Advanced usage 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 Teach (IMTech)
Universitat Politècnica de Catalunya
Barcelona, Catalonia

Last updated: June 26, 2024

Contents

1 Introduction

People reading this document are courageous readers who want to dive deeper into the realm of advanced concepts in graph theory and probability, and to apply all of these in the utmost innovative ways conceivable to the study of human languages, all of this making good, efficient and correct use of the Linear Arrangement Library. Nevertheless, readers are encouraged to ensure that they read cover-to-cover the corresponding quick guide of LAL that is available at https://cqllab.upc.edu/lal/guides/.

The quick guide presented the reader with the basic usages and utilities available in the library. In the advanced guide we present the different types of enumeration of trees and arrangements (Section 2), and advice on efficient usage of the library (Section 3).

2 Random and exhaustive generation

Recall that a syntactic dependency structure is actually a triad defined by

- A tree structure, the skeleton defining the syntactic dependency structure of a sentence.
- A linear ordering of its vertices, e.g. the natural order of words in a real sentence. In graph theory an ordering of the vertices is called a linear arrangement.
- Labels with additional information on vertices or edges.

Among the distinct scores that can be defined on syntactic dependency structures, the structural scores are defined exclusively on the tree structure and are thus independent from the linear ordering. Examples of structural scores are the mean hierarchical distance (MHD) [8]. The linear ordering scores depend on the linear ordering and are defined on both the tree structure and the linear ordering. Examples of linear ordering scores are the proportion of head initial dependencies [6] and mean dependency distance [1, 3]. In quantitative dependency syntax, one is often interested in answering two kinds of questions:

1. What is the expected value of score X in the absence of any cognitive pressure?
2. Is the value of score X significantly low (or high)?

The answer to the 1st question requires the calculation of a left p-value (or a right p-value) if one adopts a traditional p-vale testing. Given any score X, both questions can be answered generating trees if the scores is structural or generating random linear arrangements in case of linear order scores.

LAL is equipped with methods to generate trees both uniformly at random and exhaustively from different families, and, along the same lines, generate arrangements of trees, uniformly at random or exhaustively, under different formal constraints on dependency structures (projectivity, planarity or none). By uniformly random generation we mean rolling a fair die with as many sides as distinct objects. For instance, a random generator of trees, given a kind of tree, will generate all of them by rolling a die where each side corresponds to each tree. By exhaustive generation of trees, we mean generating each distinct object until, one at time (without repetition). For instance an exhaustive generator of random trees would generate all distinct trees, one after another. However, we need to introduce the different criteria that allow one to determine when two trees of the same kinds are distinct. Along the same lines a random generator of unconstrained linear arrangements will be equivalent to rolling a die where every side correspond to one of the possible shufflings of a sentence. Put differently, a random generator of a linear arrangements is equivalent to shuffling the words in a sentence as when shuffling card but each card corresponding to a distinct word token of the sentence.

We explain the different kinds of trees that can be generated with LAL, the criteria that determine when they are same or distinct, and how to generate at random or exhaustively in Section 2.1. Then, we explain along the same lines the different mechanisms to generate linear arrangements at random or exhaustively in Section 2.2.

2.1 Generating trees

The generation of trees is a crucial tool to obtain random baselines for structural measures or scores, especially when mathematical formulae to calculate such baselines are lacking. Random baselines can calculated exactly (for small trees), or approximately via random sampling (for trees of any size). We first describe the families of trees LAL can generate (Section 2.1.1). Then, we introduce the generation of trees with LAL with the help of some motivating examples (Section 2.1.2), explain some formalities about probability (Section 2.1.3), and how to generate all families of trees with LAL (Section 2.1.4). We finish this subsection with a piece of advice about exhaustive generation of trees (Section 2.1.5).

2.1.1 The different types of trees

The mathematical concept of rooted tree is not alien to people familiar with dependency syntax, as this corresponds to the tree structure or skeleton of the triad defining a syntactic dependency structure. Figure 1(b) shows the rooted three of the syntactic dependency structure in Figure 1(a). Rooted trees are trees in which there is a special node called the root and its edges are oriented; they are often drawn in a top-down fashion, namely, the root at the top, its children right below it, and so on and so forth as in Figure 2 (top row). Edges of rooted trees are considered oriented in the same direction, whether towards or away from the root – in our context, the orientation is usually away from the root as in syntactic dependency structures (Figure 1((a)-(b)). Free trees, on the other hand, are trees with no such special vertex and their edges and are not oriented; these trees need not be drawn in a top-down fashion (Figure 1(c). Therefore, a rooted tree always has a 'free' version that results unequivocally from removing the root from the rooted tree. Similarly, a free tree can be converted into a rooted tree by labeling one of the edges as the root and setting the appropriate direction of the edges.

Figure 2 shows rooted trees in the top row and free trees in the bottom row. See Code 2.1 to create a rooted tree from a free tree, and Code 2.2 to convert a rooted tree into a free tree.

```
T = lal.io.read_head_vector("free_tree", "file_with_head_vector.txt")
rT = lal.graphs.rooted_tree(T, 0) # root at vertex 0
```

Code 2.1: Creating a rooted tree from a free tree.

```
rT = lal.io.read_head_vector("rooted_tree", "file_with_head_vector.txt")
T = rT.to_free_tree()
```

Code 2.2: Turning a rooted tree into a free tree.

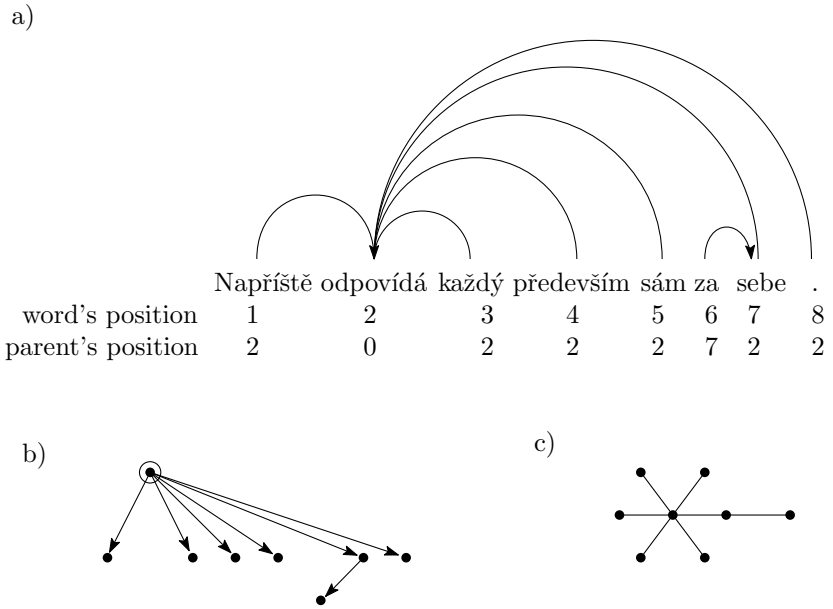


Figure 1: a) The 15*th* sentence example taken from UD 2.7 [18] Czech Github repository [16]. b) The rooted tree underlying the dependency structure in a). c) The free tree underlying the rooted tree in b).

