mtg Documentation

Release 2.1.2

Christophe Pradal

Apr 20, 2022

Contents

Contents:

CHAPTER 1

Install

Use conda to install openalea.mtg:

```
conda install openalea.mtg -c openalea
```

CHAPTER 2

Use

Simple usage:

```
from openalea.mtg import *
```

CHAPTER 3

MTG User Guide

Summary

Version 2.1.2

Release 2.1.2

Date Apr 20, 2022

Provides **MTG** or *Multiscale Tree Graph* data structure.

In order to quickly learn how to read a MTG file and plot it with PlantGL, jump to the the *Quick Start to manipulate MTGs*. If you are in a hurry and want to parse the MTG to retrieve information about it, look at the *The openalea.mtg.aml module: Long Tour* that fully describes the `openalea.mtg.aml` module.

Then, we advice you to look at the section *MTG file* to understand what is a MTG file through a detailled description of the format and a few examples (note that the section *File syntax* gives a full description of the format). The section *Illustration: exploring an apple tree orchard* explains through a full example what can be done with the MTG data in point of view of statistical analysis.

Finally, once the MTG format is understood, you may want to create your own MTG file from scratch as described in Section *Tutorial: Create MTG file from scratch*.

Note: The full guide reference is also available *Reference*.

3.1 Quick Start to manipulate MTGs

3.1.1 Reading an MTG file and activate it

A plant architecture described in a coding file can be loaded in `openalea.mtg.aml` as follows:

```
>>> from openalea.mtg.aml import MTG
>>> g1 = MTG('user/agraf.mtg')          # some errors may occur while loading the MTG
ERROR: Missing component for vertex 2532
```

Note: In order to reproduce the example, download `agraf MTG file` and the `agraf DRF file`. Other files that may be required are also available in the same directory (**smb* files) but are not compulsary.

The MTG function attempts to read a valid MTG description and parses the coding file. If errors are detected during the parsing, they are displayed on the screen and the parsing fails. In this case, no MTG is built and the user should make corrections to the coding file. If the parsing succeeds, this function creates an internal representation of the plant (or a set of plants) encoded as a MTG. In this example, the MTG object is stored in variable *g1* for further use. Note that a MTG should always be stored in a variable otherwise it is destroyed immediately after its building. The last built MTG is considered as the "active" MTG. It is used as an implicit argument by all the functions of the MTG module.

It is possible to change the active MTG using `Activate()`

```
g1 = MTG("filename1") # g1 is the current MTG
g2 = MTG("filename2") # g2 becomes the current MTG
Activate(g1)          # g1 is now again the current MTG
```

Warning: the notion of activation is very important. Each call to a function in the package MTG will look at the active MTG.

3.1.2 Plotting

Warning: PlantFrame is still in development and not all MTG files can be plotted with the current code, especially the files that have no information about positions

The following examples shows how to plot the contents of a MTG given that a dressing data file (DRF) is available. See the *File syntax* section for more information about the MTG and DRF syntax. Note that the following code should be simplified in the future.

```
from openalea.mtg.aml import MTG
from openalea.mtg.dresser import dressing_data_from_file
from openalea.mtg.plantframe import PlantFrame, compute_axes, build_scene
g = MTG('agraf.mtg')
dressing_data = dressing_data_from_file('agraf.drf')
topdia = lambda x:  g.property('TopDia').get(x)
pf = PlantFrame(g, TopDiameter=topdia,    DressingData = dressing_data)
axes = compute_axes(g, 3, pf.points, pf.origin)
diameters = pf.algo_diameter()
scene = build_scene(pf.g, pf.origin, axes, pf.points, diameters, 10000)
from  vplants.plantgl.all import Viewer
Viewer.display(scene)
```

3.1.3 Functions related to MTGs

There exists a comprehensive set of functions related to MTGs. These functions may be directly used on the active MTG or they may be combined with each other in order to define new functions on MTGs. Here are some of them.

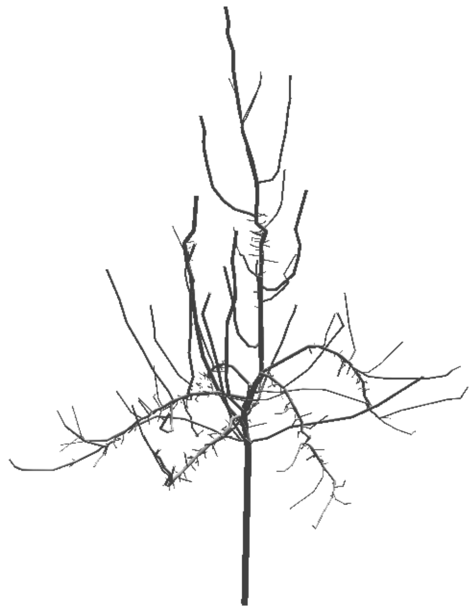


Fig. 1: **Figure 3.5** An apple tree plotted with the python script shown above

Full details may be found elsewhere either in the tutorials (e.g., *The openalea.mtg.aml module: Long Tour*) or in the *Reference* section.

- **MTG constructor**. We've already seen how to read a MTG file by using `MTG()`, which takes one mandatory argument, namely the MTG's filename.
- **Extraction of vertex sets: e.g. VtxList().** Different types of lists of vertices can be extracted from a MTG through the function `VtxList()`. Notably, the set of functions at a given scale is obtained with the optional argument **Scale**:

```
1 from openalea.mtg.aml import VtxList
2 VtxList()
3 vtx1 = VtxList(Scale=1) # vtx 1 returns a list e.g., [1]
4 vtx2 = VtxList(Scale=2)
5 vtx3 = VtxList(Scale=3)
```

  On line 2, we extract the vertices that have scale set to 1. The returned list contains only 1 element that have the index 1. Conversely, we could use the `Scale()` function to figure out what is the Scale of the vertex that have the index 1:

```
>>> from openalea.mtg.aml import Scale
>>> Scale(1)
1
```

- **Functions returning vertex attributes: e.g. Class(vtx), Index(vtx), Feature(vtx, feature_name).** The different attributes attached to a given vertex can be retrieved by these functions. The class and the index of a vertex are respectively returned by functions `Class()` and `Index()`. The value of any other attribute may be obtained by specifying its name:

```
>>> from openalea.mtg.aml import Feature, Class, Index
>>> vtxList = VtxList(Scale=2)  # get a list of vertices according to a scale
>>> v1 = vtxList[0]             # look at the first vertex
>>> # Feature(vertex_id, name)
```

(continues on next page)

(continued from previous page)

```
>>> Feature(v1, "XX")
0.0
>>> Class(v1)
'U'
>>> Index(v1)
94
```

Returns the attribute "XX" (if any) of a vertex v1. These functions return scalar (INTEGER, STRING, REAL), i.e. elementary types different from VTX.

- **Functions for moving in MTGs: e.g. Father(vtx), Complex(vtx), Successor(vtx), Predecessor(vtx).** Some functions take a VTX as an argument and return a VTX. These functions allow topological moves in the MTG, i.e. they allow to select new vertices with topological reference to given vertices. See *Father()*, *Predecessor()* , *Successor()*, and *Complex()*

  ```
  >>> from openalea.mtg.aml import Father, Successor, Predecessor
  >>> Father(v1)
  >>> Predecessor(v1)
  ```

  Note: The predecessor is a special case of Father; predecessor function is equivalent to Father(v, EdgeType-> '<'). It thus returns the father (at the same scale) of the argument

- **Functions for creating collections of vertices: e.g. Sons(vtx), Components(vtx), Axix(vtx).** These functions return sets of vertices associated with a certain vertex. Components() returns all the vertices that compose at the scale immediately superior a given vertex. Axis() returns the ordered set of vertices which compose the axis which the argument belongs to.

- **Functions for creating graphical representations of MTGs: PlantFrame(), Plot(), DressingData** Plant-Frame() enables the user to compute 3D-geometrical representations of MTGs.

The above functions can be combined together using the Python language to extract from plant databases various types of information.

documentation status:

Documentation adapted from the AMAPmod user manual version 1.8 Dec 2009.

Documentation to be revised

3.2 The `openalea.mtg.aml` module: Long Tour

3.2.1 Reading the file

This page illustrates the usage of all the functionalities available in *openalea.mtg.aml* module. All the examples uses the MTG file `code_file2.mtg`. If you are interested in the syntax, we stronly recommend you to look at Section *MTG file*.

First, let us read the MTG file with the function `MTG()`. Note that only one MTG object can be manipulated at a time. This MTG object is the **active** MTG.

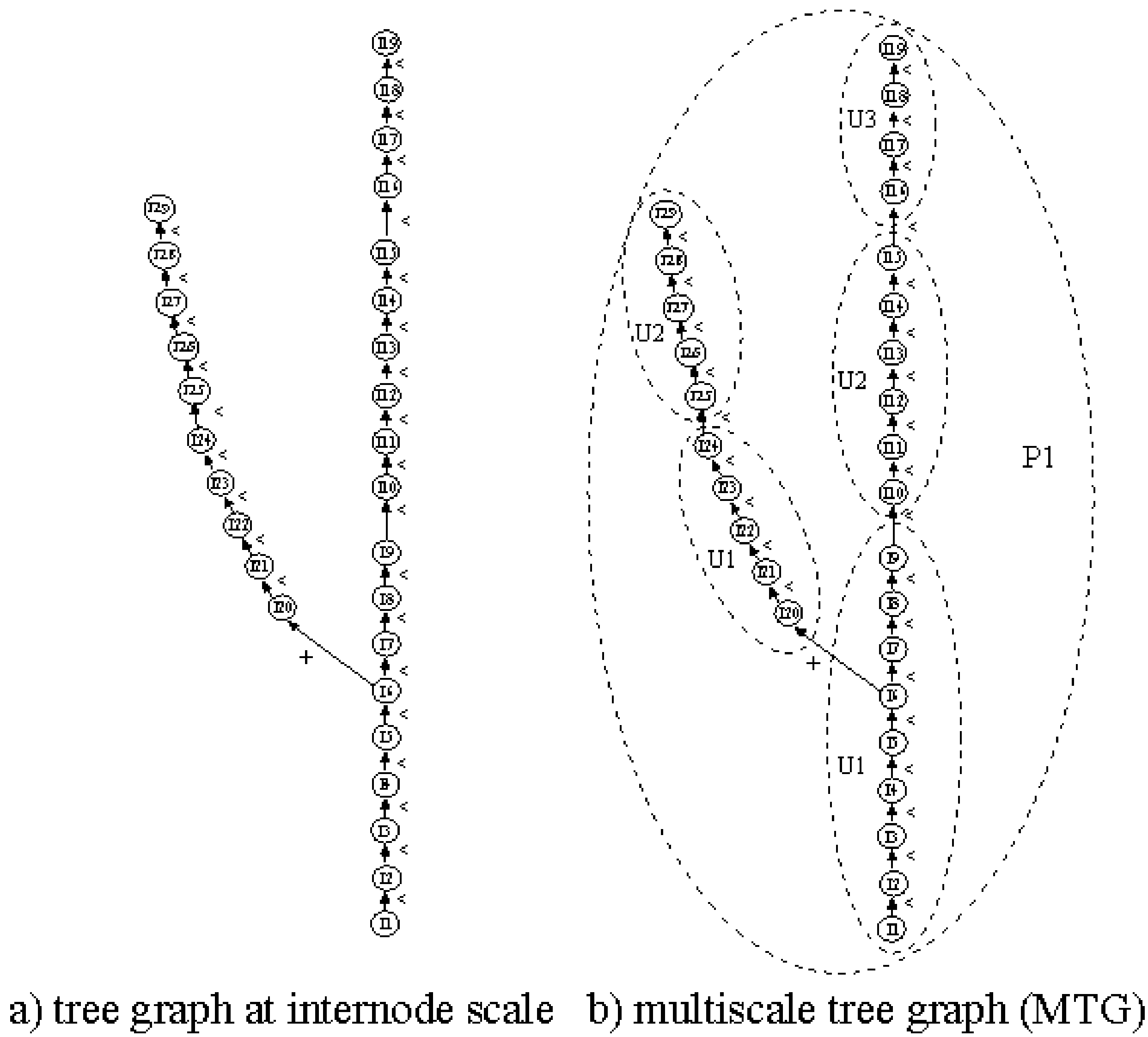


Fig. 2: **Figure 1:** Graphical representation of the MTG file code_file2.mtg used as an input file to all examples contained in this page

```
>>> from openalea.mtg.aml import *
>>> g = MTG('user/code_file2.mtg')
>>> Active() == g
True
```

The `Active()` function checks that *g* is currently the active MTG.

If a new MTG file is read, it becomes the new active MTG object. However, the function `Activate()` can be use to switch between MTG objects as follows:

```
>>> h = MTG('user/agraf.mtg')
>>> Active() == h
True
>>> Activate(g)
```

```
>>> MTGRoot()
0
```

3.2.2 Feature functions

Order, Rank and Height

`Order()` (`AlgOrder()`) look at the number of + sign that need to be crossed before reaching the vertex considered

```
>>> Order(3)
0
>>> Order(14)
1
>>> AlgOrder(3,14)
1
```

`Height()` (`AlgHeight()`) look at the number of components between the root of the vertex's branch and the vertex's position.

```
>>> Height(3)
0
>>> Height(14)
10
>>> AlgHeight(3, 14)
10
```

`Rank()` (`AlgRank()`) returns the number < sign that need to be crosssed before reaching the vertex considered.

```
>>> Rank(3)
0
>>> Rank(14)
4
>>> AlgRank(3, 14)
5
```

`Class()`, `Index()`, `Label()`, `Feature()`

`Class()` gives the type of vertex usually defined by a letter

```
>>> Class(3)
'I'
```

and `Index()` gives the other part of the label

```
>>> Index(3)
1
```

When speaking about multiscale tree graph, we also want to access the `Scale()`:

```
>>> Scale(3)
3
```

A new function called `Label()` combines the *Class* and *Index*:

```
>>> Label(3)
'I1'
```

Finally, `Feature()` returns value of a given feature coded in the MTG file.

```
>>> Feature(2, "Len")
10.0
```

ClassScale(), EdgeType(), Defined()

`ClassScale()` returns the Scale at which appears a given class of vertex:

```
>>> ClassScale('U')
3
```

`EdgeType()` returns the type of connection between two vertices (e.g., +, <)

```
>>> i=8; Class(i), Index(i)
('I', 6)
>>> i=9; Class(i), Index(i)
('U', 1)
>>> EdgeType(8,9)
'+'
```

`Defined()` tests whether a vertex's id is present in the active MTG

```
>>> Defined(1)
True
>>> Defined(100000)
False
```

3.2.3 Date functions

The following function requires MTG files to contain Date information.

