ratio. This ratio should be higher than 1 in head-initial languages and head-final languages should have a R/L ratio less than 1 [13]. The second ratio is the weight divided by the size, also called *density*. As pointed out by Kahane *et. al*, the W/S “measures the proportion of bouquets in the flux” [13].

We can calculate any tree’s dependency fluxes without much effort using the library (Code 2.15).

```
# calculate the fluxes
fluxes = lal.linarr.compute_flux(rt)
number_of_fluxes = len(fluxes)
# print the fluxes
for i in range(0, number_of_fluxes):
    print("Flux", i+1)
    print("  Dependencies:", fluxes[i].get_dependencies())
    print("  Left span:", fluxes[i].get_left_span())
    print("  Right span:", fluxes[i].get_right_span())
    print("  Weight:", fluxes[i].get_weight())
    print("  R/L ratio:", fluxes[i].get_RL_ratio())
    print("  W/S ratio:", fluxes[i].get_WS_ratio())
```

```

Flux 1
  Dependencies: ((0, 2),)
  Left span: 1
  Right span: 1
  Weight: 1
  R/L ratio: 1.0
  W/S ratio: 1.0
# ...
Flux 6
  Dependencies: ((4, 6), (5, 6))
  Left span: 2
  Right span: 1
  Weight: 1
  R/L ratio: 0.5
  W/S ratio: 0.5
# ...
Flux 9
  Dependencies: ((6, 9), (7, 9), (8, 9))
  Left span: 3
  Right span: 1
  Weight: 1
  R/L ratio: 0.3333333333333333
  W/S ratio: 0.3333333333333333

```

Code 2.15: Calculating the dependency flux of a tree and the output (we have omitted many fluxes since they do not offer much insight).

2.5.4 Mean Hierarchical distance

The mean hierarchical distance [10] is the only calculation in this section that does not rely on the linear arrangement in any way. Recall the tree structure in Figure 15. The mean hierarchical distance is the average sum of the topological distance of every word to the sentence's root. For example, the topological distance between the word **saw** and the word **Terrier** is exactly 3, and so is the distance between **saw** and **which**. The Mean Hierarchical Distance for that tree is then $7/3 = 2.333\dots$. This is calculated with the Linear Arrangement Library in the single line shown in Code 2.16.

```

mhd = lal.properties.mean_hierarchical_distance(rt)

```

Code 2.16: Calculating the Mean Hierarchical Distance for a rooted tree `rt`.

Notice that this distance indeed does not depend on the positions of the words in the sentence, since the distance is measured in the number of links needed to traverse in order to go from one word to another.

2.5.5 Omega (Ω)

Omega was defined in [19] as

$$\Omega = \frac{D_r - D}{D_r - D_m}. \quad (1)$$

In Equation 1, D denotes the sum of syntactic dependency distances (Section 2.5.1) and D_m denotes the minimum sum of edge lengths (Section 2.5.1). D_r denotes the expected value of D we would find in a uniformly random permutation of the words in the sentence [6]. The calculation of Ω for a given syntactic dependency tree (in the form of a rooted tree) can be done using Code 2.17.

```
D_r    = lal.properties.exp_sum_edge_lengths(rt)
D_m, _ = lal.linarr.min_sum_edge_lengths(rt)
D      = lal.linarr.sum_edge_lengths(rt)
Omega  = (D_r - D)/(D_r - D_m)
```

Code 2.17: Calculating Ω for a rooted tree `rt`.

3 Working with a single treebank

The section above shows how to work with only one single tree: it shows how to compute a few given metrics and see the results. This methodology, however, does not allow for an agile study of a series of trees, i.e. a *treebank*, as it is customary in quantitative dependency syntax [14]. Therefore, this library provides its users with automatic tools to process treebanks, which is simply a file with a series of *head vectors* (see Sections 2.2, 2.3 and 2.4.2). For example, the first few trees (as head vectors) of the UD 2.5 treebank for Chinese are:

```
6 1 6 5 6 8 6 12 12 11 9 0 12 16 14 12
2 0 4 2 4 5 4 7
7 7 7 5 3 3 0
9 9 6 6 4 2 8 9 11 11 0 11 17 15 17 17 22 17 17 22 22 11 24 26 26 22 26
2 0 5 3 2 2 6 9 2 9 13 11 9
3 3 15 5 3 7 9 7 5 15 15 15 15 13 0
7 3 5 5 7 7 0 7 7
6 6 4 6 6 0
```

Figure 22: Example of a treebank file `Chinese.txt`.

The whole Chinese treebank for Chinese in the form of head vector can be download from [26].