The concept of labeled (and unlabeled) trees, however, might be a new concept [19, 12]. A labeled tree (whether rooted or free) is a tree in which its vertices carry labels with them; for the sake of simplicity, consider these labels to be numbers from 1 to n, where n is the number of nodes of the original syntactic dependency structure. To determine when two trees are the same or different, graph theory uses two criteria. One where the numbers attached to each vertex are relevant and hence the two trees being compared are thus called labeled. Another where those labels are missing or simply ignored, hence the trees being compared are thus called unlabeled. Take, for example, the labeled rooted trees in Figure 2 (top left): we can see that there are two groups of distinct labeled rooted trees if we consider the labels not to be relevant (namely the trees where the root has a single child and the trees where the root has two children), which leads to the couple of trees in Figure 2 (top right).

A vertex pointed by another one in a root tree is called a child in graph theory. In case of trees of three vertices, what determines whether two labeled free trees are the same or not is the vertex that is the hub of the tree (the internal vertex), that can be 1, 2 or 3 and thus there are only 3 distinct free trees of three vertices. In case of labeled rooted trees of three vertices, what determines whether the two trees are the same is the label of the root vertex, the number of children of the root and, in case that the root has a single child, the label of the root's child. In case of rooted labeled trees of three vertices, two trees are the same only if the root has the same number of children.

A clear difference between labeled and unlabeled trees is that labeled trees are more numerous than unlabeled trees: by attaching labels to the vertices we can distinguish between the trees and hence increase the amount of distinct such trees. Note the point is not that the labeling is the same but rather that what makes a tree distinct: two trees may have totally different labels but actually same if the criterion for comparison is unlabeled. Let us insist: if we remove or ignore the labels from the labeled rooted trees in Figure 2(top left), it results in just two indistinguishable unlabeled rooted trees (Figure 2 top right). The same can be said about free trees. By labeling them as in Figure 2(bottom left), we obtain three different trees; removing or ignoring the labels leaves us with only one unlabeled free tree (Figure 2 bottom right). Finally, notice that the placement of the vertices when drawing them is irrelevant to determine whether to trees are the same or not in the sense of graph theory. Figure 2 shows in distinct rows, trees that are the same in spite of the placement of the vertices. As

an exercise, the reader is invited to try coming up with a new distinct tree that is not shown in Figure 2: bear in mind that one should be able to draw that tree so that it matches one that is already in the corresponding cell of Figure 2 because the table already contains all possible distinct trees of vertices according to the criteria that we have reviewed above (rooted/free and labeled/unlabeled).

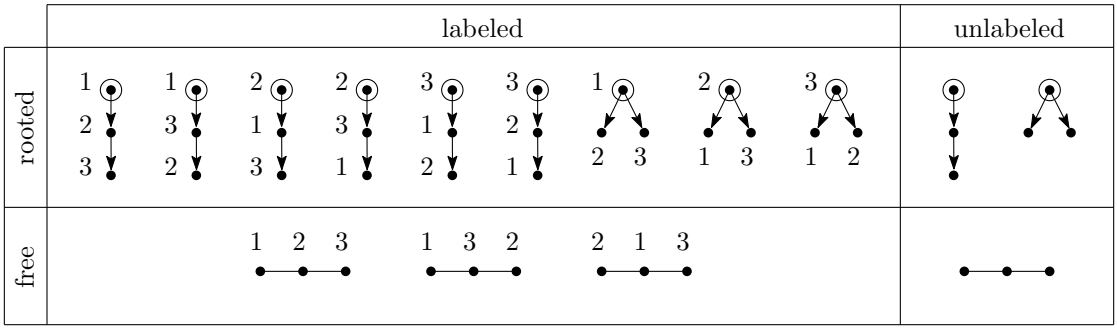


Figure 2: The four different types of trees that can be generated with the library. Every cell shows all the trees of three vertices for the corresponding combination of rooted/free and labeled/unlabeled. Labeled rooted trees (top left), unlabeled rooted trees (top right), labeled free trees (bottom left), unlabeled free trees (bottom right). In the top row, roots are indicated with a circle.

2.1.2 Motivating examples

Example 1 Suppose a Dependency Syntax researcher who investigates the mean hierarchical distance (MHD) [8] in real sentences of n words. A new type of question such a researcher might ask is "What is the expected value of the MDD in a uniformly random tree of a given number of nodes n?". This answer is rather difficult to answer exactly (with a mathematical formula) so we can try computing an approximate value. To do it, we apply the so-called Monte Carlo procedure, where we sample trees uniformly at random, and sum all their MHD into some variable of our program. Once we have finished generating (sampling) all the random trees, we divide the total sum by the number of trees generated (sampled) in order to get an approximation of the average MHD on trees of a fixed number of vertices n. This is given in Code 2.3. Now, the more trees we sample, the more precise will the estimation of the expected value be; this is generally true in Monte Carlo-like procedures.

```
import lal
n = 50               # we are given a fixed number of vertices
total_sum_MHD = 0    # variable where we accumulate the total
                     # sum of mean hierarchical distances
sample_size = 10000 # the number of trees to be generated
# generate labeled rooted trees
Gen = lal.generate.rand_lab_rooted_trees(n)
for i in range(0, sample_size):
    random_tree = Gen.yield_tree() # retrieve the random tree
    MHD_random_tree = lal.properties.mean_hierarchical_distance(random_tree)
    total_sum_MHD += MHD_random_tree # accumulate the MHD of the random tree
# calculate the expected value as the ratio of the sum of the MHD of the
# random trees and the total amount of trees generated
expected_MHD = total_sum_MHD/sample_size
```

Code 2.3: Calculating the expected value of the mean hierarchical distance on n-vertex trees.

