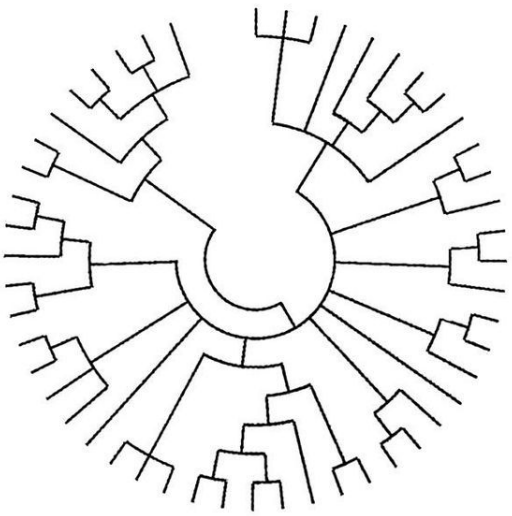
DOCUMENTATION FOR THE PYTHON LIBRARY PEDIGRAD.PY

Rémy Tuyéras
Department of Mathematics
Massachusetts Institute of Technology

Version 1.3



Typeset in LaTeX 2_ε.

Contents

Introduction

1.1. About pedigrads and `Pedigrad.py`

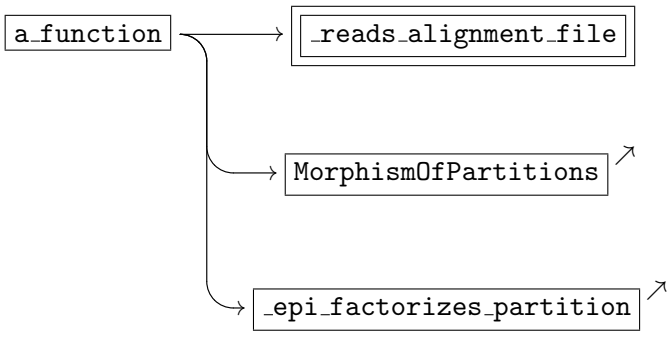
Pedigrads are mathematical tools that were initially created to model various mechanisms of genetics (see [5]). Mathematically, pedigrads are defined as functors sending the cones of a certain type of limit sketch to cones in a given category of values. Pedigrads can encode different aspects of biology depending on their 'categories of values' and associated cones. So far, the python library `Pedigrad.py` only includes pedigrads taking their values in the category of partitions whose cones are product cones (see [4]).

1.2. About this documentation

The present book contains a tutorial (see Chapter 2) explaining how to use the various methods and classes contained in `Pedigrad.py` as well as a description of the (importable and non-importable) functions of its sub-modules

- `PartitionCategory.py` (see Chapter 3)
- `SegmentCategory.py` (see Chapter 4)
- `PedigradCategory.py` (see Chapter 5)
- `AsciiTree.py` (see Chapter 6)
- `Phylogeny.py` (see Chapter 7)

A description of a function will always start with a dependcy flow chart as given below if the function depends on other procedures.



A double box indicates that the function does not have any dependencies while an arrow on the top-right corner of a box means that the function belongs to another module of the library.

Also, the python code will always be specified in text editor mode as follows.

```
1 class Pedigrad:
2   """
3   This is a comment
4   """
5   def __init__(self,alignment):  # comment:  constructor of the class
6     """
7     Another comment about the code
8     """
```

A console mode is also used for most of the examples (see below).

```
>>> print(P.loci)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13]
```

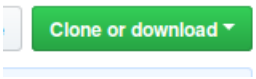
1.3. Acknowledgments

I would like to thank Maxim Wolf and Carles Boix for interesting discussions about DNA and genetics. I would also like to thank Maxim Wolf for answering many of my questions and giving me some of his time.

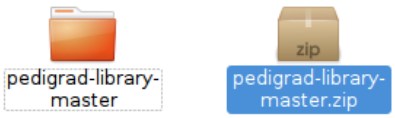
Tutorial

2.1. Installation and preparation

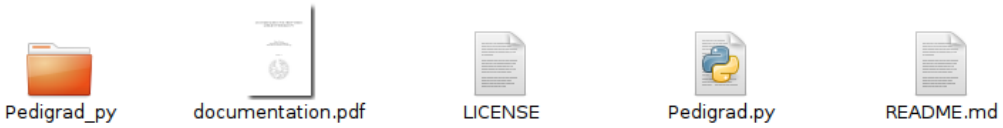
To install the library, first download the package by clicking on the green button on the right of the screen at https://github.com/remytuyeras/pedigrad-library .



The downloaded package should be a compressed file named `pedigrad-library-master.zip`. Create a new directory in which you can copy and extract the compressed file using your favorite extraction application.



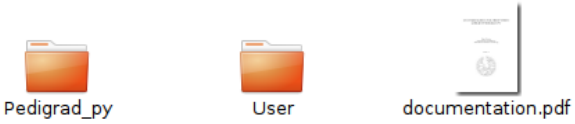
Enter the extracted directory whose inside should look as follows.



In this tutorial, the use of the library `Pedigrad` will require the creation of four files:

- ▷ two multiple sequence alignment files (usually with an extension `.fa` or `.fasta`);
- ▷ a file in which a pre-ordered set is specified (preferably with an extension `.yml`);
- ▷ a python file in which the user uses the library (the extension is obviously `.py`).

To do this properly, create a new directory in `pedigrad-library-master`, call it `User`, and copy the file `Pedigrad.py` in `User`.



Open the file `Pedigrad.py` that you copied in `User`. Once inside, you will be able to see several instances of the function

`sys.path.insert(0,−)`

in which paths appear in the second argument. Add the text ../ at the beginning of every path passed to the function `sys.path.insert(0,−)`, as shown below[1]. [UPDATE]

```
#-----------------------------------------------------------
import sys
sys.path.insert(0, 'Pedigrad_py/PartitionCategory/')
from PartitionCategory import *

#print_partition(partition): standard output

#EquivalenceRelation: .classes, .range, .closure, .quoti

#product_of_partitions(partition1,partition2): list

#coproduct_of_partitions(partition1,partition2): list

#MorphismOfPartitions: .arrow, .source, .target

#homset_is_inhabited(source,target): Boolean

#-----------------------------------------------------------
import sys
sys.path.insert(0, 'Pedigrad_py/SegmentCategory/')
from SegmentCategory import *

#SegmentObject: .colors, .topology

#CategoryOfSegments: .domain, .mask, .preorder

#-----------------------------------------------------------
import sys
sys.path.insert(0, 'Pedigrad_py/PedigradCategory/')
from PedigradCategory import *
```

```
#-----------------------------------------------------------
import sys
sys.path.insert(0, '../Pedigrad_py/PartitionCategory/')
from PartitionCategory import *

#print_partition(partition): standard output

#EquivalenceRelation: .classes, .range, .closure, .quoti

#product_of_partitions(partition1,partition2): list

#coproduct_of_partitions(partition1,partition2): list

#MorphismOfPartitions: .arrow, .source, .target

#homset_is_inhabited(source,target): Boolean

#-----------------------------------------------------------
import sys
sys.path.insert(0, '../Pedigrad_py/SegmentCategory/')
from SegmentCategory import *

#SegmentObject: .colors, .topology

#CategoryOfSegments: .domain, .mask, .preorder

#-----------------------------------------------------------
import sys
sys.path.insert(0, '../Pedigrad_py/PedigradCategory/')
from PedigradCategory import *
```

Once the paths are all updated, create, in the directory `User`, four new files with the names `unknown.fa`; `ribosome.fa`; `main.py`; and `omega.yml`. [UPDATE]

Open the file `main.py` and insert the following piece of code.

```
from Pedigrad import *
```

We are now ready to use the library – proceed to section 2.2.

2.2. Multiple sequence alignments

In this section, we are going to fill the files named `unknown.fa` and `ribosome.fa` with multiple sequence alignments by following the main example of [4].

The name of the taxa of a multiple sequence alignment should always be preceded by the symbol `>` and followed by a new line character (obtained by using the return key ⏎). The DNA sequence associated with the taxon should then be displayed. Note that the sequence can contain new line characters, but cannot contain the character `>`.

```
>Human
ATGTCAGACTGCTACACGGAGCTGGAGAAGGCAGTCATTGTCCTGGTGGAAAACTTCTACAAATATGTGT
CTAAGTACAGCCTGGTCAAGAACAAGATCAGCAAGAGCAGCTTCCGCGAGATGCTCCAGAAAGAGCTGAA
CCACATGCTGTCGGACACAGGGAACCGGAAGGCTGCGGATAAGCTCATCCAGAACCTGGATGCCAATCAT
GATGGGCGCATCAGCTTCGATGAGTACTGGACCTTGATAGGCGGCATCACCGGCCCCATCGCCAAACTCA
TCCATGAGCAGGAGCAGCAGAGCAGCAGCTAG
>Chimp
ATGTCAGACTGCTACACGGAGCTGGAGAAGGCAGTCATTGTCCTGGTGGAAAACTTCTACAAATATGTGT
CTAAGTACAGCCTGGTCAAGAACAAGATCAGCAAGAGCAGCTTCCGCGAGATGCTCCAGAAAGAGCTGAA
CCACATGCTGTCGGACACAGGGAACCGGAAGGCTGCGGATAAGCTCATCCAGAACCTGGATGCCAATCAT
GATGGGCGCATCAGCTTCGATGAGTACTGGACCTTGATAGGCGGCATCACCGGCCCCATCGCCAAACTCA
TCCGTGAGCAGGAGCAGCAGAGCAGCAGCTAG
>Mouse_lemur
ATGGCGGACTGCTACACGGAGCTGGAGAAGGCGGTCGTTGTCCTGGTGGAAAACTTCTACAAATACGTGT
CTAAGCACAGCCTGGTCAAGAACAAGATCAGCAAGAGCAGCTTCCGCAAGATGCTCCAGAAAGAGCTGAA
CCATATGCTGACGGACACGGGGAACCGCAAGGCTGCAGACAAGCTCATCCAGAACCTGGATGCCAACCAC
```

[1] Note that the highlight mode of my text editor is different from the one used in this documentation.

For the file named `unknown.fa`, we shall consider the following alignment, which the reader can now can copy and paste in `unknown.fa`.

unknown.fa

```
>TaxonA
ACGCTAGCAGTTTGGTTGCTCCAGCTG
>TaxonB
ACGTTAGTGCTCTGGTTGCCCTGGCCA
>TaxonC
ACGTTAGATCTCTGATTGGCCGAGTTA
>TaxonD
GCACCGGACATGCTATAGGTCGAATCA
>TaxonE
GTGCCGGACACACTATAGATCGAATCG
>TaxonF
ATGCTGGATAATCAATCGATCGATCCG
```

Similarly, copy and paste the following alignment in the file named `ribosome.fa`.

ribosome.fa

```
>TaxonA
ACGTGCTTGTTGGTCGCCTGT
>TaxonB
ACATGTTTGCTGGCCGTCTGT
>TaxonC
ACGTGTTTATCGACTGTCTGT
>TaxonD
ACACGCTTGCCGATCATCTAT
>TaxonE
ATGCGCTTACCGATCATCTAT
>TaxonF
ACGTGCTTACTGATCACCTGT
```

Once these texts have been copied and pasted in their respective files, open the file named `main.py` and add the following piece of code.

```
ribosome = read_alignment_file("ribosome.fa",READ_DNA)
for i in range(len(ribosome[0])):
  print(str(i)+":  "+str(ribosome[0][i]))
print("----")
for i in range(len(ribosome[1])):
  print(str(i)+":  "+str(ribosome[1][i]))
```

Save and compile `main.py` – you should obtain the following result in the console.

```
0:  TaxonA
1:  TaxonB
2:  TaxonC
3:  TaxonD
```

```
4:  TaxonE
5:  TaxonF
----
0:  ['A', 'C', 'G', 'T', 'G', 'C', 'T', 'T', 'G', 'T', 'T', 'G', 'G', 'T',
'C', 'G', 'C', 'C', 'T', 'G', 'T']
1:  ['A', 'C', 'A', 'T', 'G', 'T', 'T', 'T', 'G', 'C', 'T', 'G', 'G', 'C',
'C', 'G', 'T', 'C', 'T', 'G', 'T']
2:  ['A', 'C', 'G', 'T', 'G', 'T', 'T', 'T', 'A', 'T', 'C', 'G', 'A', 'C',
'T', 'G', 'T', 'C', 'T', 'G', 'T']
3:  ['A', 'C', 'A', 'C', 'G', 'C', 'T', 'T', 'G', 'C', 'C', 'G', 'A', 'T',
'C', 'A', 'T', 'C', 'T', 'A', 'T']
4:  ['A', 'T', 'G', 'C', 'G', 'C', 'T', 'T', 'A', 'C', 'C', 'G', 'A', 'T',
'C', 'A', 'T', 'C', 'T', 'A', 'T']
5:  ['A', 'C', 'G', 'T', 'G', 'C', 'T', 'T', 'A', 'C', 'T', 'G', 'A', 'T',
'C', 'A', 'C', 'C', 'T', 'G', 'T']
```

As can be seen, the procedure `read_alignment_file` can be used to collect the name of the taxa of the multiple sequence alignment given in its first argument as well as the DNA sequence of each of them in the form of lists of characters. See section 5.1 for a detailed description of the procedure `read_alignment_file`.

Note that the lists contained in `ribosome[1]` include all the nucleotides of the alignment. To only select the columns that present noticeable mutations, one can use the procedure `column_is_trivial` (see section 5.4).

For illustration, replace the current code contained in the file `main.py` with the following piece of code.

```
1 from Pedigrad import *
2
3 ribosome = read_alignment_file("ribosome.fa",READ_DNA)
4
5 indices = list()
6 transpose = list()
7 for i in range(len(ribosome[1][0])):
8   column = list()
9   for j in range(len(ribosome[1])):
10     column.append(ribosome[1][j][i])
11   if not(column_is_trivial(column,[])):
12     transpose.append(column)
13     indices.append(i)
14
15 for i in range(len(transpose)):
16   print(str(i+1) + ") at index " + str(indices[i])+":  " +
   str(transpose[i]))
```

Now, save and compile `main.py` – you should obtain the following result in the console.

```
1) at index 1:  ['C', 'C', 'C', 'C', 'T', 'C']
2) at index 2:  ['G', 'A', 'G', 'A', 'G', 'G']
3) at index 3:  ['T', 'T', 'T', 'C', 'C', 'T']
4) at index 5:  ['C', 'T', 'T', 'C', 'C', 'C']
5) at index 8:  ['G', 'G', 'A', 'G', 'A', 'A']
6) at index 9:  ['T', 'C', 'T', 'C', 'C', 'C']
7) at index 10:  ['T', 'T', 'C', 'C', 'C', 'T']
8) at index 12:  ['G', 'G', 'A', 'A', 'A', 'A']
9) at index 13:  ['T', 'C', 'C', 'T', 'T', 'T']
10) at index 14:  ['C', 'C', 'T', 'C', 'C', 'C']
11) at index 15:  ['G', 'G', 'G', 'A', 'A', 'A']
12) at index 16:  ['C', 'T', 'T', 'T', 'T', 'C']
13) at index 19:  ['G', 'G', 'G', 'A', 'A', 'G']
```

We are now ready to proceed to the next section of the tutorial.

2.3. Pre-ordered sets

While the mathematical formalism of [4] allows us to use pre-ordered sets with only two elements (called 'colors'), the present tutorial will use a pre-ordered set with four colors to make our the code clearer.

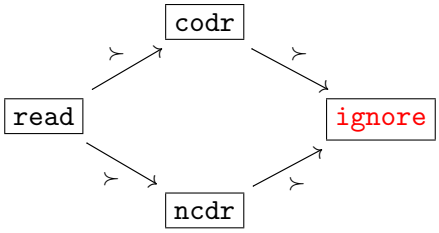
From this section on, the present tutorial follows the method described in [4] to compute the phylogeny of our set of taxa and determine whether the multiple sequence alignment of `unknown.fa` is a coding or non-coding region. In this respect, we want to use at least three colors: two colors to read coding and non-coding regions and another one to read any type of region.

To do so, open the file named `omega.yml` and copy the following piece of code.

omega.yml

```
!obj:
  - read #color to read any region
  - codr #color to read a coding region
  - ncdr #color to read a non-coding region
rel:
  - read > codr;
  - read > ncdr;
```

The previous lines define a pre-ordered set with four elements and four generating relations as follows.



The element called `ignore`, which does not appear in the code entered in `omega.yml`, is specified via the symbol `!`, appearing right before the key word `obj`. The symbol `!` formally adds an minimum element to the pre-ordered set of `omega.yml`. This minimum element will later allow us to use blanks in the specification of segments (see section 4.1 for more detail).

2.4. Categories of segments

In this section, we use the pre-order structure defined in section 2.3 to define various parsing methods that we will use to analysis the multiple sequence alignments contained in `unknown.fa` and `ribosome.fa`.

To do so, we need to use what is called a *category of quasi-homologous segments* (see [5, 4]). The definition of this 'category', usually denoted as $\mathbf{Seg}(\Omega, n)$, depends on a pre-ordered set Ω and an non-negative integer n, which stands for the length of the DNA strands in the multiple sequence alignment under study. In the present library, a category of quasi-homologous segments can be defined by using the class `SegmentCategory` (section 4.4). More advanced classes can be used, but these classes are actually subclasses of the class `SegmentCategory` so that focusing on `SegmentCategory` first may be more appropriate for this tutorial. To call an instance of `SegmentCategory`, replace the current code of `main.py`, between line 15 to line 16, with the following lines.

```
15 Seg_rb = CategoryOfSegments("omega.yml",len(ribosome[1][0]))
16
17 unknown = read_alignment_file("unknown.fa",READ_DNA)
18 Seg_ukn = CategoryOfSegments("omega.yml",len(unknown[1][0]))
```

The variables `Seg_rb` and `Seg_ukn` can now be used to specify parsing methods for the multiple sequence alignments of `unknown.fa` and `ribosome.fa` by using the set of 'colors' `read`, `codr` and `ncdr` defined in `omega.yml`. Specifically, the parsing of a piece of information is represented by what is called a *segment* in [4], which is to say an object in the category $\mathbf{Seg}(\Omega, n)$ (see below). Segments are usually represented as partitioned sequences of colored nodes as follows.

(○○○)(●●●)(●●)(●)(●)(●●)(●●●)(●●)(●)(○○○)

○ = `ignore`
● = `codr`
● = `ncdr`
● = `read`

As can be seen, the colors of the segments stand for elements of the pre-ordered structure given in `omega.yml`. From the user's point of view, the previous segments can be specified by the following regular expression.

```
[(3,3,1,'codr'), (6,2,1,'read'), (8,1,2,'read'), (10,2,1,'read'),
(12,3,1,'codr'), (15,2,1,'ncdr'), (17,1,1,'read')]
```

Each quadruple given in the previous list represents a colored patch of the segment. The specification of such a patch follows the following syntax:

(`start_position`, `number_of_nodes`, `number_of_times_repeated`, `the_color`)

where

- ▷ `start_position` is the index at which the patch starts in the segment;
- ▷ `number_of_nodes` is the length of the patch (i.e. the number of nodes);
- ▷ `number_of_times_repeated` is the number of times the patch is successively repeated (see below);

`[(2,3,5,'read')]` = (○○)$\underbrace{(\bullet\bullet\bullet)(\bullet\bullet\bullet)(\bullet\bullet\bullet)(\bullet\bullet\bullet)(\bullet\bullet\bullet)}_{\text{repeated 5 times}}$

- ▷ `the_color` is the the reading state at which the patch is set;

Structurally, a segment is not a regular expression (as given above), but an instance of the class `SegmentObject` (see section 4.3), which comes with two objects: `.topology` and

`.colors`. However, the type of regular expression given earlier can always be turned into a `SegmentObject` item by using the method `.segment` of the class `SegmentCategory`, as shown below (also see section 4.4.5).

```
s = Seg_ukn.segment([(3,3,1,'codr'), (6,2,1,'read'), (8,1,2,'read'),
(10,2,1,'read'), (12,3,1,'codr'), (15,2,1,'ncdr'), (17,1,1,'read')])
```

In this tutorial, we want to follow the main example of [4] and parse

- the file `ribosome.fa` according to a 1-nucleotide resolution topology with patches of 1 nucleotides colored in `read`;

$$\texttt{[(\_,1,1,'read')]} = (\circ)(\circ)\ldots(\circ)(\bullet)(\circ)(\circ)\ldots(\circ)$$

- the file `unknown.fa` according to
  1) a codon resolution topology with patches of 3 nucleotides colored in `codr`;

$$\texttt{[(\_,3,1,'codr')]} = (\circ)(\circ)\ldots(\circ)(\bullet\bullet\bullet)(\circ)(\circ)\ldots(\circ)$$

  2) a 2-nucleotide resolution topology with patches of 2 nucleotides colored in `ncdr`;

$$\texttt{[(\_,2,1,'ncdr')]} = (\circ)(\circ)\ldots(\circ)(\bullet\bullet)(\circ)(\circ)\ldots(\circ)$$

  3) a 2-nucleotide resolution topology refining the codon topology with patches of 3 nucleotides colored in `read`;

$$\texttt{[(\_,2,1,'read'),(\_+2,1,1,'read')]} = (\circ)(\circ)\ldots(\circ)(\bullet\bullet)(\bullet)(\circ)(\circ)\ldots(\circ)$$

$$\texttt{[(\_,1,1,'read'),(\_+1,2,1,'read')]} = (\circ)(\circ)\ldots(\circ)(\bullet)(\bullet\bullet)(\circ)(\circ)\ldots(\circ)$$

Because the parsing method meant to be used with `ribosome.fa` is already implemented in the present library, we shall focus on the parsing methods of `unknown.fa`. Our goal is to construct lists that contain the segments of the form specified earlier.

We start by constructing the list of segments described in item 1 by adding the following lines to the file `main.py`.

```
20 parse_coding = list()
21 for i in range(Seg_ukn.domain):
22   if i % 3 == 0:
23     parse_coding.append(Seg_ukn.segment([(i,3,1,'codr')]))
```

The user can display the list `parse_coding` by using the following lines of code.

```
25 def print_parsing_method(list_of_segments):
26   for i in range(len(list_of_segments)):
27     print((list_of_segments[i].topology,list_of_segments[i].colors))
28
29 print_parsing_method(parse_coding)
```

Compiling `main.py` should give the following display in the console.

```
([(0, 2)], ['codr'])
([(3, 5)], ['codr'])
([(6, 8)], ['codr'])
([(9, 11)], ['codr'])
([(12, 14)], ['codr'])
([(15, 17)], ['codr'])
([(18, 20)], ['codr'])
([(21, 23)], ['codr'])
([(24, 26)], ['codr'])
```

We now construct the list of segments described in item 2 by adding the following lines to the file `main.py`.

```
parse_ncoding = list()
for i in range(Seg_ukn.domain):
  if i+1 < Seg_ukn.domain:
    parse_ncoding.append(Seg_ukn.segment([(i,2,1,'ncdr')]))
```

We can display `parse_ncoding` as before by using the following line of code.

```
print_parsing_method(parse_ncoding)
```

Compiling `main.py` should give the following display in the console.

```
([(0, 1)], ['ncdr'])
([(1, 2)], ['ncdr'])
([(2, 3)], ['ncdr'])
([(3, 4)], ['ncdr'])
([(4, 5)], ['ncdr'])
([(5, 6)], ['ncdr'])
([(6, 7)], ['ncdr'])
([(7, 8)], ['ncdr'])
([(8, 9)], ['ncdr'])
([(9, 10)], ['ncdr'])
([(10, 11)], ['ncdr'])
([(11, 12)], ['ncdr'])
([(12, 13)], ['ncdr'])
([(13, 14)], ['ncdr'])
([(14, 15)], ['ncdr'])
([(15, 16)], ['ncdr'])
([(16, 17)], ['ncdr'])
([(17, 18)], ['ncdr'])
([(18, 19)], ['ncdr'])
([(19, 20)], ['ncdr'])
([(20, 21)], ['ncdr'])
([(21, 22)], ['ncdr'])
([(22, 23)], ['ncdr'])
([(23, 24)], ['ncdr'])
([(24, 25)], ['ncdr'])
([(25, 26)], ['ncdr'])
```

Finally, the list of segments described in item 3 can be constructed as follows.

```
parse_ground0 = list()
parse_ground1 = list()
for i in range(Seg_ukn.domain):
  if i % 3 == 0 and (i+2 < Seg_ukn.domain):
    parse_ground0.append(Seg_ukn.segment([(i,2,1,'read'),
(i+2,1,1,'read')]))
  if i % 3 == 0 and (i+2 < Seg_ukn.domain):
    parse_ground0.append(Seg_ukn.segment([(i,1,1,'read'),
(i+1,2,1,'read')]))
parse_ground = parse_ground0 + parse_ground1
```

Again, we can display `parse_ground` as before by using the following line of code.

```
print_parsing_method(parse_ground)
```

Compiling `main.py` should give the following display in the console.

```
([(0, 1), (2, 2)], ['read', 'read'])
([(3, 4), (5, 5)], ['read', 'read'])
([(6, 7), (8, 8)], ['read', 'read'])
([(9, 10), (11, 11)], ['read', 'read'])
([(12, 13), (14, 14)], ['read', 'read'])
([(15, 16), (17, 17)], ['read', 'read'])
([(18, 19), (20, 20)], ['read', 'read'])
([(21, 22), (23, 23)], ['read', 'read'])
([(24, 25), (26, 26)], ['read', 'read'])
([(0, 0), (1, 2)], ['read', 'read'])
([(3, 3), (4, 5)], ['read', 'read'])
([(6, 6), (7, 8)], ['read', 'read'])
([(9, 9), (10, 11)], ['read', 'read'])
([(12, 12), (13, 14)], ['read', 'read'])
([(15, 15), (16, 17)], ['read', 'read'])
([(18, 18), (19, 20)], ['read', 'read'])
([(21, 21), (22, 23)], ['read', 'read'])
([(24, 24), (25, 26)], ['read', 'read'])
```

2.5. Local analyses

While the class `SegmentCategory` allows us to define a syntax for our parsing methods, it does not allow us to specify a semantics. To do so, we need to go higher in the class hierarchy and use the subclass `LocalAnalysis` (see section 5.3).