3.1 Before jumping to the analysis of treebanks

The library provides automatic tools for processing a single treebank file. These files, however, are not constructed by the library and, therefore, must be made by the user either by hand or using alternative software. Either alternatives are error prone and therefore the result must be checked prior to using the library in order to avoid erroneous results and unexpected crashes in the execution of your programs. The correctness of a treebank file can be easily done in a single line. Taking the tree in the example in Figure 22 as the contents of a treebank example file `correct_treebank.txt`, the Python code we write is given in Code 3.1.

```
import lal
errlist = lal.io.check_correctness_treebank("correct_treebank.txt")
for err in errlist:
    print(err)
```

```
()
```

Code 3.1: Ensuring correctness of a treebank file `correct_treebank.txt`, and the output of the program.

As we can see, the output of the previous program is a simple `()` indicating that there are no errors whatsoever. In a treebank whose trees (as head vectors) contain errors, we will see a different output. Take, for example, the contents of the treebank file shown in Figure 23.

```
1  how are you
2  0 -1 0
3  -1 -2 -3
4  2 0 z 2 a 2 2 2
5  2 0 2 2    a 2 2 2
6  0 0 0 0 a 0 0 0
7  0 0 0 0 0 b 0 0
8  2 0 0 2 2 2
9  8 8 8 8 0 9 8 9 10 0 9 16 14 8 16 8
10 0 2 2 2 2 2 8 1 10 8
11 4 4 4 5 25 10 10 7 10 5 5 13 12 13 21 21 21 21 21 14 25 25 25 0 25 25 33 33
12 4 4 4 5 25 10 10 7 10 5 5 13 12 13 21 21 21 21 21 14 25 25 25 0 25 25
```

Figure 23: Example of a treebank file containing errors `incorrect_treebank.txt`.

Obviously, neither of the first 8 head vectors are formatted correctly, since they contain letters, or more root vertices (the number 0) than it is allowed. What about the 9th tree? That tree also contains more roots than it is allowed: it contains two zeros. It is a bit more complicated to tell why the 10th is incorrect: the second value in the sequence (the first 2) indicates a self-loop, i.e., the parent vertex of the vertex at position 2 is vertex 2, i.e., itself. What about the 11th and 12th trees? In those trees errors are almost invisible to the naked eye. To begin with, the last two values in the 11th line (the last two 33) indicate parents outside the range of vertices, in other words, the parent of the vertices at positions 27 and 28 indicate parents that do not exist in the sequence: there are only 28 vertices but the parents of the last two vertices is the vertex 33. Even the authors of the library had problems spotting the error in the 12th head vector: there is a cycle, but it is very small. If we take a close look at that vector, we see that the vertex at position 12 has as parent vertex 13, and the vertex at position 13 has as parent the vertex at position 12. This is not allowed. The program that checks the correctness of a file `incorrect_treebank.txt` and its output are given in Code 3.2.

```
import lal
errlist = lal.io.check_correctness_treebank("incorrect_treebank.txt")
for err in errlist: print(err)
```

```
(1, Error: Value at position '1' (value: 'how') is not a valid non-negative integer number.),
(1, Error: Value at position '2' (value: 'are') is not a valid non-negative integer number.),
(1, Error: Value at position '3' (value: 'you') is not a valid non-negative integer number.),
(2, Error: Value at position '2' (value: '-1') is not a valid non-negative integer number.),
(3, Error: Value at position '1' (value: '-1') is not a valid non-negative integer number.),
(3, Error: Value at position '2' (value: '-2') is not a valid non-negative integer number.),
(3, Error: Value at position '3' (value: '-3') is not a valid non-negative integer number.),
(4, Error: Value at position '3' (value: 'z') is not a valid non-negative integer number.),
(4, Error: Value at position '5' (value: 'a') is not a valid non-negative integer number.),
(5, Error: Value at position '5' (value: 'a') is not a valid non-negative integer number.),
(6, Error: Value at position '5' (value: 'a') is not a valid non-negative integer number.),
(7, Error: Value at position '6' (value: 'b') is not a valid non-negative integer number.),
(8, Error: Wrong number of roots: 2.),
(8, Error: Vertex '2' is isolated.),
(8, Error: Wrong number of edges. Number of vertices is '6'. Number of edges is '4'; should be '5'.),
(9, Error: Wrong number of roots: 2.),
(9, Error: Vertex '4' is isolated.),
(9, Error: Wrong number of edges. Number of vertices is '16'. Number of edges is '14'; should be '15'.),
(10, Error: found a self-loop at position '2'.),
(11, Error: Number at position '27' (value: 33) is out of bounds.),
(11, Error: Number at position '28' (value: 33) is out of bounds.),
(12, Error: The graph described is not a tree, i.e., it has cycles.)
```

Code 3.2: Ensuring correctness of a treebank file `incorrect_treebank.txt`, and the output of the program.