Example 2 In addition to helping to calculate random baselines, these generation methods allows one to perform traditional statistical tests. The same researcher may wish to ask: "Is the value of MHD of some syntactic dependency tree [8] significantly small?" This translates into the following questions: "What is the probability that a rooted tree has a MHD less than or equal to a given value x?" In a traditional p-value testing, one needs to calculate a left p-value to answer the question. If the p-value is below the significance level, we will conclude that that k is significantly small. Again, we apply a Monte Carlo algorithm, i.e., we generate labeled rooted trees uniformly at random, for every generated tree we calculate its mean hierarchical distance MHD, and count how many of these trees have a lower MHD than the MHD of the given tree. The more trees we generate, the more precise will the probability be. The Python code to calculate such a probability is given in Code 2.4.

```
import lal
# we are given a tree
T = lal.graphs.from_head_vector_to_rooted_tree([2,0,2,2,2,7,2,2])
n = T.get_num_nodes()
# whose mean hierarchical distance can be calculated easily
MHD_T = lal.properties.mean_hierarchical_distance(T)
# variable that counts how many generated trees have a
# MHD less than or equal to MHD_T
amount_trees_smaller_MHD = 0
# the number of trees to be generated
sample_size = 10000 # 10^4
# generate labeled rooted trees uniformly at random
Gen = lal.generate.rand_lab_rooted_trees(n)
for i in range(0, sample_size):
    # retrieve the random tree
    random_tree = Gen.yield_tree()
    # calculate the MHD of the random tree
    MHD_random_tree = lal.properties.mean_hierarchical_distance(random_tree)
    # count the random tree into 'amount_trees_smaller_MHD' when appropriate
    if MHD_random_tree <= MHD_T:
        amount_trees_smaller_MHD += 1
# estimate the left p-value as the ratio of the amount of trees with
# a MHD lower or equal to MHD_T and the sample size
pvalue = amount_trees_smaller_MHD/sample_size
print(pvalue)
```

Code 2.4: Calculating the probability that a random tree has a MHD less than a given value.

Example 3 The probability calculated in Code 2.4 and the expected value calculated in Code 2.3 are only approximations of the real values, whose error is inversely proportional to the amount of sampled trees. Nevertheless, we can calculate exact values in case the size of the vertices, i.e., the number of vertices, is small: we simply enumerate all labeled rooted trees of that given amount of vertices. This is shown in Code 2.5.

For these reasons, a common trick is to use exhaustive generation for trees up to a certain value of n (e.g. $n = 10$) and then use random generation beyond [10, 9]. Using exhaustive generation the expected value and the p-value will be exact.

```
import lal
# we are given a tree (of small number of vertices 'n')
T = lal.graphs.from_head_vector_to_rooted_tree([2,0,2,2,2,7,2,2])
n = T.get_num_nodes()
# whose mean hierarchical distance can be calculated easily
MHD_T = lal.properties.mean_hierarchical_distance(T)
# variable that counts how many generated trees
# have a MHD less than or equal to MHD_T
amount_trees_smaller_MHD = 0
# variable where we accumulate the total
# sum of mean hierarchical distances
total_sum_MHD = 0
# total amount of trees
total_num_trees = 0
# generate all n-vertex labeled rooted trees
Gen = lal.generate.all_lab_rooted_trees(n)
while not Gen.end():
    # retrieve the tree
    current_tree = Gen.yield_tree()
    # count total amount of trees
    total_num_trees += 1
    # calculate the MHD of the current tree
    MHD_current_tree = lal.properties.mean_hierarchical_distance(current_tree)
    # count the current tree into 'amount_trees_smaller_MHD' when appropriate
    if MHD_current_tree <= MHD_T:
        amount_trees_smaller_MHD += 1
    # accumulate the MHD of the current tree
    total_sum_MHD += MHD_current_tree
# calculate the expected value as the ratio of the sum of the MHD of the
# random trees and the total amount of trees generated
expected_MHD = total_sum_MHD/total_num_trees
# calculate the left p-value as the ratio of the amount of trees with
# a MHD lower or equal to MHD_T and the total amount of trees generated
pvalue = amount_trees_smaller_MHD/total_num_trees
```

Code 2.5: Calculating the left p-value and expected value in Codes 2.4 and 2.3 exactly when the number of vertices n is low.

2.1.3 What we mean by uniformly at random and exhaustive

By 'uniformly random' we mean, basically, that every tree returned by the generator has the same probability of being returned than any other tree as it happens to the sides obtained by rolling a fair die. Take, for example, the set of labeled rooted trees in Figure 2 (top left), and suppose we are given a fair 9-sided die where each of its sides has a tree from Figure 2 (top left); each tree is assigned to a distinct side of the die. Assuming it is a fair die, every side is equally likely to end up on top after rolling the die, with probability 1/9. Therefore, after rolling the die enough times, say $N = 10^6$, and counting how many times each tree appeared on top we would get that every tree appeared a similar amount of times, around $N/9$ each. This is what is called a *uniformly random* generation. Hence, the generator of n-vertex labeled rooted trees simulates a fair die of as many sides as labeled rooted trees of n vertices there are. An exhaustive generator is capable of enumerating *all* trees of a certain type,

in an order prescribed by the algorithm the generator implements. The same can be said about the other generators.

2.1.4 The full tree generation capabilities of LAL

At the beginning of this section we have given three motivating examples: two involving generation of random trees (the first to estimate a probability and the second to estimate an expected value), and one that calculates the same probability and expected value exactly, by an exhaustive enumeration of the trees when the size of the tree is low. Based on the distinctions above, LAL offers the possibility to generate trees that can be labeled or unlabeled, free or rooted, yielding 4 possible kinds of trees. When combined with the generation method (exhaustive or random) as

$$\left\{ \begin{array}{c} exhaustive \\ random \end{array} \right\} \times \left\{ \begin{array}{c} labeled \\ unlabeled \end{array} \right\} \times \left\{ \begin{array}{c} free \\ rooted \end{array} \right\},$$

one obtains 8 possible methods to generate trees.

The names of the classes to generate trees follow the same pattern so as to make it easy for users to deduce the name of the class of the object they need. The pattern can be represented with the template in Code 2.6. The numbers `1`, `2` and `3` are placeholders for `all`/`rand`, `lab`/`ulab` and `rooted`/`free`, where

(1) One of `all`/`rand` is used to indicate whether the generation must be exhaustive (`all`) or random (`rand`),

(2) One of `lab`/`ulab` is used to indicate whether the trees generated have to be labeled (`lab`) or unlabeled (`ulab`),

(3) One of `rooted`/`free` is used to indicate whether the trees should be rooted (`rooted`) or free (`free`).

```
lal.generate.1_2_3_trees
```

Code 2.6: The pattern of the names of the classes that generate trees.