In general, only the subclass `LocalAnalysis` is only meant to be used by its subclass `Pedigrad` (see section 5.5), but we will use it in this section to show how the semantics of our parsing methods is recorded internally.

As said in section 2.4, the parsing method that is to be used with the file `ribosome.fa` is already implemented in the library and can be called as follows.

```
49 Loc_rb = LocalAnalysis(NUCL_MODE,'read',"omega.yml",Seg_rb.domain)
```

As the reader may have noticed, the file `ribosome.fa` is not passed to the function and the only information related to `ribosome.fa` is the integer `Seg_rb.domain`, which corresponds to the number of columns of the alignment that it contains. The reason for not requiring `ribosome.fa` is that a `LocalAnalysis` item is only meant to give a parsing schema and it is only at the level of the class `Pedigrad` that this schema will be used with the content of the file `ribosome.fa`.

The following lines of code show us the type of information that the item `Loc_rb` contains, namely a mapping that relates the segments of a 1-nucleotide resolution parsing to a set of equivalence class – for this first example, the equivalence classes will be trivial (see below).

```
51 for i in range(len(Loc_rb.base)):
52   print("segment " + str(i+1) + ":  " + str(Loc_rb.base[i].topology)
   + " " + str(Loc_rb.base[i].colors) + " ---> " +
   str(ID_to_EQ(Loc_rb.equiv[i])))
```

Compiling the previous code gives the following mapping, where the trivial equivalence classes are represented by the empty lists given on the right-hand side.

```
segment 1:  [(0, 0)] ['read'] ---> [[]]
segment 2:  [(1, 1)] ['read'] ---> [[]]
segment 3:  [(2, 2)] ['read'] ---> [[]]
segment 4:  [(3, 3)] ['read'] ---> [[]]
segment 5:  [(4, 4)] ['read'] ---> [[]]
segment 6:  [(5, 5)] ['read'] ---> [[]]
segment 7:  [(6, 6)] ['read'] ---> [[]]
segment 8:  [(7, 7)] ['read'] ---> [[]]
segment 9:  [(8, 8)] ['read'] ---> [[]]
segment 10:  [(9, 9)] ['read'] ---> [[]]
segment 11:  [(10, 10)] ['read'] ---> [[]]
segment 12:  [(11, 11)] ['read'] ---> [[]]
segment 13:  [(12, 12)] ['read'] ---> [[]]
segment 14:  [(13, 13)] ['read'] ---> [[]]
segment 15:  [(14, 14)] ['read'] ---> [[]]
segment 16:  [(15, 15)] ['read'] ---> [[]]
segment 17:  [(16, 16)] ['read'] ---> [[]]
segment 18:  [(17, 17)] ['read'] ---> [[]]
segment 19:  [(18, 18)] ['read'] ---> [[]]
segment 20:  [(19, 19)] ['read'] ---> [[]]
segment 21:  [(20, 20)] ['read'] ---> [[]]
```

Let us now design the semantics of the other parsing methods. Following [4], we want to associate every segment of the list `parse_coding`, defined in section 2.4, with the following equivalence classes, which will be interpreted as identities during the parsing.

$$\texttt{ATA} \leftrightarrow \texttt{GTG} \qquad \texttt{CTA} \leftrightarrow \texttt{TTG} \qquad \texttt{TTA} \leftrightarrow \texttt{CTG} \tag{2.1}$$

To encode these equivalence classes, we need to use one of the (empty) global lists `N01_EQ`, `N02_EQ`, ..., `N21_EQ`. These variables are associated with a set of global strings `N01_ID`, `N02_ID`, ..., `N21_ID`, which we can use to refer to the equivalence classes stored in `N01_EQ`, `N02_EQ`, etc.

For the semantics associated with the parsing method `parse_coding`, we will use the global list `N01_EQ`. The specification of the equivalence classes of (2.1) is done via the following line of code.

```
54 N01_EQ.extend([["ATA","GTG"],["CTA","TTG"],["TTA","CTG"]])
```

To associate every `SegmentObject` item in `parse_coding` with the equivalence classes stored in `N01_EQ`, we want to create a list that only contains copies of the string `N01_ID` and whose length is equal to the length of the list `parse_coding`. The correspondence between the elements of `parse_coding` and the created list (which will be named `equiv_coding`, as shown below) is established by the indexing.

```
56 equiv_coding = list()
57 for i in range(len(parse_coding)):
58   equiv_coding.append(N01_ID)
```

Passing the lists `parse_coding` and `equiv_coding` to the procedure `LocalAnalysis` creates a structure that records the semantics of the parsing.

```
60 Loc_coding = LocalAnalysis(SEGM_MODE,parse_coding,equiv_coding,"omega.yml",
   Seg_ukn.domain)
```

The mapping can be displayed by using the following piece of code.

```
62 for i in range(len(Loc_coding.base)):
63   print("segment " + str(i+1) + ":  " + str(Loc_coding.base[i].topology)
   + " " + str(Loc_coding.base[i].colors) + " ---> " +
   str(ID_to_EQ(Loc_coding.equiv[i])))
```

Compiling gives us the following associations.

```
segment 1:  [(0, 2)] ['codr'] ---> [['ATA', 'GTG'], ['CTA', 'TTG'], ['TTA',
'CTG']]
segment 2:  [(3, 5)] ['codr'] ---> [['ATA', 'GTG'], ['CTA', 'TTG'], ['TTA',
'CTG']]
segment 3:  [(6, 8)] ['codr'] ---> [['ATA', 'GTG'], ['CTA', 'TTG'], ['TTA',
'CTG']]
segment 4:  [(9, 11)] ['codr'] ---> [['ATA', 'GTG'], ['CTA', 'TTG'],
['TTA', 'CTG']]
segment 5:  [(12, 14)] ['codr'] ---> [['ATA', 'GTG'], ['CTA', 'TTG'],
['TTA', 'CTG']]
segment 6:  [(15, 17)] ['codr'] ---> [['ATA', 'GTG'], ['CTA', 'TTG'],
['TTA', 'CTG']]
segment 7:  [(18, 20)] ['codr'] ---> [['ATA', 'GTG'], ['CTA', 'TTG'],
['TTA', 'CTG']]
segment 8:  [(21, 23)] ['codr'] ---> [['ATA', 'GTG'], ['CTA', 'TTG'],
['TTA', 'CTG']]
segment 9:  [(24, 26)] ['codr'] ---> [['ATA', 'GTG'], ['CTA', 'TTG'],
['TTA', 'CTG']]
```

We proceed similarly for the parsing method of `parse_ncoding`. This time, every segment of `parse_ncoding` should be associated with the following equivalence class.

$$\texttt{CA} \leftrightarrow \texttt{TG}$$

This equivalence class can be saved in the variable `N02_EQ` as follows.

```
65 N02_EQ.extend([["CA","TG"]])
```

Then, we proceed as before by creating a list that only contains the variable `N02_ID` and whose length is equal to the length of the list `parse_ncoding`.

```
67 equiv_ncoding = list()
68 for i in range(len(parse_ncoding)):
69   equiv_ncoding.append(N02_ID)
```

Passing the lists `parse_ncoding` and `equiv_ncoding` to the procedure `LocalAnalysis` creates a structure that records the semantics of the parsing.

```
71 Loc_ncoding = LocalAnalysis(SEGM_MODE,parse_ncoding,equiv_ncoding,"omega.yml",
   Seg_ukn.domain)
```

The mapping can be displayed by using the following piece of code.

```
73 for i in range(len(Loc_ncoding.base)):
74   print("segment " + str(i+1) + ":  " + str(Loc_ncoding.base[i].topology)
   + " " + str(Loc_ncoding.base[i].colors) + " ---> " +
   str(ID_to_EQ(Loc_ncoding.equiv[i])))
```

Compiling gives us the following associations.

```
segment 1:  [(0, 1)] ['ncdr'] ---> [['CA', 'TG']]
segment 2:  [(2, 3)] ['ncdr'] ---> [['CA', 'TG']]
segment 3:  [(4, 5)] ['ncdr'] ---> [['CA', 'TG']]
segment 4:  [(6, 7)] ['ncdr'] ---> [['CA', 'TG']]
segment 5:  [(8, 9)] ['ncdr'] ---> [['CA', 'TG']]
segment 6:  [(10, 11)] ['ncdr'] ---> [['CA', 'TG']]
segment 7:  [(12, 13)] ['ncdr'] ---> [['CA', 'TG']]
segment 8:  [(14, 15)] ['ncdr'] ---> [['CA', 'TG']]
segment 9:  [(16, 17)] ['ncdr'] ---> [['CA', 'TG']]
segment 10:  [(18, 19)] ['ncdr'] ---> [['CA', 'TG']]
segment 11:  [(20, 21)] ['ncdr'] ---> [['CA', 'TG']]
segment 12:  [(22, 23)] ['ncdr'] ---> [['CA', 'TG']]
segment 13:  [(24, 25)] ['ncdr'] ---> [['CA', 'TG']]
```

Finally, the parsing method of `parse_ground` will be used differently from the way those of `parse_coding` and `parse_ncoding` are used and will therefore not need to be associated with any list of equivalence classes.

2.6. Categories of pedigrads

We now introduce a subclass of `LocalAnalysis` called `Pedigrad` (described in section 5.5). As already mentioned at the beginning of section 2.5, the methods of this class will allow us to implement the mapping stored in a `LocalAnalysis` item with respect to the multiple sequence alignments given in section 2.2. This mapping will later be used to construct the phylogeny of our group of taxa.

Let us call our first `Pedigrad` item, which we are going to use to read the multiple sequence alignment saved in the file `ribosome.fa`. We do so by adding the following line of code to `main.py`.

```
76 P_rb = Pedigrad("ribosome.fa",READ_DNA,NUCL_MODE,'read',"omega.yml")
```

The parameters passed to the previous constructor works as follows:

- ▷ `"ribosome.fa"` tells the `Pedigrad` item to read the multiple sequence alignment contained in the file `ribosome.fa`;
- ▷ `READ_DNA` tells the `Pedigrad` item that DNA sequences are contained in the file `"align.fa"` so that any lower case nucleotide `a`, `c`, `g` and `t` is read as an upper case nucleotide (i.e. `A`, `C`, `G` and `T`);
- ▷ `NUCL_MODE` tells the `Pedigrad` item that the DNA sequences of `ribosome.fa` should be parsed with respect to their nucleotidic topology (1-nucleotide resolution);
- ▷ `'read'` tells the `Pedigrad` item to associate every segment representing a nucleotide with the color `read` – in the present case, this association is more formal than useful;
- ▷ `"omega.yml"` tells the `Pedigrad` item that the color `read` follow the rules specified in the file `"omega.yml"`;

Every `Pedigrad` item possesses an object `.local` that records the vertical section of the multiple sequence alignment associated with the segments of the object `.base` (given by the superclass `LocalAnalysis`). For instance, let us add the following lines to the file `main.py`.

```
78 for i in range(len(P_rb.local)):
79   print("segment "+str(P_rb.base[i].topology) + " " +
   str(P_rb.base[i].colors) + " ---> " + str(P_rb.local[i]))
```

Compiling should add the following lines to the output.

```
segment [(1, 1)] ['read'] ---> ['C', 'C', 'C', 'C', 'T', 'C']
segment [(2, 2)] ['read'] ---> ['G', 'A', 'G', 'A', 'G', 'G']
segment [(3, 3)] ['read'] ---> ['T', 'T', 'T', 'C', 'C', 'T']
segment [(5, 5)] ['read'] ---> ['C', 'T', 'T', 'C', 'C', 'C']
segment [(8, 8)] ['read'] ---> ['G', 'G', 'A', 'G', 'A', 'A']
segment [(9, 9)] ['read'] ---> ['T', 'C', 'T', 'C', 'C', 'C']
segment [(10, 10)] ['read'] ---> ['T', 'T', 'C', 'C', 'C', 'T']
segment [(12, 12)] ['read'] ---> ['G', 'G', 'A', 'A', 'A', 'A']
segment [(13, 13)] ['read'] ---> ['T', 'C', 'T', 'C', 'T', 'T']
segment [(14, 14)] ['read'] ---> ['C', 'C', 'T', 'C', 'C', 'C']
segment [(15, 15)] ['read'] ---> ['G', 'G', 'G', 'A', 'A', 'A']
segment [(16, 16)] ['read'] ---> ['C', 'T', 'T', 'T', 'T', 'C']
segment [(19, 19)] ['read'] ---> ['G', 'G', 'G', 'A', 'A', 'G']
```

Note that the columns of the multiple sequence alignment of `ribosome.fa` that are indexed by `0`, `4`, `6`, `7`, `11`, `17`, `18` and `20` are not present in the previous display. The reason for this is that the segments associated with these indices are not part of `P.base` due to the fact that the columns associated to these indices are trivial (i.e. they each contain a unique character).

Every `Pedigrad` item possesses a method `.partition` that allows us to associate any segment with a partition that is the product of all those partitions associated with those segments, in `.base`, that can be derived, via a morphism of segment, from the given input segment (see section 5.5.1 for more detail). Below, we give an example for three different segments.

```
81 print(P_rb.partition([(5,1,1,'read')],EXPR_MODE))
82 print(P_rb.partition([(5,1,1,'read'),(6,1,1,'read')],EXPR_MODE))
83 print(P_rb.partition([(5,1,1,'read'),(9,1,1,'read')],EXPR_MODE))
```

Compiling should display the three following partitions.

```
[0, 1, 1, 0, 0, 0]
[0, 1, 1, 0, 0, 0]
[0, 1, 2, 3, 3, 3]
```

The class `Pedigrad` also possesses a method `.isolate` that allows us to isolate the taxa associated with certain characters in every columns. The reason for such a method is that multiple sequence alignments usually contain special characters such as the dot character, which stands for a missing character. A way to deal with these characters would be to call the procedure `P_rb.isloate(not(NEW),['.'])`. If the label `NEW` is used instead of `not(NEW)`, then the procedure creates a new `pedigrad` item where the dot characters are islolated and the pedigrad `P_rb` is not modified (see section 5.5.2).

The example given below isolates all the characters `C` is a new pedigrad `Q`, but `P_rb` is untouched.

```
85 Q = P_rb.isolate(NEW,['C'])
86
87 for i in range(len(Q.local)):
88   print(str(Q.base[i].topology) + "--->" +str(Q.local[i]))
89
90 print(Q.partition([(5,1,1,'read')],EXPR_MODE))
91 print(Q.partition([(5,1,1,'read'),(6,1,1,'read')],EXPR_MODE))
92 print(Q.partition([(5,1,1,'read'),(9,1,1,'read')],EXPR_MODE))
```

Compiling the previous lines of code gives the following output. The symbols `>n`, where `n` is an integer, are used to isolate what are represented by symbols `C` in `P_rb.local`.

```
segment [(1, 1)] ['read'] ---> ['>0', '>1', '>2', '>3', 'T', '>4']
segment [(2, 2)] ['read'] ---> ['G', 'A', 'G', 'A', 'G', 'G']
segment [(3, 3)] ['read'] ---> ['T', 'T', 'T', '>0', '>1', 'T']
segment [(5, 5)] ['read'] ---> ['>0', 'T', 'T', '>1', '>2', '>3']
segment [(8, 8)] ['read'] ---> ['G', 'G', 'A', 'G', 'A', 'A']
segment [(9, 9)] ['read'] ---> ['T', '>0', 'T', '>1', '>2', '>3']
segment [(10, 10)] ['read'] ---> ['T', 'T', '>0', '>1', '>2', 'T']
segment [(12, 12)] ['read'] ---> ['G', 'G', 'A', 'A', 'A', 'A']
segment [(13, 13)] ['read'] ---> ['T', '>0', 'T', '>1', 'T', 'T']
segment [(14, 14)] ['read'] ---> ['>0', '>1', 'T', '>2', '>3', '>4']
segment [(15, 15)] ['read'] ---> ['G', 'G', 'G', 'A', 'A', 'A']
segment [(16, 16)] ['read'] ---> ['>0', 'T', 'T', 'T', 'T', '>1']
segment [(19, 19)] ['read'] ---> ['G', 'G', 'G', 'A', 'A', 'G']
[0, 1, 1, 2, 3, 4]
[0, 1, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
```

Later on, in the tutorial, we will use two other `Pedigrad` items, named `P_codr` and `P_ncdr`, which read the multiple sequence alignment of `unknown.fa` in the same ways as it is specified by the `LocalAnalysis` items `Loc_coding` and `Loc_ncoding` (see section 2.5). The following lines of code are meant to call these particular pedigrads, where, this time, the global variable `SEGM_MODE` is used (instead of `NUCL_MODE`) to mean that the specification of the object `.base` and the related equivalence classes are specified by hand (see section 5.5 and section 5.3).

```
94 P_codr = Pedigrad("unknown.fa",READ_DNA,SEGM_MODE,parse_coding,
   equiv_coding,"omega.yml")
95 P_ncdr = Pedigrad("unknown.fa",READ_DNA,SEGM_MODE,parse_ncoding,
   equiv_ncoding,"omega.yml")
```

We can have a first understanding of `P_codr` and `P_ncdr` by looking at their objects `.local`. Let us start by looking at `P_codr.local` by adding the following lines of code to `main.py`.

```
97 for i in range(len(P_codr.local)):
98   print("segment "+str(P_codr.base[i].topology) + " " +
   str(P_codr.base[i].colors) + " ---> " + str(P_codr.local[i]))
```

Compiling gives the output displayed below, in which the integers `0`, `1` and `2` are the representative elements of the three equivalence classes given in (2.1).

```
segment [(0, 2)] ['codr'] ---> ['ACG', 'ACG', 'ACG', 'GCA', 0, 'ATG']
segment [(3, 5)] ['codr'] ---> [1, 2, 2, 'CCG', 'CCG', 2]
segment [(6, 8)] ['codr'] ---> ['GCA', 0, 'GAT', 'GAC', 'GAC', 'GAT']
segment [(9, 11)] ['codr'] ---> ['GTT', 'CTC', 'CTC', 'ATG', 'ACA', 'AAT']
segment [(12, 14)] ['codr'] ---> ['TGG', 'TGG', 'TGA', 1, 1, 'CAA']
segment [(15, 17)] ['codr'] ---> [1, 1, 1, 'TAG', 'TAG', 'TCG']
segment [(18, 20)] ['codr'] ---> ['CTC', 'CCC', 'GCC', 'GTC', 'ATC', 'ATC']
segment [(21, 23)] ['codr'] ---> ['CAG', 'TGG', 'GAG', 'GAA', 'GAA', 'GAT']
segment [(24, 26)] ['codr'] ---> [2, 'CCA', 2, 'TCA', 'TCG', 'CCG']
```

Let us now look at `P_ncdr.local` by adding the following lines of code to `main.py`.

```
100 for i in range(len(P_ncdr.local)):
101   print("segment "+str(P_ncdr.base[i].topology) + " " +
    str(P_ncdr.base[i].colors) + " ---> " + str(P_ncdr.local[i]))
```

This time, we obtain the following output, in which the integer `0` is the representative element of the equivalence class `CA` $\leftrightarrow$ `TG`.

```
segment [(0, 1)] ['ncdr'] ---> ['AC', 'AC', 'AC', 'GC', 'GT', 'AT']
segment [(2, 3)] ['ncdr'] ---> ['GC', 'GT', 'GT', 'AC', 'GC', 'GC']
segment [(4, 5)] ['ncdr'] ---> ['TA', 'TA', 'TA', 'CG', 'CG', 0]
segment [(6, 7)] ['ncdr'] ---> ['GC', 'GT', 'GA', 'GA', 'GA', 'GA']
segment [(8, 9)] ['ncdr'] ---> ['AG', 'GC', 'TC', 0, 0, 'TA']
segment [(10, 11)] ['ncdr'] ---> ['TT', 'TC', 'TC', 0, 0, 'AT']
segment [(12, 13)] ['ncdr'] ---> [0, 0, 0, 'CT', 'CT', 0]
segment [(14, 15)] ['ncdr'] ---> ['GT', 'GT', 'AT', 'AT', 'AT', 'AT']
segment [(16, 17)] ['ncdr'] ---> [0, 0, 0, 'AG', 'AG', 'CG']
segment [(18, 19)] ['ncdr'] ---> ['CT', 'CC', 'GC', 'GT', 'AT', 'AT']
segment [(20, 21)] ['ncdr'] ---> ['CC', 'CT', 'CG', 'CG', 'CG', 'CG']
segment [(22, 23)] ['ncdr'] ---> ['AG', 'GG', 'AG', 'AA', 'AA', 'AT']
segment [(24, 25)] ['ncdr'] ---> ['CT', 'CC', 'TT', 'TC', 'TC', 'CC']
```

Finally, note that, even though `P_codr` and `P_ncdr` use the colors `codr` and `ncdr`, the way the pre-order structure `omega.yml` was design allows us to use segments of color `read` to query partitions, as shown below.

```
103 print(P_codr.partition([(0,2,1,'read'),(2,1,1,'read')],EXPR_MODE))
104 print(P_ncdr.partition([(0,2,1,'read'),(2,1,1,'read')],EXPR_MODE))
```

The result of these calls is given below.

```
[0, 0, 0, 1, 2, 3]
[0, 0, 0, 1, 2, 3]
```

2.7. Phylogeneses and phylogenies

The goal of the present section is to implement the algorithm described in [4, Theorem 3.37]. Before even implementing this algorithm, we need to change the file `Pedigrad.py` in order to use a non importable function from the library– this will give us the opportunity to illustrate how to import the non importable functions described in this documentation.

The non importable function that we want to use is `_preimage_of_partition` (see section 3.3). To import it, insert the following piece of code in line 16 of `Pedigrad.py`.

```
16 from PartitionCategory import _preimage_of_partition
17 def preimage_of_partition(*args):
19   return _preimage_of_partition(*args)
```

This function can allow us to better see what partitions look like. This is illustrated below with the object `.local` of our `Pedigrad` item `P_rb` defined in the file `main.py`. For this, append the following lines of code to the file `main.py`.

```
106 print("\n{{Alignment}}\n")
107 for i in range(len(P_rb.local)):
108   print("Segment "+str(P_rb.base[i].topology)+ " ---> " +
    str(preimage_of_partition(P_rb.local[i])))
```

The output associated with this part of the code (after compilation) should look as follows.

```
{{Alignment}}

Segment [(1, 1)] ---> [[0, 1, 2, 3, 5], [4]]
Segment [(2, 2)] ---> [[0, 2, 4, 5], [1, 3]]
Segment [(3, 3)] ---> [[0, 1, 2, 5], [3, 4]]
Segment [(5, 5)] ---> [[0, 3, 4, 5], [1, 2]]
Segment [(8, 8)] ---> [[0, 1, 3], [2, 4, 5]]
Segment [(9, 9)] ---> [[0, 2], [1, 3, 4, 5]]
Segment [(10, 10)] ---> [[0, 1, 5], [2, 3, 4]]
Segment [(12, 12)] ---> [[0, 1], [2, 3, 4, 5]]
Segment [(13, 13)] ---> [[0, 2, 4, 5], [1, 3]]
Segment [(14, 14)] ---> [[0, 1, 3, 4, 5], [2]]
Segment [(15, 15)] ---> [[0, 1, 2], [3, 4, 5]]
Segment [(16, 16)] ---> [[0, 5], [1, 2, 3, 4]]
Segment [(19, 19)] ---> [[0, 1, 2, 5], [3, 4]]
```

The motivated reader can verify that the previous outputs correspond to the non-trivial images of the functor $L_{\mathrm{rb}} : B_{\mathrm{rb}} \to \mathbf{Uprt}(S)$, as shown in [4, Example 2.24]. However, note that the previous collection of partitions misses the (terminal) partitions associated with the trivial columns of the alignment by definition of the constructor `Pedigrad(−)`.

Below, we define the first generation of our phylogeny, which corresponds to the 1-phylogeny ϕ_0 defined in [4, Example 3.30].