3.2 Analyzing a treebank file automatically

The Linear Arrangement Library offers its users a wide variety of metrics/scores to be computed on trees. The calculation of *all* of them for every tree in a treebank is almost trivial (Code 3.3).

```
import lal
err = lal.io.process_treebank("Chinese.txt", "output_file.csv")
print(err)
```

Code 3.3: Processing the `Chinese.txt` treebank file as explained in the introduction (Code 1.1). The contents of said file are partially reproduced in Figure 22.

The one liner in Code 3.3 will compute every metric from a broad set of scores that are essential for research in Quantitative Linguistics listed in Table 3 for every tree in a treebank file. Therefore, it will produce a `.csv` file with as many rows as sentences in the treebank plus an initial row for the header with labels defined in Table 3; the number of columns depends on the quantity of columns that every feature spans: most features span just one column, but some of them span several. Nevertheless, it is a bit too simple since it does not allow for a finer-grained processing of treebanks, as it is just a wrapper of Code 3.4 which directly uses the `treebank_processor` class.

```
import lal
tbproc = lal.io.treebank_processor()
# The object will compute all features by default
err = tbproc.init("Chinese.txt", "output_file.csv")
# Process the treebank file...
if err == lal.io.treebank_error_type.no_error: err = tbproc.process()
print(err)
```

Code 3.4: Processing the `Chinese.txt` treebank file. The contents of said file are partially reproduced in Figure 22.

Table 3: The list of scores computable by the treebank processor object. These scores are given as a “feature” available in the `lal.io.treebank.feature` enumeration. Most scores span over one column, but there are a few that span over 2 or more columns.

Feature	Score calculated
STRUCTURAL PROPERTIES OF TREES	
<code>num_nodes</code>	Number of nodes of the tree
<code>second_moment_degree</code>	Second moment of degree about zero $\langle k^2 \rangle$
<code>second_moment_degree_in</code>	Second moment of in-degree about zero $\langle k_{in}^2 \rangle$
<code>second_moment_degree_out</code>	Second moment of out-degree about zero $\langle k_{out}^2 \rangle$
<code>third_moment_degree</code>	Third moment of degree about zero $\langle k^3 \rangle$
<code>third_moment_degree_in</code>	Third moment of in-degree about zero $\langle k_{in}^3 \rangle$
<code>third_moment_degree_out</code>	Third moment of out-degree about zero $\langle k_{out}^3 \rangle$
<code>num_pairs_independent_edges</code>	Amount of pairs of independent edges
<code>hubiness</code>	Hubiness of the tree
<code>mean_hierarchical_distance</code>	Mean hierarchical distance (MHD)
<code>tree_centre</code>	The two central vertices of the tree. Spans two columns
<code>tree_centroid</code>	The two centroidal vertices of the tree. Spans two columns
<code>tree_diameter</code>	The diameter of the tree
<code>tree_caterpillar_distance</code>	The size of the maximum spanning caterpillar
<code>tree_type</code>	The type of tree. This spans as many columns as tree types the library can recognize. LAL can currently classify trees into the following categories: <code>star</code> , <code>bistar</code> , <code>caterpillar</code> [1], <code>linear</code> , <code>quasistar</code> , <code>spider</code> [15, 23], <code>two_linear</code> [17] (known in the literature as 2-linear)
NUMBER OF CROSSINGS	
<code>num_crossings</code>	The number of edge crossings C in the sentence
<code>predicted_num_crossings</code>	The predicted number of crossings, $\mathbb{E}_2[C]$
<code>exp_num_crossings</code>	Expected value of number of crossings, $\mathbb{E}[C]$
<code>var_num_crossings</code>	Variance of the number of crossings, $\mathbb{V}[C]$
<code>z_score_num_crossings</code>	z -score of the number of crossings, $C_z = (C - \mathbb{E}[C]) / \sqrt{\mathbb{V}[C]}$
SUM OF EDGE LENGTHS	
<code>sum_edge_lengths</code>	The sum of edge lengths, D
<code>exp_sum_edge_lengths</code>	Expected value of the sum of edge lengths, $\mathbb{E}[D]$
<code>exp_sum_edge_lengths_projective</code>	Expected value of D in projective arrangements, $\mathbb{E}_{pr}[D]$
<code>exp_sum_edge_lengths_planar</code>	Expected value of D in planar arrangements, $\mathbb{E}_{pl}[D]$
<code>var_sum_edge_lengths</code>	Variance of the sum of the length of edges, $\mathbb{V}[D]$
<code>z_score_sum_edge_lengths</code>	z -score of the sum of the length of edges, $D_z = (D - \mathbb{E}[D]) / \sqrt{\mathbb{V}[D]}$
<code>min_sum_edge_lengths</code>	Minimum value of D
<code>min_sum_edge_lengths_planar</code>	Minimum value of D under planarity
<code>min_sum_edge_lengths_projective</code>	Minimum value of D under projectivity
<code>max_sum_edge_lengths_planar</code>	Maximum value of D under planarity
<code>max_sum_edge_lengths_projective</code>	Maximum value of D under projectivity
<code>mean_dependency_distance</code>	Mean dependency distance (MDD)
DEPENDENCY FLUXES	

Table 3: The list of scores computable by the treebank processor object. These scores are given as a “feature” available in the `lal.io.treebank.feature` enumeration. Most scores span over one column, but there are a few that span over 2 or more columns.