We strongly encourage users to learn this pattern by heart so as to avoid having to memorize all the 8 combinations, listed in Code 2.7. We refer to the classes starting with '`all`' as '*exhaustive*' classes, and those starting with `rand` as the *random* classes.

```
lal.generate.all_lab_free_trees
lal.generate.all_lab_rooted_trees
lal.generate.all_ulab_free_trees
lal.generate.all_ulab_rooted_trees
lal.generate.rand_lab_free_trees
lal.generate.rand_lab_rooted_trees
lal.generate.rand_ulab_free_trees
lal.generate.rand_ulab_rooted_trees
```

Code 2.7: All the 8 different combinations of tree generation classes.

All of the classes listed in Code 2.7 have to be constructed with a number of vertices. In both types of classes, random and exhaustive, a tree is retrieved using the `.yield_tree()` method. The main

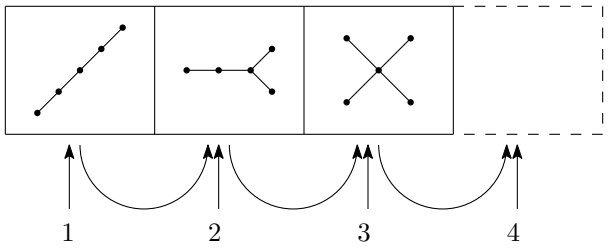


Figure 3: Graphical illustration of the generation of all 5-vertex unlabeled free trees with class `lal.generate.all_ulab_free_trees` based on Code 2.8. At step 1, the class is constructed and the tree in the first box is ready to be retrieved with the `.yield_tree()` method. Further calls to `.yield_tree()` retrieve the subsequent trees in the corresponding box. At step 4, method `.end()` returns `True`.

difference between the two types of classes is that exhaustive classes act like a pointer within the list of trees it is generating; such a pointer needs is moved forward when `.yield_tree()` method is called (Figure 3). Therefore, upon initialization of an exhaustive class, method `.yield_tree()` can be called to retrieve the first tree. In order to finish the generation, exhaustive classes have an `.end()` method that returns `True` when the iteration has reached its end.

Code 2.8 illustrates how to enumerate all trees; recall that in Code 2.8 the '2' and '3' have to be replaced by `lab`/`ulab` and `rooted`/`labeled` respectively.

```
import lal
Gen = lal.generate.all_2_3_trees(10)
while not Gen.end():
    t = Gen.yield_tree()
    # process tree 't' ...
```

Code 2.8: Generating all 10-vertex trees.

Random classes are simpler to use: basically, there is no `.end()` method to call and `.yield_tree()` returns a tree generated uniformly at random every time it is called. Code 2.9 illustrates how to generate trees uniformly at random; recall that in Code 2.9 the '2' and '3' have to be replaced by `lab`/`ulab` and `rooted`/`free` respectively.

```
import lal
Gen = lal.generate.rand_2_3_trees(10)
for i in range(0,1000):
    t = Gen.yield_tree()
    # process tree 't' ...
```

Code 2.9: Generating 10-vertex trees uniformly at random.

Evidently, a random generation can produce the same tree more than once whereas exhaustive generation will never give the same tree twice.

2.1.5 Advice on the exhaustive generation of trees

As we have already explained, exhaustive generation allows one to estimate expected values or p-values exactly. However, when generating all trees of a certain type (be it labeled or not, rooted or free), users must bear in mind that the amount of trees that will be generated might be too large for the computation to ever finish regardless of the computing power available to the user. The amount of trees in every type can be found in the Online Encyclopedia of Integer sequences [13]. In particular, the sequences of the amount of unlabeled free [14] and unlabeled rooted trees [15], which we summarize in Table 1 in order to illustrate the large amount of such trees. Therefore, if your n is too large, the consider a Monte Carlo approach. To estimate the expectation or the p-value accurately enough, use a generate a large number of samples (e.g., 10^4, 10^5 depending on the power of your computer and your patience). The number does not need to be huge to be obtain accurate enough estimates.

n	Rooted	Free
1	1	1
	...	
10	719	106
	...	
21	35.221.832	2.144.505
	...	
30	354.426.847.597	14.830.871.802

Table 1: Number of unlabeled rooted [15], unlabeled free trees [14].

2.2 Generating arrangements

Although a sentence of n words has $n!$ possible orderings, language researchers have defined some formal constraints that restrict the possible linear arrangements. Two rather frequent types of arrangements of trees are projective and planar arrangements. These are very well-known, nevertheless we remind the reader that a projective arrangement *of a rooted tree* is one in which there are no edge crossings and the root is not covered by the shadow of any edge (Figure 4(a)). A planar arrangement, relaxes the covering condition hence allowing the root to be covered (Figure 4(b)). In the latter case, since the root does not play an important role, we consider that a planar arrangement of a rooted tree is also a planar arrangement of the same tree without its root, i.e., a planar arrangement of the 'free' version of the rooted tree. There is a third formal constraint that readers must be aware of: no constraint at all, or, as we call them, unconstrained arrangements (Figure 4(c)). These arrangements can take any shape and form, including, be it by chance or not, the form of a planar arrangement and even of a projective arrangement. That is, the mapping of the vertices to positions is completely free of constraints on edge crossings and/or root coverings. Figure 5(c) shows two arrangements (different, yet equivalent), arrangements of the tree in Figure 5(b): those two arrangements are certainly not planar or projective, and it is doubtful as to whether such arrangements belong to any of the classes described in [7]. Thus, the set of unconstrained arrangements of a tree is the set of all possible arrangements, amounting to $n!$ different arrangements.