Todo: not yet implemented

Function	
DateSample(e1)	
FirstDefinedFeature(e1, e2)	
LastDefinedFeature(e1, e2)	
NextDate(e1)	
PreviousDate(e1)	

3.2.4 Functions for moving in MTGs

Trunk()

`Trunk()` returns the list of vertices constituting the bearing botanical axis of a branching system

```
>>> Trunk(2)     # vertex 2  is U1 therefore the Trunk should return index related to␣
↪U1, U2, U3
[2, 24, 31]
>>> Class(24), Index(24)
('U', 2)


>>> Trunk(3)     # vertex 3 is an internode, so we get all internode of the axis␣
↪containing vertex 3
[3, 4, 5, 6, 7, 8, 21, 22, 23, 25, 26, 27, 28, 29, 30, 32, 33, 34, 35]
>>> Class(35), Index(35)
('I', 19)
```

Father()

Topological father of a given vertex.

```
>>> Label(8)
'I6'
>>> Father(8)
7
>>> Label(9)      # Let us look at vertex 9 (with the U1 label)
'U1'
>>> Father(9)               # and look for its father's index
2
>>> Label(2)      # and its father's label that appear to also be equal to 1
'U1'
```

Axis()

`Axis()` returns the vertices of the axis to which belongs a given vertex.

```
>>> [Label(x) for x in Axis(9)]
['U1', 'U2']
```

The scale may be specified

```
>>> [Label(x) for x in Axis(9, Scale=3)]
['I20', 'I21', 'I22', 'I23', 'I24', 'I25', 'I26', 'I27', 'I28', 'I29']
```

`Ancestors()`

`Ancestors()` returns a list of ancestors of a given vertex

```
>>> Ancestors(20)    # of I29
[20, 19, 18, 17, 16, 14, 13, 12, 11, 10, 8, 7, 6, 5, 4, 3]
>>> [Class(x)+str(Index(x)) for x in Ancestors(20)]
['I29', 'I28', 'I27', 'I26', 'I25', 'I24', 'I23', 'I22', 'I21', 'I20', 'I6', 'I5', 'I4
↪', 'I3', 'I2', 'I1']
```

`Path()`

The `Path()` returns a list of vertices defining the path between two vertices

```
>>> [Class(x)+str(Index(x)) for x in Path(8, 20)]
['I20', 'I21', 'I22', 'I23', 'I24', 'I25', 'I26', 'I27', 'I28', 'I29']
```

`Sons()`

In order to illustrate the `Sons()` function, let us consider the vertex 8

```
>>> Class(8), Index(8)
('I', 6)
>>> [Class(x)+str(Index(x)) for x in Sons(8)]
['I20', 'I7']
```

`Descendants()` and `Ancestors()`

`Descendants()` an array with all the vertices, at the same scale as v, that belong to the branching system starting at v:

```
>>> [Class(x)+str(Index(x)) for x in Descendants(8)]
```

`Ancestors()` contains the vertices on the path from v back to the root (in this order) and finishes by the tree root.:

```
>>> [Class(x)+str(Index(x)) for x in Ancestors(8)]
```

`Predecessor()` and `Successor()`

`Predecessor()` returns the Father of a vertex connected to it by a ‘<’ edge, and is therefore equivalent to:

```
Father(v, EdgeType-> ‘<’).
```

Similarly, `Successor()` is equivalent to

```
Sons(v, EdgeType=’<’)[0]
```

Root()

`Root()` returns root of the branching systenme containing a given vertex and therefore is equivalent to:

```
Ancestors(v, EdgeType='<')[-1]

>>> [Class(x)+str(Index(x)) for x in Ancestors(8)]
['I6', 'I5', 'I4', 'I3', 'I2', 'I1']
>>> Root(8)
3
>>> Class(3)+str(Index(3))
'I1'
```

Todo: Complex returns Scale(v)-1 why what is it for?

```
>>> Complex(8)
2
```

Components()

Returns a list of vertices that are included in the upper scale of the vertex's id considered. The array is empty if the vertex has no components.

```
>>> Components(1, Scale=2)
[2, 9, 15, 24, 31]
>>> Components(1, Scale=3)
[3, 4, 5, 6, 7, 8, 21, 22, 23, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 25, 26, 27, 28,
↪ 29, 30, 32, 33, 34, 35]
```

ComponentRoots()

Todo: to be done. find example

Location()

Vertex defining the father of a vertex with maximum scale.

```
>>> Label(9)             # starting from a Component U1 at vertex's id 9
'U1'
>>> Father(9)            # what is its Father ?
2
>>> Label(Father(9))     # answer: another U1 of vertex's id 2
'U1'
>>> Location(9)          # what is the location of vertex 9
8
>>> Label(Location(9))   # the internode I6
'I6'
```

Extremities()

```
>>> Label(8)
'I6'
>>> Label(Extremities(8))
['I29', 'I19']
```

3.2.5 Geometric interpretation

Most of the following functions are not yet implemented. See *Quick Start to manipulate MTGs* to see the usage of `PlantFrame()` with dressing data.

You may also use the former AML code using *openalea.aml* package

PlantFrame() and Plot()

One can use openalea.aml for now:

```
>>> import openalea.aml as aml
>>> aml.MTG('code_file2.txt')
>>> pf = aml.PlantFrame(2)
>>> aml.Plot()
```

Shows the MTG file at scale 2. This is possible because Diameter and Lenmgth features are provided at that scale.

`DressingData()`	
`Plot()`	
`TopCoord()`	
RelTopCoord(e1, e2)	
BottomCoord(e1, e2)	
RelBottomCoord(e1, e2)	
Coord(e1, e2)	
BottomDiameter(e1,e2)	
TopDiameter(e1,e2)	
Alpha(e1,e2)	
Beta(e1,e2)	
Length(e1,e2)	
VirtualPattern(e1)	
PDir(e1,e2)	
SDir(e1,e2)	

3.2.6 Comparison Functions

Todo: not yet implemented

TreeMatching(e1) MatchingExtract(e1)

documentation status:

Documentation adapted from the AMAPmod user manual version 1.8 Dec 2009.

Documentation to be revised

Contents

3.3 MTG file

`openalea.mtg` provides a Multiscale Tree Graph data structure (MTG) that is compatible with the standard MTG format that was defined in the AMAPmod software. For compatibility reasons, the same interfaces have been implemented in this package. However, this is a completly new implementation written in Python that will evolve by adding new functionalities and algorthims.

3.3.1 MTG: a Plant Architecture Databases

Overview

In OpenAlea/VPlants projects, plants are formally represented by multiscale tree graphs (MTGs)[20]. A MTG consists of a set of layered tree graphs, representing plant topology at different scales (internodes, growth units, axes, etc.).

To build up MTGs from plants, plants are first broken down into plant components, organised in different scales (*Figure3.2.a* and *Figure3.2.b*). Components are given labels that specify their types (*Figure3.2.b*, *U* = growth unit, *F* = flowering site, *S* = short shoot, *I* = internode). These labels

[20] Godin, C. et Caraglio, Y., 1998. A multiscale model of plant topological structures. Journal of Theoretical Biology, 191: 1-46.

are then used to encode the plant architecture into a textual form. The resulting coding file (*Figure3.2.c*) can then be analysed by `openalea.mtg` tools to build the corresponding MTG (*Figure3.2.d*).

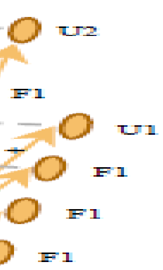
Fig. 3: **Figure 3.2,a** Starting from real plants, measurements are made.

Fig. 4: **Figure 3.2.b** Plants components are identified and labelled (e.g, U for growth unit)

/P1			
^/A91/U1			
	/I12+A91/U1		
	^<U2		
		/I8+A92/F1	
		/I9+A92/F1	
		/I10+A92/F1	
		/I11+A92/U1	
	^<A92/F1		
		/I1+U2	
	/I13+A91/U1		
		/I10+A92/F1	
		/I15+A92/F1	
			/I1+U2
	^<U2		
		/I6+A92/S1	
		/I7+A92/S1	
		/I8+A92/F1	
		/I9+A92/F1	
		/I10+A92/F1	
	^<A92/F1		
		/I1+U2	
^<U2			
^<A92/F1			
	/I1+U2		
	/I2+U2		

Fig. 5: **Figure 3.2.c** The plant components and their attributes are encoded in a MTG file

Explanations



t representation

In an MTG, the organisation of plant components at a given scale of detail is represented by a tree graph, where each component is represented by a vertex in the graph and edges represent the physical connections between them. At any given scale, the plant components are linked by two types of relation, corresponding to the two basic mechanisms of plant growth, namely the apical growth and the branching processes. Apical growth is responsible for the creation of axes, by producing new components (corresponding to new portions of stem and leaves) on top of previous components. The connection between two components resulting from the apical growth is a **precedes** relation and is denoted by a **<** character.

On the other hand, the branching process is responsible for the creation of axillary buds (these buds can then create axillary axes with their own apical growth). The connection between two components resulting from the branching process is a **bears** relation and is denoted by a **+** character. A MTG integrates – within a unique model – the different tree graph representations that correspond to the different scales at which the plant is described.

Various types of attribute can be attached to the plant components represented in the MTG, at any scale. Attributes may be geometrical (e.g., diameter of a stem, surface area of a leaf or 3D positioning of a plant component) or morphological (e.g., number of flowers, nature of the associated leaf, type of axillary production - latent bud, short shoot or long shoot -).

MTGs can be constructed from field observations using textual encoding of the plant architecture as described in[22] (see *Figure3.2.a*). Alternatively, code files representing plant architectures can also be constructed from simulation programs that generate artificial plants, or directly from any Python program, as we will illustrate it in the *Tutorial: Create MTG file from scratch*.

Todo: fix the internal link reference

The code files usually have a spreadsheet format and contain the description

[22] Godin, C., Costes, E. et Caraglio, Y., 1997. Exploring plant topology structure with the AMAPmod software : an outline. Silva Fennica, 31(3): 355-366.

of plant topology in the first few columns and the description of attributes attached to plant components on subsequent columns.

3.3.2 Coding Individuals

Different strategies have been proposed for recording topological structures of real plants, e.g.[43],[32] for plant represented at a single scale and[21],[25], for multiscale representations. In OpenAlea/Vplants, plant topological structures are abstracted as multiscale tree graphs. Describing a plant topology thus consists of describing the multiscale tree graph corresponding to this plant. The description of a given plant can be specified using a **coding language**. This language consists of a naming strategy for the vertices and the edges of multiscale graphs. A graph description consists of enumerating the vertices consecutively using their names. The name of a vertex is constructed in such a way that

[43] Room, P. et Hanan, J., 1996. Virtual plants: new perspectives for ecologists, pathologists and agricultural scientists. Trends in Plant Science Update, 1(1): 33-38.

[32] Hanan, J. et Room, P., 1997. Practical aspects of plant research. In: Plants to ecosystems - Advances in Computational Life Sciences 2nd International Symposium on Computer Challenges in Life Science. M.T. Michalewicz (Ed.). CISRO Australia, Melbourne, Australie, pp. 28-43.

[21] Godin, C. et Costes, E., 1996. How to get representations of real plants in computers for exploring their botanical organisation. In: International Symposium on Modelling in Fruit Trees and Orchard Management, Avignon (FRA) 4-8/09/95, ISHS. Acta Horticulturae, Vol. 416, pp. 45-52.

[25] Godin, C., Guédon, Y., Costes, E. et Caraglio, Y., 1997. Measuring and analyzing plants with the AMAPmod software. In: Plants to ecosystems - Advances in Computational Life Sciences 2nd International Symposium on Computer Challenges in Life Science. M.T. Michalewicz (Ed.). CISRO Australia, Melbourne, Australie, pp. 53-84.

it clearly defines the topological location of a given vertex in the overall multiscale graph. The vertices and their features are described using this formal language in a so called **code file**. Let us illustrate the general principle of this coding language by the topological structure of the plant depicted in *Figure3.3*.

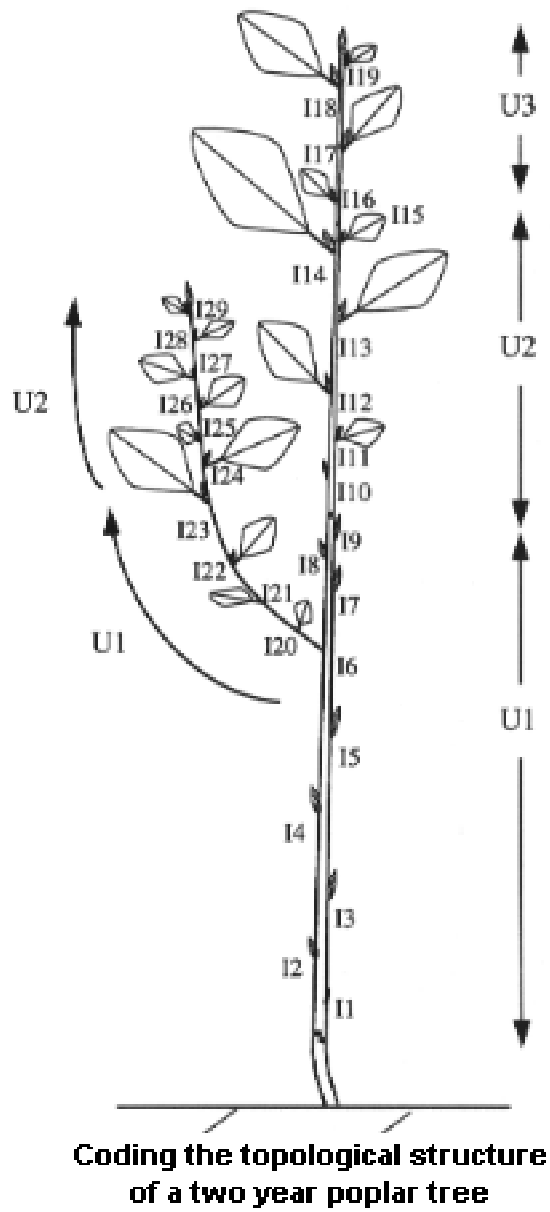


Fig. 7: **Figure 3.3** Coding the topological structure of a two year old poplar tree

Each vertex is associated with

a **label** consisting of a letter, called its **class**, and an integer, called its **index**. The class of a vertex often refers to the nature of the corresponding botanical entity, *e.g.* *I* for internode, *U* for growth unit, *B* for branching system, etc. The index of a vertex is an integer which enables the user to locally identify a vertex among its immediate neighbors. Apart from this purely structural role, indexes may be used to convey additional meaning: they can be used for instance to encode the year of growth of an entity, its rank in an axis, etc.

At a given scale, plants are inspected by working upwards from the base of the trunk and symbols representing each vertex and its relationship to its father are either written down or keyed directly into a laptop computer.