Feature	Score calculated
Maximum, average, and mean values over all fluxes in the sentence	
<code>flux_max_weight</code>	Maximum flux weight
<code>flux_mean_weight</code>	Mean flux weight
<code>flux_min_weight</code>	Minimum flux weight
<code>flux_max_left_span</code>	Maximum flux left span
<code>flux_mean_left_span</code>	Mean flux left span
<code>flux_min_left_span</code>	Minimum flux left span
<code>flux_max_right_span</code>	Maximum flux right span
<code>flux_mean_right_span</code>	Mean flux right span
<code>flux_min_right_span</code>	Minimum flux right span
<code>flux_max_size</code>	Maximum flux size
<code>flux_mean_size</code>	Mean flux size
<code>flux_min_size</code>	Minimum flux size
<code>flux_max_RL_ratio</code>	Maximum flux R/L ratio
<code>flux_mean_RL_ratio</code>	Mean flux R/L ratio
<code>flux_min_RL_ratio</code>	Minimum flux R/L ratio
<code>flux_max_WS_ratio</code>	Maximum flux W/S ratio
<code>flux_mean_WS_ratio</code>	Mean flux W/S ratio
<code>flux_min_WS_ratio</code>	Minimum flux W/S ratio
OTHER PROPERTIES	
<code>head_initial</code>	Headedness of the tree
<code>dependency_structure_type</code>	<p>The type of syntactic dependency structure of the sentence. This spans as many columns as tree types the library can recognize. LAL can currently classify syntactic dependency structures into the following types:</p> <ul style="list-style-type: none"> - projective - planar - 1EC (1-Endpoint crossing) - WG1 (Well-nested with gap degree 1)

3.2.1 Calculating Ω in a treebank

Notice that Table 3 does not contain an entry for Omega (Section 2.5.5). Fortunately, the calculation of Omega can be easily done with a simple office program that can manipulate spreadsheets. Using your favourite office program³ make a new column (called, say, “Omega”), add the formula that corresponds to Equation 1 using the following existing columns

- D is in the column `sum_edge_lengths`,
- D_r is in the column `exp_sum_edge_lengths`,
- D_m is in the column `min_sum_edge_lengths`,

as shown in Figure 24.

³We suggest LibreOffice <https://www.libreoffice.org/>.

	A	B	C	D	E
1	n	sum_edge_lengths	exp_sum_edge_lengths	min_sum_edge_lengths	Omega
2	16	32	85	24	$=(C2-B2)/(C2-D2)$
3	8	10	21	10	
4	7	21	16	8	
5	27	70	242.667	48	
6	13	28	56	17	
7	15	41	74.6667	22	
8	9	16	26.6667	13	
9	6	13	11.6667	7	

Figure 24: Screenshot of the calculation of Omega using Libre Office Calc by extending the output of Code 3.4. Only four columns are shown due to lack of space.

3.3 Custom analysis of a treebank file

As explained before, Code 3.4 computes the full set of metrics. However, notice that computing all metrics can make the resulting file too large, and the computation time a bit long. Because of this, we can choose the exact metrics that should be computed. For this, we have mainly two possibilities to do so depending on the amount of metrics to be computed. If the amount of metrics is small then it is best to start with an empty treebank processor and add the desired features (Code 3.5), and if the amount of features is large, we start with a full treebank processor and remove the metrics we are not interested in (Code 3.6).

```
import lal
tbproc = lal.io.treebank_processor()
# The object will compute all features by default
err = tbproc.init("Chinese.txt", "output_file.csv")
if err == lal.io.treebank_error_type.no_error:
    # Remove all the features
    tbproc.clear_features()
    # Now we add some metrics
    tbproc.add_feature( lal.io.treebank_feature.num_nodes )
    tbproc.add_feature( lal.io.treebank_feature.mean_hierarchical_distance )
    tbproc.add_feature( lal.io.treebank_feature.min_sum_edge_lengths )
    tbproc.add_feature( lal.io.treebank_feature.min_sum_edge_lengths_planar )
    # ...
    # Process the treebank file...
    err = tbproc.process()
print(err)
```


n	mean_hierarchical_distance	min_sum_edge_lengths	min_sum_edge_lengths_planar
16	2.4	24	24
8	2	10	10
7	1.66667	8	8
27	2.65385	48	48
13	2	17	17
15	2.35714	22	22
9	1.5	13	13
6	1.2	7	7