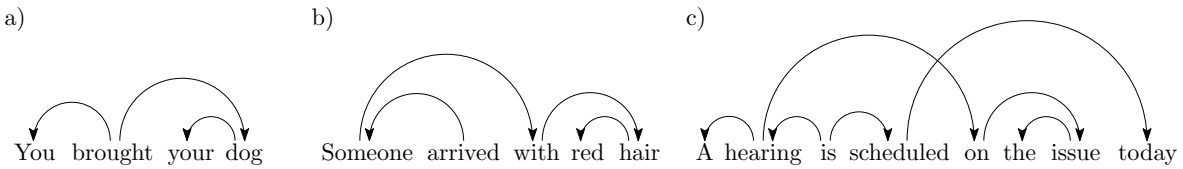
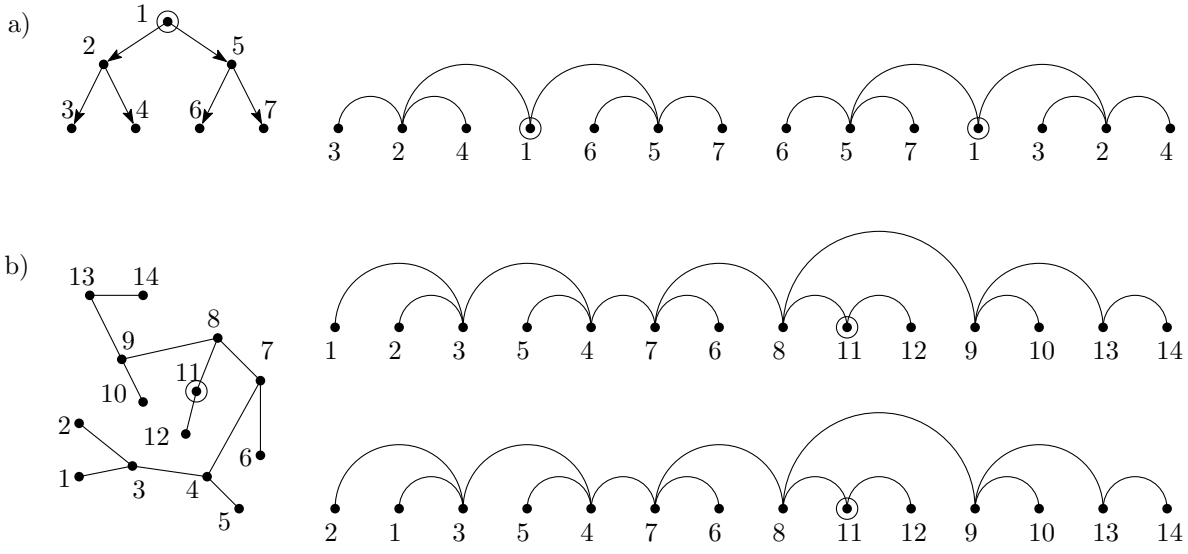


Figure 4: Real sentences in English. a) Sentence subject to projectivity. b) Sentence subject to planarity but not to projectivity. c) Sentence not subject to any formal constraint.

The Linear Arrangement Library also offers a solution to generating arrangements of rooted trees that are projective (Figure 5(c)). In other words, given a rooted tree, users can generate *all* projective arrangements of such a tree, and, alternatively, generate projective arrangements *uniformly at random*. Moreover, LAL also provides its users with similar methods for planar arrangements (Figure 5(b)) and unconstrained arrangements (Figure 5(a)).

2.2.1 The different types of arrangements

To put it a little bit more formally, an arrangement is a mapping of the vertices of a tree to positions in a linear sequence; these positions are, more often than not, the positions from 1 to n. In the context of this library we consider arrangements to be always arrangements of labeled trees. This means that two arrangements are only equal when every vertex is mapped to the same position in both arrangements, and *different* if otherwise. Put differently, two arrangements are the same if the vectors that indicate the position of every vertex are the same. There might be other criteria that could be used to determine if two arrangements are the same. Some might be evident or intuitive, but bear in mind that they are not taken into account by LAL. Two arrangements may be deemed equivalent if [1] if every arc connecting a pair of positions in one linear ordering is also found in the other linear ordering[2]. For example, Figure 5(a) shows two *different* arrangements that are *equivalent*; the same can be said about 5(b) and 5(c).



[1] To the best of our knowledge, there is no clear definite definition of *arrangement equivalence*; nevertheless we use such word to give an intuition of how two arrangements, in spite of mapping vertices to different positions, can yield the same sum of edge lengths and the same amount of edge crossings, all this while avoiding the concept of *isomorphism*.

[2] Although the notion that two arrangements are equivalent 'when for every pair of connected positions in one arrangement the same positions in the other arrangement are also connected' is attractive and appealing, we are not implying by any means that this is the only definition possible.

c)

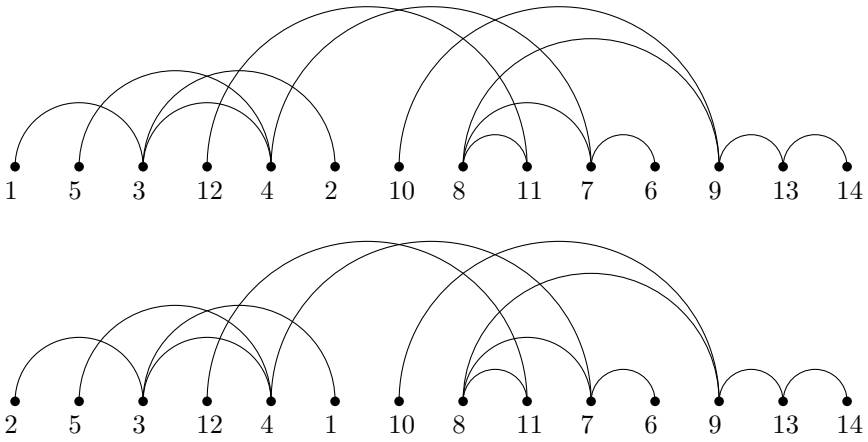


Figure 5: Pairs of different yet equivalent arrangements. a) Projective arrangements of the same rooted tree. b) Planar arrangements of the same rooted tree. c) Unconstrained arrangements of the tree in b).

2.2.2 Motivating examples

Example 1 Quantitative Linguistics researchers asked themselves: "What is the expected value of D in random arrangements that are *constrained to be projective*?". This question was explored in previous papers [2, 4, 5]; a similar question was already answered for *unconstrained* arrangements [1], yielding $\mathbb{E}[D] = (n^2 - 1)/3$, where n denotes the number of vertices. The expected value of D constrained to projective arrangements of a given tree T is equal to the average of all the values of D in every projective arrangement of T. To be a bit more precise, by listing all projective arrangements of a tree T, adding up all the values of D (one value for each arrangement) and dividing said sum by the number of projective arrangements we obtain the aforementioned expected value.