The coded string starts with the single

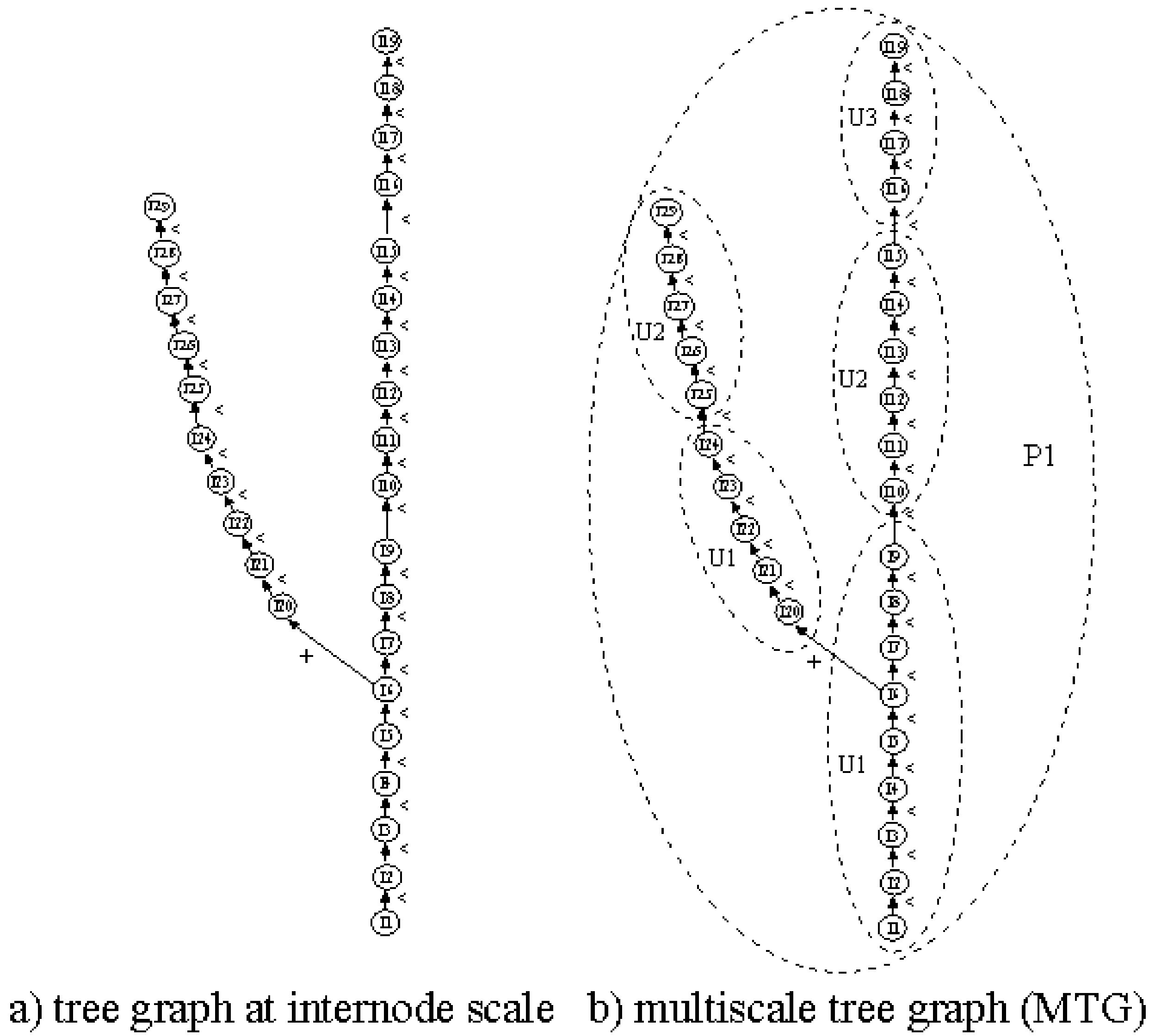


Fig. 8: **Figure 3.4**

symbol **/**. Coding a single axis (e.g. the series of internodes of the trunk depicted in *Figure3.4* a) would then yield the string:

```
/I1<I2<I3<I4<I5<I6<I7<I8<I9<I10<I11<I12<I13<I14<I15<I16<I17<I18<I19
```

For a branching structure (*Figure3.4* a), encoding a tree-like structure in a linear sequence of symbols leads us to introduce a special notation, frequently used in computer science to encode tree-like structures as strings (e.g.[39]). A square bracket is opened each time a bifurcation point is encountered during the visit (i.e. for vertices having more than one son). A square bracket is closed each time a terminal vertex has just been visited (i.e. a vertex with no son) and before backtracking to the last bifurcation point. In the above example, entity *I6* is a bifurcation point since the description process can either continue by visiting entity *I7* or *I20*. In this case, the bifurcation point *I6* is first stored in a bifurcation point stack (which is initially empty). Secondly, an opened square bracket is inserted in the output string and thirdly, the visiting process resumes at one of the two possible continuations, for example *I20*, leading to the following code :

```
/I1<I2<I3<I4<I5<I6[+I20
```

The entire branch *I20* to *I28* is then encoded like entities *I1* to *I6*. Entity *I29* has no son, and thus is a terminal entity. This results in inserting a closed square bracket in the string :

```
/I1<I2<I3<I4<I5<I6[+I20<I21<I22<I23<I124<I25
```

(continues on next page)

[39] Prusinkiewicz, P. et Lindenmayer, A., 1990. The algorithmic beauty of plants. Springer Verlag.

(continued from previous page)

The last bifurcation point can then be popped out of the bifurcation point stack and the visiting process can resume on the next possible continuation of *I6*, i.e. *I7*, leading eventually to the final output code string:

```
/
↪I1
↪<I2
↪<I3
↪<I4
↪<I5
↪<I6[+I20
↪<I21
↪<I22
↪<I23
↪<I124
↪<I25
↪<I26
↪<I27
↪<I28
↪<I29]
↪<I7
↪<I8
↪<I9
↪<I10
↪<I11
↪<I12
↪<I13
↪<I14
↪<I15<I16<I17<I18<I19]
```

Let us now extend this coding strategy to multi-scale structures. Consider a plant described at three different scales, for example the scale of internodes, the scale of growth units and the scale of plants (*Figure3.4* b). The depth first procedure explained above is generalized to multiscale structures in the following way. The multiscale coding strategy consists basically of describing the plant structure at the highest scale in a depth first order. However, during this process, each time a boundary of a macroscopic entity is crossed when passing from entity a to entity b, the corresponding macroentity label, suffixed by a '/', must be inserted into the code string just before the label of b and after the edge type of (a,b). If more than one macroscopic boundary is crossed at a time, corresponding labels suffixed by '/' must be inserted into the code string at the same location, labels of the most macroscopic entities first. In the multiscale graph of *Figure3.4* b for example, the depth first visit is carried out at the internode level (highest scale). The visit starts by entering in vertex I1 at the scale of internodes. However, to reach this entity from the outside, we cross boundaries of P1 and U1, in this order. Then the depth first visit starts by creating the code string :

```
/P1/U1/I1
```

Then, the coding procedes through vertices *I1* to *I6*, with no new macroscopic boundary encountered. *I6* is a bifurcation point and as explained above, this vertex is stored in the bifurcation point stack, a '[' is inserted in the code string and the depth first process continues on the son of *I6* whose label is *I20*. Since to reach *I20* from *I6* the macroscopic boundary of the first growth unit of the branch is crossed, on *I20* the generated code string is

```
/P1/U1/I1<I2<I3<I4<I5<I6[+U1/I20
```

Similarly on the new branch, coding continues and crosses a growth unit boundary between internodes I24 and I25 :

```
/P1/U1/I1<I2<I3<I4<I5<I6[+U1/I20<I21<I22
```

(continues on next page)

(continued from previous page)

Once the end of the branch is reached at entity I29, a ‘]’ is inserted in the code string and the process backtracks to bifurcation point I6 in order to resume the visit at the internode scale on the next son of I6, i.e. I7. Then coding goes through to the end of the poplar trunk since there are no more bifurcation points. Between entities I7 and I19, two growth unit boundaries are crossed which generate the final code string :

```
/P1/U1/I1<I2<I3<I4<I5<I6[+U1/I20<I21<I22<I23<I24<U2/I25<I26<I27]<I28<I29<I7<I8<I9<U2/I10<I11<I12<I13<I14<I15<U3/I16<I17<I18<I19]
```

It is often the case in practical applications that a number of attributes are measured on certain plant entities. Measured values can be attached to corresponding entities using a bracket notation, '{. . . }'. For instance, assume that one wants to note the length and the diameter of observed growth units. For each measured growth unit, a pair of ordered values defines respectively its measured length and diameter. Then, the precedent code string would become:

```
/P1/U1{10,5.9}/I1<I2<I3<I4<I5<I6[+U1{7,3.5}/I20<I21<I22<I23<I24<U2{4,2.1}/I25<I26<I27<I28<I29]<I7<I8<I9<U2{8,4.3}/I10<I11<I12<I13<I14<I15<U3{7.5,3.9}/I16<I17<I18<I19
```

In this string, we can read that the first growth unit of the trunk, U1, has length 10 cm and diameter 5.9 mm (units are assumed to be known and fixed).

In practical applications, coding plants as raw sequences of symbols becomes quite unreadable.

In order to give the user a better feedback of the plant topology in the code itself, we can slightly change the above code format in order to achieve better legibility. Each square bracket is replaced by a new line and an indentation level corresponding to the nested degree of this square bracket. Similarly, a new line is created after each feature set and the feature values are written in specific columns. The following table gives the final code corresponding to the example in *Figure3.3* .

```
↪                                    Length      Diameter
/
↪P1/
↪U1
↪                                    10          5.9
/
↪I1
↪<I2
↪<I3
↪<I4
↪<I5
↪<I6
```

(continues on next page)

(continued from previous page)

```
↪+U1
↪ 7 3.5
↪/
↪I20
↪<I21
↪<I22
↪<I23
↪<I24
↪<U2
↪ 4 2.1
↪/
↪I25
↪<I26
↪<I27
↪<I28
↪<I29
```

(continues on next page)

(continued from previous page)

```
↪<I7
↪<I8
↪<I9
↪<U2␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪                                8             4.3
/
↪I10
↪<I11
↪<I12
↪<I13
↪<I14
↪<I15
↪<U3␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪              7.5           3.9
/
↪I16
↪<I17
↪<I18
↪<I19
```

```
ENTITY-CODE
/P1/U1
^/I1<I2<I3<I
    +U1
    ^/I20<I2
    ^/I25<I2
<I7<I8<I9<U2
/I10<I11<I12
/I16<I17<I18
```

3.3.3 Exploring a simple example

Reading the MTG file

Once a plant database has been created, it can be analyzed using the **openalea.newmtg** python package. The different objects, methods and models contained in **openalea.newmtg** can be accessed through Python language.

The formal representation of a plant, and more generally of a set of plants, can be built using the function *MTG()*:

```
from openalea.mtg.aml import MTG
g = MTG('wij.mtg')
```

The procedure MTG attempts to build the plant formal representation, checking for syntactic and semantic correctness of the code file. If the file is not consistent, the procedure outputs a set of errors which have to be corrected before applying a new syntactic analysis. Once the file is syntactically consistent, the MTG is built (cf. *Figure3.4* b) and is available in the variable g.

Warning: However, for efficiency reasons, the latest constructed MTG is said to be **active** : it will be considered as an implicit argument of most of the functions dealing with MTGs. See `Activate()`

To get the list of all vertices contained in **g**, for instance, we write:

```
from openalea.mtg.aml import VtxList
```

(continues on next page)

(continued from previous page)

```
vlist = VtxList()
```

instead of:

```
vlist = VtxList(g)
```

The function *VtxList()* extracts the set of vertices from the active MTG and returns the result in variable vlist.

Once the MTG is loaded, it is frequently useful to make sure that the database actually corresponds to the observed data. Part of this checking process has already been done by the *MTG()* function. But, some high-level checking may still be necessary to ensure that the database is completely consistent. For instance, in our example, we might want to check the number of plants in the database. Since plants are represented by vertices at scale 1, the set of plants is built by:

```
plants = VtxList(Sc
```

Like vlist, the set plants is a set of vertices. The number of plants can be obtained by computing the size of the set plants.:

```
plant_nb = len(plants
```

Note: In the former AML language, the function Size was used to get the length. Here a call to the standard python function `len()` is used.

Each plant constituting the database can be individually and interactively accessed via Python.

For instance, assuming the plant corresponding to the example of *Figure3.4* b is represented by a vertex (at scale 1) with label *P1*. Plant *P1* can be identified in the database by selecting the vertex at scale 1 having index 1:

```
from opena
mtg.
aml
impor
Index
plant1_
list
=
[p
for
p
in
plant
if
Index
plant1
=
plant
list[

#
plant
list
is
a
list
of
verti
```

Note: former AML code: plant1 = Foreach _p In plants : Select(_p, Index(_p)==1)

The lambda expression (line 2) select the *plants* vertex *p* that fulfills the conditon *Index==1*. Thus, *plant1* contains the vertex representing plant *P1*. Now it is possible to apply new functions to this vertex in order to explore the nature of plant *P1*. Assume for instance we want to know the number of growth units composing *P1*:

```
from openalea.mtg.aml import Components
gu_nb = len(Compon
#should be 1
Components(1
[2]
len(Componen
33
```

Todo: clarify this example and following comments

the *Components* function applies

to a vertex *v* and returns the vertices composing *v* at the next superior scale. Since *plant1* is a vertex at scale 1, representing plant *P1*, components of *plant1* are vertices at scale 2, i.e. growth units. It is also possible to compute the number of internodes composing a plant by simply specifying the optional argument Scale in function Components:

```
internode_nb = len(Compon
 Scale=1)) # should return 1
```

3D representation

Example 1

Many such direct queries can be made on the plant database which provide interactive access to it. However, a complementary synthesizing view of the database may be obtained by a graphical reconstruction of plant geometry. Geometrical parameters, like branching and phyllotactic angles, diameters, length, shapes, are read from the database. If they are not available, mean values can be inferred from samples or can be inferred from additional data describing plant general geometry[19]. A 3D interpretation of the MTG provides the user with natural feedback on the database. Built-in function *PlantFrame()* computes the 3D-geometry of plants. For example:

```
from openalea.mtg.plantframe import PlantFrame
frame1 = PlantFrame
```

Warning: PlantFrame from openalea.mtg.aml is obsolet, use PlantFrame from openalea.mtg

Todo: script that leads to the picture in figure 3.5

Example 2

[19] Godin, C., Bellouti, S. et Costes, E., 1996. Restitution virtuelle de plantes réelles : un nouvel outil pour l'aide à l'analyse de données botaniques et agronomiques. In: L'interface des mondes réels et virtuels, 5èmes Journées Internationales Informatiques, Montpellier, France 22-24/05/96, pp. 369-378.

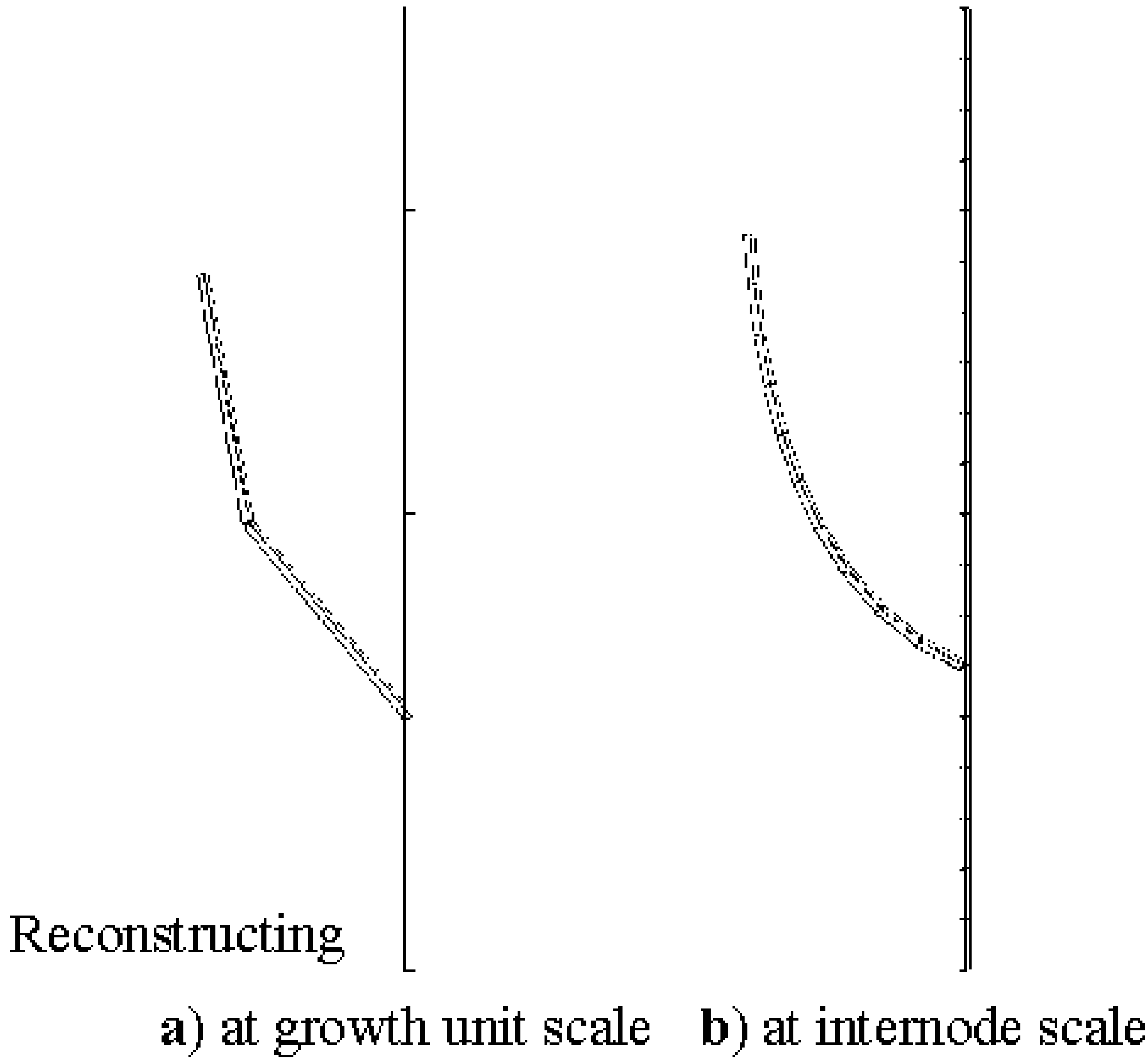


Fig. 9: **Figure 3.5**

Todo: in progress

```
from openalea.mtg.aml import MTG
from openalea.mtg.dresser import dressing_data_from_file
from openalea.mtg.plantframe import PlantFrame, compute_axes, build_scene
g = MTG('agraf.mtg')
dressing_data = dressing_data_from_file('agraf.drf')
topdia = lambda x: g.property('TopDia').get(x)
pf = PlantFrame TopDiamete
```

(continues on next page)

(continued from previous page)

```
axes = compute_axes(g, 3, pf.points, pf.origin)
diameters = pf.algo_diameter()
scene = build_scene(pf.g, pf.origin, axes, pf.points, diameters, 10000)
from vplants.plantgl.all import Viewer
Viewer.display(sc
```

Note: the previous example uses many functions that have not been introduced yet but they will be desribe later on.