Code 3.5: Processing the `Chinese.txt` treebank file with very few features. The contents of said treebank file are partially reproduced in Figure 22; the output given corresponds to those trees.

```
import lal
tbproc = lal.io.treebank_processor()
# The object will compute all features by default
err = tbproc.init("Chinese.txt", "Chinese_output.csv")
if err == lal.io.treebank_error_type.no_error:
    # Now we remove the metrics we are not interested in
    tbproc.remove_feature( lal.io.treebank_feature.tree_centre )
    tbproc.remove_feature( lal.io.treebank_feature.tree_diameter )
    # ...
    # Process the treebank file...
    err = tbproc.process()
print(err)
```

Code 3.6: Processing the `Chinese.txt` treebank file with less features than those computed by default. The contents of said treebank file are partially reproduced in Figure 22.

One can further tune the processing of a treebank by renaming the columns (Code 3.7).

```

import lal
tbproc = lal.io.treebank_processor()
# The object will compute all features by default
err = tbproc.init("Chinese.txt", "output_file.csv")
if err == lal.io.treebank_error_type.no_error:
    # Shorthand helper
    tf = lal.io.treebank_feature
    # Remove all the features
    tbproc.clear_features()
    # Now we add some metrics
    tbproc.add_feature( tf.num_nodes )
    tbproc.add_feature( tf.mean_hierarchical_distance )
    tbproc.add_feature( tf.min_sum_edge_lengths )
    tbproc.add_feature( tf.max_sum_edge_lengths_planar )
    # ...
    # Rename some columns
    tbproc.set_column_name( tf.num_nodes, "n" )
    tbproc.set_column_name( tf.mean_hierarchical_distance, "MHD" )
    tbproc.set_column_name( tf.min_sum_edge_lengths, "Dmin" )
    tbproc.set_column_name( tf.max_sum_edge_lengths_planar, "DMax_planar" )
    # Process the treebank file...
    err = tbproc.process()
print(err)

```

n	MHD	Dmin	DMax_planar
16	2.4	24	105
8	2	10	26
7	1.66667	8	21
27	2.65385	48	305
13	2	17	75
15	2.35714	22	105
9	1.5	13	36
6	1.2	7	15

Code 3.7: Processing the `Chinese.txt` treebank file with very few features; the columns have been renamed. The contents of said treebank file are partially reproduced in Figure 22; the output given corresponds to those trees.

3.4 Notes on corner cases

Most of the scores in Table 3 are non-negative integer values (whole numbers larger than or equal to 0) which are defined on all trees and graphs. However, a few of them are rational numbers which do not take a well-defined value under certain circumstances. For example, the mean dependency distance is not well defined for the only one-vertex tree, since the score would be calculated as $0/0$. Table 4 lists the scores for which there undefined cases and the output produced by the `treebank_processor` class.

Score	Corner case	Processor's output
hubiness	$n \leq 3$	nan
MDD	$m = 0$	nan
MHD	$n \leq 1$	nan
(min, mean, max) fluxes	$n \leq 1$	nan
head initial	$m = 0$	nan

Table 4: Corner cases to always bear in mind. The scores in the left-most column are not defined in the cases listed in the second column (n denotes the number of vertices – words, and m denotes the number of edges – dependencies). The third indicates the value output by said class, where ‘nan’ indicates ‘Not A Number’. The cases $m = 0$ happen in trees when $n \leq 1$.

3.5 Correctness outside treebanks

Many users will create their own data sets with perhaps head vectors in them scattered over several files. One way to check whether or not those head vectors are correct is to write a small script to output those head vectors into a file, run LAL as explained above and locate the erroneous head vectors (if any) in the original data files. Since this may seem daunting given the difficulty of getting back to the original sentence, users can use the two functions presented in Section 2.4.2 (Code 2.6).

4 Working with a treebank collection

More often than not treebanks are part of a whole, e.g., the treebanks of the Universal Dependencies collection, the set of treebanks of the novels of a particular author, etc. Such a collection of treebanks is referred to as a *treebank collection*. In order to create a treebank collection we only need to put all the treebank files within the same collection into a folder, and create a single text file, referred to as the *main file* of the collection, which lists all treebank files within the collection and gives them an identifier. The main file must be located within the same directory as the folder containing all the treebanks. The format of such a file is very simple. Each of its lines must have two columns: the first must contain an identifier of the treebank, for example, an ISO code of a language. The second column must contain the name of the folder of the treebanks and the treebank file. When working with a treebank collection users must remember the location and name of the *main file*.

For example, suppose that we have a collection of 90 treebank files of the Universal Dependencies 2.5 treebank, which we put inside the folder *ud25*. The contents of the main file for this collection could be those given in Figure 25.

```
afr ud25/Afrikaans.txt
akk ud25/Akkadian.txt
amh ud25/Amharic.txt
grc ud25/Ancient_Greek.txt
arb ud25/Arabic.txt
hye ud25/Armenian.txt
```

Figure 25: Partial contents of a main file of a treebank collection.