In this library, phylogenies are implemented through the class `Phylogeny`, which is described in section 7.2 of this documnetation. The idea behind this type of structure is that, instead of being encoded by a unique evoluationary tree, as it is common, it is encoded by a collection of evolutionary trees – one for each taxa. The following lines define a 1-phylogeny whose histories 'end' (or start, if seen as lists) with the taxa themselves.

```
110 p = list()
111 for i in range(len(P_rb.taxa)):
112   p.append([[i]])
113 pgy = Phylogeny(p)
```

We can display the content of the 1-phylogeny `pgy` by using the object `.history` associated with item of the class `Phylogeny`. To do so, let us add the following lines to `main.py`.

```
115 print("\n{{Mathematical phylogeny}}\n")
116 for i in range(len(pgy.phylogeneses)):
117   print(P_rb.taxa[i] + " = " + str(pgy.phylogeneses[i].history))
```

Compiling thiese lines should give us the following display on the standard output.

```
{{Mathetical phylogeny}}

TaxonA = [[0]]
TaxonB = [[1]]
TaxonC = [[2]]
TaxonD = [[3]]
TaxonE = [[4]]
TaxonF = [[5]]
```

As described in the statement of [4, Theorem 3.37], our algorithm is a repetition of a construction called an *extension* and it stops once a *complete* extension is reached, which broadly means an extension that is equal to its extension. For the sake of exposition, we will

first illustrate the first loop of the algorithm and then run the whole code on a `while` loop. In this respect, add the following lines to `main.py` – the commented part will be uncommented later on.

```
119 keep_going = True
120
121 if keep_going:
122 #while(keep_going):
```

We first start by creating groups of taxa that may have possibly originated from a common ancestor. The idea is to gather those taxa that are not present in each other first generations (see [4, Definition 3.28]), which means the last list of the list `pgy.phylogeneses[i].history`. We do so by using the method `pgy.set_up_friendship()` (described section 7.2.7) as shown below.

```
124   print("\n{{Set up friendship}}\n")
125   friendship = pgy.set_up_friendships()
126   for i in range(len(P_rb.taxa)):
127     print("----->"+str(P_rb.taxa[i])+"'s network:  ")
128     for j in print(len(friendship[0][i])):
129       print("> common ancestor with " + str(P_rb.taxa[friendship[0][i]
    [j]]) + ":  "+ str(friendship[1][i][j]))
```

Compiling the previous lines of code gives the following output. As can be seen, common ancestors are represented by lists of taxa that they are supposed to generate.

```
{{Set up friendship}}

----->TaxonA's network:
> common ancestor with TaxonB: [0, 1]
> common ancestor with TaxonC: [0, 2]
> common ancestor with TaxonD: [0, 3]
> common ancestor with TaxonE: [0, 4]
> common ancestor with TaxonF: [0, 5]
----->TaxonB's network:
> common ancestor with TaxonA: [0, 1]
> common ancestor with TaxonC: [1, 2]
> common ancestor with TaxonD: [1, 3]
> common ancestor with TaxonE: [1, 4]
> common ancestor with TaxonF: [1, 5]
----->TaxonC's network:
> common ancestor with TaxonA: [0, 2]
> common ancestor with TaxonB: [1, 2]
> common ancestor with TaxonD: [2, 3]
> common ancestor with TaxonE: [2, 4]
> common ancestor with TaxonF: [2, 5]
----->TaxonD's network:
> common ancestor with TaxonA: [0, 3]
> common ancestor with TaxonB: [1, 3]
> common ancestor with TaxonC: [2, 3]
> common ancestor with TaxonE: [3, 4]
> common ancestor with TaxonF: [3, 5]
```

```
----->TaxonE's network:
> common ancestor with TaxonA: [0, 4]
> common ancestor with TaxonB: [1, 4]
> common ancestor with TaxonC: [2, 4]
> common ancestor with TaxonD: [3, 4]
> common ancestor with TaxonF: [4, 5]
----->TaxonF's network:
> common ancestor with TaxonA: [0, 5]
> common ancestor with TaxonB: [1, 5]
> common ancestor with TaxonC: [2, 5]
> common ancestor with TaxonD: [3, 5]
> common ancestor with TaxonE: [4, 5]
```

To decide which of these ancestors is the most probable, we can use the function `pgy.socre` to assess them. As is done in [4], we use two scoring systems: the cardinalities of the large and exact selections (see [4, Definition 3.7 & 3.8]). In the code given below, these cardinalities are stored in the variable `scores[i][j][1]` and `scores[i][j][2]`.

```
131   print("\n{{Score friendship}}\n")
132   scores = pgy.score(P_rb.local,pgy.set_up_friendships())
133   for i in range(range(scores)):
134     print("----->"+ str(P_rb.taxa[i]))
135     for j in range(len(scores[i])):
136     print(">" + str(P_rb.taxa[scores[i][j][0]]) + ":  L =
    "+str(scores[i][j][1]) + ", E = "+str(scores[i][j][2]))
```

Adding the previous piece of code to the file `main.py` and compiling it gives the scores displayed below. The cardinalities of the large selections are denoted as `L` while the cardinalities of the exact selections are denoted as `E`. The reader who is reading [4] may here recognize the scores used in [4, Example 3.31] and [4, Example 3.22].

```
{{Score friendship}}

----->TaxonA
>TaxonB: L = 16, E = 1
>TaxonC: L = 15, E = 1
>TaxonD: L = 12, E = 0
>TaxonE: L = 12, E = 0
>TaxonF: L = 17, E = 1
----->TaxonB
>TaxonA: L = 16, E = 1
>TaxonC: L = 14, E = 1
>TaxonD: L = 15, E = 2
>TaxonE: L = 11, E = 0
>TaxonF: L = 14, E = 0
----->TaxonC
>TaxonA: L = 15, E = 1
>TaxonB: L = 14, E = 1
>TaxonD: L = 12, E = 0
>TaxonE: L = 14, E = 0
>TaxonF: L = 15, E = 0
```

```
----->TaxonD
>TaxonA: L = 12, E = 0
>TaxonB: L = 15, E = 2
>TaxonC: L = 12, E = 0
>TaxonE: L = 17, E = 2
>TaxonF: L = 14, E = 0
----->TaxonE
>TaxonA: L = 12, E = 0
>TaxonB: L = 11, E = 0
>TaxonC: L = 14, E = 0
>TaxonD: L = 17, E = 2
>TaxonF: L = 16, E = 0
----->TaxonF
>TaxonA: L = 17, E = 1
>TaxonB: L = 14, E = 0
>TaxonC: L = 15, E = 0
>TaxonD: L = 14, E = 0
>TaxonE: L = 16, E = 0
```

For each taxon, we are going to choose a set of closest relatives based on the scores computed above. Note that each series of scores is relative to a taxon so that `TaxonA` may be the best choice from the point of view of `TaxonB`, but taxon `TaxonB` may not be the best choice from the point of view `TaxonA`. In [4], the set of closest relatives associated with each taxon was encoded via the notion of *set of dominant ancestors*. To process the scores stored in the list `scores`, we want to use the method `.choose` associated with the `Phylogeny` item `pgy` (see section 7.2.9). To do so, add the following lines of code to the file `main.py`.

```
138   print("\n{{Choose friendship}}\n")
139   choose_friends = pgy.choose(scores)
140   for i in range(len(choose_friends)):
141     print("----->"+ str(P_rb.taxa[i])+"'s closet relative(s):  ")
142       for j in range(len(choose_friends[i])):
143         print(">" + str(P_rb.taxa[choose_friends[i][j]]))
```

Compiling should now add the following lines to the standard output.

```
{{Choose friendship}}

----->TaxonA's closet relative(s):
>TaxonF
----->TaxonB's closet relative(s):
>TaxonA
----->TaxonC's closet relative(s):
>TaxonA
----->TaxonD's closet relative(s):
>TaxonE
----->TaxonE's closet relative(s):
>TaxonD
----->TaxonF's closet relative(s):
>TaxonA
```

As can be seen above, the procedure `pgy.choose()` chose a closest relative for each taxon. These closest relatives together with the taxa with which they are associated will formally

represent the ancestor from which they are supposed to originate. To create these formal representatives, we want to use the method `.set_up_competition` associated with the class `Phylogeny`. The name of this method comes from the fact that once ancestors have been defined for each taxon, we want to determine which one of these are more likely to survive. Then, these individuals, seen as the latest possible survivors, should be placed as early as possible in the phylogeny.

Below, we apply the method `.set_up_competition` on the list `choose_friends` created earlier. Because the list `p` is, for now, quite trivial, the ancestors created by this method are only the list of the taxa shown ealier. However, later in the algorithm, the method `.set_up_competition` will form ancestors in a less obvious way by considering the 'first' generations of the phylogeny, which means the last list appended to the lists of `pgy.phylogeneses` (this will be explained later in the tutorial).

To create our list of ancestors, we need to add the following piece of code to `main.py`.

```
145   print("\n{{Set up competition}}\n")
146   competition = pgy.set_up_competition(choose_friends)
147   for i in range(len(competition)):
148     print("----->"+str(P_rb.taxa[i])+"'s team:  ")
149     for j in range(len(competition[i])):
150       print(">" + str(P_rb.taxa[competition[i][j]]))
151   adjusted_competition = ([range(len(competition))],[competition])
```

Compiling the previous piece of code gives the following output.

```
{{Set up competition}}

----->TaxonA's team:
>TaxonA
>TaxonF
----->TaxonB's team:
>TaxonA
>TaxonB
----->TaxonC's team:
>TaxonA
>TaxonC
----->TaxonD's team:
>TaxonD
>TaxonE
----->TaxonE's team:
>TaxonD
>TaxonE
----->TaxonF's team:
>TaxonA
>TaxonF
```

As can be seen, each `team` includes the taxon with which it is associated and its closest relatives. As was the case for friendships, these teams have to ranked so that only the best ones are chosen (ties are allowed). This can be done by using the method `.score` again. Note that while the output of `.set_up_friendships()` was matching the type of input required for `.score`, the output of `.set_up_competition` needs to be completed (see `adjusted_competition` above) to fit the format required by `.score`. While the part

```
[range(len(competition))]
```

of `adjusted_competition` only gives formal representatives to the internal lists of the list `[competition]`, the method `.set_up_friendships()` outputs a more complex labeling system that is more representatives of the process that gave rise to the returned friendships (see section 7.2.7).

To score the set of best teams contained in `adjusted_competition`, copy the following piece of code into `main.py`.

```
153   print("\n{{Score competition}}\n")
154   scores = pgy.score(P_rb.local,adjusted_competition)
155   print(scores)
156     for j in range(len(scores[0])):
157       print(">" + str(P_rb.taxa[scores[0][j][0]]) + ":  L =
    "+str(scores[0][j][1]) + ", E = "+str(scores[0][j][2]))
```

Compiling `main.py` should now add the following text to the standard output.

```
{{Score competition}}

>TaxonA: L = 17, E = 2
>TaxonB: L = 16, E = 2
>TaxonC: L = 15, E = 1
>TaxonD: L = 17, E = 9
>TaxonE: L = 17, E = 9
>TaxonF: L = 17, E = 2
```

As before, we want to choose the taxa associated with the best scores by using the method `.choose` (see section 7.2.9). Below, the procedure `pgy.choose` is applied to the list `scores`, defined earlier, and returns the 'labels' of the ancestors that possess the best scores. By construction of `adjusted_competition`, the labels returned by `pgy.choose` correspond to the indices of the ancestors in the list `competition`. This means that we can recover the list characterizing the ancestors by indexing the output `pgy.choose` in `competition`, as shown in the code displayed below.

Also, in the piece of code given below, we create a list `c` that contains the pairs of the form `(t,competition[t])` for every taxon `t` returned by `pgy.choose`. This list will later be used to add the new ancestor `competition[t]` to the phylogenesis of `t`, while the other taxa will keep their ancestors for one more generation.

```
159   print("\n{{Choose competition}}\n")
160   choose_competitors = pgy.choose(scores)
161   c = list()
162   for i in range(len(choose_competitors[0])):
163     c.append((choose_competitors[0][i],competition[choose_competitors[0]
    [i]]))
164     print("--->"+P_rb.taxa[choose_competitors[0][i]]+"'s winners:  ")
165     for j in range(len(competition[choose_competitors[0][i]])):
166       print(">"+P_rb.taxa[competition[choose_competitors[0][i]][j]])
```

After adding the previous piece of code to `main.py` and compiling it, we obtain two winners for the competitions (see the output displayed below). Even though these winners represent the same ancestor, this may not always be the case.

```
{{Choose competition}}

----->TaxonD's winners:
>TaxonD
>TaxonE
----->TaxonE's winners:
>TaxonD
>TaxonE
```

We now want to add the ancestor `[TaxonD,TaxonE]` (represented by a list) to the phylogeneses of the taxa `TaxonD` and `TaxonE`. We can do so by using the method `.extend` associated with the class `Phylogeny` (see section 7.2.4).

In the piece of code given below, the procedure `pgy.extend(c)` returns a Boolean value that indicates whether the adding of the ancestors contained in `c` actually adds new information to the phylogeny `pgy`. If the value `False` is returned, then the ancestors stored in `c` are only repetitions of ancestors already considered in `pgy` for the same taxa. In this case, extending `pgy` with the ancestor contained in `c` will only generate an infinite loop. We therefore want to make the algorithm terminate when this happens – we can do so by giving the output of `pgy.extend(c)` to the variable `keep_going`, which is used to control the `while` loop of the algorithm (see the beginning of the section).

```
168   print("\n{{Extension}}\n")
169   keep_going = pgy.extend(c)
170   print("CONTINUE: "+len(keep_going))
```

It is possible to display the phylogeny `pgy` at any time via the method `.print_tree()`. We can add the following lines of code to the file `main.py` to display the added generation to the phylogeny.

```
172   #Display current phylogeny
173   if keep_going == True:
174   for i in range(len(pgy.phylogeneses)):
175     print("\nTree for " + str(P_rb.taxa[pgy.phylogeneses[i].taxon])+":")
176     try:
177       pgy.phylogeneses[i].print_tree()
178     expect:
179       {}
```

Compiling `main.py` should display the following text on the standard output.

```
{{Extension}}

CONTINUE: True

Tree for TaxonA:
[UPDATE]

Tree for TaxonB:
[UPDATE]

Tree for TaxonC:
[UPDATE]
```

```
Tree for TaxonD:
[UPDATE]

Tree for TaxonE:
[UPDATE]

Tree for TaxonF:
[UPDATE]
```

We now exit the loop of the algorithm (see the change of indentation in the lines of code) and add the following piece of code, which is to display the final phylogeny in form of ascii trees.

```
181 print("\n{{Phylogeny}}")
182 for i in range(len(pgy.phylogeneses)):
183   print("\nTree for " + str(P_rb.taxa[pgy.phylogeneses[i].taxon])+":")
184   pgy.phylogeneses[i].print_tree()
```

The reader who also reads [4] may also want to see what the phylogeny looks like structurally. We therefore add the following lines of code to display the lists

`pgy.phylogeneses[i].history`,

which give the sequences of inclusions considered in [4, Definition 3.23].

```
186 print("\n{{Mathematical phylogeny}}\n")
187 for i in range(len(pgy.phylogeneses)):
188   print(P_rb.taxa[i] + " = " + str(pgy.phylogeneses[i].history))
```

To run the algorithm, it suffices to uncomment the `while` condition of line 122 and turn line 121 into a comment.

```
121 #if keep_going:
122 while(keep_going):
```

Compiling `main.py` should give a detailed description of the algorithm described in [4, Example 3.39].

The final output associated with the lines of code comprised from line 181 to line 188 is given below.

```
{{Phylogeny}}

Tree for TaxonA:
[UPDATE]

Tree for TaxonB:
[UPDATE]

Tree for TaxonC:
[UPDATE]

Tree for TaxonD:
[UPDATE]
```

```
Tree for TaxonE:
[UPDATE]

Tree for TaxonF:
[UPDATE]

{{Mathematical phylogeny}}

TaxonA = [[0], [0], [0], [0, 5], [0, 2, 5], [0, 1, 2, 5], [0, 1, 2, 3, 4,
5]]
TaxonB = [[1], [1], [0, 1], [0, 1], [0, 1, 5], [0, 1, 2, 5], [0, 1, 2, 3,
4, 5]]
TaxonC = [[2], [2], [2], [2], [0, 2, 5], [0, 1, 2, 5], [0, 1, 2, 3, 4, 5]]
TaxonD = [[3], [3, 4], [3, 4], [3, 4], [3, 4], [3, 4], [0, 1, 2, 3, 4, 5]]
TaxonE = [[4], [3, 4], [3, 4], [3, 4], [3, 4], [3, 4], [0, 1, 2, 3, 4, 5]]
TaxonF = [[5], [5], [5], [0, 5], [0, 2, 5], [0, 1, 2, 5], [0, 1, 2, 3, 4,
5]]
```

2.8. Agreements

Presentation of the module PartitionCategory.py

3.1. Description of _image_of_partition

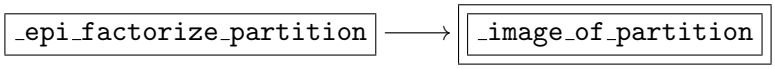
The function _image_of_partition takes a list of elements and returns the list of its elements without repetition in the order in which they can be accessed from the left to the right.

```
1 def _image_of_partition(partition):
2   """ the source code of this function can be found in iop.py """
3   return the_image
```

This corresponds to returning the image object of the underlying partition of the list.

```
>>> print(_image_of_partition([3,3,2,1,1,2,4,5,6,5,2,6]))
[3, 2, 1, 4, 5, 6]
>>> print(_image_of_partition(['A',4,'C','C','G',4,0,0,1,'a','A']))
['A', 4, 'C', 'G', 0, 1, 'a']
```

3.2. Description of _epi_factorize_partition



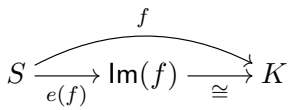
The function _epi_factorize_partition relabels the elements of a list with non-negative integers. It starts with the integer 0 and attributes a new label by increasing the previously attributed label by 1. The first element of the list always receives the label 0 and the highest integer used in the relabeling equals the length of the image (section 3.1) of the list decreased by 1.

```
1 def _epi_factorize_partition(partition):
2   """ the source code of this function can be found in efp.py """
3   return epimorphism
```

Even though a list already encodes an epimorphism, the goal of the function

_epi_factorize_partition

is to return a canonical *choice* of epimorphism.

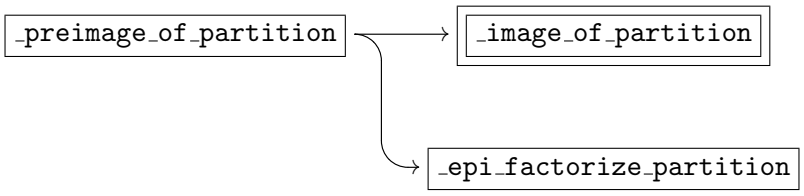


This choice ensures that two partitions characterized by the same set of universal properties are *equal* as python lists. In [4], this type of construction is formulated in terms of 'epi-skeletal' structure for the object S (see the diagram above).

```
>>> p = [3,3,2,1,1,2,4,5,6,5,2,6]
>>> print(_epi_factorize_partition(p))
[0, 0, 1, 2, 2, 1, 3, 4, 5, 4, 1, 5]
>>> im = _image_of_partition(p)
>>> print(_epi_factorize_partition(im))
[0, 1, 2, 3, 4, 5]
>>> print(_epi_factorize_partition(['A',4,'C','C','G',4,0,0,1,'a','A']))
[0, 1, 2, 2, 3, 1, 4, 4, 5, 6, 0]
```

Note that the function does not literally relabel the input list, but allocates a new space in the memory to store the relabeled list.

3.3. Description of _preimage_of_partition



The function _preimage_of_partition takes a list and returns the list of the lists of indices that index the same element.

```
1 def _preimage_of_partition(partition):
2    """ the source code of this function can be found in piop.py """
3    return the_preimage
```

From the point of view of partitions, the returned list the_preimage (see code above) is the preimage of the underlying epimorphism $f : S \to K$ of the input partition, where the preimage of f is defined as the K-indexed set of the fibers of the epimorphism.

$$\mathsf{PreIm}(f) = \{f^{-1}(k)\}_{k\in K}$$

Note that, from an implementation viewpoint, the set K might not be equipped with an obvious order relation, which makes it difficult to define the preimage of $f : S \to K$ as a python list. To rectify this flaw, the preimage is computed with respect to the canonical epimorphism $e(f) : S \to \mathsf{Im}(f)$ whose codomain is equipped with the natural order on integers (see section 3.2).

$$\mathsf{PreIm}(f) := \{e(f)^{-1}(k)\}_{k\in \mathsf{Im}(f)}$$

```
>>> p = ['a','a',2,2,3,3,'a']
>>> print(_preimage_of_partition(p))
[[0, 1, 6], [2, 3], [4, 5]]
>>> print(_epi_factorize_partition(p))
[0, 0, 1, 1, 2, 2, 0]
>>> p = [2,1,0,6,5,4,2,1,0]
>>> print(_preimage_of_partition(p))
```

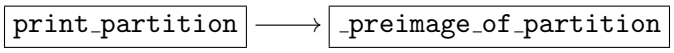
```
[[0, 6], [1, 7], [2, 8], [3], [4], [5]]
>>> print(_epi_factorize_partition(p))
[0, 1, 2, 3, 4, 5, 0, 1, 2]
```

In the first example given above:

- the list `[0,1,6]` is the fiber of the element `'a'` and its index in the preimage is 0;
- the list `[2,3]` is the fiber of the element `2` and its index in the preimage is 1;
- the list `[4,5]` is the fiber of the element `3` and its index in the preimage is 2.

The preimage will always orders its fibers with respect to the order in which the elements of the input list appear.

3.4. Description of print_partition

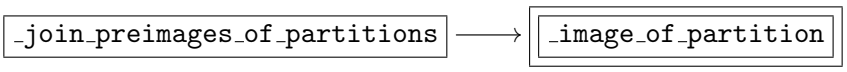


The function `print_partition` is a debug function that takes a list of elements and prints its preimage on the standard output (i.e. the console).

```
1 def print_partition(partition):
2   print(_preimage_of_partition(partition))
```

See section 3.3 for examples.

3.5. Description of _join_preimages_of_partitions



The function `_join_preimages_of_partitions` takes two lists of lists of indices (the indices can be repeated and should only be non-negative integers) as well as a Boolean and returns the list of the maximal unions of internal lists that intersect within the concatenation of the two input lists (see the examples below).

```
1 def _join_preimages_of_partitions(preimage1,preimage2,speed_mode):
2   """ the source code of this function can be found in jpop.py """
3   return the_join
```

While the two input lists `preimage1` and `preimage2` could be two outputs of the procedure

`_preimage_of_partition(−)`

for two input lists of the same length, the Boolean value `speed_mode` would indicate whether one of the two input lists may contain at least two different sublists with the same index, as shown below.

`[[0, 1], [0, 4], [2, 3, 4, 5]]`

Note that the global variable `FAST` is reserved to this use.

```
>>> print(FAST)
True
```

From the point of view of partitions, the composition of `_preimage_of_partition` with `_join_preimages_of_partitions` would amount to computing the coproduct of two partitions. Since a category of partitions is also a partially ordered set, this coproduct is also the *join* of the two partitions, which explains the name of the procedure.

For illustration, if we consider the following two lists of lists of indices

```
>>> p1 = [[0, 3], [1, 4], [2]]
>>> p2 = [[0, 1], [2], [3], [4]]
```

we can notice that

- the internal list [0,3] of p1 intersects with the internal lists [0,1] and [3] in p2;
- the internal list [0,1] of p1 intersects with the internal lists [1,4] and [1] in p2;
- the internal list [1,4] of p1 intersects with the internal list [4] in p2;

and

- the internal list [2] of p1 only intersects with the internal list [2] in p2,

so that we have

```
>>> print(_join_preimages_of_partitions(p1,p2,FAST))
[[1, 4, 0, 3], [2]]
```

In terms of implementation, the program

(3.1) `_join_preimages_of_partitions(p1,p2,FAST)`

considers each internal list of p1 and searches for the lists of p2 that intersect it. If an intersection is found between two internal lists, it merges the two internal lists in p1 and empties that of p2 (the list is emptied and *not* removed in order to preserve a coherent indexing of the elements of p2). The function continues until all the possible intersections have been checked.

Here is a detail of what program (3.1) does with respect to the earlier example:

The element 0 of [0,3] is searched in the list [0,1] of p2;

The element 0 is found;

The lists [0,3] and [0,1] are merged in p1 and [0,1] is emptied from p2 as follows:

```
p1 = [[0, 3, 1], [1, 4], [2]]
p2 = [[], [2], [3], [4]]
```

Because the element 0 has now been found in p2 and the third input was set to FAST, no other sublist of p2 is supposed to contain the element 0 and the search of the element 0 stops here. Note that if not(FAST) were given in the third argument, then the earlier union operation would also be operated on the remaining sublists of p2.

The element 3 of [0, 3] is searched in the list [] of p2;

The element 3 is not found (continues);

The element 3 of [0, 3] is searched in the list [2] of p2;

The element 3 is not found (continues);

The element 3 of [0, 3] is searched in the list [3] of p2;

The element 3 is found;

The lists [0,3] and [3] are merged in p1 and [3] is emptied from p2 as follows:

```
p1 = [[0, 3, 1], [1, 4], [2]]
p2 = [[], [2], [], [4]]
```

The element 3 has now been found in p2 and does not need to be searched again.

All elements of the initial list [0, 3] have been searched.

The first lists of p1 is appended to p2 in order to ensure the transitive computation of the maximal unions through the next interations.

The list [0, 3, 1] of p1 is emptied as follows:

```
p1 = [[], [1, 4], [2]]
p2 = [[], [2], [], [4], [0, 3, 1]]
```

Repeat the previous procedure with respect to the list `[1, 4]` of `p1`. We obtain the following pair:

```
p1 = [[], [], [2]]
p2 = [[], [2], [], [], [], [1, 4, 0, 3]]
```

Repeat the previous procedure with respect to the remaining list `[2]` of `p1`. We obtain the following pair:

```
p1 = [[], [], []]
p2 = [[], [], [], [], [], [1, 4, 0, 3], [2]]
```

The function stops because there is no more list to process in `p1`. The output is all the non-empty lists of `p2`; i.e. `[[1, 4, 0, 3], [2]]`

Note that, because of the iterative nature of the previous algorithm, the procedure

`_join_preimages_of_partitions(−,−,−)`

does not necessarily presents its output in the same way as the procedure

`_preimage_of_partition(−)`

does. For instance, while the index `0` will always be contained in the first list of the output of `_preimage_of_partition`, it might not be contained in the first list of the output of `_join_preimages_of_partitions` as illustrated below.

```
>>> l = _preimage_of_partition([1,2,4,5,1,2,3,2,2,1,3])
>>> print(l)
[[0, 4, 9], [1, 5, 7, 8], [2], [3], [6, 10]]
>>> m = _preimage_of_partition([1,2,5,4,2,2,5,6,5,7,8])
>>> print(m)
[[0], [1, 4, 5], [2, 6, 8], [3], [7], [9], [10]]
>>> print(_join_preimages_of_partitions(l,m))
[[3], [6, 10, 2, 1, 5, 7, 8, 0, 4, 9]]
```

Interestingly, the procedure `_join_preimages_of_partitions` can also be used to compute the intersection-free closure of a set of sets, as shown below.

```
>>> a = [[1,0,2,1,3,2,5,4,6],[15,15,18,0,13],[7,11,12,22],[23,12]]
>>> closure_of_a = _join_preimages_of_partitions(a,a,not(FAST))
>>> print(closure_of_a)
[[15, 18, 0, 13, 1, 2, 3, 5, 4, 6], [23, 12, 7, 11, 22]]
```

3.6. Description of `EquivalenceRelation` (class)

`EquivalenceRelation` ⟶ `_join_preimages_of_partitions`

The class `EquivalenceRelation` possesses two objects, namely

- `.classes` (list of lists);
- `.range` (integer);

and three methods, namely

- `.__init__` (constructor);
- `.closure`;
- `.quotient`.