Todo: CECHK that it workds in openalea.mtg : computes a 3D-geometrical interpretation of *P1* topology at scale 2, i.e. in terms of growth units (*Figure3.5* a). Like in the previous example, PlantFrame takes Scale as an optional argument which enables us to build the 3D-geometrical interpretation of P1 at the level of internodes (*Figure3.5* b):

Refinements of this

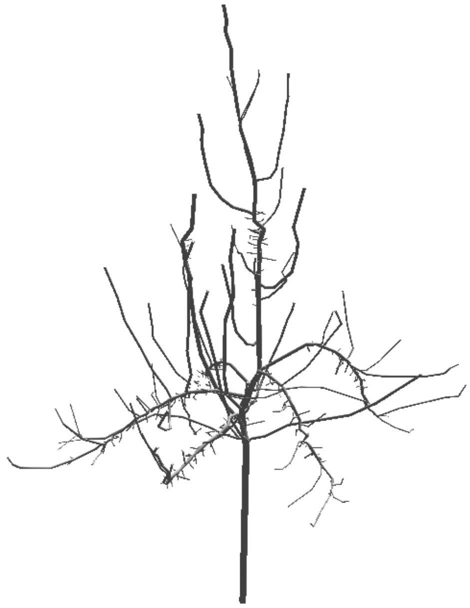


Fig. 10: **Figure 3.5** An apple tree plotted with the python script above

3D geometrical reconstruction may be obtained with the possibility to change the shape of the different plant components, possibly at different scales, to tune geometrical features (length, diameter, insertion angle, phyllotaxy, …) as functions of the topological position of entities in the plant structure.

Extraction of plant entity features

When attributes of entities are available in MTGs, it is possible to retrieve their values by using the function *Feature()*:

```
first_gu = Trunk(2)[0]
first_gu_diameter = Feature(first_gu, "Diameter")
```

Note: Here *Diameter* is a property/feature contained in the MTG header. Feature's names can be found in MTG's header, or directly by instrospection using this python syntax:: [x for x in g.property_names()]

The first line retrieves the vertex corresponding to the first growth unit of the trunk of *P1* (function

Trunk() returns the ordered set of components of vertex P1, and operator @ with argument 1 selects the first element of this set). Then, in the second line, the diameter of this growth unit is extracted from the database. Variable first_gu_diameter then contains the value 5.9 (see the code file). Similarly the length of the first growth unit can be retrieved:

```
first_gu_length = Feature(fi
gu, "Length")
```

Variable *first_gu_length* contains value 10.

The user can simplify this extraction by creating alias names using lambda function:

```
>>> diameter = lambda x: g.property(
```

(continues on next page)

(continued from previous page)

```
>>> length = lambda x: g.property('Length').get(x)
```

It is then possible with these functions to build data arrays corresponding to feature values associated with growth units:

```
>>> growth_unit_set = VtxList(Sc
>>> [length(x) for x in growth_unit_set]
```

(continues on next page)

(continued from previous page)

```
[10.0, 7.0, 4.0, 8.0, 7.5]
```

Here, VtxList should contain the index 2 and therefore the second line returns *10cm*, as expected. Moreover, new synthesized attributes can be defined by creating new functions using these basic features. For example, making the simple assumption that the general form of a growth unit is a cylinder, we can compute the volume of a growth unit:

```
from math import pi
volume = lambda x: pi * diameter(x / 4. * length(x)
```

(continues on next page)

(continued from previous page)

Now, the user can use this new function on any growth unit entity as if it were a feature recorded in the MTG. For instance, the volume of the first growth unit is computed by:

```
first_
↪gu_
↪volume␣
↪=␣
↪volume(fir
↪gu)
```

Todo: trunk and plant volumes using numpy.sum ?

The total volume of the trunk:

```
trunk_
↪volume␣
↪=␣
↪sum([volum
↪for␣
↪x␣
↪in␣
↪growth_
↪unit_
↪set])
```

(continues on next page)

(continued from previous page)

Todo: how and purpose of volume for the whole plant. Isnt' it the volume of the trunk ?

The wood volume of the whole plant can be computed by:

```
plant_volume = sum[volume for gu in Components
```

Extracting more information from plant databases

As illustrated in the previous section, plant databases can be investigated by building appropriate Python lambda functions. Built-in words of the **openalea.mtg.aml** module may be combined in various ways in order to create new queries. In this way, more and more elaborated types of queries can be constructed by creating user-defined functions which are equivalent to computing programs. In order to illustrate this procedure, let us assume that we would like to study distributions of numbers of internodes per growth units, such distributions being an important basic prerequisite for botanically-based 3D plant simulations (e.g.[2,9,37,3]). At a first stage, we consider all the growth units contained in the plant database together. We first need to define a function which returns the number of internodes of a given growth unit. Since in the database, each growth unit (at scale 2) is composed of internodes (at scale 3) we compute the set of internodes constituting a given growth unit *x* as follows:

```
internode_set = lambda x: Components
```

The object returned by function internode_set() is a set of vertices. The number of internodes of a given growth unit is thus the size of this set:

```
internode_nb = lambda x: len(internode_set(x))
```

(continues on next page)

[2] Barthélémy, D., 1991. Levels of organization and repetition phenomena in seed plants. Acta Biotheoretica, 39: 309-323.

[9] de Reffye, P., Dinouard, P. et Barthélémy, D., 1991. Modélisation et simulation de l'architecture de l'Orme du Japon Zelkova serrata (Thunb.) Makino (Ulmaceae): la notion d'axe de référence. In: 2ème Colloque International sur l'Arbre, Montpellier (FRA) 9-14/09/90. Naturalia Monspeliensa, Vol. hors-série, pp. 251-266.

[37] Jaeger, M. et de Reffye, P., 1992. Basic concepts of computer simulation of plant growth. In: The 1990 Mahabaleshwar Seminar on Modern Biology, Mahabaleshwar (IND) . Journal of Biosciences, Vol. 17, pp. 275-291.

[3] Bouchon, J., de Reffye, P. et Barthélémy, D. (Eds), 1997. Modélisation et simulation de l'architecture des végétaux. Science Update. INRA Editions, Paris, France, 435 pp.

(continued from previous page)

Second, the entities on which the previous function has to be applied, must be located in the database. A set of vertices is created by selecting plant entities having a certain property.

The set of growth units is the set of entities at scale 2

```
gu_
↪set
↪=
↪VtxList(Sc
```

Third, we have to apply function internode_nb() to each element of the selected set of entities:

```
>>> sample1 = [internode_nb(x) for x in gu_set]
>>> sample1
[9, 5, 5, 6, 4]
```

Todo: in all the documentation, we should also emphasize the puire Pythonic style. For instance in the example above, we could have created a generator g.components() and then for [len([x for x in g.components(y)]) for y in gu_set]

We use

the list-comprehension Python syntax in order to browse the whole set of growth units of the database, and to apply our internode_nb() function to each of them.

Now, we want to get the distribution of the number of internodes on a more restricted set of growth units. More precisely, we would like to study the distribution of internode numbers of different populations corresponding to particular locations in the plant structure. We thus have to define these populations first and then to iterate the function internode_nb() on each entity of this new population like in the previous example. Let us consider for example the population made of the growth units composing branches of order 1. Consider again the whole set of growth units gu_set. Among them, those which are located on branches (defined as entities of order 1 in AML) are defined by:

```
>>> gu1 = [x for x in VtxList(Sc
>>> [Order(x) for x in gu1]
[0, 1, 1, 0, 0]
```

Here again, we use the Python list comprehension in order to browse the whole set of growth units of the database, and to apply the Order function to each of them. Then, in order to select growth unit vertices whose order is 1 (all the growth units in the corpus which are located on branches), change the above command into:

```
>>> [x for x in VtxList(Sc
if Order(x) == 1]
[9, 15]
```

Eventually, after the sample of values is built, the above function is applied to the selected entities :

Todo: figure out what was the input data for the following plots and use either pylab or Histogram or both .

::

sample = Foreach _x In gu1 :

internode_number(_x

At this stage, a set of values has been extracted from the plant database corresponding to a topologically selected set of entities. This sample of data can be further investigated with appropriate AML tools. For example, AML provides the built-in function `Histogram()` which builds the histogram corresponding to a set of values.:

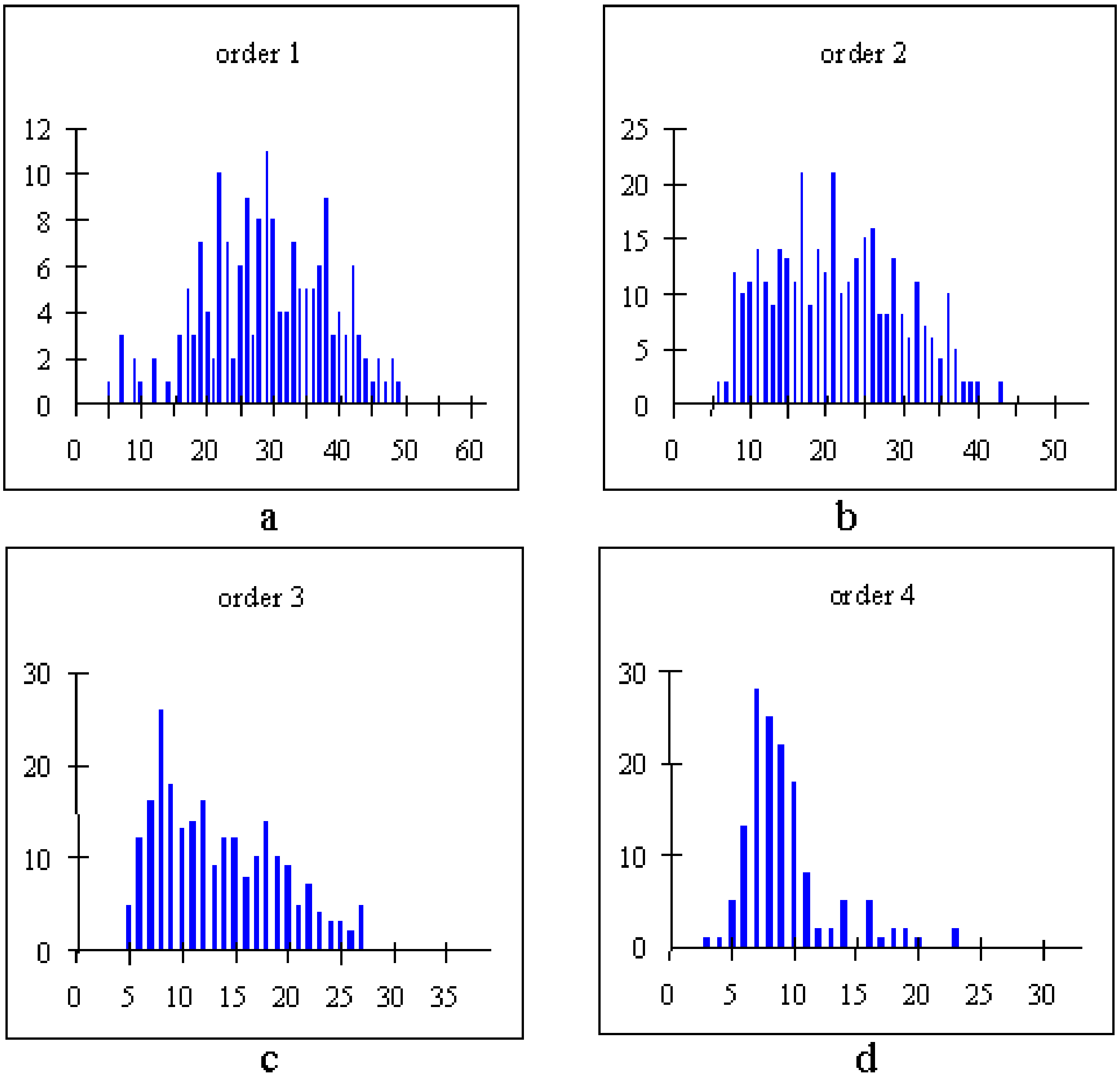
```
histo1 = Histogram(
Plot(histo1)
```

This plot gives the graph depicted in Figure 3-6a. Similarly, by selecting samples corresponding to different topological situations, we would obtain the series of plots in Figure 3-6 [4].

3.3.4 Types of extracted data

various types of data can be extracted from MTGs. For each plant component in the database, attributes can be extracted or synthesised using the Python language. The wood volume of a component, for instance, can be synthesised from the diameter and the length of this component measured in the field. The type of measurement carried out in the context of architectural analysis emphasises the use of discrete variables which can be either symbolic, e.g. the type of axillary production at a given node (latent bud, short shoot or long shoot) or numeric (number of flowers in a branching structure). In general, a plant component can be qualified by a set of attributes, called a multivariate attribute. A plant component, for instance, could be described by a multivariate attribute made up of the volume, the number of leaves, the azimuth and the botanical type of the constituent.

Multivariate attributes correspond to



Different distributions of the number of internodes par growth unit, in different topological situations

Fig. 11: **Figure 3.6**

the first category of data that can be extracted from MTGs. A second and more complex category of particular importance is defined by sequences of – possibly multivariate – attributes. The aim of this category is to represent biological sequences that can be observed in the plant architecture. These sequences may have two origins: they can correspond to changes over time in the attributes attached to a given plant component. In this case, the sequences represent the trajectories of the components with respect to the considered attributes and the index parameter of the sequences is the observation date. Sequences can also correspond to paths in the tree topological structures contained in MTGs. In this case, the index parameter of the sequences is a spatial index that denotes the rank of the successive components in the considered paths. Spatially-indexed sequence is a versatile data type for which the attributes of a component in the path can be either directly extracted or synthesised from the attributes of the borne components. In the later case, all the information contained in the branching system can be efficiently summarised into a sequence of multivariate attributes, corresponding to the main axis of the branching system.

A third category of object can be extracted from MTGs, namely trees of – multivariate – attributes. Like sequences, these objects are intended to preserve part of the plant organisation in the extracted data.