4.1 Before jumping to the analysis of a treebank collection

Similarly as before, we can ensure that a treebank collection has been generated correctly. This can be done by replicating Code 3.1 as many times as treebanks in our collection. But it is actually much

easier to do when using the Linear Arrangement Library by using the appropriate function (Code 4.1). Recall the main file `ud25.txt` given in Figure 25 at the beginning of this section.

```
import lal
errlist = lal.io.check_correctness_treebank_collection("ud25.txt")
for err in errlist: print(err)
```

Code 4.1: Ensuring correctness of the treebank collection `ud25.txt`.

The output of the `check` function for treebank collections is different from the output of the same function for treebank files. However, it is just a notch more complicated. It has a really easy format: each error is returned as a tuple of four values. The first value is the name of the treebank file where the error occurred. The second is the line number within the main file where the treebank is located at. The third is the line number within the treebank file where the error occurred. The last is the error message. Notice that the last two values of this tuple of four values are the errors as shown in Code 3.2. For example, checking correctness of the main file, say `incorrect_collection.txt` containing a single line (Figure 26) can also be done quite easily (Code 4.2).

```
TEST_FILE incorrect_treebank.txt
```

Figure 26: Contents of the main file for the `incorrect_collection.txt`.

```
import lal
errlist=lal.io.check_correctness_treebank_collection("incorrect_collection.txt")
for err in errlist: print(err)
```

Code 4.2: Ensuring correctness of an incorrect treebank collection `incorrect_collection.txt`.

Code 4.2 will produce an output similar to that in Code 3.2, partially reproduced in the following figure.

```
(incorrect_collection.txt, 1, 1, Error: Value at position '1' (value: 'n') is not a valid non-negative ...)
(incorrect_collection.txt, 1, 1, Error: Value at position '2' (value: 'k2') is not a valid non-negative ...)
(incorrect_collection.txt, 1, 1, Error: Value at position '3' (value: 'Dmin') is not a valid non-negative ...)
(incorrect_collection.txt, 1, 2, Error: Value at position '2' (value: '-1') is not a valid non-negative ...)
...
```

The only differences being in the string `'incorrect_collection.txt, 1'` at the beginning of each line. Notice that this also checks the correctness of the main file.

4.2 Analyzing a treebank collection automatically

Users can analyze automatically a treebank in a similar way as they can do to analyze a single treebank file. The simplest solution is also with the one liner in Code 4.3.

```
import lal
err = lal.io.process_treebank_collection("ud25.txt", "output_directory")
print(err)
```

Code 4.3: The simplest way to process a treebank collection with LAL. If all goes well, this prints 0.

Processing a treebank collection will produce as many files as treebanks in the collection, one output file for every treebank, stored within the specified output directory, and finally will all be joined into a single file. If no output filename is given then the library uses a file name of the form `*_full.txt`, where `*` is the name of the main file. The file that joins all the data will contain an extra column with the identifier of the treebank as specified in the main file (recall the first column in the main file). It goes without saying that the set of features computed is the same for all treebanks. However, for similar reasons explained before in Section 3.2, Code 4.3 is too simple as it is just a wrapper of Code 4.4, which allows a more fine-grained processing of a treebank collection. This time, however, for such a purpose we use the *treebank collection processor* class.

```
import lal
tbcolproc = lal.io.treebank_collection_processor()
# The object will compute all features by default
err = tbcolproc.init("ud25.txt", "output_directory")
if err == lal.io.treebank_error_type.no_error:
    # Process the treebank collection...
    err = tbcolproc.process("output_ud25_joined.csv")
print(err)
```

Code 4.4: Processing the `ud25.txt` treebank collection. If all goes well, this prints 0.

The treebank collection processor checks the correctness of a collection treebank unless it is told not to do so; readers will see in Section 4.4 how to prevent a treebank collection processor to check for correctness. An example of the file containing all the joined output files is given in Figure 27. In that figure we simply depict the first column. The remaining columns are the features calculated, whichever these may be.

treebank	n	second_moment_degree	...
afr	40	5.75	...
afr	29	5.65517	...
afr	50	6.28	...
...			
akk	11	3.63636	...
akk	8	3.5	...
akk	2	1	...
...			

Figure 27: A possible output for the `ud25.txt` treebank collection main file.