The constructor .__init__ takes between 1 and 2 arguments: the first argument should either be an empty list or a list of lists of indices (i.e. non-negative integers) and the second argument, which is optional when the first argument is not an empty list, should be an integer that is greater than or equal to the maximum index contained in the first input, if it exists.

```
 1 class EquivalenceRelation:
 2   #The objects of the class are:
 3   #.classes (list of lists);
 4   #.range (integer);
 5   #The following constructor takes between 1 and 2 arguments,
 6   #the first one being a list and the second being an integer.
 7   def __init__(self,*args):
 8   """ the source code of this constructor can be found in cl_er.py """
 9   def closure(self):
10     self.classes = _join_preimages_of_partitions(self.classes,
   self.classes,not(FAST))
11   def quotient(self):
12   """ the source code of this function can be found in cl_er.py """
13     return the_quotient
```

If the first input is not empty, then it is stored in the object .classes while the object .range receives:

- either the second input, when this second input is given;
- or the maximum index contained in the first input when no second input is given.

```
>>> eq1 = EquivalenceRelation([[0,1,2,9],[7,3,8,6],[4,9,5]])
>>> print(eq1.classes)
[[0, 1, 2, 9], [7, 3, 8, 6], [4, 9, 5]]
>>> print(eq1.range)
9
>>> eq2 = EquivalenceRelation([[0,1,2,9],[7,3,8,7],[9,15]],18)
>>> print(eq2.classes)
[[0, 1, 2, 9], [7, 3, 8, 7], [9, 15]]
>>> print(eq2.range)
18
```

If the first input is empty, then the second argument is required. In this case, the object .classes receive the lists containing all the singleton lists containing the integers from 0 to the integer given in the second argument, which is, for its part, stored in the object .range.

```
>>> eq3 = EquivalenceRelation([],5)
>>> print(eq3.classes)
[[0], [1], [2], [3], [4], [5]]
```

The method .closure() replaces the content of the object .classes with the transitive closure of its classes. After this procedure, the object .classes describes an actual equivalence relation (modulo the singleton equivalence classes, which do not need to be specified for obvious reasons).

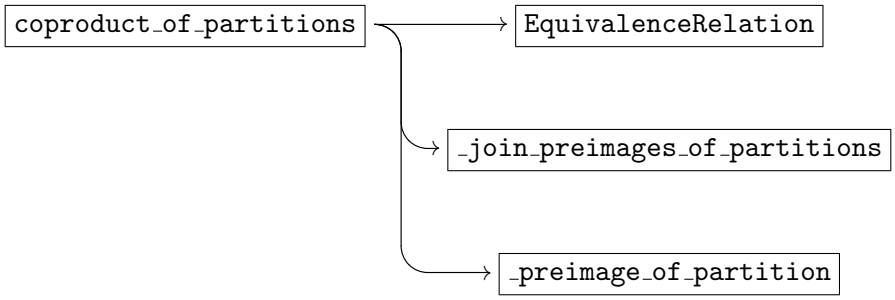
```
>>> eq1.closure()
>>> print(eq1.classes)
[[7, 3, 8, 6], [4, 9, 5, 0, 1, 2]
```

```
>>> eq2.closure()
>>> print(eq2.classes)
[[7, 3, 8], [9, 15, 0, 1, 2]]
```

The method `.quotient()` returns a list of integers whose length is equal to the integer contained in the object `.range` decreased by 1 and whose non-trivial fibers are those contained in the object `.classes` after a call of the method `.closure()`.

```
>>> print(eq1.quotient())
[1, 1, 1, 0, 1, 1, 0, 0, 0, 1]
>>> print(eq2.quotient())
[1, 1, 1, 0, 2, 3, 4, 0, 0, 1, 5, 6, 7, 8, 9, 1, 10, 11, 12]
```

3.7. Description of coproduct_of_partitions



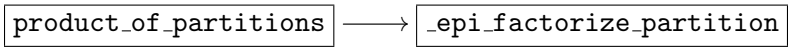
The function `coproduct_of_partitions` takes two lists of the same legnth and returns their coproduct (or join) as partitions. Specifically, the procedure outputs the quotient of the join of their preimages. If the two input lists do not have the same length, then an error message is outputted and the program is aborted.

```
 1 def coproduct_of_partitions(partition1,partition2):
 2   if len(partition1) == len(partition2):
 3     #Returns the coproduct of two partitions as the quotient of the
 4     #equivalence relation induced by the join of the preimages
 5     #of the two partitions.
 6     the_join = EquivalenceRelation(_join_preimages_of_partitions(
   _preimage_of_partition(partition1),_preimage_of_partition(partition2),
   FAST))
 7     return the_join.quotient()
 8   else:
 9     print("Error:  in coproduct_of_partitions:  lengths do not match.")
10     exit()
```

Note that the outputs of the procedure `coproduct_of_partitions` do not necessarily belong to the set of outputs of the procedure `_epi_factorize_partition`. The reason comes from the way in which the procedure `_join_preimages_of_partitions` is implemented (see the end of section 3.5).

```
>>> l = [1,2,4,5,1,2,3,2,2,1,3]
>>> m = [1,2,5,4,2,2,5,6,5,7,8]
>>> c = coproduct_of_partitions(l,m)
>>> print(c)
[1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1]
>>> print(_epi_factorize_partition(c))
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

3.8. Description of product_of_partitions



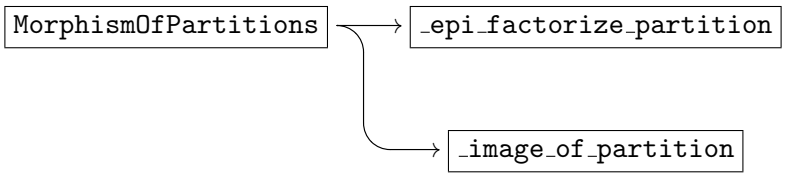
The function product_of_partitions takes two lists and returns a list that is the relabeling of the zipping of the two lists (i.e. the list of pairs of elements with corresponding indices in each of the input lists) via the procedure _epi_factorize_partition.

```
 1 def product_of_partitions(partition1,partition2):
 2   #The following line checks if the product of the two lists is possible.
 3   if len(partition1) == len(partition2):
 4     #Constructs the list of pairs of element with the
 5     #same index in the two lists, and then relabels
 6     #the pairs using _epi_factorize_partition.
 7     return _epi_factorize_partition(zip(partition1,partition2))
 8   else:
 9     print("Error:  in product_of_partitions:  lengths do not match.")
10     exit()
```

The function outputs an error if the two input lists do not have the same length.

```
>>> product_of_partitions([1,1,1,1,2,3],['a','b','c','c','c','c'])
[0, 1, 2, 2, 3, 4]
>>> product_of_partitions([1,1,1,1,2],['a','b','c','c','c','c'])
Error:  in product_of_partitions:  lengths do not match.
```

3.9. Description of MorphismOfPartitions (class)



The class MorphismOfPartitions possesses three objects, namely

- .arrow (list)
- .source (list)
- .target (list)

and a constructor .__init__. The consructor .__init__ takes two lists as well as an optional argument and stores, in the object .arrow, the list that describes, if it exists, the (unique) morphism of partitions from the first input list (seen as a partition) to the second input list (seen as a partition). If the morphism does not exist, then the method returns an error message unless the value False was given as a third input.

The canonical epimorphisms associated with the partitions of the first and second input lists (see section 3.2) are stored in the objects .source and .target, respectively.

```
1 class MorphismOfPartitions:
2   #The objects of the class are:
3   #.arrow (list);
4   #.source (list);
5   #.target (list).
6   def __init__(self,source,target,*args):
7   """ the source code of this constructor can be found in cl_mop.py """
```

The list that is contained in the object .arrow is the image of the function source $\mapsto$ target(source) that can be constructed from the parametrization $t \mapsto$ source$[t]$ and $t \mapsto$ target$[t]$ given by the list structure of the two input lists target and source. This is illustrated below in more detail.

For illustration, let us consider the following pair of lists.

```
>>> p1 = [0,1,2,3,3,4,5]
>>> p2 = [0,1,2,3,3,3,1]
```

To construct the function source $\mapsto$ target(source), we can first try to construct the graph (source$[t]$, target$[t]$), for which we use the procedure zip.

```
>>> p3 = zip(p1,p2)
>>> print(p3)
[(0, 0), (1, 1), (2, 2), (3, 3), (3, 3), (4, 3), (5, 1)]
```

The image of the zipping is then as follows:

```
>>> p4 = _image_of_partition(p3)
>>> print(p4)
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 1)]
```

We can see that for each pair (x,y) in p4, every component x is mapped to a unique image y so that p4 defines the *graph* of the morphism of partitions between p1 and p2. The constructor .__init__ then records in .arrow the second projections of the pairs contained in p4, which also corresponds to the image of the underlying graph encoded by p4.

```
>>> m = MorphismOfPartitions(p1,p2)
>>> print(m.arrow)
[0, 1, 2, 3, 3, 1]
```

If there exists no morphism from the first input list to the second input list, then the function outputs an error message.

For example, if we modify p2 as follows

```
>>> p2 = [0,1,2,3,6,3,1]
```

then the image of the zipping of p1 and p2 is as follows:

```
>>> p4 = _image_of_partition(zip(p1,p2))
>>> print(p4)
[(0, 0), (1, 1), (2, 2), (3, 3), (3, 6), (4, 3), (5, 1)]
```

As can be seen, the argument 3 is 'mapped' to two different images, namely 3 and 6. In this case, the constructor .__init__ exits the program with an error message.

Here is a complete example summarizing the previous explanation.

```
>>> m = MorphismOfPartitions([1,6,5,3,3,4,2],[1,2,5,4,4,4,2])
>>> print(m.source)
[0, 1, 2, 3, 3, 4, 5]
>>> print(m.target)
[0, 1, 2, 3, 3, 3, 1]
>>> print(m.arrow)
[0, 1, 2, 3, 3, 1]
>>> m = MorphismOfPartitions([1,6,5,3,3,4,2],[1,2,5,4,6,4,2])
Error:  in MorphismOfPartitions.__init__:  source and target are not
compatible.
>>> m = MorphismOfPartitions([1,6,5,3,3,4,2],[1,2,5,4,6,4,2],False)
```

Finally, note that the constructor of `MorphismOfPartitions` can be used to test whether there is a morphism between two partitions by using the key words `try` and `except` and setting the thrid argument to `False`, as illustrated below.

```
>>> try:
>>>   m = MorphismOfPartitions([1,6,5,3,3,4,2],[1,2,5,4,4,4,2],False)
>>>   print("True")
>>> except:
>>>   print("False")
True
>>> try:
>>>   m = MorphismOfPartitions([1,6,5,3,3,4,2],[1,2,5,4,6,4,2],False)
>>>   print("True")
>>> except:
>>>   print("False")
False
```

Chapter 4

Presentation of the module SegmentCategory.py

4.1. Description of _read_pre_order

The function _read_pre_order takes the name of a file (called name_of_file) and returns a Boolean value and a list of lists.

```
1 def _read_pre_order(name_of_file):
2   """ the source code of this function can be found in rpo.py """
3   return (flag_zero_obj,the_pre_order)
```

The input file is supposed to contain some specific code lines that describe a pre-ordered set. The code follows a certain syntax that will be detailed in this section. Because this syntax is compatible with the yaml grammar, these pre-ordered sets will usually be specified with the extension .yml in our examples. Here is an example of such a file.

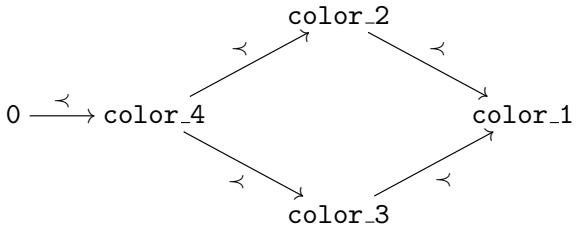
Omega.yml

```
 1 #This is an example of pre-ordered set.
 2
 3 !obj:
 4 - color_1 #object for color 1
 5 - color_2 #object for color 2
 6 - color_3 #object for color 3
 7 - color_4 #object for color 4
 8
 9 rel:  #these are generating relations for the preorder.
10 - color_1 > color_2, color_3;
11 - color_1 > color_3, color_1;
12 - color_3 > color_4;
13 - color_2 > color_4;
```

Diagrammatically, this pre-ordered set can be described by the graph given below, whose square commutes. We will see below why there is an additional object 0 that is not specified

in the yaml file.

(4.1)



4.1.1. Comments. Comments are allowed and are specified as in python and in yaml, which is to say by the character `#`.

4.1.2. Objects. The set of elements of the pre-ordered set should always be specified in terms of a list of words succeeding one of the key words '`obj:`' or '`!obj:`' as shown below.

```
obj:[name1][separator][name2][separator] (etc.)
!obj:[name1][separator][name2][separator] (etc.)
```

The labels `[name1]` and `[name2]` stand for words that should only be made of the characters `0-9`, `@`, `A-Z`, `_` and `a-z` while the label `[separator]` stands for any character that is not in this list of characters. Repetitions of names are taken care by the parser, which ignores all repetitions. The following specification of objects is equivalent to that given in `Omega.yml`.

Objects.yml

```
1 #Note that we now use different types of separator.
2 !obj:  color_1 ;
3 ,, color_2 ; color_1 ;
4 --> color_3.!color_4 *
```

While the key word `obj:` is only supposed to present the list of the elements of the pre-ordered set, the key word `!obj:` should only be used to formally *add* a formal initial object (i.e. a mimimum element) to the pre-ordered set. This element was represented by `0` in diagram (4.1).

4.1.3. Relations. The pre-order relations of the set should be specified after the key word `rel:` by a list of phrases satisfying the following syntax.

```
[dominant_object]>[predecessor1][separator][predecessor2][separator] (etc.);
```

Such a line means that `[dominant_object]` is greater than or equal to all the elements listed after the symbol '`>`'. The list of predecessors stops at the symbol '`;`'. Note that the symbol '`;`' can be omitted for the last line of the file. The following specification of relations is equivalent to that given in `Omega.yml`.

Omega_bis.yml

```
1 #The pre-ordered set can be very compact.
2 !obj:  1 2 3 4
3 rel:  1>2,3 ; 3>4 ; 2>4
```

The labels `[dominant_object]` and `[predecessor(`n`)]` stand for words that should only be made of the characters `0-9`, `@`, `A-Z`, `_` and `a-z` while the label `[separator]` stands for any character that is not in this list of characters.

4.1.4. Output. Finally, the output of the function `_read_pre_order` is a pair

```
(flag_zero_obj,the_pre_order)
```

whose first component `flag_zero_obj` is a Boolean value specifying by `False` or `True` whether the word `obj:` (`False`) or `!obj:` (`True`) was used to define the list of objects and whose

second component the_pre_order is a list of lists giving the *non-transitive* down-closures of the elements of the pre-ordered set in the order in which these were given in the input file.

Specifically, a list in the list the_pre_order will always start with the element of which it is supposed to give the down-closure, as shown below.

```
[dominant_object, predecessor1, predecessor2, (etc.)]
```

For instance, the ouput of our previous example Omega.yml is as follows.

```
>>> omega = _read_pre_order("Omega.yml")
>>> print("Formal initial object:  "+str(omega[0]))
Formal initial object:  True
>>> for i in range(len(omega[1])):
...   print(omega[1][i])
['color_1', 'color_2', 'color_3']
['color_2', 'color_4']
['color_3', 'color_4']
['color_4']
```

4.2. Description of _transitive_closure

The function _transitive_closure takes a list of lists (called down_closure) and returns another list of lists whose pointer is the same as that given in the input (i.e. the input is therefore modified by the function to make the output).

```
1 def _transitive_closure(down_closure):
2   """ the source code of this function can be found in tc.py """
3   return down_closure
```

The input down_closure is supposed to take the form of a list of the (potentially) non-transitive down-closures of the elements of a given pre-ordered set. In other words, the input down_closure should take the same form as that taken by the second outputs of the procedure _read_pre_order(−) (see section 4.1). Specifically, the down-closure of an element dominant_object is the unique list of down_closure starting with this element (see below).

```
[dominant_object, predecessor1, predecessor2, (etc.)]
```

The output down_closure is then the list of the transitive down-closures of the elements of that pre-ordered set.

Here is an example with the procedure _read_pre_order. Let us consider the following pre-order specification.

Omega.yml

```
1 #This is a non-transitive presentation of a pre-ordered set.
2
3 #The list of the elements of the set (without the formal minimum).
4 !obj:  1 2 3 4 5 6
5
6 #The list of the non-trivial (generating) relations.
7 rel:  1 > 2; 2 > 3; 3 > 5; 5 > 6;
```

As shown below, we can see that the output of the procedure _read_pre_order does not include the transitive relation 1 > 2 > 3 in the down-closure of 1. On the other hand, the relation 1 > 3 does appear in the down-closure of 1 even it is not specified in Omega.yml.

```
>>> preorder = _read_pre_order("Omega.yml")
>>> print("Formal initial object:  "+str(preorder[0]))
>>> for i in range(len(preorder[1])):
...   print(preorder[1][i])
['1', '2']
['2', '3']
['3', '5']
['4']
['5', '6']
['6']
>>> omega = _transitive_closure(preorder[1])
>>> for i in range(len(omega)):
...   print(omega[i])
['1', '2', '3', '5', '6']
['2', '3', '5', '6']
['3', '5', '6']
['4']
['5', '6']
['6']
```

4.3. Description of SegmentObject (class)

The class SegmentObject possesses two objects, namely

- .topology (list of 2-tuples);
- .colors (list of indices),

and two methods, namely

- .__init__ (constructor)
- .patch

The idea of behind the two objects .colors and .topology is that they contain all the needed information to define a mathetical segment, that is to say a pair (t, c) where t is the *topology*, usually encoded by an order preserving surjection, and c is the *coloring map*, usually encoded by a function going to a pre-ordered set.

```
1 class SegmentObject:
2   #The objects of the class are:
3   #.topology (list of 2-tuples);
4   #.colors (list of indices);
5   def __init__(self,topology,colors):
6   """ the source code of this constructor can be found in cl_so.py """
7   def patch(self,position):
8   """ the source code of this function can be found in cl_so.py """
9     return the_index
```

The constructor .__init__ takes two lists, specifically a lists of indices and a list of pairs of increasing indices, and allocate them in the object .colors and .topology, respectively

For instance, we could first define an uncolored topology of the form

$$(\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ)$$

by using the list t defined below.

```
>>> t = list()
>>> for i in range(5):
...   t = t + [(3*i,3*i+2)]
```

Then, we could add colors to this topology, say to form the following Boolean segment.

$$(\circ\circ\circ)(\bullet\bullet\bullet)(\bullet\bullet\bullet)(\circ\circ\circ)(\circ\circ\circ)$$

In this case, the colors would be specified by a list of colors whose length is equal to `len(t)` and the associated segment would be defined via the constructor `.__init__` as follows.

```
>>> c = [0,1,1,0,0]
>>> s = SegmentObject(t,c)
```

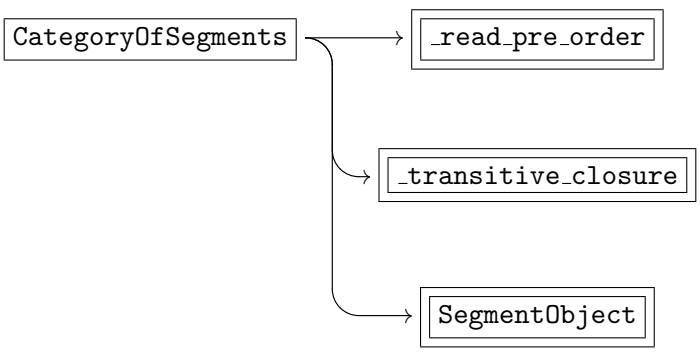
For its part, the method `.patch` takes an integer and returns the index of the first pair `(a,b)` of `.topology` that bounds the integer, that is to say that the input integer is greater than or equal to the first component `a` and also is less than or equal to the second component `b`. If no such index exists, then the procedure returns `-1`.

For example, we can use the method `.patch` to recover all the information contained in our previous segment `s`, as illustrated below.

```
>>> for i in range(15):
...   print("Position "+str(i)+" belongs to "+str(s.topology[s.patch(i)])+"
    and its color is "+str(s.colors[s.patch(i)]))
Position 0 belongs to (0, 2) and its color is 0
Position 1 belongs to (0, 2) and its color is 0
Position 2 belongs to (0, 2) and its color is 0
Position 3 belongs to (3, 5) and its color is 1
Position 4 belongs to (3, 5) and its color is 1
Position 5 belongs to (3, 5) and its color is 1
Position 6 belongs to (6, 8) and its color is 1
Position 7 belongs to (6, 8) and its color is 1
Position 8 belongs to (6, 8) and its color is 1
Position 9 belongs to (9, 11) and its color is 0
Position 10 belongs to (9, 11) and its color is 0
Position 11 belongs to (9, 11) and its color is 0
Position 12 belongs to (12, 14) and its color is 0
Position 13 belongs to (12, 14) and its color is 0
Position 14 belongs to (12, 14) and its color is 0
```

Finally, note that the pre-ordered set that is usually associated with a segment (t, c) can be specified through the class `CategoryOfSegments` defined in section 4.4.

4.4. Description of `CategoryOfSegments` (class)



The class `CategoryOfSegments` possesses three objects, namely

- `.domain` (integer)
- `.mask` (Boolean)
- `.preorder` (list of lists)

and five methods, namely

- `.__init__` (constructor)
- `.can_switch_to`
- `.homset`
- `.topology`
- `.segment`

In addition, the class `CategoryOfSegments` may be used as a super class (this is allowed by an argument `object`; see below). In this library, the class `CategoryOfSegments` is the parent of the classes `LocalAnalysis` and `Pedigrad` (see section 5.3 and section 5.5).

```
1 class CategoryOfSegments(object):
2   #The objects of the class are:
3   #.domain (integer);
4   #.mask (Boolean);
5   #.preorder (list of lists);
```

The three objects `.domain`, `.mask`, and `.preorder` give all the needed information to specify a category of quasi-homologous segments $\mathbf{Seg}(\Omega \,|\, n)$ (as defined in [5, 4]) for some pre-ordered set $(\Omega, \preceq)$ and non-negative integer n. Specifically, the object

- `.domain` contains the integer n defining what is called the *domain* [5, 4] of the segments of $\mathbf{Seg}(\Omega \,|\, n)$, which can informally be identified as the length of the segments;
- `.mask` indicates whether the pre-ordered set $(\Omega, \preceq)$ contains a formal initial object (i.e. a minimum element), which is usually used as an 'ignore' state or a mask;
- `.preorder` contains the down-closure of all the elements of the pre-ordered set Ω. This type of data corresponds to the type of data outputted by the following procedure composition (see section 4.2 and section 4.1)

`_transitive_closure(_read_pre_order(-)[1])`

```
6   def __init__(self,name_of_file,length_of_segments):
7   """ the source code of this constructor can be found in cl_cos.py """
```

For its part, the constructor `.__init__` takes two inputs, namely the name of file that contains a pre-order structure and an integer, and stores

- the integer in the object `.domain`;
- a Boolean value specifying whether the pre-order structure comprises a formal initial object in `.mask`;
- the down-closures of the elements of the pre-order in `.preorder`.

The other methods of the class are very 'categorical', in the sense that they either answers question that are relevant to the category structure of $\mathbf{Seg}(\Omega \,|\, n)$ or provide items that would be directly accessible from its structure.

```
 8   def can_switch_to(self,color1,color2):
 9   """ the source code of this function can be found in cl_cos.py """
10       return (color2 in self.preorder[i])
11   def identity(self,source,target):
12     return (segment1.topology == segment2.topology)\
13     and (segment1.colors == segment2.colors)
13   def homset(self,source,target):
14   """ the source code of this function can be found in cl_cos.py """
15     return flag
16   def topology(self,expression):
17   """ the source code of this function can be found in cl_cos.py """
18     return the_topology
19   def segment(self,expression):
20   """ the source code of this function can be found in cl_cos.py """
21     return the_segment
```

4.4.1. Comparing colors. The method `.can_switch_to` takes two strings that are meant to be names of elements in the pre-ordered set and returns a Boolean value indicating whether the first string is greater than or equal to the second one with respect to the pre-order structure.

For illustration, consider the following pre-order specification.

omega.yml

```
1 !obj:  0 1 2
2 rel:  2 > 1; 0 > 2
```

We can then use the method `.can_switch_to` to verify whether a certain a pre-order relation holds as follows.

```
>>> Seg = CategoryOfSegments("omega.yml",50)
>>> print(Seg.can_switch_to("2","1"))
True
>>> print(Seg.can_switch_to("1","2"))
False
>>> print(Seg.can_switch_to("1","1"))
True
>>> print(Seg.can_switch_to("0","1"))
True
```

4.4.2. Identities. The method `.identity` takes two `SegmentObject` items and returns a Boolean value indicating whether there is a morphism of segments from the first input to the second input.

The following example uses the method `.segment`, described in section 4.4.5, to create two `SegmentObject` items from regular expressions.

```
>>> Seg = CategoryOfSegments("omega.yml",50)
>>> s1 = Seg.segment([(0,3,10,"1"),(30,6,2,"2")])
>>> s2 = Seg.segment([(0,6,7,"1")])
>>> print(Seg.identity(s1,s2))
False
>>> print(Seg.identity(s1,s1))
```

```
True
```

4.4.3. Hom-sets. The method `.homset` takes two `SegmentObject` items (see section 4.3) and returns a Boolean value specifying if there is a morphism from the first one to the second one. Theoretically, this function parses the two segments as if the object `.mask` was set to the value `True`, in which case the topologies of the segments may lack some patches, which are then assumed to have the color of the formal initial object of the pre-ordered set $(\Omega, \preceq)$ (see section 4.4.5).

The following example uses the method `.segment`, described in section 4.4.5, to create two `SegmentObject` items from regular expressions.