Tree structures represent the raw organisation of the components that compose branching structures of the plant at a certain scale of analysis.

Data extracted from MTGs can thus be ordered according to their level of structural complexity: unstructured data, sequences, trees. These levels correspond to different degrees to which the structural information contained in the MTG is summarised and are associated with different statistical analysis techniques.

3.3.5 Statistical exploration and model building using other Openalea/VPlants packages

To explore plant architecture, users are frequently led to create data samples according to topological criteria on plant architecture. A wide range of AML primitives that apply to MTGs enable the user to express these topological criteria and select corresponding plant components. Samples of the three main structural data types can be created as described below:

Multivariate samples: Simple data samples can be created by computing the set of - possibly multivariate - attributes associated with a selected set of components, e.g. the number of flowers borne by components that appeared in the plant structure during 1995. The packages openalea.stat_tool and openalea.sequence_analysis provides a core of tools for exploring these objects. However, a very large panoply of methods are available in other statistical packages for analysing multivariate samples (the user can export data to other softwares such as RPy).

Samples of mul-

tivariate sequences: In the context of plant architecture analysis, MTG objects present two advantages. On the one hand, part of the plant organisation is directly preserved in the sample through the notion of ''sequence" discussed above. On the other hand, the structural complexity of samples of sequences still remains tractable and efficient exploratory tools and statistical models can be designed for them[28][29]. The openalea.sequence_analysis system includes mainly classes of stochastic processes such as (hidden) Markov chains, (hidden) semi-Markov chains and renewal processes for the analysis of discrete-valued sequences. A set of exploratory tools dedicated to sequences built from numeric variables is also available, including sample (partial) autocorrelation functions and different types of linear filters (for instance symmetric smoothing filters to extract trends or residuals).

Samples of multivariate trees: The analysis of samples of tree structured data is a

[28] Guédon, Y., Barthélémy, D. et Caraglio, Y., 1999. Analyzing spatial structures in forests tree architectures. In: Salamandra (Ed) Empirical and process-based models for forest tree and stand growth simulation, Oeiras, Portugal 21-27/09/1997, pp. 23-42.

[29] Guédon, Y. et Costes, E., 1999. A statistical approach for analyszing sequences in fruit tree architecture. In: Wagenmakers P.S., van der Werf W., Blaise Ph. (Eds), 5th International Symposium on Computer modelling in fruit research and orchard management, Wageningen, The Netherlands 28-31/07/1998. Acta Horticulturae, pp. 271-280.

challenging problem. A sample of trees could represent a set of comparable branching systems considered at different locations in a plant or in several plants. Similarly, the development of a plant can be represented by a set of trees, representing different steps in time of a branching system. Plant organisation for this type of object is relatively well preserved in the raw data. However, this requires a higher degree of conceptual and algorithmic complexity. We are currently investigating methods for computing distances between trees[13] which could be used as a basis for dedicated statistical tools.

OpenAlea/VPla contains a large set of tools for analysing these different types of samples, with special emphasis on tools dedicated to the analysis of samples of discrete-valued sequences. These tools fall into one of the three following categories:

- exploratory analysis relying on descriptive methods (graphi-

[13] Ferraro, P. et Godin, C., 1998. Un algorithme de comparaison d'arborescences non ordonnées appliqué à la comparaison de la structure topologique des plantes. In: SFC'98, Recueil des Actes, Montpellier, France 21-23/09/1998, Agro Monpellier, pp. 77-81.

cal display, computation of characteristics such as sample autocorrelation functions, etc.),

- parametric model building,
- comparison techniques (between individual data).

The aim of building a model is to obtain adequate but parsimonious representation of samples of data. A parametric model may then serve as a basis for the interpretation of a biological phenomenon. The elementary loop in the iterative process of model building is usually broken down into three stages:

1.

The specification stage consists of determining a family of candidate models on the basis of the results given by an exploratory analysis of the data and some biological knowledge.

2. The estimation stage, consists of inferring the model parameters on the basis of the data sample. This model is chosen from within the family determined at the specification stage. Automatic methods of model selection are available for classes of models such as (hidden) Markov chains dedicated to the analysis of stationary discrete-valued sequences. The estimation is always made by algorithms based on the maximum likelihood criterion. Most of these algorithms are iterative optimisation schemes which can be considered

as applications of the Expectation-Maximisation (EM) algorithm to different families of models,[12,26,27]. The EM algorithm is a general-purpose algorithm for maximum likelihood estimation in a wide variety of situations best described as incomplete data problems.

3. The validation stage, consists of checking the fit between the estimated model and the data to reveal inadequacies and thus modify the a priori specified family of models. Theoretical characteristics can be computed from the estimated model parameters to fit the empirical characteristics extracted from the data and used in the exploratory analysis.

The parametric approach based on the process of model building is com-

[12] Dempster, A.P., Laird, N.M. et Rubin, D.B., 1977. Maximum likelihood from incomplete data via the EM algorithm (with discussion). Journal of the Royal Statistical Society, Series B, 39: 1-38.

[26] Guédon, Y., 1998. Analyzing nonstationary discrete sequences using hidden semi- Markov chains. Document de travail du programme Modélisation des plantes, 5-98. CIRAD, Montpellier, France, 41 pp.

[27] Guédon, Y., 1998. Hidden semi-Markov chains: a new tool for analyzing nonstationary discrete sequences. In: 2nd International Symposium on Semi-Markov models: theory and applications, J. Janssen et N. Limnios (Eds), Compiègne, France 09-11/12/1998, Université de Technologie de Compiègne, pp. 1-7.

plemented by a non-parametric approach based on structured data alignment (either sequences or trees). Distance matrices built from the piece by piece alignments of a sample of structured data can be explored by clustering methods to reveal groups in the sample.

documentation status

Documentation adapted from the AMAPmod user manual version 1.8.

3.3.6 Biblio

3.4 Illustr exploring an apple tree orchard

Todo: This section has to be validated (e.g., translate aml code into python)

Let us

now illustrate the usage of openalea.mtg package together with other packages such as openalea.sequence_

in a real application. To do this, we shall consider an apple tree orchard and show how a plant architecture database can be created from observations[24]. Then, we shall use this database to illustrate the use of specific tools employed to explore plant architecture databases.

3.4.1 Biolo context and data collection

The application is part of a general selection

[24] Godin, C., Guédon, Y. et Costes, E., 1999. Exploration of plant architecture databases with the AMAPmod software illustrated on an apple-tree bybird family. Agronomie, 19(3/4): 163-184.

program, conducted at INRA (Institut National de la Recherche Agronomique), and aims to improve apple tree species as regards morphological characters and more classical criteria such as fruit quality and disease resistance. In this particular example, two apple tree clones were chosen for their contrasting growth and branching habits. The first clone ('Wijcik') exhibits a very particular growth and branching habit, characterised by short internodes, great diameters and the absence of long axillary branches. By contrast, the second clone ('Baujade') exhibits many long and flexible branches. A population of 102 hybrids was obtained by crossing these two clones. The objective of this work was to study how morphological characters, such as the length of the internodes or the number of long lateral branches, are distributed within the progeny.

Creation of the database: The branching systems borne by the three-year-old annual shoot of the trunk is described for each individual. The branching system is first broken down into axes i.e. linear portions of stem derived from the same bud. Each axis is then divided into portions created during the same year (called annual shoots). When cessation and resumption of growth occur within a year, the annual shoot can be split into growth units, i.e. portions created over the same period (or between two resting periods). Finally, the growth units can be divided into internodes, i.e. portions of stem between two leaves. Regarding these successive decompositions, a given branching system is simultaneously considered at four scales. The different plant components and their connections are represented into a code file as explained previously.

In order to

give a quantitative idea of the total resources necessary for an application of this size, it should be noted that all the measures were carried out by a team of 6 persons over 5 days. The collected data, initially recorded on paper, were then computer-entered by 1 person over 20 days using a text editor and consists of a file of approximately 16000 lines of code. The corresponding MTG is constructed in 45 seconds on a SGI-INDY workstation. It contains about 65000 components and some 15000 attributes. The overall size of the database is 7 Mb.

3.4.2 3D visualisation of real plants

To build the database associated with the collected data, the AMAPmod system is launched and an MTG is built from the encoded plant file:

```
plant_database = MTG("wij.mtg")
```

The primitive MTG attempts to build a formal representation of the orchard, checking for syntactic and semantic correctness of the code file. If the file is not consistent, the procedure outputs a set of errors which must be corrected before applying a new syntactic analysis. Once the file is syntactically consistent, the MTG is built and is available in the variable plant_database. However, for efficiency reasons, the latest constructed MTG is said to be ‘’active”: it will be considered as an implicit argument for most of the primitives dealing with MTGs. For example, to obtain the set of vertices representing the plants contained in the database, i.e. vertices at scale 1, the primitive VtxList is used and applies by default to the active MTG plant_database:

```
plant_list = VtxList(Sc
```

It is then possible to obtain an initial feedback on the collected data by displaying a 3D geometrical interpretation of a plant from the MTG. This notably allows the user to rapidly browse the overall database. For instance, a geometric interpretation of the 5th plant in the set of plants described in the MTG can be computed and plotted using the primitive PlantFrame as follows, (Figure 3-7a):

```
geom_
↪struct␣
↪=␣
↪PlantFrame
↪list[4])
Plot(geom_
↪struct)
```

Todo: continue to adapt the documenation from here including example here above

Such reconstructions can be carried out even if no geometric information is available in the collected data. In this case, algorithms are used to infer the missing data where possible (otherwise, default information is used)[19]. In other cases, plants are precisely digitised and the algorithms can provide accurate 3D geometric reconstructions[7][22][46][48].

Apart from giving a natural view of the plants contained in the database, these 3D reconstructions play another important role: they can be used as a support to graphically visualise how various sorts of information are distributed in the plant architecture. Figure 3-7b for example shows the organisation of plant components according to their branching order (trunk components have order 0, branch components have order 1, etc.). This would be obtained by the following commands:

```
color_order(_x) = Switch Order(_x) Case 0: MediumGrey
```

(continues on next page)

[19] Godin, C., Bellouti, S. et Costes, E., 1996. Restitution virtuelle de plantes réelles : un nouvel outil pour l'aide à l'analyse de données botaniques et agronomiques. In: L'interface des mondes réels et virtuels, 5èmes Journées Internationales Informatiques, Montpellier, France 22-24/05/96, pp. 369-378.

[7] Costes, E., Sinoquet, H., Godin, C. et Kelner, J.J., 1999. 3D digitizing based on tree topology : application to study th variability of apple quality within the canopy. Acta Horticulturae, in press.

[22] Godin, C., Costes, E. et Caraglio, Y., 1997. Exploring plant topology structure with the AMAPmod software : an outline. Silva Fennica, 31(3): 355-366.

[46] Sinoquet, H., Adam, B., Rivet, P. et Godin, C., 1998. Interactions between light and plant architecture in an agroforestry walnut tree. Agroforestry Forum, 8(2): 37-40.

[48] Sinoquet, H., Rivet, P. et Godin, C., 1997. Assessment of the three-dimensional architecture of walnut trees using digitising. Silva Fennica, 31(3): 265-273.

(continued from previous page)

```
Case 1: DarkGrey Case 2: LightGrey Case 3: Black
Default: White
Plot(geom_struct, Color=colo
order)
```

This representation emphasises different informations related to the branching order: it can be seen in Figure 3-7b that the maximum branching order is 4, that this order is reached only once in the tree crown, and that this occurs at a floral site (black component).

The use of

the 3D representation of plant structure can also be illustrated in the context of plant growth analysis. The year in which each component grew can be retrieved from a careful analysis of the plant morphological makers. If this information is recorded in the MTG, it is then possible to colour the different components accordingly. Figure 3-7c shows, for instance, that a branch appeared on the trunk during the first year of growth. This information can then be linked to other data, e.g. the branching order of a component or the number of fruits borne by a component, and thus provides deeper insight into the plant growth process.

Thanks to the multiscale nature of the plant representation, more or less detailed information can be projected onto the plant structure. Let us consider again the context of plant growth analysis. Plant growth is characterised by rhythms that result in the production of long internodes during periods of high activity and short internodes during rest periods (indicated on the plant by scares close together). These informations, at the level of internodes, can be projected onto the plant 3D structure (Figure 3-7d). Like the year of growth, this information enables us to access plant growth dynamics, but now, at an intra-year scale.

Finally, another use for the virtual reconstruction of measured plants is illustrated in Figure 3-8a and 8b. These plants have been reconstructed from the MTG at the scale of each leafy internode. This enables us to obtain a natural representation of the plant which can be used for instance in models that are intended to describe the interaction of the plant and its environment (e.g. light) at a detailed level, e.g.[41]. More generally, the user can plot a set of plants from the database (Figure 3-9):

```
orchard = aml.PlantFrame
list)
aml.Plot(orcha
```

3.4.3 Extra of data samples

Visualizing informations

[41] Rapidel, B., 1995. Etude expérimentale et simulation des transferts hydriques dans les plantes individuelles. Application au caféier (Coffea arabica L.). Thèse Doctorat, Université des Sciences et Techniques du Languedoc (USTL), Montpellier, France, 246 pp.

projected onto the 3D representation of plants is one way to explore the database. More quantitative explorations can be carried out and the most simple of these consists of studying how specific characters are distributed in the architecture of the plant population. To do this, samples of components are created corresponding to some topological or morphological criteria, and the distributions of one or several characters (target characters) are studied on this sample. This data extraction always follows the three following steps:

Firstly, a sample of components is created to study the target character. Secondly, the character itself is defined. It may be more or less directly derived from the data recorded in the field. For example, it is straightforward to define the diameter of a component if this has been measured in the field. On the other hand, the maximum branching order of the components that are borne by a given component needs some computation. Thirdly, the target character is computed for each component of the selected sample of components.

The output of these three operations is a set of values that can be analysed and visualised in various ways. Let us assume for instance that we wish to determine the distribution of the number of internodes produced during a specific growth period for all the plants in the database. It is first necessary to determine the sample of components on which we wish to study this distribution. In our case, we assume that we are interested in the growth units of the trunk that are produced during the first year of growth. This would be written as:

```
sample = Foreach _component In growth_unit_list:
    Select(_component, Order(_component) == 0 And
    Index(_component) == 90)
```

The variable sample thus contains the set of growth units whose order is 0 (i.e. which are parts of trunks) and whose growth year is 1990 (assuming 1990 corresponds to the first year of growth). The second step consists of defining the target character. This can be done by defining a corresponding function:

```
nb_of_internodes = lambda x: len(Compor
```

The number of internodes of a component _x (assumed to be a growth unit) is defined as the size of the set of components that compose this growth unit _x (assuming that growth units are composed of internodes). Finally, this function is applied to each component in the previously selected sample and the corresponding histogram is plotted (Figure 3-10):

```
sample_values = Histogram(_component In sample :nb_of_internodes(component))
Plot(sample_values)
```

This example illustrates the kind of interaction a user may expect from the exploration of tree architecture. In the field, the growth units of the trunks produced during the first year of growth present a variable length, ranging roughly from 10 to 100 internodes. However, the quantitative exploration of the database shows that the histogram exhibits two relatively well-separated sub-populations of components (Figure 3-10). The sub-population of short components corresponds to the first annual shoots of the trunk, made up of two successive intra-annual growth units, while the sub-population of long components corresponds to the first annual shoots made up of a single growth unit.