4.3 Custom analysis of a treebank collection

A custom analysis of a treebank collection is done in the same way it is done for a single treebank file. We just have to indicate what features are to be added or removed, depending on the case.

```
tbcolproc = lal.io.treebank_collection_processor()
# -----
# (1) Either we start with all metrics and then remove some of them
err = tbcolproc.init("ud25.txt", "output_directory")
if err == lal.treebank_error.no_error:
    tbcolproc.remove_feature(lal.io.treebank_feature.var_num_crossings)
    tbcolproc.remove_feature(lal.io.treebank_feature.var_sum_edge_lengths)
    # ...
# -----
# (2) Or we clear the features and then we add some
err = tbcolproc.init("ud25.txt", "output_directory")
if err == lal.treebank_error.no_error:
    tbcolproc.clear_features()
    tbcolproc.add_feature(lal.io.treebank_feature.num_nodes)
    tbcolproc.add_feature(lal.io.treebank_feature.mean_hierarchical_distance)
    # ...
```

Code 4.5: Adding or removing features to a treebank collection processor.

4.4 Hints on efficiency

In Section 4.3, we have seen how to generate a single output file for a whole collection of treebanks. If that file turns out to be too large, the library allows one to obtain the output in different files. Code 4.6 shows how to stop the `treebank.collection_processor` from joining all the separate output files.

```
import lal
tbcolproc = lal.io.treebank_collection_processor()
# The object will compute all features by default
err = tbcolproc.init("ud25.txt", "output_directory")
if err == lal.io.treebank_error_type.no_error:
    # Do not join the files
    tbcolproc.set_join_files(False)
    # Process the treebank collection...
    err = tbcolproc.process()
print(err)
```

Code 4.6: Do not join the individual files into a single one.

If computations on a collections of treebank are too slow, the library offers two approaches. One is restricting the analysis to the features that are actually needed as explained in the preceding sections (Code 4.5 in Section 4.3). The other approach is taking advantage of parallelizing of computations. The speed of computation should increase proportionally to the number of cores of your computer. For this we simply we specify the number of threads as in Code 4.7.

```

import lal
tbcolproc = lal.io.treebank_collection_processor()
# The object will compute all features by default
err = tbcolproc.init("ud25.txt", "output_directory")
if err == lal.io.treebank_error_type.no_error:
    # Use parallel threads to speed up the computations
    tbcolproc.set_number_threads(4)
    # Process the treebank collection...
    err = tbcolproc.process()
print(err)

```

Code 4.7: We can indicate a number of threads to speed up the computation.

In case a treebank collection has already been checked, there is the option to stop the treebank collection processor to also check it, in order to avoid redundancy of correctness checks. Code 4.8 shows how to do this.

```

import lal
tbcolproc = lal.io.treebank_collection_processor()
# The object will compute all features by default
err = tbcolproc.init("ud25.txt", "output_directory")
if err == lal.io.treebank_error_type.no_error:
    # Do not check for errors in the collection
    tbcolproc.set_check_before_process(False)
    # Process the treebank collection...
    err = tbcolproc.process()
print(err)

```

Code 4.8: We can prevent the treebank collection processor to check for errors in a treebank collection.

References

- [1] Frank Harary and Allen J. Schwenk. “The number of caterpillars”. In: *Discrete Mathematics* 6 (4 1973), pp. 359–365. ISSN: 0012365X. DOI: [10.1016/0012-365X\(73\)90067-8](https://doi.org/10.1016/0012-365X(73)90067-8).
- [2] Yossi Shiloach. “A Minimum Linear Arrangement Algorithm for Undirected Trees”. In: *SIAM Journal on Computing* 8.1 (1979), pp. 15–32. DOI: <https://doi.org/10.1137/0208002>.
- [3] F.R.K. Chung. “On optimal linear arrangements of trees”. In: *Computers & Mathematics with Applications* 10.1 (1984), pp. 43–60. ISSN: 0898-1221. DOI: [https://doi.org/10.1016/0898-1221\(84\)90085-3](https://doi.org/10.1016/0898-1221(84)90085-3).
- [4] M. A. Iordanskii. “Minimal numberings of the vertices of trees — Approximate approach”. In: *Fundamentals of Computation Theory*. Ed. by Lothar Budach, Rais Gatič Bukharajev, and Oleg Borisovič Lupanov. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 214–217. ISBN: 978-3-540-48138-6.
- [5] Robert A. Hochberg and Matthias F. Stallmann. “Optimal one-page tree embeddings in linear time”. In: *Information Processing Letters* 87.2 (2003), pp. 59–66. ISSN: 0020-0190. DOI: [https://doi.org/10.1016/S0020-0190\(03\)00261-8](https://doi.org/10.1016/S0020-0190(03)00261-8).