```
>>> Seg = CategoryOfSegments("omega.yml",50)
>>> s1 = Seg.segment([(0,3,10,"1"),(30,6,2,"2")])
>>> s2 = Seg.segment([(0,6,7,"1")])
>>> print(Seg.homset(s1,s2))
True
>>> s1 = Seg.segment([(0,3,10,"1"),(30,7,2,"2")])
>>> s2 = Seg.segment([(0,6,7,"1")])
>>> print(Seg.homset(s1,s2))
False
```

4.4.4. Constructing a topology. The method `.topology` takes a list of 3-tuples and returns a list of 2-tuples. The input list is meant to encode a regular expression that describes the topology of a segment. The 3-tuples contained by the input list specifies

- where a patch starts;
- how long the patch is;
- how many such patches are repeated successively.

Specifically, the input satisfies the following syntax:

```
[(start_position, length_of_patch, number_of_such_patches), (etc.)  ]
```

Also, note that the start positions of the 3-tuples contained in the regular expression need to be given in an increasing order. For instance, a topology of the form

$$(\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ)$$

would be encoded by the following lists of 3-tuples:

```
[(0, 3, 2), (6, 4, 1), (10, 3, 4)]
```

However, as is shown, the expression given below is not a valid example.

```
>>> Seg = CategoryOfSegments("omega.yml",22)
>>> expr = [(10, 3, 4), (6, 4, 1), (0, 3, 2)]
>>> s = Seg.topology(expr)
Error:  in CategoryOfSegments.topology:  expression is not well-formed
```

The output of the procedure is the implementation of the regular expression into a topology (a list of 2-tuples). The 2-tuples contained in the output are the start and end positions of each patch. For instance, the output that would be produced for our first example (given above) would be as follows.

```
[(0, 2), (3, 5), (6, 9), (10, 12), (13, 15), (16, 18), (19, 21)]
```

Also, depending on whether the object `.mask` is `True` or `False`, the regular expression either needs to cover the whole topology or only needs to give a few relevant patches whose color might not be that of the initial object of the pre-order structure.

1) If `.mask` contains `True`, then only a few patches can be specified so that the remaining patches of the topology are implicit. For instance, the expression `[(5,3,1)]` can specify either one of the following topologies when `.domain` is equal to `15`.

(4.2)

$$(\circ\circ\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ\circ\circ\circ\circ) \qquad (\circ)(\circ)(\circ)(\circ)(\circ)(\circ\circ\circ)(\circ)(\circ)(\circ)(\circ)(\circ)(\circ)(\circ)$$

In practice, it is always recommended to associate the implicit part of the topology with the 'mask' color – given by the initial object of $(\Omega, \preceq)$.

```
>>> Seg = CategoryOfSegments("omega.yml",15)
>>> expr = [(5,3,1)]
>>> s = Seg.topology(expr)
>>> print(s)
[(5, 7)]
```

2) If `.mask` contains `False`, then all the patches of the topology must be specified so that the only way to specify the topology given on the left-hand side of (4.2) is to give the following expression.

```
>>> expr = [(0,5,1), (5,3,1), (8,7,1)]
>>> s = Seg.topology(expr)
>>> print(s)
[(0, 4), (5, 7), (8, 14)]
```

4.4.5. Constructing a segment. The method `.segment` takes is very similar to the procedure `.topology`. It takes a list of 4-tuples and returns a `SegmentObject` item. The input list is meant to encode a regular expression that describes the segment. The 4-tuples contained by the input list specifies

- where a patch starts;
- how long the patch is;
- how many such patches are repeated successively;
- the color name (or state) associated with the group of patches.

Specifically, the input satisfies the following syntax:

```
[(start_position, length_of_patch, number_of_such_patches, color), (etc.)  ]
```

Also, note that the start positions of the 4-tuples contained in the regular expression need to be given in an increasing order. For instance, a segment of the form

$$(000)(000)(1111)(222)(222)(222)(222)$$

would be encoded by the following lists of 3-tuples:

```
[(0,3,2,'0'), (6,4,1,'1'), (10,3,4,'2')]
```

However, as is shown, the expression given below is not a valid example.

```
>>> expr = [(10,3,4,'2'), (6,4,1,'1'), (0,3,2,'0')]
>>> s = Seg.segment(expr)
Error:  in CategoryOfSegments.segment:  expression is not well-formed
```

In addition, the color names associated with the patches of the segment must be elements of the pre-order structure specified in the object `.preorder`.

```
>>> Seg = CategoryOfSegments("omega.yml",22)
>>> expr = [(0,3,2,'0'), (6,4,1,'1'), (10,3,4,"red")]
>>> s = Seg.segment(expr)
Error:  in CategoryOfSegments.segment:  color 'red' not found in .preorder
```

The output of the procedure is the implementation of the regular expression into a `SegmentObject` item. For instance, the segment that would be produced for our first example (given above) would be given by the following pair of lists.

```
topology = [(0, 2), (3, 5), (6, 9), (10, 12), (13, 15), (16, 18), (19, 21)]
colors = ['0', '0', '1', '2', '2', '2', '2']
```

Also, depending on whether the object `.mask` is `True` or `False`, the regular expression either needs to cover the whole segment or only needs to give a few relevant patches whose color is not that of the initial object potentially specified in the pre-order structure.

1) If `.mask` contains `True`, then only a few patches can be specified so that the remaining patches of the segment have an implicit topology. For instance, the expression `[(5,3,1,'1')]` can specify either one of the following segments when `.domain` is equal to `15`.

(4.3)

$$(\circ\circ\circ\circ\circ)(\bullet\bullet\bullet)(\circ\circ\circ\circ\circ\circ\circ) \qquad (\circ)(\circ)(\circ)(\circ)(\circ)(\bullet\bullet\bullet)(\circ)(\circ)(\circ)(\circ)(\circ)(\circ)(\circ)$$

Note that, in practice, this polymorphism is never taken into account as it is always associated with the 'mask' color – given by the initial object of $(\Omega, \preceq)$.

```
>>> Seg = CategoryOfSegments("omega.yml",15)
>>> expr = [(5,3,1,'1')]
>>> s = Seg.segment(expr)
>>> print(s.topology)
[(5, 7)]
>>> print(s.colors)
['1']
```

2) If `.mask` contains `False`, then all the patches of the topology must be specified so that the only way to specify the topology given on the left-hand side of (4.3) is to give an expression as follows.

```
>>> expr = [(0,5,1,'0'), (5,3,1,'1'), (8,7,1,'0')]
>>> s = Seg.segment(expr)
>>> print(s.topology)
[(0, 4), (5, 7), (8, 14)]
>>> print(s.colors)
['0', '1', '0']
```

Chapter 5

Presentation of the module PedigradCategory.py

5.1. Description of read_alignment_file

The function `read_alignment_file` takes the name of a file and an integer and returns a pair of lists.

```
1 def read_alignment_file(name_of_file,reading_mode):
2   """ the source code of this function can be found in raf.py """
3   return (names,alignment)
```

The input file is given in terms of a string and should be as given in the following example, where `This is a text` is a text that does not contain the character '>'.

Align.fa

```
1 >Name of taxon1
2 This is a text
3
4 >Name of taxon2
5 This is a
6 text
```

The second input `reading_mode` specifies whether the text specifically contains a DNA sequence or any regular text. In the former case, the value of `reading_mode` must be `1`, which will imply that all lower case letters `a`, `c`, `g` and `t` are read as upper case letters `A`, `C`, `G` and `T`. The global variable `READ_DNA` can be used for this purpose.

```
>>> READ_DNA
1
```

The output `names` is a list of strings containing the names placed after the charater '>' in the input file while the output `alignment` is a list of lists of characters that spell each text displayed after the names saved in `names`.

As can be seen in the example given below, the index of a name in `names` corresponds to the index of its associated text in `alignment`.

```
>>> output = read_alignment_file("Align.fa",READ_DNA)
>>> for i in range(len(output[0])):
...   print(str(i)+":  "+str(output[0][i]))
0:  Name of taxon1
1:  Name of taxon2
>>> for i in range(len(output[1])):
...   print(str(i)+":  "+str(output[1][i]))
0:  ['T', 'h', 'i', 's', ' ', 'i', 's', ' ', 'A', ' ', 'T', 'e', 'x', 'T']
1:  ['T', 'h', 'i', 's', ' ', 'i', 's', ' ', 'A', 'T', 'e', 'x', 'T']
```

5.2. Description of ID_to_EQ

The function ID_to_EQ takes a string and returns a list of lists, which describes an equivalence relation whose equivalence classes (the internal lists) can be used during the parsing of a sequence alignment to identify certain molecular patches together.

```
1 def ID_to_EQ(name_ID):
2   """ the source code of this function can be found in ite.py """
3     return N21_EQ
4   else:
5     print("Error:  in ID_to_EQ: name_ID is not recognized")
5     exit()
```

The function ID_to_EQ is meant to be used with two sets of global variables. One sets consists of strings that define the valid inputs of ID_to_EQ while the other set consists of the possible outputs of ID_to_EQ, which are lists of lists that are

- either already equipped with specific equivalence classes describing a well-known evolution model (transition mutations, codon translation, etc.)
- or empty so that they can be 'edited' (see below).

We now give three examples of global variables that are associated with pre-determined equivalence classes, the first one being the trivial one, which is to say that it will involve no other identification than the reflexive ones (i.e. the identities).

```
>>> ID_to_EQ(NUCL_ID)
[[]]
```

Using the previous equivalence relation implies that the characters of a sequence alignment are read as-is (see section 5.3 for more details).

The next equivalence relation describes transition mutations (A ↔ G and C ↔ T), which commonly occur on the nucleotides of DNA strands. These are often silent mutations so that one sometimes reasonably wants to identify A with G and C with T.

```
>>> ID_to_EQ(TRAN_ID)
[['A', 'G'], ['C', 'T']]
```

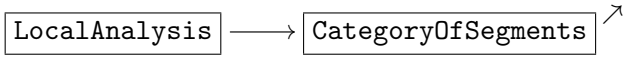
The last equivalence relation contains the equivalence classes induced by the codon table, that is to say those codons coding for the same amino acids.

```
>>> ID_to_EQ(AMIN_ID)
[['TTT', 'TTC'], ['TTA', 'TTG'], ['CTT', 'CTC', 'CTA', 'CTG'], ['TCT',
'TCC', 'TCA', 'TCG', 'AGT', 'AGC'], ['TAT', 'TAC'], ['CAT', 'CAC'], ['CAA',
'CAG'], ['TGT', 'TGC'], ['CGT', 'CGC', 'CGA', 'CGG'], ['ATT', 'ATC',
'ATA'], ['GTT', 'GTC', 'GTA', 'GTG'], ['ACT', 'ACC', 'ACA', 'ACG'], ['GCT',
'GCC', 'GCA', 'GCG'], ['AAT', 'AAC'], ['AAA', 'AAG'], ['GAT', 'GAC'],
['GAA', 'GAG'], ['AGA', 'AGG'], ['GGT', 'GGC', 'GGA', 'GGG']]
```

The user can also design its own equivalence relation (up to 21) via the (empty) global variables N01_EQ, N02_EQ, ..., N21_EQ, which are associated with the global strings N01_ID, N02_ID,..., N21_ID. The way an equivalence relation can be used with its associated string will be explained in section 5.3.

```
>>> print(N05_ID)
n5
>>> print(ID_to_EQ(N05_ID))
[]
>>> N05_EQ.append(["A","G"])
>>> print(N05_EQ)
[['A', 'G']]
>>> print(ID_to_EQ(N05_ID))
[['A', 'G']]
>>> N04_EQ.extend([["AC","TC"],["CC", "TC"]]))
>>> print(ID_to_EQ(N04_ID))
[['AC', 'TC'], ['CC', 'TC']]
```

5.3. Description of LocalAnalysis (subclass)



The class LocalAnalysis is a subclass of CategoryOfSegments (section 4.4) that possesses three objects, namely

- .equiv (list of strings)
- .base (list of SegmentObject items)

and one method, namely a constructor .__init__.

```
1 class LocalAnalysis(CategoryOfSegments):
2   #The objects of the class are:
3   #.equiv (list of strings);
4   #.base (list of SegmentObjects);
5   def __init__(self,analysis_mode,*args):
6   """ the source code of this constructor can be found in cl_la.py """
```

The idea behind the class LocalAnalysis is to specify the shape of a local analysis [4] via the two objects .equiv and .base. It is however important to understand that a LocalAnaylsis item does not strictly define a local analysis in the sense of [4] because it is not associated with a sequence alignment yet. The sequence alignment is only meant to be specified at the level of a Pedigrad item (section 5.5).

In other words, the present structure only specifies the schema of a local analysis, by describing its pattern of construction, rather than the associated functor $L : B \to \mathbf{Uprt}(S)$ induced from the data of a sequence alignment. Even though the domain B is contained in the object .base, the mapping rules are not properly defined. Instead, one specifies, in the object .equiv, recipes for these mapping rules. Such a pattern specification is then used, in section 5.5, to generate what is regarded as a pedigrad [5, 4].

The constructor .__init__ of LocalAnalysis takes between 4 and 5 arguments and uses them to initialize the objects of the class according to certain pre-determined or personalized patterns. The first argument taken by .__init__ is always a string. However, the form of its second and third arguments may vary depending on the string contained in the first argument. On the other hand, the last two arguments always keep the same form, namely a

string and an integer, which are given to the constructor of `CategoryOfSegments` to initialize the (super) objects `.domain`, `.mask`, and `.preorder` (see section 4.4)

More specifically, the first argument of `.__init__` is meant to be part of a set of global variables containing strings, which are all displayed below.

```
EXPR_MODE = 'exp'
SEGM_MODE = 'seg'
NUCL_MODE = 'nu'
TRAN_MODE = 'tr'
TRN0_MODE = 'tr0nu'
TRN1_MODE = 'tr1nu'
TRN2_MODE = 'tr2nu'
AMN0_MODE = 'aa0'
AMN1_MODE = 'aa1'
AMN2_MODE = 'aa2'
AMIN_MODE = 'aa'
```

Depending on the string contained in the first argument, the constructor `.__init__` may take four or five arguments. In addition, the second and third arguments may also take different forms.

The procedures `.__init__(NUCL_MODE,−)` and `.__init__(TRAN_MODE,−)` take three arguments whose last two arguments should be as described above, namely a string and an integer meant to be fed to the method `.__init__` of `CategoryOfSegments`. For its part, the first argument should be a string that is the name of an element in the pre-order structure stored in the object `.preorder`.

The following example illustrates the two types of local analysis schemas that are obtained from the global variables `NUCL_MODE` and `NUCL_MODE`. The pre-order structure `omega.yml` used below is that given in section 4.4.1.

```
>>> Loc = LocalAnalysis(NUCL_MODE,'1',"omega.yml",10)
>>> print(Loc.domain)
10
>>> print(Loc.mask)
True
>>> print(Loc.preorder)
[['0', '2', '1'], ['1'], ['2', '1']]
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(0, 0)] ['1'] ---> nu
segment 1:  [(1, 1)] ['1'] ---> nu
segment 2:  [(2, 2)] ['1'] ---> nu
segment 3:  [(3, 3)] ['1'] ---> nu
segment 4:  [(4, 4)] ['1'] ---> nu
segment 5:  [(5, 5)] ['1'] ---> nu
segment 6:  [(6, 6)] ['1'] ---> nu
segment 7:  [(7, 7)] ['1'] ---> nu
segment 8:  [(8, 8)] ['1'] ---> nu
segment 9:  [(9, 9)] ['1'] ---> nu
```

```
>>> Loc = LocalAnalysis(TRAN_MODE,'2',"omega.yml",10)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(0, 0)] ['2'] ---> tr
segment 1:  [(1, 1)] ['2'] ---> tr
segment 2:  [(2, 2)] ['2'] ---> tr
segment 3:  [(3, 3)] ['2'] ---> tr
segment 4:  [(4, 4)] ['2'] ---> tr
segment 5:  [(5, 5)] ['2'] ---> tr
segment 6:  [(6, 6)] ['2'] ---> tr
segment 7:  [(7, 7)] ['2'] ---> tr
segment 8:  [(8, 8)] ['2'] ---> tr
segment 9:  [(9, 9)] ['2'] ---> tr
```

From the point of view of the function `ID_to_EQ` (see section 5.2), the previous two procedures specify how the images of the local analyses should be 'quotiented' via the strings contained in the object `.equiv`. Specifically, the type of local analysis returned by

`.__init__(NUCL_MODE,−)`

is meant to read the columns of a sequence alignment without identification between the characters while the type of local analysis returned by `.__init__(TRAN_MODE,−)` is meant to identify `A` with `G` and `C` with `T`.

```
>>> Loc = LocalAnalysis(NUCL_MODE,'1',"omega.yml",10)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(0, 0)] ['1'] ---> [[]]
segment 1:  [(1, 1)] ['1'] ---> [[]]
segment 2:  [(2, 2)] ['1'] ---> [[]]
segment 3:  [(3, 3)] ['1'] ---> [[]]
segment 4:  [(4, 4)] ['1'] ---> [[]]
segment 5:  [(5, 5)] ['1'] ---> [[]]
segment 6:  [(6, 6)] ['1'] ---> [[]]
segment 7:  [(7, 7)] ['1'] ---> [[]]
segment 8:  [(8, 8)] ['1'] ---> [[]]
segment 9:  [(9, 9)] ['1'] ---> [[]]
>>> Loc = LocalAnalysis(TRAN_MODE,'2',"omega.yml",10)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(0, 0)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 1:  [(1, 1)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 2:  [(2, 2)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 3:  [(3, 3)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 4:  [(4, 4)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 5:  [(5, 5)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 6:  [(6, 6)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 7:  [(7, 7)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 8:  [(8, 8)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 9:  [(9, 9)] ['2'] ---> [['A', 'G'], ['C', 'T']]
```

The three procedures

.__init__(TRN0_MODE,−), .__init__(TRN1_MODE,−) and .__init__(TRN2_MODE,−)

take three arguments whose last two arguments are as described above, namely a string and an integer meant to be fed to the method .__init__ of CategoryOfSegments. For its part, the first argument should be a list of two strings that are the names of elements in the pre-order structure stored in the object .preorder.

```
>>> Loc = LocalAnalysis(TRN0_MODE,['2','1'],"omega.yml",10)
>>> for i in range(len(Loc.base)):
...     print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(0, 0)] ['2'] ---> tr
segment 1:  [(1, 1)] ['1'] ---> nu
segment 2:  [(2, 2)] ['1'] ---> nu
segment 3:  [(3, 3)] ['2'] ---> tr
segment 4:  [(4, 4)] ['1'] ---> nu
segment 5:  [(5, 5)] ['1'] ---> nu
segment 6:  [(6, 6)] ['2'] ---> tr
segment 7:  [(7, 7)] ['1'] ---> nu
segment 8:  [(8, 8)] ['1'] ---> nu
segment 9:  [(9, 9)] ['2'] ---> tr
>>> Loc = LocalAnalysis(TRN2_MODE,['2','1'],"omega.yml",10)
>>> for i in range(len(Loc.base)):
...     print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(0, 0)] ['1'] ---> nu
segment 1:  [(1, 1)] ['1'] ---> nu
segment 2:  [(2, 2)] ['2'] ---> tr
segment 3:  [(3, 3)] ['1'] ---> nu
segment 4:  [(4, 4)] ['1'] ---> nu
segment 5:  [(5, 5)] ['2'] ---> tr
segment 6:  [(6, 6)] ['1'] ---> nu
segment 7:  [(7, 7)] ['1'] ---> nu
segment 8:  [(8, 8)] ['2'] ---> tr
segment 9:  [(9, 9)] ['1'] ---> nu
```

From the point of view of the function ID_to_EQ (see section 5.2), a procedure of the form

.__init__(TRNX_MODE,−)

generates a local analysis that is meant to identify A with G and C with T on every column whose position is X modulo 3 and that is meant to read the other columns as-is.

```
>>> Loc = LocalAnalysis(TRN0_MODE,['2','1'],"omega.yml",10)
>>> for i in range(len(Loc.base)):
...     print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(0, 0)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 1:  [(1, 1)] ['1'] ---> [[]]
segment 2:  [(2, 2)] ['1'] ---> [[]]
```

```
segment 3:  [(3, 3)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 4:  [(4, 4)] ['1'] ---> [[]]
segment 5:  [(5, 5)] ['1'] ---> [[]]
segment 6:  [(6, 6)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 7:  [(7, 7)] ['1'] ---> [[]]
segment 8:  [(8, 8)] ['1'] ---> [[]]
segment 9:  [(9, 9)] ['2'] ---> [['A', 'G'], ['C', 'T']]
>>> Loc = LocalAnalysis(TRN2_MODE,['2','1'],"omega.yml",10)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(0, 0)] ['1'] ---> [[]]
segment 1:  [(1, 1)] ['1'] ---> [[]]
segment 2:  [(2, 2)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 3:  [(3, 3)] ['1'] ---> [[]]
segment 4:  [(4, 4)] ['1'] ---> [[]]
segment 5:  [(5, 5)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 6:  [(6, 6)] ['1'] ---> [[]]
segment 7:  [(7, 7)] ['1'] ---> [[]]
segment 8:  [(8, 8)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 9:  [(9, 9)] ['1'] ---> [[]]
```

The three procedures

`.__init__(AMN0_MODE,−)`, `.__init__(AMN1_MODE,−)` and `.__init__(AMN2_MODE,−)`

take three arguments whose last two arguments are as described above, namely a string and an integer meant to be fed to the method `.__init__` of `CategoryOfSegments`. For its part, the first argument should be a string that is the name of an element in the pre-order structure stored in the object `.preorder`.

```
>>> Loc = LocalAnalysis(AMN0_MODE,'1',"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(0, 2)] ['1'] ---> aa
segment 1:  [(3, 5)] ['1'] ---> aa
segment 2:  [(6, 8)] ['1'] ---> aa
segment 3:  [(9, 11)] ['1'] ---> aa
segment 4:  [(12, 14)] ['1'] ---> aa
segment 5:  [(15, 17)] ['1'] ---> aa
segment 6:  [(18, 20)] ['1'] ---> aa
>>> Loc = LocalAnalysis(AMN1_MODE,'1',"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(1, 3)] ['1'] ---> aa
segment 1:  [(4, 6)] ['1'] ---> aa
segment 2:  [(7, 9)] ['1'] ---> aa
segment 3:  [(10, 12)] ['1'] ---> aa
segment 4:  [(13, 15)] ['1'] ---> aa
segment 5:  [(16, 18)] ['1'] ---> aa
```

```
>>> Loc = LocalAnalysis(AMN2_MODE,'1',"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(2, 4)] ['1'] ---> aa
segment 1:  [(5, 7)] ['1'] ---> aa
segment 2:  [(8, 10)] ['1'] ---> aa
segment 3:  [(11, 13)] ['1'] ---> aa
segment 4:  [(14, 16)] ['1'] ---> aa
segment 5:  [(17, 19)] ['1'] ---> aa
```

From the point of view of the function ID_to_EQ (see section 5.2), a procedure of the form

.__init__(AMNX_MODE,−)

generates a local analysis that reads the codons of a sequence alignment with the assumption that the codon topology starts at position X and identifies them according to the codon translation table.

```
>>> Loc = LocalAnalysis(AMN2_MODE,'1',"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(2, 4)] ['1'] ---> [['TTT', 'TTC'], ['TTA', 'TTG'], ['CTT',
'CTC', 'CTA', 'CTG'], ['TCT', 'TCC', 'TCA', 'TCG', 'AGT', 'AGC'], ['TAT',
'TAC'], ['CAT', 'CAC'], ['CAA', 'CAG'], ['TGT', 'TGC'], ['CGT', 'CGC',
'CGA', 'CGG'], ['ATT', 'ATC', 'ATA'], ['GTT', 'GTC', 'GTA', 'GTG'], ['ACT',
'ACC', 'ACA', 'ACG'], ['GCT', 'GCC', 'GCA', 'GCG'], ['AAT', 'AAC'], ['AAA',
'AAG'], ['GAT', 'GAC'], ['GAA', 'GAG'], ['AGA', 'AGG'], ['GGT', 'GGC',
'GGA', 'GGG']]
segment 1:  [(5, 7)] ['1'] ---> [['TTT', 'TTC'], ['TTA', 'TTG'], ['CTT',
'CTC', 'CTA', 'CTG'], ['TCT', 'TCC', 'TCA', 'TCG', 'AGT', 'AGC'], ['TAT',
'TAC'], ['CAT', 'CAC'], ['CAA', 'CAG'], ['TGT', 'TGC'], ['CGT', 'CGC',
'CGA', 'CGG'], ['ATT', 'ATC', 'ATA'], ['GTT', 'GTC', 'GTA', 'GTG'], ['ACT',
'ACC', 'ACA', 'ACG'], ['GCT', 'GCC', 'GCA', 'GCG'], ['AAT', 'AAC'], ['AAA',
'AAG'], ['GAT', 'GAC'], ['GAA', 'GAG'], ['AGA', 'AGG'], ['GGT', 'GGC',
'GGA', 'GGG']]
segment 2:  [(8, 10)] ['1'] ---> [['TTT', 'TTC'], ['TTA', 'TTG'], ['CTT',
'CTC', 'CTA', 'CTG'], ['TCT', 'TCC', 'TCA', 'TCG', 'AGT', 'AGC'], ['TAT',
'TAC'], ['CAT', 'CAC'], ['CAA', 'CAG'], ['TGT', 'TGC'], ['CGT', 'CGC',
'CGA', 'CGG'], ['ATT', 'ATC', 'ATA'], ['GTT', 'GTC', 'GTA', 'GTG'], ['ACT',
'ACC', 'ACA', 'ACG'], ['GCT', 'GCC', 'GCA', 'GCG'], ['AAT', 'AAC'], ['AAA',
'AAG'], ['GAT', 'GAC'], ['GAA', 'GAG'], ['AGA', 'AGG'], ['GGT', 'GGC',
'GGA', 'GGG']]
segment 3:  [(11, 13)] ['1'] ---> [['TTT', 'TTC'], ['TTA', 'TTG'], ['CTT',
'CTC', 'CTA', 'CTG'], ['TCT', 'TCC', 'TCA', 'TCG', 'AGT', 'AGC'], ['TAT',
'TAC'], ['CAT', 'CAC'], ['CAA', 'CAG'], ['TGT', 'TGC'], ['CGT', 'CGC',
'CGA', 'CGG'], ['ATT', 'ATC', 'ATA'], ['GTT', 'GTC', 'GTA', 'GTG'], ['ACT',
'ACC', 'ACA', 'ACG'], ['GCT', 'GCC', 'GCA', 'GCG'], ['AAT', 'AAC'], ['AAA',
'AAG'], ['GAT', 'GAC'], ['GAA', 'GAG'], ['AGA', 'AGG'], ['GGT', 'GGC',
'GGA', 'GGG']]
```

```
segment 4:  [(14, 16)] ['1'] ---> [['TTT', 'TTC'], ['TTA', 'TTG'], ['CTT',
'CTC', 'CTA', 'CTG'], ['TCT', 'TCC', 'TCA', 'TCG', 'AGT', 'AGC'], ['TAT',
'TAC'], ['CAT', 'CAC'], ['CAA', 'CAG'], ['TGT', 'TGC'], ['CGT', 'CGC',
'CGA', 'CGG'], ['ATT', 'ATC', 'ATA'], ['GTT', 'GTC', 'GTA', 'GTG'], ['ACT',
'ACC', 'ACA', 'ACG'], ['GCT', 'GCC', 'GCA', 'GCG'], ['AAT', 'AAC'], ['AAA',
'AAG'], ['GAT', 'GAC'], ['GAA', 'GAG'], ['AGA', 'AGG'], ['GGT', 'GGC',
'GGA', 'GGG']]
segment 5:  [(17, 19)] ['1'] ---> [['TTT', 'TTC'], ['TTA', 'TTG'], ['CTT',
'CTC', 'CTA', 'CTG'], ['TCT', 'TCC', 'TCA', 'TCG', 'AGT', 'AGC'], ['TAT',
'TAC'], ['CAT', 'CAC'], ['CAA', 'CAG'], ['TGT', 'TGC'], ['CGT', 'CGC',
'CGA', 'CGG'], ['ATT', 'ATC', 'ATA'], ['GTT', 'GTC', 'GTA', 'GTG'], ['ACT',
'ACC', 'ACA', 'ACG'], ['GCT', 'GCC', 'GCA', 'GCG'], ['AAT', 'AAC'], ['AAA',
'AAG'], ['GAT', 'GAC'], ['GAA', 'GAG'], ['AGA', 'AGG'], ['GGT', 'GGC',
'GGA', 'GGG']]
```

The procedure `.__init__(AMIN_MODE,−)`, takes three arguments whose last two arguments are as described above, namely a string and an integer meant to be fed to the method `.__init__` of `CategoryOfSegments`. For its part, the first argument should be a list of three strings that are the names of elements in the pre-order structure stored in the object `.preorder`. From the point of view of the function `ID_to_EQ` (see section 5.2), the procedure

`.__init__(AMIN_MODE,−)`

generates a local analysis that reads all the codons of the sequence and identifies them according to the codon translation table.