In order to sep- a- rate and char- ac- terise these two sub- populations, we can make the as- sump- tion that the global distribution is a mixture of two parametric distributions, more precisely, two negative binomial distributions. The parameters of this model can be estimated from the above histogram as follows:

```
mixture␣
↪=␣
↪Estimate(s
↪value,
↪␣
↪
↪"MIXTURE
↪",
↪
↪"NEGATIVE_
↪BINOMIAL
↪",
↪␣
↪
↪"NEGATIVE_
↪BINOMIAL
↪")
Plot(mixture
```

For all para- met- ric mod- els in the sys- tem, the function Estimate performs both parameter estimation and computation of various quantities (likelihood of the observed data for the estimated model, theoretical characteristics, etc) involved in the validation stage. As demonstrated by the cumulative distribution functions in Figure 3-11b, the data are well fitted by the estimated mixture of two negative binomial distributions. The weights of the two components of the mixture are very close (0.49 / 0.51), the first being centred on 21 internodes and the second on 53 internodes (Figure 3-11a). Due to the small overlap of these two mixture components (Figure 3-11a), the extracted sample can be optimally split up into two optimal sub-populations with a threshold fixed at 37.

As illustrated in this example, using AMAPmod, the user can query the database, make assumptions and look for data regularities. This interactive exploration process enables the user to build a rich and detailed mental representation of the architectural database, which relies on various complementary viewpoints.

3.4.4 Extraction and analysis of biological sequences

The previous section illustrates the extraction of a simple sample type, made up of numeric values. In this section, we consider a more complex sample type, made up of sequences of values. For example, in the apple tree database, let us consider sequences of lateral productions along trunks. Our aim is to analyse how lateral branches are distributed along the trunks of hybrids.

The sequences are coded as follows: for each plant, the 90 annual shoot of the trunk is described node by node from the base to the top. Each node is qualified by the type of lateral production (latent bud: 0, one-year-delayed short shoot: 1, one-year-delayed long shoot: 2 and immediate shoot: 3). This sample of sequences is built as follows:

```
AML> seq = Foreach _component In growth_unit_sample :
Foreach _node In Axis(_component, Scale -> 4) :
Switch lateral_type(_node)
Case BUD: 0 Case SHORT: 1 Case LONG: 2
Case IMMEDIATE: 3 Default: Undef
```

(continues on next page)

(continued from previous page)

The AML variable growth_unit_sa contains the set of growth units of interest (assumed to be selected before).

For each component in this set, the array of nodes that compose its main axis is browsed by the second Foreach construct. Finally, for each node, a function lateral_type() (defined elsewhere) is used to encode the nature of the lateral production at that node.

Figure 3-12 illustrates the diversity of annual shoot branching structures encountered in the studied hybrid family, which results from the different branching habits of the two parents. In our context, we wish to characterise and classify the hybrids according to their branching habits. The difficulty arises from the fact that the branching pattern is made of a succession of branching zones which are not characterised by a single type of lateral production but by a combination of types (e.g. short shoots interspersed with latent buds). We shall use this example to illustrate how parametric models may be used in AMAPmod to identify and characterize successive branching zones along these annual shoots.

We assume that sequences have a two-level structure, where annual shoots are made up of a succession of zones, each zone being characterised by a particular combination of lateral production types. To model this two-level structure, we use a hierarchical model with two levels of representation. At the first level, a semi-Markov chain (Markov chain with null self-transitions and explicit state occupancy distributions) represents the succession of zones along the annual shoots and the lengths of each zone[6,28,29]. Each zone is represented by a state of the Markov chain and the succession of zones are represented by transitions between states. The second level consists of attaching to each state of the semi-Markov chain a discrete distribution which represents the lateral productions types observed in the corresponding zone. The whole model is called a hidden semi-Markov chain[26,27].

The model parameters

[6] Costes, E. et Guedon, Y., 1997. Modelling the sylleptic branching on one-year-old trunks of apple cultivars. Journal of the American Society for Horticultural Science, 122(1): 53-62.

[28] Guédon, Y., Barthélémy, D. et Caraglio, Y., 1999. Analyzing spatial structures in forests tree architectures. In: Salamandra (Ed) Empirical and process-based models for forest tree and stand growth simulation, Oeiras, Portugal 21-27/09/1997, pp. 23-42.

[29] Guédon, Y. et Costes, E., 1999. A statistical approach for analyszing sequences in fruit tree architecture. In: Wagenmakers P.S., van der Werf W., Blaise Ph. (Eds), 5th International Symposium on Computer modelling in fruit research and orchard management, Wageningen, The Netherlands 28-31/07/1998. Acta Horticulturae, pp. 271-280.

[26] Guédon, Y., 1998. Analyzing nonstationary discrete sequences using hidden semi- Markov chains. Document de travail du programme Modélisation des plantes, 5-98. CIRAD, Montpellier, France, 41 pp.

[27] Guédon, Y., 1998. Hidden semi-Markov chains: a new tool for analyzing nonstationary discrete sequences. In: 2nd International Symposium on Semi-Markov models: theory and applications, J. Janssen et N. Limnios (Eds), Compiègne, France 09-11/12/1998, Université de Technologie de Compiègne, pp. 1-7.

are estimated from the extracted sample of sequences by the function Estimate:

```
hsmc = Estimate(s
"HIDDEN_SEMI-MARKOV", initial_hsmc, Segmentati
= True)
```

The first argument seq represents the extracted sequences, “HIDDEN_SEM MARKOV” specifies the family of models and initial_hsmc is an initial hidden semi-Markov chain which summarises the hypotheses made in the specification stage. An optimal segmentation of the sequences is required by the optional argument Segmentation set at True.

The hidden semi-Markov chain built from the 90 annual shoots of the 102 hybrids is depicted in Figure 3-13 with the following convention: each state is represented by a box numbered in the lower right corner. The possible transitions between states are represented by directed edges with the attached probabilities noted nearby. Transient states are surrounded by a single line while recurrent states are surrounded by a double line. State i is said to be recurrent if starting from state i, the first return to state i always occurs after a finite number of transitions. A nonrecurrent state is said to be transient. The state occupancy distributions which represent the length of the zones in terms of number of nodes are shown above the corresponding boxes. The possible lateral productions observed in each zone are indicated inside the boxes, the font sizes being roughly proportional to the observation probabilities(for state 3, these probabilities are 0.1, 0.62 and 0.28 while for state 4, these probabilities are 0.01, 0.07 and 0.92 for latent bud, one-year-delayed short shoot and one-year-delayed long shoot respectively). State 0 which is the only transient state is also the only initial state as indicated by the edge entering in state 0. State 0 represents the basal non-branched zone of the annual shoots. The remaining five states constitute a recurrent class which corresponds to the stationary phase of the sequences.

Building a parametric model gives us a global insight into the structure of the 90 annual shoot of the trunk for the 102 hybrids. The adequacy of the estimated model to the data is checked by examining the fitting of theoretical characteristic distributions computed from the model parameters to the corresponding observed characteristic distributions extracted from the data. Counting characteristic distributions for example focus on the number of occurrences of a given feature per sequence. The two features of interest are the number of series (or clumps) and the number of occurrences of a given lateral production type per sequence. The fits of counting distributions (Figure 3-14) can be plotted by the following function:

```
Plot(hsmc, "Counting")
```

In addition, the optimal segmentation of the observed sequences in successive zones (Figure 3-12) can be extracted from the model as a by-product of estimation of model parameters by the following function:

```
segmented_seq = ExtractDat
```

(continues on next page)

(continued from previous page)

segmented_seq represents the observed sequences augmented by a variable which contains the corresponding optimal state sequences (Figure 3-12). A careful examination of this optimal segmentation help us highlight a discriminating property: it suggests using the absence of state 4 in this optimal segmentation as a discrimination rule between hybrids closer to the Wijcik parent than to the Baujade parent (and conversely). State 4 corresponds to a dense long branching zone characteristic of the Baujade parent. Two sub-populations close to each of the parents are extracted by the function ValueSelect relying on the absence/presence of state 4 on the 1st variable:

```
wijcik_seq = ValueSelec
seq, 1, 4, Mode -> Reject)
baujade_seq = ValueSelec
seq, 1, 4, Mode -> Keep)
```

(continues on next page)

(continued from previous page)

Simply counting the number of axillary long shoots per sequence would not have been sufficient, since for a given number of long shoots, these can be either scattered (Figure 3-12c) or aggregated in a dense zone (Figure 3-12d). This is confirmed by comparing the empirical distributions of the number of series with the number of occurrences of axillary long shoots per sequence extracted from the two hybrid sub-populations. The empirical distributions of the number of series/number of occurrences of axillary long shoots (coded by 2) per sequence for the sub-population close to the Wijcik parent can be simultaneously plotted by the following function (Figure 3-15a):

```
AML> Plot(Extra
seq, "NbSeries
", 2, 2), ExtractHis
seq, "NbOccurre
", 2, 2))
```

These empirical distributions are very similar for the sub-population close to the Wijcik parent, (Figure 3-15a). Most of the series are thus composed of a single long shoot. These empirical distributions are very different for the sub-population close to the Baujade parent, (Figure 3-15b). In this case, the series are frequently composed of several successive long shoots.

The studied sample of sequences encompasses a broad spectrum of branching habits ranging from the Wijcik to the Baujade parent one. Hence, the building of a parametric model is mainly used for identifying a discrimination rule to separate the initial sample of branching sequences into two sub-samples.

documentation status

Documentation adapted from the AMAPmod user manual version 1.8.

3.4.5 Biblio

3.5 Tutor Create MTG file from scratch

This tutorial briefly introduces the main features of the package and should show you the contents and potential of the **openalea.mtg** library.

All the examples can be tested in a Python interpreter.

3.5.1 MTG creation

Let us consider the following example:

```
import openalea.mtg as mtg

g = mtg.MTG()

print len(g)
print g.nb_vertices()
```

(continues on next page)

(continued from previous page)

```
7  print g.nb_scales()
8
9  root = g.root
10 print g.scale(root
```

- First, the package is imported (**line 1**).
- Then, a mtg is instantiated without parameters (**line 3**).
- However, as for a `Tree`, the mtg is not empty (**line 5-

7).

- There is always a root node at scale 0 (**line 9-10**).

3.5.2 Simp edition

We add a component *root1* to the root node, which will be the root node of the tree at the scale 1.

```
root1 = g.add_component(

# Edit the tree at scale 1 by
```

(continues on next page)

(continued from previous page)

```
# to the vertex `root1`.
v1 = g.add_child(root
v2 = g.add_child(root
v3 = g.add_child(root

g.parent(v1) == root1
g.complex(v1 == root
v3 in g.siblings(v
```

3.5.3 Trave the mtg at one scale

The mtg can be traversed at any scales like a regular tree. Their are three traversal algorithms working on Tree data structures (container_algo_traversal):

- `pre_order`
- `post_order`
- `level_order`

These methods take as parameters a tree like data structure, and a vertex. They will traverse the subtree rooted on this vertex in a specific order. They will return an iterator on the traversed vertices.

```
from openalea.container.traversal.tree import *
```

(continues on next page)

(continued from previous page)

```
print list(g.components
print list(pre_order(g, root1))
print list(post_order(g, root1))
print list(level_order(g, root1))
```

Warning: On **MTG** data structure, methods that return collection of vertices always return an iterator rather than `list`, `array`, or `set`.

You have to convert the

length.

it- er- a- tor into a list if you want to dis- play it, or com- pute its

```
>>> print le
Traceback (m
  File "<std
TypeError: o
```

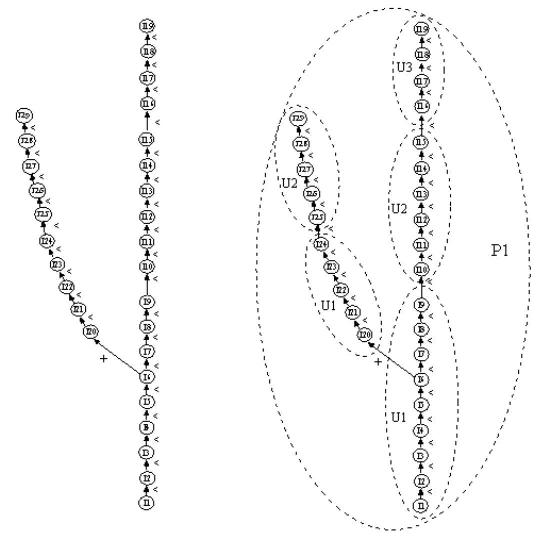
Use rather:

```
>>> componen
>>> print co
[1, 2, 3, 4]
```

3.5.4 Full ex- am- ple: how to cre- ate an MTG

```
from␣
↪openalea.
↪mtg.
↪mtg␣
↪import␣
↪*
```

(continues on next page)



a) tree graph at internode scale b) multiscale tree graph (MTG)

Fig. 12: **Figure 1:** Graphical representation of the MTG file code_file2.mtg used as an input file to all examples contained in this page

(continued from previous page)

```
from openalea.mtg.aml import *

g = mtg.MTG()
plant_id = g.add_component
root, label='P1')

#first u1
u1 = g.add_component
id, label='U1', Length=10,
```

(continues on next page)

(continued from previous page)

```
i = g.add_component(i, label='I1')
i = g.add_child(i, label='I2', edge_type='<')
i = g.add_child(i, label='I3', edge_type='<')
i = g.add_child(i, label='I4', edge_type='<')
i = g.add_child(i, label='I5', edge_
```

(continues on next page)

(continued from previous page)

```
i6 = i = g.add_child(i, label='I6', edge_type='<')

#u2 branch
i, u2 = g.add_child_and_complex(i6, label='I20', edge_type='+', Length=7, Diameter=3.5)
g.node(u2).label='U2'
g.node(u2).edge_type='+'
i = g.add_child(i, label='I21', edge_
```

(continues on next page)

(continued from previous page)

```
i = g.add_child(i, label='I22', edge_type='<')
i = g.add_child(i, label='I23', edge_type='<')
i = g.add_child(i, label='I24', edge_type='<')

#u3 branch
i, u3 = g.add_child_and_complex(i, label='I25', edge_type='
```

(continues on next page)

(continued from previous page)

```
g.node(u3).label='U3'
g.node(u3).edge_type='<'
i = g.add_child(i, label='I25', edge_type='<' )
i = g.add_child(i, label='I26', edge_type='<' )
i = g.add_child(i, label='I27', edge_type='<' )
```

(continues on next page)

(continued from previous page)

```
i = g.add_child(i, label='I28', edge_type='<')
i = g.add_child(i, label='I29', edge_type='<')

#continue u1
i = g.add_child(i6, label='I7', edge_type='<')
i = g.add_child(i, label='I8', edge_type='<')
```

(continues on next page)

(continued from previous page)

```
i = g.add_child(i, label='I9', edge_type='<')

# u2 main axe
i, c = g.add_child_and_complex(i, label='I10', edge_type='<', Length=8, Diameter=4.3)
g.node(c).label='U2'
g.node(c).edge_type='<'
i = g.add_child(i, label='I11',
```

(continues on next page)

(continued from previous page)

```
i = g.add_child(i, label='I12', edge_type='<')
i = g.add_child(i, label='I13', edge_type='<')
i = g.add_child(i, label='I14', edge_type='<')
i = g.add_child(i, label='I15', edge_type='<')
```

(continues on next page)

(continued from previous page)

```


# u3 main axe
i, c = g.add_child_and_complex(i, label='I16', edge_type='<', Length=7.5, diameter=3.9 )
g.node(c).label='U3'
g.node(c).edge_type='<'
i = g.add_child(i, label='I17', edge_type='<' )
```

(continues on next page)

(continued from previous page)

```
i = g.add_child(i, label='I18', edge_type='<')
i = g.add_child(i, label='I19', edge_type='<')


#fat_mtg(g)

print g.is_valid()
print g

for id in g.vertices():
    print g[id]
from openalea.mtg.io import *
```

(continues on next page)

(continued from previous page)

```
print list(g.property_names())
properties = [(p, 'REAL') for p in g.property_names() if p not in ['edge_type', 'index', 'label']]
print properties
mtg_lines = write_mtg(g, properties)
f = open('test.mtg', 'w')
f.write(mtg_lines)
f.close()
```

Authors

Christophe Pradal <christophe pradal __at__ cirad fr>, Thomas Cokelaer <thomas cokelaer __at__ sophia inria fr>

3.6 PlantFrame (3D reconstruction of plant architecture)

Section contents

In this section, we introduce

the PlantFrame vocabulary that we use throughout *vplants* and give a series of examples.