- [6] Ramon Ferrer-i-Cancho. “Euclidean distance between syntactically linked words”. In: *Physical Review E* 70.5 (2004), p. 5. ISSN: 1063651X. DOI: [10.1103/PhysRevE.70.056135](https://doi.org/10.1103/PhysRevE.70.056135).
- [7] Ryan McDonald et al. “Non-Projective Dependency Parsing using Spanning Tree Algorithms”. In: *HLT-EMNLP*. 2005, pp. 523–530.
- [8] Daniel Gildea and David Temperley. “Optimizing Grammars for Minimum Dependency Length”. In: *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*. Prague, Czech Republic: Association for Computational Linguistics, June 2007, pp. 184–191. URL: <https://www.aclweb.org/anthology/P07-1024>.
- [9] H. Liu. “Dependency direction as a means of word-order typology: a method based on dependency treebanks”. In: *Lingua* 120.6 (2010), pp. 1567–1578.
- [10] Yingqi Jing and Haitao Liu. “Mean Hierarchical Distance: Augmenting Mean Dependency Distance”. In: *Proceedings of the Third International Conference on Dependency Linguistics (Depling 2015)* Depling (2015), pp. 161–170.
- [11] Juan Luis Esteban, Ramon Ferrer-i-Cancho, and Carlos Gómez-Rodríguez. “The scaling of the minimum sum of edge lengths in uniformly random trees”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2016.6 (June 2016), p. 063401. DOI: [10.1088/1742-5468/2016/06/063401](https://doi.org/10.1088/1742-5468/2016/06/063401). URL: <https://doi.org/10.1088/1742-5468/2016/06/063401>.
- [12] Juan Luis Esteban and Ramon Ferrer-i-Cancho. “A Correction on Shiloach’s Algorithm for Minimum Linear Arrangement of Trees”. In: *SIAM Journal on Computing* 46.3 (2017), pp. 1146–1151. DOI: [10.1137/15M1046289](https://doi.org/10.1137/15M1046289). eprint: <https://doi.org/10.1137/15M1046289>. URL: <https://doi.org/10.1137/15M1046289>.
- [13] Sylvain Kahane, Chunxiao Yan, and Marie-Amélie Botalla. “What are the limitations on the flux of syntactic dependencies? Evidence from UD treebanks”. In: Sept. 2017, pp. 73–82. URL: <https://hal.archives-ouvertes.fr/hal-01675325>.
- [14] H. Liu, C. Xu, and J. Liang. “Dependency distance: A new perspective on syntactic patterns in natural languages”. In: *Physics of Life Reviews* 21 (2017), pp. 171–193.
- [15] Patrick Bennett, Sean English, and Maria Talanda-Fisher. “Weighted Turán problems with applications”. In: *Discrete Mathematics* 342.8 (2019), pp. 2165–2172. ISSN: 0012-365X. DOI: <https://doi.org/10.1016/j.disc.2019.04.007>. URL: <https://www.sciencedirect.com/science/article/pii/S0012365X19301268>.
- [16] Daniel Zeman et al. *Universal Dependencies 2.5*. <http://hdl.handle.net/11234/1-3105>. LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University. 2019.
- [17] Tanay Wakhare, Eric Wityk, and Charles R. Johnson. “The proportion of trees that are linear”. In: *Discrete Mathematics* 343.10 (2020), p. 112008. ISSN: 0012-365X. DOI: <https://doi.org/10.1016/j.disc.2020.112008>. URL: <https://www.sciencedirect.com/science/article/pii/S0012365X20301941>.
- [18] Lluís Alemany-Puig, Juan Luis Esteban, and Ramon Ferrer-i-Cancho. “Minimum projective linearizations of trees in linear time”. In: *Arxiv* (2021). arXiv: [2102.03277](https://arxiv.org/abs/2102.03277). URL: <https://arxiv.org/abs/2102.03277>.
- [19] Ramon Ferrer-i-Cancho et al. “Optimality of syntactic dependency distances”. In: *Physical Review E* 105 (1 Jan. 2022), p. 014308. DOI: [10.1103/PhysRevE.105.014308](https://doi.org/10.1103/PhysRevE.105.014308). URL: <https://arxiv.org/abs/2007.15342>.
- [20] Lluís Alemany-Puig, Ramon Ferrer-i-Cancho, and Juan Luis Esteban. *The Linear Arrangement Library on Github*. <https://github.com/lluissalemanyapuig/linear-arrangement-library>.
- [21] *Anaconda*. <https://www.anaconda.com>.
- [22] Dimitri van Heesch. *Doxygen*. <https://doxygen.nl>.

- [23] *Spider Graph* – Wolfram Math World. <https://mathworld.wolfram.com/SpiderGraph.html>. Accessed: 2022-10-04.
- [24] *Spyder*. <https://www.spyder-ide.org>.
- [25] *Stack Overflow*. <https://stackoverflow.com>.
- [26] *The Linear Arrangement Library*. <https://cqlab.upc.edu/lal/>.
- [27] *UD_Cantonese-HK*. https://github.com/UniversalDependencies/UD_Cantonese-HK/blob/master/yue_hk-ud-test.conllu.
- [28] *Universal Dependencies*. <https://universaldependencies.org>.