```
>>> Loc =LocalAnalysis(AMIN_MODE,['0','1','2'],"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(0, 2)] ['0'] ---> aa
segment 1:  [(3, 5)] ['0'] ---> aa
segment 2:  [(6, 8)] ['0'] ---> aa
segment 3:  [(9, 11)] ['0'] ---> aa
segment 4:  [(12, 14)] ['0'] ---> aa
segment 5:  [(15, 17)] ['0'] ---> aa
segment 6:  [(18, 20)] ['0'] ---> aa
segment 7:  [(1, 3)] ['1'] ---> aa
segment 8:  [(4, 6)] ['1'] ---> aa
segment 9:  [(7, 9)] ['1'] ---> aa
segment 10:  [(10, 12)] ['1'] ---> aa
segment 11:  [(13, 15)] ['1'] ---> aa
segment 12:  [(16, 18)] ['1'] ---> aa
segment 13:  [(2, 4)] ['2'] ---> aa
segment 14:  [(5, 7)] ['2'] ---> aa
segment 15:  [(8, 10)] ['2'] ---> aa
segment 16:  [(11, 13)] ['2'] ---> aa
segment 17:  [(14, 16)] ['2'] ---> aa
segment 18:  [(17, 19)] ['2'] ---> aa
```

Finally the user can design their own local analysis by using the procedures

`.__init__(EXPR_MODE,−)` and `.__init__(SEGM_MODE,−)`,

which take four more arguments as described below:

- the first argument of .__init__(EXPR_MODE,−) should be a list of regular expressions specifying SegmentObject items (section 4.3). This list is then turned into a list of SegmentObject items, which is to be stored in the object .base;

- the first argument of .__init__(SEGM_MODE,−) should be a list of SegmentObject items (section 4.3), which is to be stored in the object .base;

- the second argument should be a list of strings, which is to be stored in the object .equiv. More specifically, the strings should be among those contained in the global variables NUCL_ID, TRAN_ID, AMIN_ID, NO1_ID, NO2_ID, . . . , and N21_ID;

- and, as mentioned many times above, the last two arguments should be a string and an integer, which are to be passed to the method .__init__ of CategoryOfSegments (see section 4.4).

Below, we show two examples of how the previous two procedures can be used. Note how theses examples uses the (empty) global variables NO1_EQ and NO2_EQ (see section 5.2).

```
>>> NO1_EQ.append(["A","G"])
>>> NO2_EQ.extend([["AC","TC"],["CC", "TC"]])
>>> base = [[(5,1,1,'1')], [(6,2,1,'2')], [(8,1,1,'1')], [(9,2,2,'2')]]
>>> equiv = [NO1_ID, NO2_ID, NO1_ID, NO2_ID, NO2_ID]
>>> Loc = LocalAnalysis(EXPR_MODE,base,equiv,"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(5, 5)] ['1'] ---> n1
segment 1:  [(6, 7)] ['2'] ---> n2
segment 2:  [(8, 8)] ['1'] ---> n1
segment 3:  [(9, 10), (11, 12)] ['2', '2'] ---> n2
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(5, 5)] ['1'] ---> [['A', 'G']]
segment 1:  [(6, 7)] ['2'] ---> [['AC', 'TC'], ['CC', 'TC']]
segment 2:  [(8, 8)] ['1'] ---> [['A', 'G']]
segment 3:  [(9, 10), (11, 12)] ['2', '2'] ---> [['AC', 'TC'], ['CC',
'TC']]
```

Alternatively, the procedure .__init__(SEGM_MODE,−) would require to have a pre-defined environment in which SegmentObject items can be defined. Note the difference between the previous lines of codes and the following ones, which use the class CategoryOfSegments.

```
>>> NO1_EQ.append(["A","G"])
>>> NO2_EQ.extend([["AC","TC"],["CC", "TC"]])
>>> Seg = CategoryOfSegments("omega.yml",21)
>>> base = [Seg.segment([(5,1,1,'1')]), Seg.segment([(6,2,1,'2')]),
    Seg.segment([(8,1,1,'1')]), Seg.segment([(9,2,2,'2')])]
>>> equiv = [NO1_ID, NO2_ID, NO1_ID, NO2_ID, NO2_ID]
>>> Loc = LocalAnalysis(SEGM_MODE,base,equiv,"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
```

```
segment 0:  [(5, 5)] ['1'] ---> n1
segment 1:  [(6, 7)] ['2'] ---> n2
segment 2:  [(8, 8)] ['1'] ---> n1
segment 3:  [(9, 10), (11, 12)] ['2', '2'] ---> n2
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(5, 5)] ['1'] ---> [['A', 'G']]
segment 1:  [(6, 7)] ['2'] ---> [['AC', 'TC'], ['CC', 'TC']]
segment 2:  [(8, 8)] ['1'] ---> [['A', 'G']]
segment 3:  [(9, 10), (11, 12)] ['2', '2'] ---> [['AC', 'TC'], ['CC',
'TC']]
```

5.4. Description of column_is_trivial

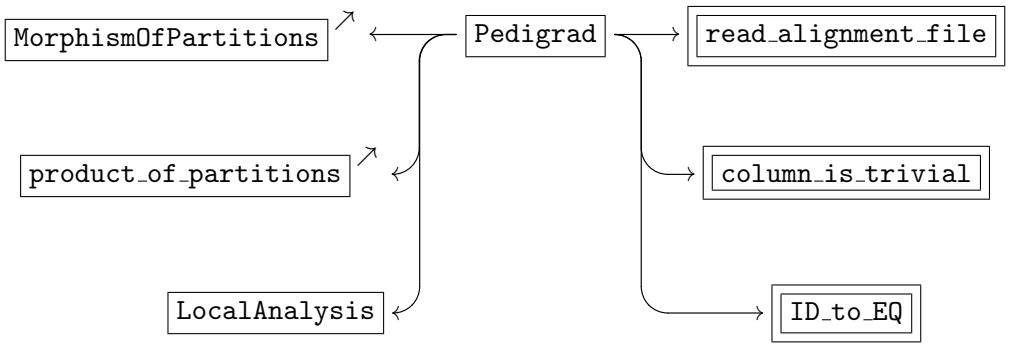
The function column_is_trivial takes two lists of elements and returns True if, apart from the elements in the second input list, the first input list contains copies of a unique element; returns False if, apart from the elements in the second input list, the first input list contains at least two different elements.

```
1 def column_is_trivial(column,exceptions):
2   """ the source code of this function can be found in cit.py """
3   return flag
```

As will be seen later in other functions, there are often special characters that need to be handled differently from the other characters. For instance, the character '.' is often used in alignments to mean that a character is actually missing. These special characters sometimes need to be ignored in the analysis.

```
>>> p = ['e','e','e','e','e']
>>> print(column_is_trivial(p,[]))
True
>>> p = ['e','e','a','a','e']
>>> print(column_is_trivial(p,[]))
False
>>> p = [0,'.',0,0,'.',0,1,0]
>>> print(column_is_trivial(p,['.']))
False
>>> print(column_is_trivial(p,['.',1]))
True
```

5.5. Description of Pedigrad (subclass)



The class Pedigrad is a subclass of LocalAnalysis (section 5.3) that possesses two objects, namely

- .local (list of lists)
- .taxa (list)

and two methods, namely

- .__init__ (constructor)
- .partition
- .reduce
- .agree

The constructor .__init__ takes from 5 to 6 arguments, which are meant to be used with the function read_alignment_file and the constructor of LocalAnalysis. More specifically, the constructor of Pedigrad usually takes

- the name of a file and an integer, given to the function read_alignment_file;
- a list of arguments, given to the constructor of LocalAnalysis.

```
1 class Pedigrad(LocalAnalysis):
2   #The objects of the class are:
3   #.local (list);
4   #.taxa (list).
5   def __init__(self,name_of_file,reading_mode,*args):
6   """ the source code of this constructor can be found in cl_ped.py """
```

The list of arguments *args (shown above) usually contains those arguments that would be given to the constructor of LocalAnalysis (see .__init__ in section 5.3), except for the last argument, which

- does not need to be specified if the name of the file is the name of an existing file;
- needs to be specified if the name of the file is empty (see the examples given later).

Throughout this section, we will use the preorder structure omega.yml given in section 4.4.1. We will however use various multiple sequence alignment. For our first example, we will consider the following alignment file.

Align.fa

```
1 >A
2 aaaaaaaaaf
3 >B
4 bbbbbbbbbf
5 >C
6 ccccccccf
7 >D
8 ddddddddf
9 >E
10 eeeeeeeeef
```

The following lines of code illustrate the syntax with which an instance of Pedigrad can be constructed and compares it to that used to construct an instance of LocalAnalysis.

```
>>> P = Pedigrad("Align.fa",not(READ_DNA),TRN2_MODE,['2','1'],"omega.yml")
>>> L = LocalAnalysis(TRN2_MODE,['2','1'],"omega.yml",500)
>>> P = Pedigrad("Align.fa",not(READ_DNA),AMN0_MODE,'1',"omega.yml")
>>> L = LocalAnalysis(AMN0_MODE,'1',"omega.yml",500)
>>> P = Pedigrad("",not(READ_DNA),TRN2_MODE,['2','1'],"omega.yml")
TypeError:  __init__() takes exactly 3 arguments (2 given)
>>> P = Pedigrad("",not(READ_DNA),TRN2_MODE,['2','1'],"omega.yml",500)
```

If the name of the file that is passed to the contructor, in its first argument, is not empty, then the first two arguments are passed to the procedure `read_alignment_file(−,−)` and the constructor proceeds to the initialization of the objects `.local`, `.taxa` and `.base` as follows:

- it stores the first output of `read_alignment_file` in the object `.taxa`;
- it appends the remaining lists of arguments with the length of the second output of `read_alignment_file` and gives the resulting tuples of arguments to the constructor of `LocalAnalysis` (see below).

```
super(Pedigrad, self).__init__(*args+(len(alignment[0]),))
```

Once the object `.base` of the super class has been (pre-)initialized by this process, the `SegmentObject` items that it contains are used to parse the sequence alignment. The parsed data is then stored in the object `.local` as follows:

- ▷ For every patch `(x,y)` occurring in the topology of i-th segment contained `.base`, the characters appearing from position `x` to position `y`, in the sequence alignment, are collected for each line of the sequence alignment (i.e. each list of characters contained in the second output of `read_alignment_file`) and put together to form a list of strings;
- ▷ This list of strings is then relabeled with respect the indices of the lists contained in the equivalence classes that are meant to be given to the constructor of `LocalAnalysis`;
- ▷ If the relabeled list of strings is not trivial according to the procedure

  `column_is_trivial(−,[])`,

  then it is stored at position i in the object `.local`;
- ▷ If the lists of strings happens to be trivial, then it is not stored and the associated segment is removed from the object `.base` (thus shifting the indexing).

```
>>> print(NUCL_EQ)
[[]]
>>> P = Pedigrad("Align.fa",not(READ_DNA),NUCL_MODE,'2',"omega.yml")
>>> print(P.taxa)
[' A', ' B', ' C', ' D', ' E']
>>> print(P.domain,len(P.base),len(P.local))
(10, 9, 9)
```

```
>>> for i in range(len(P.base)):
...   print("segment " + str(i) + ":  " + str(P.base[i].topology) + ",
    "+str(P.base[i].colors) + " ---> " + str(P.local[i]))
segment 0:  [(0, 0)], [‘2’] ---> [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
segment 1:  [(1, 1)], [‘2’] ---> [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
segment 2:  [(2, 2)], [‘2’] ---> [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
segment 3:  [(3, 3)], [‘2’] ---> [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
segment 4:  [(4, 4)], [‘2’] ---> [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
segment 5:  [(5, 5)], [‘2’] ---> [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
segment 6:  [(6, 6)], [‘2’] ---> [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
segment 7:  [(7, 7)], [‘2’] ---> [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
segment 8:  [(8, 8)], [‘2’] ---> [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
```

When the `SegmentObject` items of the base only contain patches of length 1 and the equivalence classes are empty (as above), then the list of lists stored in `.local` corresponds to the transpose of the sequence alignment (when seen as a matrix), up to removal of those columns (in the alignment) that only contain the same character.

Below, we give several other examples, some of which present bases whose `SegmentObject` items contain patches of length greater than 1.

Our first example illustrates how the lists of strings contained in the object `.local` are relabeled when the equivalence class contained in `NUCL_EQ` is replaced with non-trivial ones.

```
>>> NUCL_EQ.remove([])
>>> NUCL_EQ.extend([[‘e’],[‘d’],[‘c’],[‘b’],[‘a’]])
>>> print(NUCL_EQ)
[[‘e’], [‘d’], [‘c’], [‘b’], [‘a’]]
>>> P = Pedigrad("Align.fa",not(READ_DNA),NUCL_MODE,‘2’,"omega.yml")
>>> for i in range(len(P.base)):
...   print("segment " + str(i) + ":  " + str(P.base[i].topology) + ",
    "+str(P.base[i].colors) + " ---> " + str(P.local[i]))
segment 0:  [(0, 0)], [‘2’] ---> [4, 3, 2, 1, 0]
segment 1:  [(1, 1)], [‘2’] ---> [4, 3, 2, 1, 0]
segment 2:  [(2, 2)], [‘2’] ---> [4, 3, 2, 1, 0]
segment 3:  [(3, 3)], [‘2’] ---> [4, 3, 2, 1, 0]
segment 4:  [(4, 4)], [‘2’] ---> [4, 3, 2, 1, 0]
segment 5:  [(5, 5)], [‘2’] ---> [4, 3, 2, 1, 0]
segment 6:  [(6, 6)], [‘2’] ---> [4, 3, 2, 1, 0]
segment 7:  [(7, 7)], [‘2’] ---> [4, 3, 2, 1, 0]
segment 8:  [(8, 8)], [‘2’] ---> [4, 3, 2, 1, 0]
```

This second example is similar to the previous one, but the relabelling only occurs where the lists of strings conained in the object `.local` are detected. Note that the following example uses the modified version of `NUCL_EQ` designed in the previous example (see the first lines below).

```
>>> print(NUCL_EQ)
[[‘e’], [‘d’], [‘c’], [‘b’], [‘a’]]
>>> print(TRAN_EQ)
[[‘A’, ‘G’], [‘C’, ‘T’]]
```

```
>>> P = Pedigrad("Align.fa",not(READ_DNA),TRN2_MODE,['2','1'],"omega.yml")
>>> for i in range(len(P.base)):
...   print("segment " + str(i) + ":  " + str(P.base[i].topology) + ",
    "+str(P.base[i].colors) + " ---> " + str(P.local[i]))
segment 0:  [(0, 0)], ['1'] ---> [4, 3, 2, 1, 0]
segment 1:  [(1, 1)], ['1'] ---> [4, 3, 2, 1, 0]
segment 2:  [(2, 2)], ['2'] ---> ['a', 'b', 'c', 'd', 'e']
segment 3:  [(3, 3)], ['1'] ---> [4, 3, 2, 1, 0]
segment 4:  [(4, 4)], ['1'] ---> [4, 3, 2, 1, 0]
segment 5:  [(5, 5)], ['2'] ---> ['a', 'b', 'c', 'd', 'e']
segment 6:  [(6, 6)], ['1'] ---> [4, 3, 2, 1, 0]
segment 7:  [(7, 7)], ['1'] ---> [4, 3, 2, 1, 0]
```

We finish with the following example, in which the local analysis is designed by the user.

```
>>> N01_EQ.extend([['aaa','eee'],['bbb']])
>>> alignment = read_alignment_file("Align.fa",not(READ_DNA))
>>> domain = len(alignment[1][0])
>>> base = list()
>>> equiv = list()
>>> for i in range(domain/3):
...   base.append([(3*i,3,1,'1')])
...   equiv.append(N01_ID)
>>> P = Pedigrad("Align.fa",not(READ_DNA),EXPR_MODE,base,equiv,"omega.yml")
>>> print(P.domain,len(P.base),len(P.local))
(10, 3, 3)
>>> for i in range(len(P.base)):
...   print("segment " + str(i) + ":  " + str(P.base[i].topology) + ",
    "+str(P.base[i].colors) + " ---> " + str(P.local[i]))
segment 0:  [(0, 2)], ['1'] ---> [0, 1, 'ccc', 'ddd', 0]
segment 1:  [(3, 5)], ['1'] ---> [0, 1, 'ccc', 'ddd', 0]
segment 2:  [(6, 8)], ['1'] ---> [0, 1, 'ccc', 'ddd', 0]
```

If the name of the file passed to the contructor, in its first argument, is empty, then the objects `.local` and `.taxa` are initialized with empty lists while the other objects `.equiv`, `.base`, `.domain`, `.mask` and `.preorder` are initialized via the constructor of the class `LocalAnalysis`.

Let us now focus on the other methods of the class, which recover usual or canonical operations on pedigrads.

```
7   def partition(self,*args):
8   """ the source code of this constructor can be found in cl_ped.py """
9       return the_image
10   def isolate(self,update_mode,exceptions):
11   """ the source code of this constructor can be found in cl_ped.py """
12     if update_mode == NEW:
13       return new_pedigrad
14   def agree(self,ground,pulling_condition):
15   """ the source code of this constructor can be found in cl_ped.py """
16     return agreeing_segments
```

5.5.1. Partition. The method `.partition` takes

- either no argument;
- or the regular expression of `SegmentObject` item and a string (specifically `EXPR_MODE`);

- or a SegmentObject item and a string (specifically SEGM_MODE).

If no argument is given, then the procedure returns a terminal partition. Otherwise, the procedure returns (see the formula below) the image of the right Kan extension $\mathsf{Ran}L$ [4, Definition 4.29] of the underlying local analysis $L : B \to \mathbf{Uprt}(S)$ on the segment τ given as input.

$$\mathsf{Ran}L(\tau) = \prod_{\upsilon \in B} \prod_{\mathbf{Seg}(\Omega\,|\,n)(\tau,\upsilon)} L(\upsilon)$$

This image is obtained from the product of those partitions, in the object .local, whose indices correspond to the indices of those segments υ, in the object .base, toward which there is a morphism of segments from the input SegmentObject item τ.

$$\mathbf{Seg}(\Omega\,|\,n)(\tau,\upsilon) \neq \emptyset$$

For illustration, let us change the sequence alignment to the following one.

Align.fa

```
 1 >A
 2 CCCAGTTAG
 3 >B
 4 CCGATATAA
 5 >C
 6 GGCTTATAG
 7 >D
 8 GGCGTATAG
 9 >E
10 ACCATATAA
```

For simplicity, we shall suppose that the global variable is as initially provided by the library NUCL_EQ (empty) and we shall consider the following pedigrad.

```
>>> print(NUCL_EQ)
[[]]
>>> P = Pedigrad("Align.fa",not(READ_DNA),NUCL_MODE,'2',"omega.yml")
>>> for i in range(len(P.base)):
...   print("segment " + str(i) + ":  " + str(P.base[i].topology) + ",
    "+str(P.base[i].colors) + " ---> " + str(P.local[i]))
segment 0:  [(0, 0)], ['2'] ---> ['C', 'C', 'G', 'G', 'A']
segment 1:  [(1, 1)], ['2'] ---> ['C', 'C', 'G', 'G', 'C']
segment 2:  [(2, 2)], ['2'] ---> ['C', 'G', 'C', 'C', 'C']
segment 3:  [(3, 3)], ['2'] ---> ['A', 'A', 'T', 'G', 'A']
segment 4:  [(4, 4)], ['2'] ---> ['G', 'T', 'T', 'T', 'T']
segment 5:  [(5, 5)], ['2'] ---> ['T', 'A', 'A', 'A', 'A']
segment 6:  [(8, 8)], ['2'] ---> ['G', 'A', 'G', 'G', 'A']
```

First, recall that if no argument is given to the procedure .partition, then the procedure returns the terminal partition, whose (unique) value is chosen to be 0.

```
>>> print(P.partition())
[0, 0, 0, 0, 0]
```

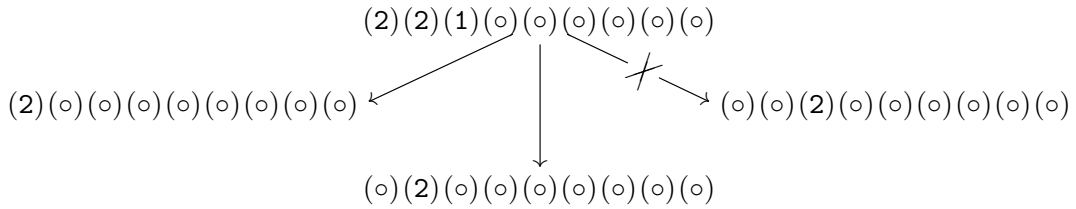
Let us now look at the outputs of the procedure partition when the regular expression of a segment is given as an input. For instance, suppose that one wants to compute the image of the following segment via the previously specified pedigrad.

(5.1) $$(2)(2)(1)(\circ)(\circ)(\circ)(\circ)(\circ)(\circ)$$

This would be encoded by the following specification.

```
>>> print(P.partition([(0,1,1,'2'),(1,1,1,'2'),(2,1,1,'1')],EXPR_MODE))
[0, 0, 1, 1, 2]
```

Recall that the earlier output is the product of those partitions, in `.local`, whose indices correspond to the indices of those segments, in `.base`, toward which there is a morphism of segment from the input `SegmentObject` item.



This calculation is made more explicit in the following lines of code, in which the segments of the base are given by the regular expressions `[(0,1,1,'2')]`, `[(1,1,1,'2')]`, and `[(2,1,1,'2')]`.

```
>>> s1 = P.segment([[(0,1,1,'2'),(1,1,1,'2'),(2,1,1,'1')])
>>> s2 = P.segment([(0,1,1,'2')]))
>>> print(P.homset_is_inhabited(s1,s2))
True
>>> s3 = P.segment([(1,1,1,'2')]))
>>> print(P.homset_is_inhabited(s1,s3))
True
>>> s4 = P.segment([(2,1,1,'2')]))
>>> print(P.homset_is_inhabited(s1,s4))
False
>>> p1 = P.partition(s1,SEGM_MODE)
>>> p2 = P.partition(s2,SEGM_MODE)
>>> print(product_of_partitions(p1,p2))
[0, 0, 1, 1, 2]
```

For the sake of comparison, we can now compute the image of the following segment.

(5.2) (2)(2)(2)(∘)(∘)(∘)(∘)(∘)(∘)

```
>>> print(P.partition([(0,1,1,'2'),(1,1,1,'2'),(2,1,1,'2')],EXPR_MODE))
[0, 1, 2, 2, 3]
```

The difference between the image of segment (5.1) and that of segment (5.2) comes from non-triviality if the following image.

```
>>> print(P.partition([(2,1,1,'2')],EXPR_MODE))
[0, 1, 0, 0, 0]
```

5.5.2. Isolate. The method `.isolate` takes an integer and a list of characters and may or may not return an output depending on the value of the integer, specifically:

- if the integer is equal to the global variable `NEW` (i.e. equal to 1), then the procedure returns a new `Pedigrad` item;
- otherwise, the procedure changes the ambient pedigrad directly.