Now the question is: how do we calculate it using the library? One way is by following the steps of [2, 4, 5] and obtain an approximate value by random sampling (i.e., by using a Monte Carlo-like procedure). This is shown in Code 2.10.

```
import lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,5,2,5,0,5,6])
sample_size = 1000 # the number of arrangements to be generated
sum_D_projective = 0 # the sum of all values of D in projective arrangements
# generate random projective arrangements of rt
Gen = lal.generate.rand_projective_arrangements(rt)
for i in range(0, sample_size):
    rand_arr = Gen.yield_arrangement()
    # compute the sum of edge lengths
    rand_D = lal.linarr.sum_edge_lengths(rt, rand_arr)
    sum_D_projective += rand_D # accumulate the value of D
# calculate the expected value of D in random projective arrangements
expected_D_projective = sum_D_projective/sample_size
```

Code 2.10: Calculating (via random sampling) the expected value of D in uniformly random projective arrangements.

Another way of calculating such expected value is by making good use of state-of-the-art research:

the library implements the algorithm devised in [11] to calculate the expected value of D in projective arrangements of trees *exactly*, i.e., without random sampling and without the need of enumerating all projective arrangements. This is shown in Code 2.11.

```
import lal
# create a rooted tree
rt = lal.graphs.from_head_vector_to_rooted_tree([3,5,2,5,0,5,6])
# the exact expected value of D in random projective arrangements
exp_D_projective = lal.properties.exp_sum_edge_lengths_projective(rt)
```

Code 2.11: Exact calculation of the expected value of D in uniformly random projective arrangements.

Code 2.10 can be easily modified to answer the following question: is the value of D of a given tree significantly small assuming that the linear arrangements must be projective? The question can be answered by calculating a left p-value and comparing it against the significance level. The left p-value can be estimated by sampling (uniformly at random) projective arrangements of the given tree and counting how many arrangements yield a lower value of D. This is given in Code 2.12. Notice that this combines the two types of linear arrangements that we explained in the Quick Guide: implicit and explicit arrangements. `D_rt` is calculated using the implicit linear arrangement of rt whereas `rand_D` is calculated using the explicit arrangement `rand_arr`.

```
import lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,5,2,5,0,5,6])
D_rt = lal.linarr.sum_edge_lengths(rt) # the current value of D
sample_size = 1000 # the number of arrangements to be generated
amount_D_lower = 0 # the amount of arrangements with a lower D
# generate N random projective arrangements of rt
Gen = lal.generate.rand_projective_arrangements(rt)
for i in range(0, sample_size):
    rand_arr = Gen.yield_arrangement()
    # compute the sum of edge lengths
    rand_D = lal.linarr.sum_edge_lengths(rt, rand_arr)
    if rand_D <= D_rt: amount_D_lower += 1 # accumulate the value of D
# calculate the right p-value of D in random projective arrangements
pvalue = amount_D_lower/sample_size
```

Code 2.12: Calculating the right p-value that a given tree has a value of D that is large enough.

Example 2 Consider the following question: "What is the expected value of the maximum flux weight in random projective arrangements of a rooted tree?". How about: "What is the expected minimum R/L ratio in random planar arrangements?". Users of LAL can give an approximate answer these questions using the generators of random projective and planar arrangements, or, if the tree is small enough, an exact answer by enumerating all projective/planar arrangements. Question (1) is answered in Code 2.13, and question (2) is answered in Code 2.14. Both codes use the tree in Figure 6.

```
import lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,5,2,5,0,5,6])
sample_size = 1000 # the number of arrangements to be generated
sum_max_weight = 0 # the sum of all maximum weights
# generate N random projective arrangements of rt
Gen = lal.generate.rand_projective_arrangements(rt)
for i in range(0, sample_size):
    rand_arr = Gen.yield_arrangement()
    fluxes = lal.linarr.compute_flux(rt, rand_arr) # compute dependency fluxes
    max_weight = max([ flux.get_weight() for flux in fluxes ]) # maximum weight
    sum_max_weight += max_weight # accumulate the maximum weight
# calculate the expected maximum weight
expected_max_weight = sum_max_weight/sample_size
```

Code 2.13: Calculating the expected maximum weight of a rooted tree restricted to its projective arrangements.

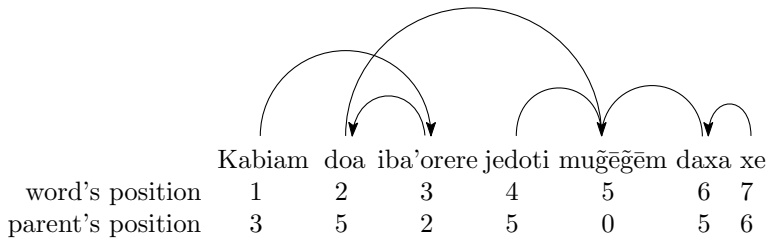


Figure 6: The 61*st* sentence example taken from UD 2.7 [18] Munduruku Github repository [17].

```
import lal
rt = lal.graphs.from_head_vector_to_rooted_tree([3,5,2,5,0,5,6])
sample_size = 1000 # the number of arrangements to be generated
sum_min_RL_ratio = 0 # the sum of all minimum R/L ratios
# generate N random planar arrangements of rt
Gen = lal.generate.rand_planar_arrangements(rt)
for i in range(0, sample_size):
    rand_arr = Gen.yield_arrangement()
    fluxes = lal.linarr.compute_flux(rt, rand_arr) # compute the fluxes
    # find the minimum R/L ratio
    min_RL_ratio = min([ flux.get_RL_ratio() for flux in fluxes ])
    sum_min_RL_ratio += min_RL_ratio # accumulate the minimum R/L ratio
# calculate the expected minimum R/L ratio
expected_min_RL_ratio = sum_min_RL_ratio/sample_size
```

Code 2.14: Calculating the expected minimum R/L ratio of a rooted tree restricted to its planar arrangements.

2.2.3 What we mean by uniformly at random and exhaustive

Following the description in Section 2.2.1, an exhaustive generation (enumeration) of arrangements of a tree will generate all different arrangements, which, recall, does not take into account any symmetry of the arrangements. A uniformly random generation of the arrangements follows the same ideas explained in Section 2.1.3 regarding random generation of trees: if a rooted tree has s arrangements of a certain formal class (e.g., unconstrained, planar or projective), then any of such arrangements has probability $1/s$ of being generated, as if every arrangement was put on the side of a s-sided fair die. In case of exhaustive generation, each of the s possible orderings will be produced, one by one.