3.6.1 The problem setting

PlantFrame is a method to compute the geometry of each organ of a Plant Architecture. Geoemtrical data is associated to some vertices of the architecture (aka **MTG**). But often, geometrical information is missing on some vertex. Constraints have to be solved to compute missing values.

The stages of the Pla

1. Insert a scale at the axis level.
2. Project all the constraints at the finer scale.
3. Apply different *Knowledge Sources* (i.e. KS) on the MTG to compute the values at some nodes.
4. Solve the constraints.
5. Visualise the geom-

e-try using a 3D Turtle.

3.6.2 Wher are the data?

The *tutorial* package comes with a few datasets. The data are in *share/data/Plan* directory from the root.

```
>
↪>
↪>
↪␣
↪impor
↪opena
↪mtg
>
↪>
↪>
↪␣
↪from␣
↪opena
↪deplo
↪share
↪data␣
↪impor
↪share
↪data
```

(continues on next page)

(continued from previous page)

```
>>> impor
vplan
tutor
>>> data
= share
data(
tutor
'Plan
'
```

3.6.3 Visua of a digitized Tree

First, we load the digitized Walnut `noylum2.mtg`

```
>>> from openalea.mtg import *
>>> g = MTG(data/
'noylum2.mtg
')
```

(continues on next page)

(continued from previous page)

Then, a file containing a set of default geometric parameters is loaded to build a DressingData (`walnut.drf`)

```
>>> drf = data/'walnut.drf'
>>> dressing_data = dresser.dressing_data_from_file(drf)
```

Another solution is to create the default parameters directly

::

```
>
↪>
↪>
↪
↪dress
↪data
↪=
↪plant
↪Dress
```

Geometric parameters are missing. How to compute them? Use the PlantFrame, a geometric solver working on multiscale tree structure.

Create the solver

and solve the prob- lem

```
>
↪>
↪>
↪
↪pf
↪=
↪plantframe
↪PlantFrame
↪

↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪TopDiamete
↪'TopDia',

↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪
↪DressingDa
↪= dressing_data)
```

(continues on next page)

(continued from previous page)

Visualise the plant in 3D

```
>>> pf.plot(gc=Tr
```

3.6.4 Simp visualisation of a monopodial plant

First, we load the MTG monopodial_ mtg

```
>>> from openalea.mtg import *
>>> g = MTG(data/'monopodia plant.mtg')
```

```
>>> def coloring(m vertex):
        try:
```

(continues on next page)

(continued from previous page)

```
mtg.property('diam')[vertex]
return "g"
except: return "r"

>>> def legend(mtg, vertex):
try:
```

(continues on next page)

(continued from previous page)

```
                return "diam: "+str(mtg.property('diam')[vertex]
            except: return "diam: NA"
>>> def label(mtg, vertex):
            return mtg.label(vert
>>> g.plot(roots
        node=dict(
```

(continues on next page)

(continued from previous page)

```
user/image
```

```
>
↪>
↪>
↪␣
↪def␣
↪legend(mtg
↪␣
↪vertex):
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪try:
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
```

(continues on next page)

(continued from previous page)

```
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪except:␣
↪return␣
↪
↪"diam:␣
↪NA
↪"

>
↪>
↪>
↪␣
↪g.
↪plot(roots
↪␣
↪node=dict(
↪␣
↪label=labe
↪␣
↪legend=leg
↪␣
↪prog=
↪"dot
↪")
```

user/image

The mtg *monopodial_plant.mtg* is loaded. To draw it, just run:

```
>
↪>
↪>
↪␣
↪pf␣
↪=␣
↪plantframe
↪PlantFrame
↪␣
↪TopDiamete
↪'diam
↪')
```

(continues on next page)

(continued from previous page)

```
>
↪>
↪>
↪␣
↪pf.
↪plot()
```

You can also define a function to compute the diameter:

```
>>> def diam(v):
        d = g.node(v).diam
        return d/10. if d else None
>>> pf = plantframe
PlantFrame
 TopDiamete
>>> pf.plot()
```

The **diameter** is defined for each vertex of the MTG. To take into account the diameter, we have to define a visitor function.

```
diam = g.property('diam')

def visitor(g, v, turtle):
    if g.edge_type(v) == '+':
        angle = 90 if g.order(v) == 1 else 30
        turtle.down(angle
```

(continues on next page)

(continued from previous page)

```
turtle.setId(v)
if v in diam:
turtle.setWidth(d
2.
)
turtle.F(10)
turtle.rollL()

pf = plantframe
PlantFrame
pf.plot(g,
visitor=vi
```

3.7 Using MTG within VisuAlea

Nodes have been implemented within VisuAlea so as to manipulate MTG files. See OpenAlea wiki for examples and details about VisuAlea.

The following dataflows illustrates how MTG files can be manipulated within VisuAlea.

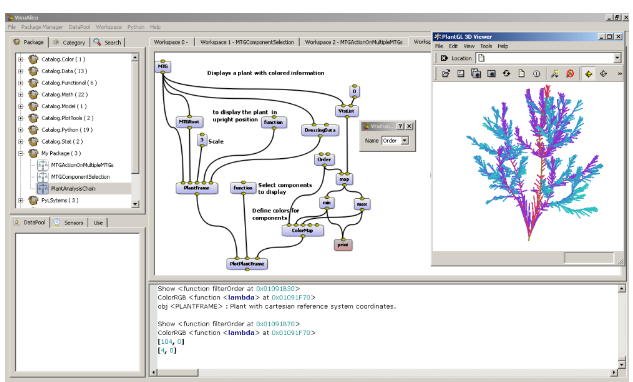


Fig. 13: MTG manipulation within VisuAlea

3.8 File syntax

Todo: revise the entire document to check tabulation of the examples

Contents

3.8.1 General conventions

In general, words can be separated by any combination of whitespace characters (SPACE, TAB, EOL). In certain files, TABs or EOL are meaningful (e.g. MTG coding files), and therefore are not considered as a whitespace character in these files.

Comments may be introduced anywhere in a file using the sharp sign #, meaning that the rest of the line is a comment. In some files, block comments can be introduced by bracketing the comment text with (# and #).

Files used by AML

can be located anywhere in the UNIX hierachical file system, provided the user can access them. All references to files from within a file or from AML must be given explicitly. References to files must always be made relatively to the location where the reference is made.

In various files, user-defined names must be given to objects, attributes, etc. Unless specified otherwise, names always consist of strings of alphanumeric characters (including underscore '_') starting by a non-numeric character. A name may start by an underscore. Some names correspond to reserved keywords. Since reserved keywords always start in AMAPmod with uppercase letter, it is advised, though not mandatory, to define user-defined names starting with lowercase letter to avoid name collision.

3.8.2 MTGs

Coding strategy

A plant multiscale topology is represented by a string of characters (see 3.2.2). The string is made up of a series of labels representing plant components (a label is made up of an alphabetic character in A-Z,a-z and a numeric index) and of symbols representing either the physical relationships between the components. Character '/' is used for decomposition relationship (see next paragraph), '+' is used for branching relationship and '<' for successor relationship. For example:

```
/I1<I2<I3<I4+I5<I6
```

is a string representing 6 components with labels I1, I2, I3, I4, I5, I6. I1 to I4 are a sequence of components defining an axis which bears a second axis made up of the sequence of components I5 and I6. In this string every component is connected with at most one subsequent component (either by a '<' or by a '+')

As illustrated by this example, the name of an entity is built by concatenating the consecutive entity labels encountered while moving along the plant structure from the plant basis to the considered entity. For example, consider the decomposition of a plant in terms of axes. Assume this plant is made of 3 axes: axis A1 bears axis A2, which itself bears axis A3. Then, the respective names of the axes are:

```
/A1
/A1+A2
/A1+A2+A3
```

Symbol '+' refers to the type of connection between A1 and A2, A2 and A3 respectively. Now, consider another plant considered at the scale of growth units. A growth unit U90 bears a growth unit U91 which is itself followed on the same axis by U92. The respective names of these growth units are

```
/U90
/U90+U91
/U90+U91<U92
```

These two examples illustrate how to define the name of plant entities when only one scale of description is considered. When several scales are considered, this strategy can be extended as explained in section 3.2.2.

Assume for instance that axis A1 of the previous example is composed of 3 consecutive growth units and that axis A2 is borne by the second growth units of A1. Then the name of A2 is defined as

```
/A1/U1<U2+A2
```

Relative names

Every name of an entity is thus the concatenation of a series of pairs (relation symbol,label)

: name = relation label relation label relation label relation. . . label relation label

Let us consider any prefix p of a name n of an entity x of the plant, made of a series of pairs (relation label). According to the recursive construction of entity names, this prefix defines the name of an entity y on the path from the plant basis to the entity with name n. The name of x has thus the form:

```
n = p m
```

where m is a series label relation . . . label relation. Entity x has absolute name n. Alternatively we can say that x has relative name m with respect position p, i.e. relatively to entity y.

Examples

```
/S1/A1/E1+A3 has relative name /A1/E1+A3 in position /S1
/S1/A1/E3+A1/E4+S1/U2/E3+U1/E5+U4/E4 has relative name +U1/E5+U4/E4 in position /S1/A1/E3+A1/E4+S1/U2/E3
```

Coding files

The coding of a plan

- a header which contains a description of the coding parameters,

- the code of the plant architecture.

The header contains

- the set of all entity classes used in the MTG description,
- a detailed description of the topologi-

cal properties of these classes,

- and the set of all attributes used for any entity in the plant description.

In a MTG coding file, TABs are meaningful. They correspond to column separators. Consequently, a MTG coding file should be edited using a spreadsheet editor. If a sharp '#' is inserted on a line, every character until the next TAB on the same line is considered as a comment and is not interpreted.

Header

General parameter section

For historical reasons, two forms of plant architecture coding have been developed, denoted FORM-A et FORM-B. FORM-A is the most general and should be employed. FORM-B is available for ascendant compatibility with former coding forms employed in the AMAP laboratory [Rey et al, 97]. Whatever the coding form used the plant built by AMAPmod is the same. The form of the coding language must be specified in the coding file by specifying either FORM-A or FORM-B following the keyword CODE, in the next column, for example : CODE: FORM-A This definition is mandatory.

Class definition section

Classes must then be declared. This

is done in a section beginning with keyword CLASSES. Then a line is defined for each class of the MTG. The first column, entitled SYMBOL, contains the symbolic character denoting a class used in the MTG. This symbol most be an alphabetic character (either upper or lower-case letter). Two classes either at identical or different scales must have different symbolic characters. The second column, entitled SCALE, represents the scale at which this class appears in the MTG. There are no a priori limitation related to the number of classes, however, these must be consecutive integer greater or equal to 0. Scale i, i>1, can only appear if scale i-1 has appeared before.

```
CLASSES
SYMBOL SCALE DECOMPOSIT INDEXATION DEFINITION
$ 0 FREE FREE IMPLICIT
P 1 CONNECTED FREE
```

(continues on next page)

(continued from previous page)

```
U␣
↪␣
↪␣
↪2␣
↪␣
↪␣
↪
↪<-
↪LINEAR␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪EXPLICIT
I␣
↪␣
↪␣
↪2␣
↪␣
↪␣
↪
↪<-
↪LINEAR␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪EXPLICIT
E␣
↪␣
↪␣
↪3␣
↪␣
↪␣
↪NONE␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪IMPLICIT
```

Symbol $ rep-re-sent the entire database and is defined by definition at scale 0. Keyword DECOMPOSITION defines the types of decomposition that can have a vertex (i.e. a plant constituent) : CONNECTED, LINEAR, <-LINEAR, +-LINEAR, FREE, NONE. Key word CONNECTED means that the decomposition graph of a vertex at the next scale is connected. Keyword LINEAR means that the decomposition graph of a vertex at the next scale is a linear sequence of vertices. Besides, if this all the constituents of this sequence are connected using a single type of edge (respectively < or +), then keyword <-LINEAR et +-LINEAR can respectively be used. Keyword FREE allows any type of decomposition structure while keyword NONE, specifies that the components of a unit must not be decomposed. Column INDEXATION is not used. Column DEFINITION must be filled with value EXPLICIT if any entity of that class has feature values (i.e. attributes). IMPLICIT should be used otherwise.

This section is mandatory.

Topological constraints section

Topological constraints are described in the next section, beginning with keyword DESCRIPTION. Here, each line defines for a pair of classes at the same scale one allowed type of connection. It contains 4 columns, LEFT, RIGHT, RELTYPE, and MAX. For any class in column LEFT, the column RIGHT defines a list of class (appearing at the same scale) which can be connected to it using a connection of type RELTYPE. The maximum number of connections of type RELTYPE that can be made on an entity from column is defined in column MAX. If column MAX contains a question mark '?', the number of connections is not bounded. If a class does not appear in the column LEFT, then entities of this class cannot be connected to other entities in the MTG.

```
DESCRIPTION:
LEFT RIGHT RELTYPE MAX
U U,I + ?
U U,I < 1
I I + ?
E E <
```

(continues on next page)

(continued from previous page)

```
E
↪
↪
↪E
↪
↪
↪+
↪
↪
↪1
```

Let us resume on the example from the above CLASS section with its DESCRIPTION section. Since class P does not appear in the left column, a P cannot be connected to any other entity at scale 1, e.g. to any other P. Entities of type U can be connected to entities of either type I or U, for any of the connection types < et +. An entity of type U can be connected by relation + to any number of Us or Is. However, they can only be connected by relation < to at most one entity of either type U or I. Entities of type I cannot be connected by relation < to any type of entity, while they can be connected to other I's by relation +. At scale 3, any E can be connected to only one other E by either relation + or <. This section is mandatory but can contain no topology description.

Attribute section

The third and last part of the header

contains a list of names defining the features that can be attached to plant entities and their types. This part begins with keyword FEATURES. Thelist of names appears in column NAME and the corresponding types in column TYPE. The name of an attribute might be either a reserved keyword (see a list below) or a user-defined name. The types of attributes can be INT (integer), REAL (real number), STRING (string of characters from {A...Za...z-+. /} and which are bounded to 14 characters max), DD/MM, DD/MM/YY, MM/YY, DD/MM-TIME, DD/MM/YY-TIME (Dates), GEOMETRY (geometric objects defined in a .geom file), APPEARANCE (appearance objects defined in a .app file), OBJECT (general object defined in generic type of file).

```
FEATURES:
NAME                TYPE

Alias               STRING
Date                DD/MM
NbEl                INT
State               STRING
flowerNb            INT
len                 INT
```

(continues on next page)

(continued from previous page)

```
TopDiameter
↪REAL
geom
↪
↪
↪
↪GEOMETRY
↪
↪
↪
↪geom1.
↪geom
appear
↪
↪APPEARANCE
↪
↪material.
↪app
```

Certain names of attributes are reserved keywords. They all start by an uppercase letter. If they appear in the feature list, they must be in the same order as in the following description. Alias, of type STRING (formerly ALPHA), must come first if used. It allows the user to define aliases for plant entities to simplify some code strings. Date, is used to define the observation date of an entity. NbEl (NumBer of ELements), defines the number of components on any entity at the next scale. Length is the length of an entity. BottomDiameter et TopDiameter respectively define the bottom and top stretching values of a tapered transformed that is applied to the geometric symbol representing this entity (for branch segments associated with cylinder as a basic geometrc model, this defines cone frustums). State of type STRING defines the state of an entity at the time of observation. This state can be D (Dead), A (Alive), B (Broken) , P (Pruned), G (Growing), V (Vegetative), R (Resting), C (Completed), M (Modified). These letters can be combined to form a string of characters, provided they consistent with one another. Such state descriptions are checked during the parsing of the MTG and possible inconsistencies are detected.