```
>>> print(NEW,not(NEW))
(1, False)
```

If the input integer is equal to NEW, then the procedure returns a pedigrad that has the same attributes as the ambient pedigrad, except for its object .local whose characters that belong to the input list are given new and distinguished labels so that they give rise to singleton parts in the partitioning given by the images of the pedigrad.

If the input integer is not equal to NEW, then the procedure replaces those characters of each internal list of self.local that belong to the input list with new and distinguished characters so that these give rise to singleton parts in the partitioning given by the images of the pedigrad.

For illustration, let us suppose that our sequence alignment file Align.fa now contains some special characters, as shown below, in red.

Align.fa

```
 1 >A
 2 CCCAGTT.G
 3 >B
 4 CCGA..T!A
 5 >C
 6 GGCT..T!G
 7 >D
 8 GGCG..T.G
 9 >E
10 ACCATATAA
```

The following example illustrates the types of change that are made to the pedigrad when the method isolate is called with the value NEW and the list of characters ['.','!']. Below, we first display the object .local of the ambient pedigrad and then compare it to the returned pedigrad.

```
>>> P = Pedigrad("align.fa",READ_DNA,NUCL_MODE,'2',"omega.yml")
>>> for i in range(len(P.base)):
...   print("segment "+str(i)+":  "+str(P.base[i].topology)+"
    "+str(P.base[i].colors)+" ---> "+str(P.local[i]))
segment 0:  [(0, 0)] ['2'] ---> ['C', 'C', 'G', 'G', 'A']
segment 1:  [(1, 1)] ['2'] ---> ['C', 'C', 'G', 'G', 'C']
segment 2:  [(2, 2)] ['2'] ---> ['C', 'G', 'C', 'C', 'C']
segment 3:  [(3, 3)] ['2'] ---> ['A', 'A', 'T', 'G', 'A']
segment 4:  [(4, 4)] ['2'] ---> ['G', '.', '.', '.', 'T']
segment 5:  [(5, 5)] ['2'] ---> ['T', '.', '.', '.', 'A']
segment 6:  [(7, 7)] ['2'] ---> ['.', '!', '!', '.', 'A']
segment 7:  [(8, 8)] ['2'] ---> ['G', 'A', 'G', 'G', 'A']
>>> Q = P.isolate(NEW,['.','!'])
>>> for i in range(len(P.base)):
...   print("segment "+str(i)+":  "+str(Q.base[i].topology)+"
    "+str(Q.base[i].colors)+" ---> "+str(Q.local[i]))
```

```
segment 0:  [(0, 0)] ['2'] ---> ['C', 'C', 'G', 'G', 'A']
segment 1:  [(1, 1)] ['2'] ---> ['C', 'C', 'G', 'G', 'C']
segment 2:  [(2, 2)] ['2'] ---> ['C', 'G', 'C', 'C', 'C']
segment 3:  [(3, 3)] ['2'] ---> ['A', 'A', 'T', 'G', 'A']
segment 4:  [(4, 4)] ['2'] ---> ['G', '>0', '>1', '>2', 'T']
segment 5:  [(5, 5)] ['2'] ---> ['T', '>0', '>1', '>2', 'A']
segment 6:  [(7, 7)] ['2'] ---> ['>0', '>1', '>2', '>3', 'A']
segment 7:  [(8, 8)] ['2'] ---> ['G', 'A', 'G', 'G', 'A']
```

Note that the call of the method `isolate` with the value `NEW` do not change the outputs of the procedure `P.partition` are not changed because the list `P.local` has not changed either. Only the list `Q.local` sees the relabeling of `P.local`, which makes the procedure `Q.partition` give different outputs.

```
>>> print(Q.partition([(4,1,1,'2'),(5,1,1,'2')],EXPR_MODE))
[0, 1, 2, 3, 4]
>>> print(P.partition([(4,1,1,'2'),(5,1,1,'2')],EXPR_MODE))
[0, 1, 1, 1, 2]
```

The example given below illustrates the types of change that are made to the pedigrad when the method `isolate` is called with the value `not(NEW)` and the list of characters `['.','!']`. We first show the object `.local` of the ambient pedigrad, which has not changed so far, and compare it to the pedigrad after the call.

```
>>> for i in range(len(P.base)):
...   print("segment "+str(i)+":  "+str(P.base[i].topology)+"
    "+str(P.base[i].colors)+" ---> "+str(P.local[i]))
segment 0:  [(0, 0)] ['2'] ---> ['C', 'C', 'G', 'G', 'A']
segment 1:  [(1, 1)] ['2'] ---> ['C', 'C', 'G', 'G', 'C']
segment 2:  [(2, 2)] ['2'] ---> ['C', 'G', 'C', 'C', 'C']
segment 3:  [(3, 3)] ['2'] ---> ['A', 'A', 'T', 'G', 'A']
segment 4:  [(4, 4)] ['2'] ---> ['G', '.', '.', '.', 'T']
segment 5:  [(5, 5)] ['2'] ---> ['T', '.', '.', '.', 'A']
segment 6:  [(7, 7)] ['2'] ---> ['.', '!', '!', '.', 'A']
segment 7:  [(8, 8)] ['2'] ---> ['G', 'A', 'G', 'G', 'A']
>>> P.isolate(not(NEW),['.','!'])
>>> for i in range(len(P.base)):
...   print("segment "+str(i)+":  "+str(P.base[i].topology)+"
    "+str(P.base[i].colors)+" ---> "+str(P.local[i]))
segment 0:  [(0, 0)] ['2'] ---> ['C', 'C', 'G', 'G', 'A']
segment 1:  [(1, 1)] ['2'] ---> ['C', 'C', 'G', 'G', 'C']
segment 2:  [(2, 2)] ['2'] ---> ['C', 'G', 'C', 'C', 'C']
segment 3:  [(3, 3)] ['2'] ---> ['A', 'A', 'T', 'G', 'A']
segment 4:  [(4, 4)] ['2'] ---> ['G', '>0', '>1', '>2', 'T']
segment 5:  [(5, 5)] ['2'] ---> ['T', '>0', '>1', '>2', 'A']
segment 6:  [(7, 7)] ['2'] ---> ['>0', '>1', '>2', '>3', 'A']
segment 7:  [(8, 8)] ['2'] ---> ['G', 'A', 'G', 'G', 'A']
```

This time, the outputs of the procedure `P.partition` are different, or, in fact, equal to those of the procedure `Q.partition`.

```
>>> print(P.partition([(4,1,1,'2'),(5,1,1,'2')],EXPR_MODE))
[0, 1, 2, 3, 4]
```

5.5.3. Agree. The method .agree takes a list of SegmentObject items and a list of non-negative integers (i.e. a partition) and returns all those segments contained in the first input for which there is a morphism of partitions from the second input to the outputs of the procedure self.partition(−,SEGM_MODE) at these segments. More mathematically, the method .agree computes the set of segments τ for which there are morphisms of partitions of the form $u \to P(\tau)$, where P denotes the right Kan extension $\mathsf{Ran}L$ of the underlying local analysis $L : B \to \mathbf{Uprt}(S)$.

For illustration, let us consider the pedigrad defined in section 5.5.1. For convenience, we recall the alignment file used thereof.

Align.fa

```
 1 >A
 2 CCCAGTTAG
 3 >B
 4 CCGATATAA
 5 >C
 6 GGCTTATAG
 7 >D
 8 GGCGTATAG
 9 >E
10 ACCATATAA
```

The following example uses the method .agree to collect all those positions of the alignment that agree with the assumption that taxon A and taxon B are close relatives.

```
>>> u = EquivalenceRelation([[0,1]],4)
>>> print(u)
[0, 0, 1, 2, 3]
>>> agreement = P.agree(P.base,u.quotient())
>>> for seg in agreement:
...   print("P("+str(seg.topology)+") = "+str(P.partition(seg,SEGM_MODE)))
P([(0, 0)]) = [0, 0, 1, 1, 2]
P([(1, 1)]) = [0, 0, 1, 1, 0]
P([(3, 3)]) = [0, 0, 1, 2, 0]
```

Similarly, we can compute the agreement lists of the pairs of taxa (B,C) and (D,E).

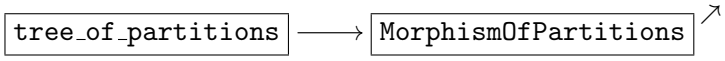
```
>>> u = EquivalenceRelation([[1,2]],4)
>>> print(u)
[1, 0, 0, 2, 3]
>>> agreement = P.agree(P.base,u.quotient())
>>> for seg in agreement:
...   print("P("+str(seg.topology)+") = "+str(P.partition(seg,SEGM_MODE)))
P([(4, 4)]) = [0, 1, 1, 1, 1]
P([(5, 5)]) = [0, 1, 1, 1, 1]
>>> u = EquivalenceRelation([[3,4]],4)
>>> print(u)
[1, 2, 3, 0, 0]
```

```
>>> agreement = P.agree(P.base,u.quotient())
>>> for seg in agreement:
...   print("P("+str(seg.topology)+") = "+str(P.partition(seg,SEGM_MODE)))
P([(2, 2)]) = [0, 1, 0, 0, 0]
P([(4, 4)]) = [0, 1, 1, 1, 1]
P([(5, 5)]) = [0, 1, 1, 1, 1]
```

Chapter 6

Presentation of the module AsciiTree.py

6.1. Description of tree_of_partitions



The function `tree_of_partitions` takes a list of partitions that can successively be related by morphisms of partitions and returns the actual lists of morphisms of partitions between these.

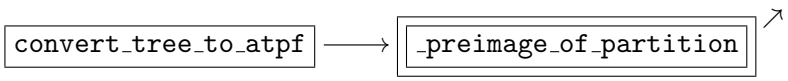
```
1 def tree_of_partitions(partitions):
2   """ the source code of this function can be found in top.py """
3   return the_tree
```

The input list should always start with the target of the first arrow, then present its source, which should also be the target of the next arrow, etc.

```
>>> l = [[0,0,0,0,0,0], [0,0,0,0,0,1], [0,0,2,2,2,1], [0,0,3,1,1,2],
    [0,1,2,3,4,5]]
>>> tree = tree_of_partitions(l)
>>> for i in range(len(tree)):
...   print(tree[len(tree)-1-i].source)
...   print("  |\n  | "+"arrow num "+str(len(tree)-i) + ":  " + str(
    tree[len(tree)-1-i].arrow) + "\n  |\n  V")
... print(tree[0].target)
[0, 1, 2, 3, 4, 5]
  |
  | arrow num 4:  [0, 0, 1, 2, 2, 3]
  |
  V
[0, 0, 1, 2, 2, 3]
  |
  | arrow num 3:  [0, 1, 1, 2]
  |
  V
```

```
[0, 0, 1, 1, 1, 2]
  |
  | arrow num 2:  [0, 0, 1]
  |
  V
[0, 0, 0, 0, 0, 1]
  |
  | arrow num 1:  [0, 0]
  |
  V
[0, 0, 0, 0, 0, 0]
```

6.2. Description of convert_tree_to_atpf

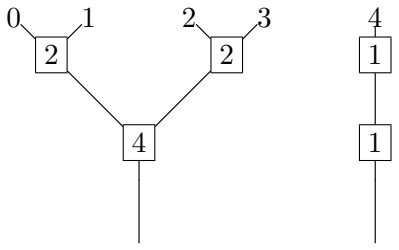


The function convert_tree_to_atpf takes a list of morphisms of partitions (as returned by the procedure tree_of_partitions) and converts it into its associated ascii tree preformat (abbrev. atpf), which is defined as any ATPF term that be can constructed from the following double grammar rules.

The grammar terms	and their associated weights
$\texttt{ATPF} := [\texttt{Tree}_1, \texttt{Tree}_2, \ldots, \texttt{Tree}_k];$	
$\texttt{Tree} := \big(\texttt{weight}(\texttt{Tree}), [\texttt{Tree}_1, \texttt{Tree}_2, \ldots, \texttt{Tree}_k]\big),$	$\texttt{weight}(\texttt{Tree}) := \sum_i \texttt{weight}(\texttt{Tree}_i);$
$\texttt{Tree} := \big(\texttt{weight}(\texttt{Tree}), [\texttt{Leaf}_1, \texttt{Leaf}_2, \ldots, \texttt{Leaf}_k]\big),$	$\texttt{weight}(\texttt{Tree}) := \sum_i \texttt{weight}(\texttt{Leaf}_i);$
$\texttt{Leaf} := (\texttt{weight}(\texttt{Leaf}), l),$ where l is a list	$\texttt{weight}(\texttt{Leaf}) := \texttt{len}(l);$

The idea behind such construction is to give access to the number of leaves contained in each fork of a tree. This type of information will later be used to display trees with ascii characters on the console. For illustration, the following line gives an example of an atpf that describes the forest displayed below it.

```
atpf = [(4, [(2, [0, 1]), (2, [2, 3])]), (1, [(1, [4])])]
```



Note that the number of levels in the trees can be linked to what one could call the *depth* of the bracketing structure in the atpf. Specifically, we define the *depth* of an afpt according to the following recursive equations, relative to the definition of the terms given above.

$$\begin{aligned}
&\texttt{depth}(\texttt{ATPF}) := \max\{\texttt{depth}(\texttt{Tree}_i) \mid i = 1, \ldots, k\};\\
&\texttt{depth}(\texttt{Tree}) := \max\{\texttt{depth}(\texttt{Tree}_i) \mid i = 1, \ldots, k\} + 1;\\
&\texttt{depth}(\texttt{Tree}) := \max\{\texttt{depth}(\texttt{Leaf}_i) \mid i = 1, \ldots, k\} + 1;\\
&\texttt{depth}(\texttt{Leaf}) := 1;
\end{aligned}$$

The procedure convert_tree_to_atpf then returns a list and an integer, where the list is the atpf of the input (i.e. of the tree) and the integer is equal to the depth of the atpf, which is equal to len(tree)+1.

```
1 def _convert_tree_to_atpf(tree):
2   """ the source code of this function can be found in ctta.py """
3   return (the_atpf,len(tree)+1)
```

Since the depth is only returned for parsing purposes (see section 6.3 and section 6.4), the main task of the procedure convert_tree_to_atpf is to compute the atpf associated with the input list of composable morphisms of partitions by considering the successive preimages of each morphisms of partitions contained in the list.

For illustration, consider the following list of partitions:

```
>>> a = [0,1,0,0,0,0]
>>> b = [0,2,0,0,0,1]
>>> c = [0,4,2,3,3,5]
```

This list induces an obvious sequence of morphisms of partitions that can be constructed by the procedure tree_of_partitions (see section 6.1).

Since the lengths of these partitions is 6, the atpf will induce a higher level bracketing of the elemens of the list [0,1,2,3,4,5]. To start with, one considers the preimage of the last partition c whose preimage is given by the following list of lists:

```
>>> _preimage_of_partition(c)
[[0], [1], [2], [3, 4], [5]]
```

To start constructing the atpf, we follow the recursive definition of the grammar of atpfs (given above) by taking the lists [0], [1, 2], [3, 4], and [5] to be the initial values of the recursion so that the first level of the atpf is given by the following list, in which the red numbers are the weight of the list terms (see grammar rule for Leaf).

```
the_atpf = [(1, [0]), (1, [1]]), (1, [2]]), (2, [3, 4]), (1, [5])]
```

For the next level, we need to compute the preimage of the arrow encoding the morphism of partitions c → b. Specifically, the arrow of the morphism c → b is as follows (when c and b are relabeled as [0,1,2,3,3,4] and [0,1,0,0,0,2], respectively).

```
>>> f = MorphismsOfPartitions(c,b)
>>> for i in range(len(f.arrow)):
...   print("f:  "+str(i)+" |--> "+str(f.arrow[i]))
f:  0 |-> 0
f:  1 |-> 1
f:  2 |-> 0
f:  3 |-> 0
f:  4 |-> 2
```

Since b has three elements in its image, the morphism f (c → b) possesses three fibers, which are given by the following list of lists:

```
>>> fiber = _preimage_of_partition(f.arrow)
>>> print(fiber)
[[0, 2, 3], [1], [4]]
```

Following the atpf grammar, we now need to replace

```
fiber[0][0] with the_atpf[0] where fiber[0][0] = 0
fiber[0][1] with the_atpf[2] where fiber[0][0] = 2
fiber[0][2] with the_atpf[3] where fiber[0][0] = 3
fiber[1][0] with the_atpf[1] where fiber[0][0] = 1
fiber[2][0] with the_atpf[4] where fiber[0][0] = 4
```

so that the fiber is turned into the following list.

```
fiber = [[(1, [0]), (1, [2]]), (2, [3, 4])], [(1, [1]])], [(1, [5])]]
```

To complete the construction of the next level of the atpf, there remains to compute the weight for each internal list. Precisely, we can see that

- the weight of `[(1, [0]), (1, [2]]), (2, [3, 4])]` is 1+1+2 = 4;
- the weight of `[(1, [1]])]` is 1;
- the weight of `[(1, [5])]]` is 1.

We then equip each list with its weight by using tuples, as shown below.

```
the_atpf = [(4,[(1, [0]), (1, [2]]), (2, [3, 4])]), (1,[(1, [1]]))], (1,[(1,
[5])])]
```

We then repeat the previous procedure with, this time, the fiber of the morphism `b` $\rightarrow$ `a` and the earlier list so that the final atpf is of the following form.

```
the_atpf = [(5, [(4, [(1, [0]), (1, [2]), (2, [3, 4])]), (1, [(1, [5])])]),
(1, [(1, [(1, [1])])])]
```

6.3. Description of `convert_atpf_to_atf`

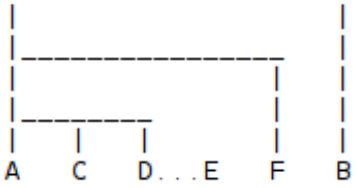
The function `convert_atpf_to_atf` takes an atpf and its depth (see section 6.2) and returns the associated ascii tree format (abbrev. atf), which is a modified version of an atpf in which one substracts all the weights by the rightmost weight of the next level, as shown by the following grammar rules

$$
\begin{aligned}
&\texttt{ATPF} := [\texttt{Tree}_1, \texttt{Tree}_2, \ldots, \texttt{Tree}_k];\\
&\texttt{Tree} := \big((\texttt{weight}(\texttt{Tree}), \texttt{weight}(\texttt{Tree}) - \texttt{weight}(\texttt{Tree}_k)), [\texttt{Tree}_1, \texttt{Tree}_2, \ldots, \texttt{Tree}_k]\big);\\
&\texttt{Tree} := \big((\texttt{weight}(\texttt{Tree}), \texttt{weight}(\texttt{Tree}) - \texttt{weight}(\texttt{Tree}_k)), [\texttt{Leaf}_1, \texttt{Leaf}_2, \ldots, \texttt{Leaf}_k]\big);\\
&\texttt{Leaf} := ((\texttt{weight}(\texttt{Leaf}), 0), l), \text{ where } l \text{ is a list;}
\end{aligned}
$$

Note that this function uses the depth of the atpf in order to differentiate between the leaves and the intermediate levels of the tree, which require two different types of treatement.

```
1 def convert_atpf_to_atf(atpf,depth):
2   """ the source code of this function can be found in cata.py """
3   return the_atf
```

The reason for this is that the procedure `print_atf` is to display ascii trees whose trunks are on the left of the screen, as show below.



Substracting the rightmost weights of the atpf from the weight placed below it, in the tree, allows `print_atf` (see section 6.4) to know when it needs to stop printing the horizontal level of the tree. Intuitively, the following pictures shows what atpf would look like without

convertion into an atf, where the red underscore symbols sticking out toward the right are symbolize the amount of weight subtracted in the atf.

```
|                   |
|_________________  |
|                 | |
|_________________| |
|   |   |       | |
A   C   D...E   F   B
```

6.4. Description of print_atf

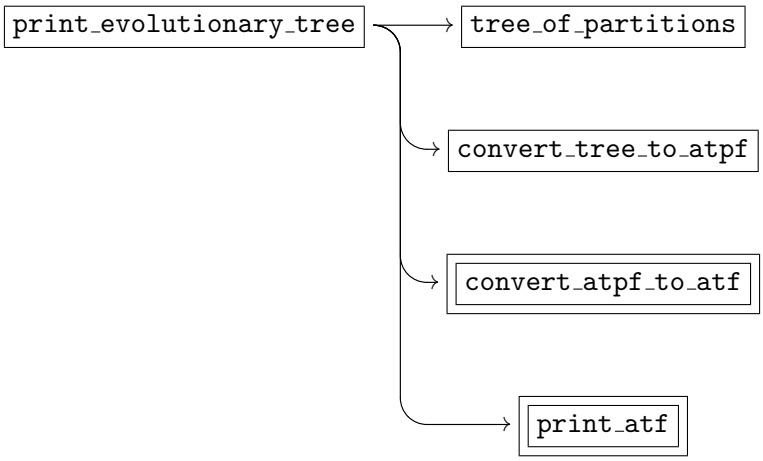
The function print_atf takes an atf and its depth and prints the ascii tree associated with the atf on the standard output.

```
1 def print_atf(atf,depth):
2   """ the source code of this function can be found in patf.py """
```

The following example shows how print_atf can be combined with the procedures convert_tree_to_atpf and convert_atpf_to_atf.

```
>>> l = [[0,1,0,0,0,0], [0,2,0,0,0,1], [0,4,2,3,3,5]]
>>> tree = tree_of_partitions(l)
>>> atpf = convert_tree_to_atpf(tree)
>>> atf = convert_atpf_to_atf(*atpf)
>>> print_atf(atf,atpf[1])
|                   |
|_________________  |
|                 | |
|_____            | |
|   |   |       | |
A   C   D...E   F   B
```

6.5. Description of print_evolutionary_tree



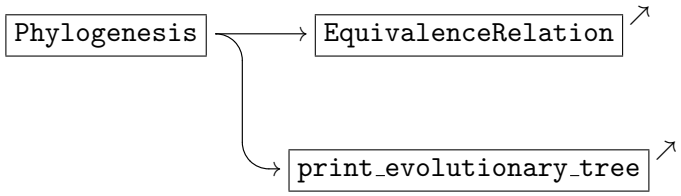
The function print_evolutionary_tree takes a list of partitions between which a sequence of composable morphisms exists and returns the tree encoded by this sequence of morphisms.

```
1 def print_evolutionary_tree(atf,depth):
2   #Returns a sequence of morphisms of partitions.
3   tree = tree_of_partitions(partitions)
4   #Returns an ascii tree pre-format and its depth.
5   atpf = convert_tree_to_atpf(tree)
6   #Returns the ascii tree format of the atpf.
7   atf = convert_atpf_to_atf(*atpf)
8   #Prints the atf on the standard output.
9   print_atf(atf,atpf[1])
```

Chapter 7

Presentation of the module `Phylogeny.py`

7.1. Description of `Phylogenesis` (class)



The class `Phylogenesis` possesses two objects, namely

- `.taxon` (non-negative integer);
- `.history` (list of lists of indices);

and three methods, namely

- `.__init__` (constructor)
- `.partitions`
- `.print_tree`

A `Phylogenesis` item is meant to be part of another structure called a `Phylogeny` (see section 7.2). The object `.taxon` stores an integer that allows us to identify the `Phylogenesis` item with respect to other `Phylogenesis` items in the `Phylogeny` structure. For its part, the object `.history` is meant to store the historical record showing the relatedness of the taxon with the other taxa contained by the `Phylogeny` structure. The list of lists contained in the object `.history` should:

- contain at least one singleton list and this list should be its first one;
- be such that every list should contain its predecessor list;

```
 1 class Phylogenesis:
 2   #The objects of the class are:
 3   #.taxon (non-negative integer);
 4   #.history (list of lists of indices);
 5   def __init__(self,history):
 6   """ the source code of this constructor can be found in cl_pgs.py """
 7   def partitions(self):
 8   """ the source code of this function can be found in cl_pgs.py """
 9     return partitions
10   def print_tree(self):
11   #Returns the evolutionary tree described by the list of lists outputted
12   #by the procedure partitions().
13     return print_evolutionary_tree(self.partitions())
```

The constructor `.__init__` takes a non-empty list of lists whose first list is a singleton and allocate

- the index contained in the first list of the inut list to the object `.taxon`,
- the list of list to the object `.history`.

Before terminating, the procedure checks whether the list of lists is made of indices and whether each list preceding another is contained in the successor list.

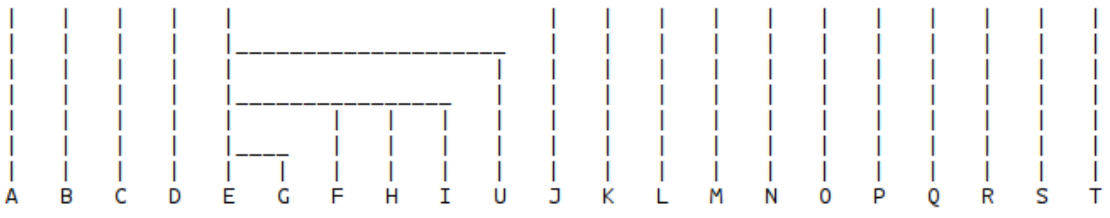
```
>>> history = [[4],[4,6],[5,4,7,8,6],[6,5,4,7,8,20]]
>>> phylogenesis = Phylogenesis(history)
>>> print(phylogenesis.taxon)
4
>>> print(phylogenesis.history[len(phylogenesis.history)-1])
[6, 5, 4, 7, 8, 20]
```

The method `.partitions()` returns the sequence of partitions induced by the list of lists contained in the object `.history` over the set of indices ranging from 0 to the maximum index of the last list of the object `.history`.

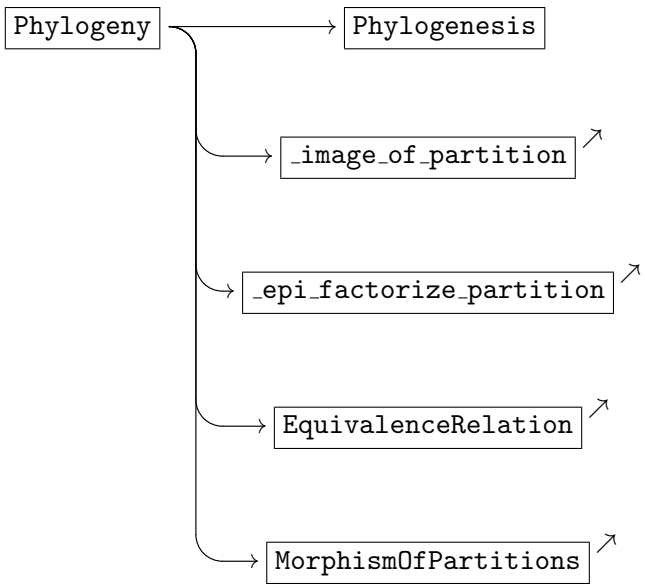
```
>>> p = phylogenesis.partitions()
>>> for i in range(len(p)):
>>> print(p[i])
[1, 2, 3, 4, 0, 0, 0, 0, 0, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]
[1, 2, 3, 4, 0, 0, 0, 0, 0, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
[1, 2, 3, 4, 0, 5, 0, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[1, 2, 3, 4, 0, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

The method `.print_tree()` returns the evolutionary tree describing the sequence of partitions returned by `.partition()`.

```
>>> phylogenesis.print_tree()
```



7.2. Description of Phylogeny (class)



The class Phylogeny possesses one object, namely

- .phylogeneses (list of Phylogenesis items)

and thirteen methods, namely

- .__init__ (constructor)
- .coalescent
- .extend
- .count_uniformity
- .boolean_partition
- .make_friends
- .set_up_friendship
- .score
- .choose
- .set_up_competition
- .compute (todo)

The object .phylogenesis is supposed to contained a list of Phylogenesis items. The taxon associated with the i-th phylogenesis should be indexed by the interger i itself and any label appearing in the Phylogenesis items of the list should have its own Phylogenesis item included in the list.

```
1 class Phylogeny:
2   #The objects of the class are:
3   #.phylogeneses (lists of Phylogenesis items);
4   def __init__(self,phylogeneses):
5   """ the source code of this constructor can be found in cl_pgs.py """
```

7.2.1. Constructor. The constructor .__init__ takes a list of lists of lists containing indices and use every internal list of the input to create a Phylogenesis item, which is stored in the object .phylogeneses.