In case of unconstrained arrangements of a sentence of n words, the exhaustive generator will generate each of the $n!$ possible orderings, one by one, whereas the random generator will be equivalent to rolling a fair die with $n!$ sides, each corresponding to a distinct linear arrangement. The latter is equivalent to shuffling cards each corresponding to a distinct word token of the sentence, producing a random ordering of the word tokens. If other formal constraints are considered, the idea is the same but then the number of possible orderings will be $\leq n!$.

2.2.4 The full arrangement generation capabilities of LAL

The combination of the generation methods and the formal constraints as

$$\left\{ \begin{array}{c} exhaustive \\ random \end{array} \right\} \times \left\{ \begin{array}{c} unconstrained \\ planar \\ projective \end{array} \right\}$$

gives 6 ways of generating arrangements.

The general pattern of usage of the arrangement generator classes is similar to the usage of tree generator classes. The names of the random classes can be easily deduced by learning the simple pattern given in Code 2.15.

```
lal.generate.1_2_arrangements
```

Code 2.15: The pattern of the names of the classes that generate arrangements.

As before, the numbers `1` and `2` are just placeholders for `all/rand` and `projective/planar`, where

(1) One of `all`/`rand` is used to indicate whether the generation must be exhaustive (`all`) or random (`rand`),

(2) One of ∅/`projective`/`planar` is used to indicate whether the arrangements generated have to be `projective` or `planar`. By ∅ we mean that the restriction on the arrangements can be omitted to refer implicitly to the unconstrained class of arrangements.

All the possible combinations of classes are given in Code 2.16.

```
lal.generate.all_arrangements
lal.generate.all_projective_arrangements
lal.generate.all_planar_arrangements
lal.generate.rand_arrangements
lal.generate.rand_projective_arrangements
lal.generate.rand_planar_arrangements
```

Code 2.16: All the 4 different combinations of arrangement generation classes.

Similarly as before (Section 2.1.4), the *exhaustive* classes for arrangements and the *random* classes are used in a slightly different way. Code 2.17 illustrates how to generate all arrangements of a tree; again, recall that in Code 2.17 the `2` has to be replaced by either nothing (∅), `planar`, or `projective`. In case projective arrangements are to be generated then the input tree must be a rooted tree. Recall that the generation of arrangements using the `all_arrangements` class includes the arrangements generated by `all_projective_arrangements` and `all_planar_arrangements`.

```
import lal
input_tree = ... # make a tree
Gen = lal.generate.all_2_arrangements(input_tree)
while not Gen.end():
    arr = Gen.yield_arrangement()
    # use arrangement 'arr' ...
```

Code 2.17: Enumerating all arrangements of a tree.

Code 2.18 illustrates how to generate arrangements uniformly at random. Recall that the `2` has to be replaced by either nothing (∅), `planar`, or `projective`. Moreover, recall that in case projective arrangements are to be generated then the input tree must be a rooted tree.

```
import lal
input_tree = ... # make a tree
Gen = lal.generate.rand_2_arrangements(input_tree)
for i in range(0,1000):
    t = Gen.yield_arrangement()
    # use arrangement 'arr' ...
```

Code 2.18: Generating uniformly random arrangements of a tree.

2.3 Generating trees and arrangements at the same time

Above we have argued that research questions on linear ordering structures can be answered generating linear arrangements (at random or exhaustively). However, in previous research, a hybrid approach where both the tree and the linear arrangements is generated at random has been considered [3, 9, 10]. This can be done with LAL generating uniformly at random (u.a.r.) a labeled tree (rooted or free) and then interpreting vertex labels as vertex positions with the notion of *implicit* arrangement that we explained in the Quick Guide: in an implicit linear arrangement, the label of a vertex indicates its position. This allows LAL users to calculate expected values and p-values with respect to the ensemble of distinct labeled trees arranged with non-equivalent arrangements (whose size is n^{n-2} for free trees and n^{n-1} for rooted trees).

In the motivating examples above, we have shown how to calculate via random sampling the expected value of D in projective arrangements of a given tree (Code 2.10). Here we show how to adapt such code to calculate the expected value of D in projective trees of a fixed size n. This is shown in Code 2.19. Since trees need to be projective, the labels of their vertices must yield a projective arrangement: this is checked in the function `is_projective`. Within that function, we use the method `lal.linarr.num_crossings` which calculates the number of edge crossings of a tree; this function, in turn, uses an implicit arrangement. Moreover, notice that the sample size is not used in a for loop but, rather, in a while loop. That is because when sampling labeled trees the resulting syntactic structure

need not be projective (for this case). This is why we filter generated trees according to projectivity and finish the while loop when the appropriate amount of projective trees has been found; this filtering strategy is known as a *rejection method*.

```
import lal
# fuction to test whether a tree is projective or not
def is_projective(rt):
    if lal.linarr.num_crossings(rt) > 0: return False
    r = rt.get_root()
    edges = rt.get_edges()
    root_covered = any([((u<r and r<v) or (v<r and r<u)) for (u,v) in edges])
    return not root_covered
# we are given a fixed size
n = 10
# the number of arrangements to be generated
n_projective_found = 0
sample_size = 1000
# the sum of all values of D in projective arrangements
sum_D_projective = 0
# generate random labeled rooted trees
Gen = lal.generate.rand_lab_rooted_trees(n)
while n_projective_found < sample_size:
    rt = Gen.yield_tree()
    # only if the random tree is projective
    if is_projective(rt):
        # compute the sum of edge lengths
        rand_D = lal.linarr.sum_edge_lengths(rt)
        # accumulate the value of D
        sum_D_projective += rand_D
        n_projective_found += 1
# calculate the expected value of D in
expected_D_projective = sum_D_projective/sample_size
```

Code 2.19: Calculating (via random sampling plus a rejection method) the expected value of D in uniformly random projective rooted trees.