The following lines show how a phylogeny can be created via the consutructor of the class.

```
>>> p = list()
>>> for i in range(8):
>>>   p.append([[i], [i, (i+10) % 8], [i, (i+10) % 8, (i+5*i+13) % 8]]))
>>> pgy = Phylogeny(p)
>>> for i in range(len(pgy.phylogeneses)):
>>>   print("taxon:  "+str(pgy.phylogeneses[i].taxon))
>>>   print(pgy.phylogeneses[i].history)
taxon:  0
[[0], [0, 2], [0, 2, 5]]
taxon:  1
[[1], [1, 3], [1, 3, 3]]
taxon:  2
[[2], [2, 4], [2, 4, 1]]
taxon:  3
[[3], [3, 5], [3, 5, 7]]
taxon:  4
[[4], [4, 6], [4, 6, 5]]
taxon:  5
[[5], [5, 7], [5, 7, 3]]
taxon:  6
[[6], [6, 0], [6, 0, 1]]
taxon:  7
[[7], [7, 1], [7, 1, 7]]
```

7.2.2. Elementary methods. `Phylogeny` items can be updated, modified or analyzed through the methods `.coalescent`; `.extend`; and `.make_friends`, which we present in the following sections.

```
 6   def coalescent(self):
 7   """ the source code of this constructor can be found in cl_pgs.py """
 8     return coalescent
 9   def extend(self,extension):
10   """ the source code of this constructor can be found in cl_pgs.py """
11       return False
12   def make_friends(self,taxon):
13   """ the source code of this constructor can be found in cl_pgs.py """
14     return (friends,coalescence_hypothesis)
```

7.2.3. Coalescent. The method `.coalescent()` returns the list of the first generations (*i.e.* the last lists) of the objects `.history` of each of the `Phylogenesis` item contained in the object `.phylogeneses`. The k-th list of the output is the first generation of the history of taxon k.

```
>>> coalescent = pgy.coalescent()
>>> for i in range(len(coalescent)):
>>>   print("1st generation of " + str(i) + "'s history:  " + " " +
    str(coalescent[i]))
1st generation of 0's history:  [0, 2, 5]
1st generation of 1's history:  [1, 3, 3]
1st generation of 2's history:  [2, 4, 1]
1st generation of 3's history:  [3, 5, 7]
1st generation of 4's history:  [4, 6, 5]
1st generation of 5's history:  [5, 7, 3]
1st generation of 6's history:  [6, 0, 1]
1st generation of 7's history:  [7, 1, 7]
```

7.2.4. Extend. The method `.extend` takes a list of pairs of the form `(t,l)` where `t` is the label of a taxon of the phylogeny (accessible through `pgy.phylogeneses[−].taxon`) and `l` is a list of taxa and updates the object `.phylogeneses` as follows:

- for all taxa `t` contained in the input of `.extend`:
  1) if *all* the lists `l` contains the last list of `self.phylogeneses[t].history`, if *at least one* of the lists `l` strictly contains the last list

     `self.phylogeneses[t].history`,

     and if the indices in the list `l` coupled with the taxa `t` are within the range of the list `self.phylogeneses[t].pedigrad.taxa`, then every list `l` is appended to the object `.history` of `self.phylogeneses[t]` and the value `True` is returned;
  2) if there is no strict inclusion of the last list of `self.phylogeneses[t].history` into `l`, then the object `.phylogeneses` is not modified and the value `False` is returned;
  3) otherwise, an error message is returned and the procedure exit the program;
- in any terminating case, for all other taxa `t` of the phylogeny that do not appear in the input of `.extend`, the last list of `self.phylogeneses[t].history` (*i.e.* the first generation of the history of the phylogenesis of `t`) is again repeated (*i.e.* appended again) in the list `self.phylogeneses[t].history`.

The following code lines illustrate these various cases.

```
>>> extension = [(5,[3,7,7,5]),(7,[7,1])]
>>> flag = pgy.extend(extension)
>>> print(flag)
False
>>> for i in range(len(pgy.phylogeneses)):
>>>   print(pgy.phylogeneses[i].history)
[[0], [0, 2], [0, 2, 5]]
[[1], [1, 3], [1, 3, 3]]
[[2], [2, 4], [2, 4, 1]]
[[3], [3, 5], [3, 5, 7]]
[[4], [4, 6], [4, 6, 5]]
[[5], [5, 7], [5, 7, 3]]
[[6], [6, 0], [6, 0, 1]]
[[7], [7, 1], [7, 1, 7]]
>>> extension = [(5,[3,7,7,5]),(7,[7,1]),(1,[1,3,4,5])]
>>> flag = pgy.extend(extension)
>>> print(flag)
```

```
True
>>> for i in range(len(pgy.phylogeneses)):
>>>   print(pgy.phylogeneses[i].history)
[[0], [0, 2], [0, 2, 5], [0, 2, 5]]
[[1], [1, 3], [1, 3, 3], [1, 3, 4, 5]]
[[2], [2, 4], [2, 4, 1], [2, 4, 1]]
[[3], [3, 5], [3, 5, 7], [3, 5, 7]]
[[4], [4, 6], [4, 6, 5], [4, 6, 5]]
[[5], [5, 7], [5, 7, 3], [3, 7, 5]]
[[6], [6, 0], [6, 0, 1], [6, 0, 1]]
[[7], [7, 1], [7, 1, 7], [7, 1]]
>>> extension = [(-1,["error"])]
>>> pgy.extend(extension)
>>> for i in range(len(pgy.phylogeneses)):
>>>   print(pgy.phylogeneses[i].history)
[[0], [0, 2], [0, 2, 5], [0, 2, 5]]
[[1], [1, 3], [1, 3, 3], [1, 3, 4, 5]]
[[2], [2, 4], [2, 4, 1], [2, 4, 1]]
[[3], [3, 5], [3, 5, 7], [3, 5, 7]]
[[4], [4, 6], [4, 6, 5], [4, 6, 5]]
[[5], [5, 7], [5, 7, 3], [3, 7, 5]]
[[6], [6, 0], [6, 0, 1], [6, 0, 1]]
[[7], [7, 1], [7, 1, 7], [7, 1]]
>>> extension = [(5,[1,4,80]),(7,["error"])]
>>> pgy.extend(extension)
>>> for i in range(len(pgy.phylogeneses)):
>>>   print(pgy.phylogeneses[i].history)
Error:  in Phylogeny.extend:  the extension is not compatible with the
phylogenesis of taxon 5
>>> extension = [(5,[3,7,7,5]),(7,[7,1]),(1,[1,3,4,5,11])]
>>> pgy.extend(extension)
>>> for i in range(len(pgy.phylogeneses)):
>>>   print(pgy.phylogeneses[i].history)
Error:  in Phylogeny.extend:  ancestor [1, 3, 4, 5, 11] of 1 is not covered
by the pedigrad
```

7.2.5. Make friends. The method `.make_friends` takes the label of a taxon (i.e. a non-negative integer) and returns a pair of lists `(friends,hypothesis)` where

- the list `friends` contains all those taxa that have not coalesced with the input taxon, which means that there are not in the first generation of the phylogenesis of the taxon
- the list `hypothesis` contains the (sorted) lists obtained by making the union of the first generation of the input taxon with the first generation of one of the taxon in `friends`.

For illustration, suppose that we consider the last phylogeny constructed in section 7.2.4, which means that we have the following setting.

```
>>> for i in range(len(pgy.phylogeneses)):
>>>   print(pgy.phylogeneses[i].history)
[[0], [0, 2], [0, 2, 5], [0, 2, 5]]
[[1], [1, 3], [1, 3, 3], [1, 3, 4, 5]]
[[2], [2, 4], [2, 4, 1], [2, 4, 1]]
[[3], [3, 5], [3, 5, 7], [3, 5, 7]]
[[4], [4, 6], [4, 6, 5], [4, 6, 5]]
[[5], [5, 7], [5, 7, 3], [3, 7, 5]]
[[6], [6, 0], [6, 0, 1], [6, 0, 1]]
[[7], [7, 1], [7, 1, 7], [7, 1]]
```

In this case, the output of the method `.make_friends` for taxa 1, 2 and 7 is as follows.

```
>>> friends_made = pgy.make_friends(1)
>>> friends = friends_made[0]
>>> coalescence_hypothesis = friends_made[1]
>>> for i in range(len(friends)):
>>>   print("taxa 1 and "+ str(friends[i])+" have ancestor
    "+str(coalescence_hypothesis[i]))
taxa 1 and 0 have ancestor [0, 1, 2, 3, 4, 5]
taxa 1 and 2 have ancestor [1, 2, 3, 4, 5]
taxa 1 and 6 have ancestor [0, 1, 3, 4, 5, 6]
taxa 1 and 7 have ancestor [1, 3, 4, 5, 7]
>>> friends_made = pgy.make_friends(2)
>>> friends = friends_made[0]
>>> coalescence_hypothesis = friends_made[1]
>>> for i in range(len(friends)):
>>>   print("taxa 2 and "+ str(friends[i])+" have ancestor
    "+str(coalescence_hypothesis[i]))
taxa 2 and 0 have ancestor [0, 1, 2, 4, 5]
taxa 2 and 3 have ancestor [1, 2, 3, 4, 5, 7]
taxa 2 and 5 have ancestor [1, 2, 3, 4, 5, 7]
taxa 2 and 6 have ancestor [0, 1, 2, 4, 6]
taxa 2 and 7 have ancestor [1, 2, 4, 7]
>>> friends_made = pgy.make_friends(7)
>>> friends = friends_made[0]
>>> coalescence_hypothesis = friends_made[1]
>>> for i in range(len(friends)):
>>>   print("taxa 7 and "+ str(friends[i])+" have ancestor
    "+str(coalescence_hypothesis[i]))
taxa 7 and 0 have ancestor [0, 1, 2, 5, 7]
taxa 7 and 2 have ancestor [1, 2, 4, 7]
taxa 7 and 3 have ancestor [1, 3, 5, 7]
taxa 7 and 4 have ancestor [1, 4, 5, 6, 7]
taxa 7 and 5 have ancestor [1, 3, 5, 7]
taxa 7 and 6 have ancestor [0, 1, 6, 7]
```

7.2.6. Algorithm for constructing phylogenies. The next set of methods are procedures meant to be used to implement the algorithm described in [4], which constructs a phylogeny according ot the definition given thereof.

```
15   def set_up_friendships(self):
16   """ the source code of this constructor can be found in cl_pgs.py """
17     return (friendships,coalescence_hypotheses)
15   def score(self,partitions,friendship_network):
16   """ the source code of this constructor can be found in cl_pgs.py """
17     return score_cardinality_adjusted
18   def choose(self,scores):
19   """ the source code of this constructor can be found in cl_pgs.py """
20     return result
21   def set_up_competition(self,best_fit):
22   """ the source code of this constructor can be found in cl_pgs.py """
23     return coalescence_hypothesis
```

7.2.7. Set up friendships. The method `.set_up_friendships()` returns a pair of lists `(friendships,hypotheses)` containing the lists of the two different outputs of the method `.make_friends` for every taxon of the phylogeny. More specifically,

- `friendships` is the list of lists whose t-th list contains the first output of the procedure `self.make_friends` for taxon t;
- `hypotheses` is the list of lists whose t-th list contains the second output of the procedure `self.make_friends` for taxon t;

The following code lines give a description of the output of `.set_up_friendships()` for the phylogeny used in section 7.2.5.

```
>>> friendships_made = pgy.set_up_friendships()
>>> friendships = friendships_made[0]
>>> coalescence_hypotheses = friendships_made[1]
>>> for t in range(len(friendships)):
>>>   for r in range(len(friendships[t])):
>>>     print("taxa " + str(t) + " and " + str(friendships[t][r])+" have
    ancestor "+str(coalescence_hypotheses[t][r]))
taxa 0 and 1 have ancestor [0, 1, 2, 3, 4, 5]
taxa 0 and 3 have ancestor [0, 2, 3, 5, 7]
taxa 0 and 4 have ancestor [0, 2, 4, 5, 6]
taxa 0 and 6 have ancestor [0, 1, 2, 5, 6]
taxa 0 and 7 have ancestor [0, 1, 2, 5, 7]
taxa 1 and 0 have ancestor [0, 1, 2, 3, 4, 5]
taxa 1 and 2 have ancestor [1, 2, 3, 4, 5]
taxa 1 and 6 have ancestor [0, 1, 3, 4, 5, 6]
taxa 1 and 7 have ancestor [1, 3, 4, 5, 7]
taxa 2 and 0 have ancestor [0, 1, 2, 4, 5]
taxa 2 and 3 have ancestor [1, 2, 3, 4, 5, 7]
taxa 2 and 5 have ancestor [1, 2, 3, 4, 5, 7]
taxa 2 and 6 have ancestor [0, 1, 2, 4, 6]
taxa 2 and 7 have ancestor [1, 2, 4, 7]
taxa 3 and 0 have ancestor [0, 2, 3, 5, 7]
taxa 3 and 1 have ancestor [1, 3, 4, 5, 7]
taxa 3 and 2 have ancestor [1, 2, 3, 4, 5, 7]
taxa 3 and 4 have ancestor [3, 4, 5, 6, 7]
taxa 3 and 6 have ancestor [0, 1, 3, 5, 6, 7]
```

```
taxa 4 and 0 have ancestor [0, 2, 4, 5, 6]
taxa 4 and 1 have ancestor [1, 3, 4, 5, 6]
taxa 4 and 2 have ancestor [1, 2, 4, 5, 6]
taxa 4 and 3 have ancestor [3, 4, 5, 6, 7]
taxa 4 and 7 have ancestor [1, 4, 5, 6, 7]
taxa 5 and 0 have ancestor [0, 2, 3, 5, 7]
taxa 5 and 1 have ancestor [1, 3, 4, 5, 7]
taxa 5 and 2 have ancestor [1, 2, 3, 4, 5, 7]
taxa 5 and 4 have ancestor [3, 4, 5, 6, 7]
taxa 5 and 6 have ancestor [0, 1, 3, 5, 6, 7]
taxa 6 and 2 have ancestor [0, 1, 2, 4, 6]
taxa 6 and 3 have ancestor [0, 1, 3, 5, 6, 7]
taxa 6 and 4 have ancestor [0, 1, 4, 5, 6]
taxa 6 and 5 have ancestor [0, 1, 3, 5, 6, 7]
taxa 6 and 7 have ancestor [0, 1, 6, 7]
taxa 7 and 0 have ancestor [0, 1, 2, 5, 7]
taxa 7 and 2 have ancestor [1, 2, 4, 7]
taxa 7 and 3 have ancestor [1, 3, 5, 7]
taxa 7 and 4 have ancestor [1, 4, 5, 6, 7]
taxa 7 and 5 have ancestor [1, 3, 5, 7]
taxa 7 and 6 have ancestor [0, 1, 6, 7]
```

7.2.8. Scoring system. The method `.score` takes a list of lists of non-negative integers (i.e. partitions) and a pair of lists, say `(friendships,hypotheses)`, where

- `friendships` is a list of lists;
- `hypotheses` is a list of length `len(friendships)` whose `t`-th element is a list of length `len(friendships[t])` whose elements are lists of integers ranging from `0` to

  `len(self.phylogeneses)-1`

  (preferrably sorted from smallest to greatest);

and returns a list of length `len(friendships)` whose `t`-th element is a list of triples of the form `(r,large,exact)` where

- `r` runs over the elements of `friendships[t]` ,
- `large` is the large score [4] of the hypothetical ancestor `hypotheses[t][r]` within the set of ancestors contained in `hypotheses[t]` for the list of partitions given in the input,
- `exact` is the exact score [4] of the hypothetical ancestor `hypotheses[t][r]` within the set of ancestors contained in `hypotheses[t]` for the list of partitions given in the input.

This means that `large` is the number of partitions belonging to the first input list for which there is a morphism of partition `x.quotient()` $\to$ `partitions[i]` where we take

`x = EquivalenceRelation([hypotheses[t][r]],len(self.phylogeneses)-1)`

and `exact` is the number of `SegmentObject` items that were counted in the large score for which either the equality

`hypotheses[t][r] = hypotheses[t][s]`

holds or the intersection of `hypotheses[t][r]` with `hypotheses[t][s]` is empty for every index `s` different from `r` that goes from `0` to `len(hypotheses[t])`.

The second input of the method .score can, for instance, be taken to be the output of the procedure

```
self.set_up_friendships().
```

For example, consider the following file containing a sequence alignment.

Align.fa

```
>Alice
ACGCTAGCGCGATCGATCGGATCGATCGATC
>Bob
ACGACTTAGCGGATCTGATACTCCCTCGATC
>Carles
ACGACCTAGCGGATCTTATAACTCACCGATC
>Doug
ACGCTAGCGGCTGATAACGATCGTATCGATC
>Eric
ACGCATGCGCGATCGACGGATCGTATCTATC
>Fred
ACGACCTAGCAGATTTCTAATCTCAACGATC
>Gary
ACGCGAGCATCTGAACACGATTGTAACGATC
>Haley
ACGCTACGCGACGATCGGCTTTAGATCGATC
```

Then, we can use the method .score, as shown below, for the Phylogeny item pgy that was considered in the example of section 7.2.5 and for the list of partitions contained in the object .local of the Pedigrad item deduced from the previous alignment when parsing the nucleotides.

```
>>> P = Pedigrad("Align.fa",READ_DNA,NUCL_MODE,'2',"omega.yml")
>>> l = pgy.score(P.local,pgy.set_up_friendships())
>>> for t in range(len(l)):
>>>   for (r,large,exact) in l[t]:
>>>     print("taxa " + str(t) + " and " + str(r)+" coalesce with large
    score "+str(large)+" and with exact score "+str(exact))
taxa 0 and 1 coalesce with large score 1 and with exact score 1
taxa 0 and 3 coalesce with large score 2 and with exact score 0
taxa 0 and 4 coalesce with large score 1 and with exact score 0
taxa 0 and 6 coalesce with large score 1 and with exact score 0
taxa 0 and 7 coalesce with large score 1 and with exact score 0
taxa 1 and 0 coalesce with large score 1 and with exact score 0
taxa 1 and 2 coalesce with large score 2 and with exact score 1
taxa 1 and 6 coalesce with large score 0 and with exact score 0
taxa 1 and 7 coalesce with large score 0 and with exact score 0
taxa 2 and 0 coalesce with large score 2 and with exact score 2
taxa 2 and 3 coalesce with large score 0 and with exact score 0
taxa 2 and 5 coalesce with large score 0 and with exact score 0
taxa 2 and 6 coalesce with large score 0 and with exact score 0
taxa 2 and 7 coalesce with large score 0 and with exact score 0
```

```
taxa 3 and 0 coalesce with large score 2 and with exact score 0
taxa 3 and 1 coalesce with large score 0 and with exact score 0
taxa 3 and 2 coalesce with large score 0 and with exact score 0
taxa 3 and 4 coalesce with large score 2 and with exact score 1
taxa 3 and 6 coalesce with large score 1 and with exact score 0
taxa 4 and 0 coalesce with large score 1 and with exact score 0
taxa 4 and 1 coalesce with large score 1 and with exact score 0
taxa 4 and 2 coalesce with large score 1 and with exact score 0
taxa 4 and 3 coalesce with large score 2 and with exact score 1
taxa 4 and 7 coalesce with large score 0 and with exact score 0
taxa 5 and 0 coalesce with large score 2 and with exact score 0
taxa 5 and 1 coalesce with large score 0 and with exact score 0
taxa 5 and 2 coalesce with large score 0 and with exact score 0
taxa 5 and 4 coalesce with large score 2 and with exact score 1
taxa 5 and 6 coalesce with large score 1 and with exact score 0
taxa 6 and 2 coalesce with large score 0 and with exact score 0
taxa 6 and 3 coalesce with large score 1 and with exact score 0
taxa 6 and 4 coalesce with large score 0 and with exact score 0
taxa 6 and 5 coalesce with large score 1 and with exact score 0
taxa 6 and 7 coalesce with large score 2 and with exact score 1
taxa 7 and 0 coalesce with large score 1 and with exact score 0
taxa 7 and 2 coalesce with large score 0 and with exact score 0
taxa 7 and 3 coalesce with large score 1 and with exact score 0
taxa 7 and 4 coalesce with large score 0 and with exact score 0
taxa 7 and 5 coalesce with large score 1 and with exact score 0
taxa 7 and 6 coalesce with large score 2 and with exact score 1
```

7.2.9. Choose. The method `.choose` takes a list of lists of triples `(r,l,e)` where `l` and `e` are non-negative integers and returns a list of lists whose i-th list is the list of those elements `r` of the i-th internal list of the input list for which the associated pairs `(l,e)` are equal to the greatest local maxima of the function Γ : `(e,l)` $\mapsto$ `(l,e)` ordered by the lexicographical order and relative to the pairs of the i-th internal list of the input list (see the example below and [4]).

```
>>> k1 = [[("a",1,0),("b",2,0),("c",1,1),("d",3,3)]]
>>> choose = pgy.choose(k1)
>>> print(choose)
[['d']]
>>> k2 = [[("a",1,0),("b",2,0),("c",1,1),("d",3,3),("e",3,3)]]
>>> choose = pgy.choose(k2)
>>> print(choose)
[['d', 'e']]
>>> k3 = [[("a",1,0),("b",2,0),("c",8,1),("d",3,3),("e",3,3)]]
>>> choose = pgy.choose(k3)
>>> print(choose)
[['c']]
>>> k4 = [[("a",1,0),("b",9,0),("c",8,1),("d",3,3),("e",3,3)]]
>>> choose = pgy.choose(k4)
>>> print(choose)
[['b']]
```

```
>>> k = k1 + k2 + k3 + k4
>>> choose = pgy.choose(k)
>>> print(choose)
[['d'], ['d', 'e'], ['c'], ['b']]
```

The following example shows the type of output that one gets if one gives the list `l` of the example of section 7.2.8 to the method `.choose`.

```
>>> choose = pgy.choose(l)
>>> for t in range(len(choose)):
>>>   print("Closest relatives of taxon " + str(t) + " are " +
    str(choose[t]))
Closest relatives of taxon 0 are [3]
Closest relatives of taxon 1 are [2]
Closest relatives of taxon 2 are [0]
Closest relatives of taxon 3 are [4]
Closest relatives of taxon 4 are [3]
Closest relatives of taxon 5 are [4]
Closest relatives of taxon 6 are [7]
Closest relatives of taxon 7 are [6]
```

7.2.10. Set up competition. The method `.set_up_competition` takes a list of lists of integers whose length must be equal to the length of `self.phylogeneses` (i.e. the number of taxa of the phylogeny) and returns a list of lists of integers whose length is also equal to the length of `self.phylogeneses` and whose t-th internal list is the union of the t-th list of `self.coalescent()` with the `r`-th list of `self.coalescent()` for every element `r` in the t-th internal list of the input list.

The following example shows the ouput of the method `.set_up_competition` when it is given the list `choose` constructed in the last example of section 7.2.9.

```
>>> competition = pgy.set_up_competition(choose)
>>> for t in range(len(competition)):
>>>   print("Next competing generation representing taxon " + str(t) + " is
    " + str(competition[t]))
Next competing generation representing taxon 0 is [0, 2, 3, 5, 7]
Next competing generation representing taxon 1 is [1, 2, 3, 4, 5]
Next competing generation representing taxon 2 is [0, 1, 2, 4, 5]
Next competing generation representing taxon 3 is [3, 4, 5, 6, 7]
Next competing generation representing taxon 4 is [3, 4, 5, 6, 7]
Next competing generation representing taxon 5 is [3, 4, 5, 6, 7]
Next competing generation representing taxon 6 is [0, 1, 6, 7]
Next competing generation representing taxon 7 is [0, 1, 6, 7]
```

Bibliography

[1] G. F. Estabrook, F. R. McMorris, (1980), *When is one estimate of evolutionary relationships a refinement of another?*, Journal of Mathematical Biology, Volume 10, Issue 4, pp 367–373.

[2] J. Felsenstein, (2003), *Inferring Phylogenies*, Sinauer Associates, Sunderland, Massachusetts.

[3] W. J. Le Quesne, (1 June 1969), *A method of selection of characters in numerical taxonomy*, Systematic Zoology, Volume 18, Issue 2, pp 201–205.

[4] R. Tuyéras, *Categorical approach to tree inference*, arXiv:????

[5] R. Tuyéras, *Category Theory for Genetics*, arXiv:1708.05255

[6] E.O. Wilson, (1965), *A consistency test for phylogenies based on contemporaneous species*, Systematic Zoology, Volume: 14, pp 214–220.