We also show how to adapt Code 2.13 which calculates the expected maximum dependency flux weight in the space of uniformly random non-equivalently arranged trees. Such adaptation is shown in Code 2.20: users need to choose a fixed size n, have to sample trees of that size with a generator, and the calculation of the score is done without indicating an explicit linear arrangement, rather, using the implicit arrangement.

```
import lal
# we are given a fixed size
n = 10
# the number of labeled trees to generate
sample_size = 1000
# the sum of all maximum weights
sum_max_weight = 0
# generate N random labeled free trees
Gen = lal.generate.rand_lab_free_trees(n)
for i in range(0, sample_size):
    # retrieve the tree
    t = Gen.yield_tree()
    # compute the dependency fluxes
    fluxes = lal.linarr.compute_flux(t)
    # find the maximum weight
    max_weight = max([ flux.get_weight() for flux in fluxes ])
    # accumulate the maximum weight
    sum_max_weight += max_weight
# calculate the expected maximum weight
expected_max_weight = sum_max_weight/sample_size
```

Code 2.20: Calculating the expected maximum weight in the space of uniformly random non-equivalently arranged trees.

Finally, let us conclude with some remarks for mathematically-inclined readers. The method to generate in one shot the tree and the linear arrangement can be seen as sampling *non-equivalently arranged labeled trees*. Thus, by generating uniformly at random one of the 3-vertex trees in Figure 2(top left) we are generating u.a.r. both a tree and an arrangement for its vertices, where such arrangements are considered non-equivalent. Notice that the number of labeled trees sampled in this way is smaller than ensemble defined by all unlabeled trees and the $n!$ possible linear arrangements of each. In case of rooted trees, $n^{n-1} \leq t_n n!$, where t_n denotes the number of unlabeled rooted trees (Table 1). In case of free trees, $n^{n-2} \leq t'_n n!$, where t'_n denotes the number of unlabeled free trees (Table 1).

3 Advice on efficient usage

3.1 Improving a generator's performance

All generators listed in Code 2.7 modify the tree object so that subsequent calls to other functions of the library can be executed in a shorter amount of time. For example, once the tree has been generated, the overall execution is likely to be faster if a rooted tree object contains (internally) precomputed the type of tree or all the sizes of the subtrees. This means that all tree generators perform more tasks than just generating the tree, which we call *postprocessing actions* in the generation. However, in some situations it is advisable to deactivate some, or all, of these actions. These situations are those in which the information an action calculates is not needed: for example, in the current version the type of tree is barely used to do any calculations. These actions can be deactivated one by one, or all at the same time. This can be done as shown in Code 3.1.

```
import lal
Gen = lal.generate.all_ulab_rooted_trees(10)
Gen.set_calculate_size_subtrees(False) # do not calculate subtree sizes
Gen.set_calculate_tree_type(False) # do not compute the tree's type
Gen.set_normalize_tree(False) # do not normalize the tree
# alternatively, one can deactivate all prostprocessing
# actions in a single call
Gen.deactivate_all_postprocessing_actions()
```

Code 3.1: Deactivating options in a generator to improve its performance.

3.2 Free vs rooted trees

Many functions in the library admit both types of trees: free and rooted trees. Whenever a function admits both trees, it usually implies a conversion from rooted tree to free tree (read the documentation of the appropriate functions). Therefore, it is advisable to use Code 2.2 to convert a rooted tree into a free tree in those cases where it is indicated that a conversion is done. Besides in some of the functions, there is also a conversion in the generator of planar arrangements (which admits both free and rooted trees). Such conversion can be avoided when initializing the generator with a free tree instead of a free tree.

3.3 LAL's release compilation

Users acquainted with a more advanced use of the library can opt to choose to use the release compilation instead of the debug compilation: on the one hand, such a compilation does not contain any debug symbols hence making finding wrong usages of LAL a bit more difficult, but on the other it can lower the execution time: for example, Code 2.5 takes ~ 110 seconds using the debug compilation, but only ~ 14 seconds using the release one. In order to use LAL's release compilation simply change the `import` line as shown in Code 3.2. This, however, is discouraged for users who are still learning how to use LAL since many checks

```
import laloptimized as lal
```

Code 3.2: Import LAL's release compilation.

References

[1] R. Ferrer-i-Cancho. "Euclidean distance between syntactically linked words". In: *Physical Review E* 70 (2004), p. 056135.

[2] 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.

[3] H. Liu. "Dependency distance as a metric of language comprehension difficulty". In: *Journal of Cognitive Science* 9 (2008), pp. 159–191.

[4] Y. Albert Park and Roger Levy. "Minimal-length linearizations for mildly context-sensitive dependency trees". In: *Proceedings of the 10th Annual Meeting of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT) conference*. Boulder, Colorado, USA: Association for Computational Linguistics, 2009, pp. 335–343.

[5] David Gildea and David Temperley. "Do Grammars Minimize Dependency Length?" In: *Cognitive Science* 34.2 (2010), pp. 286–310. DOI: https://doi.org/10.1111/j.1551-6709.2009.01073.x.

[6] 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.

[7] Carlos Gómez-Rodríguez, John Carroll, and David Weir. "Dependency Parsing Schemata and Mildly Non-Projective Dependency Parsing". In: *Computational Linguistics* 37.3 (2011), pp. 541–586. DOI: https://doi.org/10.1162/COLI_a_00060.

[8] 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.

[9] J. L. Esteban, R. Ferrer-i-Cancho, and C. Gómez-Rodríguez. "The scaling of the minimum sum of edge lengths in uniformly random trees". In: *Journal of Statistical Mechanics* (2016), p. 063401.

[10] Carlos Gómez-Rodríguez, Morten H. Christiansen, and Ramon Ferrer-i-Cancho. "Memory limitations are hidden in grammar". In: (2019). arXiv: 1908.06629 [cs.CL].

[11] Lluís Alemany-Puig and Ramon Ferrer-i-Cancho. "Linear-time calculation of the expected sum of edge lengths in projective linearizations of trees". In: *in preparation* (2021). in preparation: --. URL: --.

[12] *Labeled Graph*. https://mathworld.wolfram.com/LabeledGraph.html.

[13] Neil J. A. Sloane. *Online Encyclopedia of Integer Sequences*. https://oeis.org.

[14] Neil J. A. Sloane. *Online Encyclopedia of Integer Sequences*. https://oeis.org/A000055.

[15] Neil J. A. Sloane. *Online Encyclopedia of Integer Sequences*. https://oeis.org/A000081.

[16] *UD_Czech-CAC*. https://github.com/UniversalDependencies/UD_Czech-CAC/blob/master/cs_cac-ud-train.conllu.

[17] *UD_Munduruku-CAC*. https://github.com/UniversalDependencies/UD_Munduruku-TuDeT/blob/master/myu_tudet-ud-test.conllu.

[18] *Universal Dependencies*. https://universaldependencies.org.

[19] *Unlabeled Graph*. https://mathworld.wolfram.com/UnlabeledGraph.html.