This

section is mandatory but can contain no features.

Coding section

The section containing the code of a MTG starts by keyword MTG.

The next line contains a list of column names. In the first column, the keyword TOPO indicates that this column and the next unlabelled column are reserved for the topological code. On the same line, all the names that appear in the FEATURE section of the header must appear, in the same order, one column after the other, starting with the first feature name in a column sufficiently far from the TOPO column to leave enough space for the topological code (see examples below).

The topological code must necessarily start by a '/' like in:

```
/P1/A1...
```

It can spread on all the columns before the first feature column.

Since entity names have a nested

definition, a plant description can be made on a single line. However, if one wants to declare feature values attached to some entity, the plant code must be interrupted after the label of this entity, attributes must be entered on the same line in corresponding columns, and the plant code must continue at the next line.

Note that in the current implementation of the parser, an entity which has no features uses obviously 0 bytes of memory for recording features, however, assuming that the total number of features is F, if an entity has at least one feature value defined, it uses a constant space F*14 bytes to record its feature (whatever the actual number of features defined for this entity).

Example

Here is an example of a coding file corresponding to plant illustrated on Figure 4-1:

```
CODE:
FORM-
A
CLASSES:
SYMBOL
SCALE
DECOMPOSIT
INDEXATION
DEFINITION

$
0
FREE
FREE
IMPLICIT
P
1
CONNECTED
```

(continues on next page)

(continued from previous page)

```
A␣
↪␣
↪␣
↪2␣
↪␣
↪␣
↪
↪<-
↪LINEAR␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪EXPLICIT
S␣
↪␣
↪␣
↪2␣
↪␣
↪␣
↪CONNECTED␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪EXPLICIT
U␣
↪␣
↪␣
↪3␣
↪␣
↪␣
↪NONE␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪IMPLICIT
DESCRIPTION:
LEFT␣
↪␣
↪␣
↪␣
↪RIGHT␣
↪␣
↪␣
↪RELTYPE␣
↪MAX
A␣
↪␣
↪␣
↪A,
↪S␣
↪+␣
↪␣
↪␣
```

(continues on next page)

(continued from previous page)

```
U␣
↪␣
↪␣
↪U␣
↪␣
↪␣
↪
↪<␣
↪␣
↪␣
↪1
U␣
↪␣
↪␣
↪U␣
↪␣
↪␣
↪+␣
↪␣
↪␣
↪?
↪
FEATURES:
NAME␣
↪␣
↪␣
↪␣
↪TYPE
MTG:
TOPO
/
↪P1/
↪A1
/
↪P1/
↪A1/
↪U1
↪<U2+S1
/
↪P1/
↪A1/
↪U1
↪<U2+S2
/
↪P1/
↪A1/
↪U1
↪<U2+A1
/
↪P1/
↪A1/
↪U1
↪<U2+A1/
↪U1
↪<U2+S1
/
↪P1/
↪A1/
↪U1
↪<U2
↪<U3+S1
```

(continues on next page)

(continued from previous page)

```
/
↪P1/
↪A1/
↪U1
↪<U2
↪<U3+A2
/
↪P1/
↪A1/
↪U1
↪<U2
↪<U3+A2/
↪U1
↪<U2
↪<U3+A3
/
↪P1/
↪A1/
↪U1
↪<U2
↪<U3+A2/
↪U1
↪<U2
↪<U3+A3/
↪U1+S1
/
↪P1/
↪A1/
↪U1
↪<U2
↪<U3+A2/
↪U1
↪<U2
↪<U3
↪<U4
/
↪P1/
↪A1/
↪U1
↪<U2
↪<U3
↪<U4
```

In this example, certain names use frequently the same prefix which can be long (this bit of code contains 225 characters). We are going to introduce successively different strategies in order to simplify this first coding scheme.

The first simplification consists of giving a name (alias) to an entity name which is used frequently in the name of others.

```
# before the header is identical to the previous one
FEATURES:
NAME TYPE
```

(continues on next page)

(continued from previous page)

```
Alias␣
↪␣
↪␣
↪ALPHA
MTG:
TOPO␣
↪␣
↪␣
↪␣
↪Alias
/
↪P1/
↪A1␣
↪␣
↪A1
(A1)/
↪U1
↪<U2+S1␣
↪␣
↪␣
↪Branch1
(A1)/
↪U1
↪<U2+S2
(A1)/
↪U1
↪<U2+A1
(A1)/
↪U1
↪<U2+A1/
↪U1
↪<U2+S1
(A1)/
↪U1
↪<U2
↪<U3+S1
(A1)/
↪U1
↪<U2
↪<U3+A2␣
↪␣
↪␣
↪␣
↪A2
(A2)/
↪U1
↪<U2
↪<U3+A3
(A2)/
↪U1
↪<U2
↪<U3+A3/
↪U1+S1␣
↪␣
↪Branch2
(A2)/
↪U1
↪<U2
↪<U3
↪<U4
```

(continues on next page)

(continued from previous page)

```
/
↪P1/
↪A1/
↪U1
↪<U2
↪<U3
↪<U4
```

An alias can be associated with a given entity by defining its name in column Alias. This name can then be reused in the topological section by enclosing it between parentheses. If an alias is used as a prefix of an entity, the code of this entity must be given relatively to this alias. For entity A2, for instance, we can see that its name is /U1<U2<U3+A2 relatively to position A1 which is an alias for /P1/A1. The absolute name of A2is thus, /P1/A1/U1<U2<U3+A2. The code part of this file has now a size of 173 characters, i.e. 78% of the initial code.

The code of the MTG can be further simplified. We can avoid completely the repetition of bit of codes. Assume that entity y has a code of the form XY where X represents the code of some entity x. For example X is /P1/A1 and Y is /U1<U2<U3+A2 in the previous example. If X already appears in column of the topological section, then we may consider that if subsequently Y appears at a different line, but shifted to the right by one column, then Y is actually follows X which is thus its prefix. Then Y is a relative name with respect to position X. In our example, this leads to

```
/P1/A1                # code of x
/P1/A1/U1<U2<U3+A2    # code of y
```

which becomes

```
#column1      #column2
/P1/A1                  # code de x
```

(continues on next page)

(continued from previous page)

```
␣
↪␣
↪␣
↪␣
↪/
↪U1
↪<U2
↪<U3+A2␣
↪␣
↪␣
↪␣
↪
↪#␣
↪code␣
↪de␣
↪y
```

The fact that the code of y is shifted one column to the right, allows us to interpret /U1<U2<U3+A as the continuation of /P1/A1 leading to the absolute name /P1/A1/U1<U2<U3+A2 which is actually the code of y.

By applying this new rule on the complete previous example we obtain the following code

```
MTG:
TOPO
#column1
#column2
#column3
#column4
#column5
/
P1/
A1
/
U1
<U2
+S1
```

(continues on next page)

(continued from previous page)

```
+S2
+A1/
U1
<U2+S1
<U3
+A2/
U1
<U2
<U3
+A3/
U1+S1
```

(continues on next page)

(continued from previous page)

```
<U4
<U4
```

Now the number of characters used in the code is now 63 and corresponds to 28% of the initial code. However, this compressed code raises two new problems. The first problem is that the number of columns necessary has greatly increased. The second is that it is difficult to recognise the structural organisation of the plant in the way the code displays it.

To address both problem, a new syntactic notation is introduced. Each time a relative code starts with character ^ in a given cell, the current relative code must be interpreted with respect to the position whose code is the latest code defined in the same column just above the current cell. Using the ^ notation:

```
MTG:
TOPO
/P1/A1
^/U1<U2
                +S1
                +S2
                +A1/U1<U2+S1
```

(continues on next page)

(continued from previous page)

```
^
↪
↪<U3
␣
↪␣
↪␣
↪␣
↪+A2/
↪U1
↪<U2
↪<U3
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪+A3/
↪U1+S1
␣
↪␣
↪␣
↪␣
↪^
↪
↪<U4
^
↪
↪<U4
```

Here the number of columns used is equal to the number of orders in the plant (i.e. 3), which bounds the total number of columns required and best reflects in the code the botanical structure of the plant. Entities of order i are defined in column i which greatly improves the code leagibility. Finaly, the number of characters used is 69, i.e. 31% of the initial extended code.

In some cases, a series of consecutive entities must be coded, which produces long lines of code just as this one:

```
A1/U87<U88<U89<U90<U91<U92<U93+A2
```

Such a line can be abbreviated by using the << sign

```
A1/
↪U87
↪<
↪<U93+A2
```

U87<<U93 is a syntactic shorthand for U87<U89<U90

Symbol ++ is defined similarly: U87++U93 is a shorthand for U87+U89+U90

Note that in such cases, the entities implicitly defined cannot have attributes: for instance, the code:

```
TOPO            diam    flowers
/A1/U87<<U93    10.3    2
```

Means that an axis A1 is made of a series of 7 growth units, labelled from U87 to U93 and that U93 has a diameter of 10.3 and bears 2 flowers. In some cases, we want to express that the attributes are shared by all entities. This can be expressed as follows:

```
TOPO            diam    flowers
```

(continues on next page)

(continued from previous page)

```
/
↪A1/
↪U87
↪<.
↪
↪<U93␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪1
```

which means that every growth units from U87 to U93 has exactly 1 flower. Notation +.+ is defined similarly.

Here follows the complete code of plant of Figure 4-1:

```
CODE:␣
↪␣
↪␣
↪FORM-
↪A
CLASSES:
SYMBOL␣
↪␣
↪SCALE␣
↪␣
↪␣
↪DECOMPOSIT
↪␣
↪␣
↪INDEXATION
↪␣
↪DEFINITION

↪$␣
↪␣
↪␣
↪0␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪IMPLICIT
P␣
↪␣
↪␣
↪1␣
↪␣
↪␣
↪CONNECTED␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪IMPLICIT
A␣
↪␣
↪␣
↪2␣
↪␣
↪␣
↪
↪<-
↪LINEAR␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪EXPLICIT
```

(continues on next page)

(continued from previous page)

```
S␣
↪␣
↪␣
↪2␣
↪␣
↪␣
↪CONNECTED␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪EXPLICIT
U␣
↪␣
↪␣
↪3␣
↪␣
↪␣
↪NONE␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪IMPLICIT
DESCRIPTION:
LEFT␣
↪␣
↪␣
↪␣
↪RIGHT␣
↪␣
↪␣
↪RELTYPE␣
↪MAX
A␣
↪␣
↪␣
↪A,
↪S␣
↪+␣
↪␣
↪␣
↪?
↪
U␣
↪␣
↪␣
↪U␣
↪␣
↪␣
↪
↪<␣
↪␣
↪␣
↪1
```

(continues on next page)

(continued from previous page)

```
U␣
↪␣
↪␣
↪U␣
↪␣
↪␣
↪+␣
↪␣
↪␣
↪?
↪
FEATURES:
NAME␣
↪␣
↪␣
↪␣
↪TYPE
MTG:
TOPO
/
↪P1/
↪A1
^
↪/
↪U1
↪<U2
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪+S1
␣
↪␣
↪␣
↪␣
↪+S2
␣
↪␣
↪␣
↪␣
↪+A1/
↪U1
↪<U2+S1
^
↪
↪<U3
␣
↪␣
↪␣
↪␣
↪+A2/
↪U1
↪<
↪<U3
```

(continues on next page)

(continued from previous page)

```
↪+A3/
↪U1+S1
↪<U4
^
↪<U4
```

Examples of coding strategies in different classical situations

Non linear growth units

Until now we have only used linear growth units, i.e. entities whose decomposition in a linear set of entities. It is possible to define branching growth-units, which are not a part of an axis. The plant illustrated in Figure 4-2 illustrates such non-linear entities.

```
CODE:␣
↪␣
↪␣
↪FORM-
↪A
CLASSES:
SYMBOL␣
↪␣
↪SCALE␣
↪␣
↪␣
↪DECOMPOSIT
↪␣
↪␣
↪INDEXATION
↪␣
↪DEFINITION

↪$␣
↪␣
↪␣
↪0␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪IMPLICIT
F␣
↪␣
↪␣
↪1␣
↪␣
↪␣
↪CONNECTED␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪IMPLICIT
U␣
↪␣
↪␣
↪2␣
↪␣
↪␣
↪NONE␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪IMPLICIT
```

(continues on next page)

(continued from previous page)

```
DESCRIPTION:
LEFT␣
↪␣
↪␣
↪␣
↪RIGHT␣
↪␣
↪␣
↪RELTYPE␣
↪MAX
F␣
↪␣
↪␣
↪F␣
↪␣
↪␣
↪+␣
↪␣
↪␣
↪?
↪
F␣
↪␣
↪␣
↪F␣
↪␣
↪␣
↪
↪<␣
↪␣
↪␣
↪1
U␣
↪␣
↪␣
↪U␣
↪␣
↪␣
↪+␣
↪␣
↪␣
↪?
↪
U␣
↪␣
↪␣
↪U␣
↪␣
↪␣
↪
↪<␣
↪␣
↪␣
↪1
FEATURES:
NAME␣
↪␣
↪␣
↪␣
↪TYPE
```

(continues on next page)

(continued from previous page)

```
MTG:
TOPO
/
↪F1/
↪U1
↪<U2
␣
↪␣
↪␣
↪␣
↪+U3
↪<U4
↪<F2/
↪U1
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪+U2
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪+U3
␣
↪␣
↪␣
↪␣
↪+U5+F3/
↪U1
```

Sympodial plants

Sympodial plants often contain apparent axes made up of series of modules (or axes). At a macroscopic scale, the plant is described in terms of apparent axes connected to one another (Figure 4-3) depict a typical sympodial plant:

```
CODE: FORM-A
CLASSES:
SYMBOL SCALE DECOMPOSITION INDEXATION DEFINITION
$ 0 FREE FREE IMPLICIT
S 1 +-LINEAR FREE IMPLICIT
```

(continues on next page)

(continued from previous page)

```
A
2
<-
LINEAR
FREE
IMPLICIT
A
2
<-
LINEAR
FREE
IMPLICIT
DESCRIPTION:
LEFT
RIGHT
RELTYPE
MAX
S
S
+
?
A
A,
a
+
1
```

(continues on next page)

(continued from previous page)

```
A␣
↪␣
↪␣
↪A␣
↪␣
↪␣
↪+␣
↪␣
↪␣
↪;
↪1
FEATURES:
NAME␣
↪␣
↪␣
↪␣
↪TYPE
MTG:
TOPO
/
↪S1
^
↪/
↪A1+A2
␣
↪␣
↪␣
↪␣
↪+S1
␣
↪␣
↪␣
↪␣
↪^
↪/
↪a1+A2+A3
^
↪+A3
␣
↪␣
↪␣
↪␣
↪+S1
␣
↪␣
↪␣
↪␣
↪^
↪/
↪a1+A2
^
↪+A4+A5
```

Note in this example the role of ^ which enables us to preserve the structure of the plant into the code itself. Indeed, apparent axes appear in columns corresponding to their apparent order.

Dominant axes

Similarly, dominant axes in a plant can be identified using macroscopic units Figure 4-4 illustrates how to code dominant axes:

```
CODE:
FORM-
A
CLASSES:
SYMBOL
SCALE
DECOMPOSIT
INDEXATION
DEFINITION
$
0
FREE
FREE
IMPLICIT
D
1
+-
LINEAR
FREE
IMPLICIT
A
2
NONE
FREE
IMPLICIT
```

(continues on next page)

(continued from previous page)

```
DESCRIPTION:
LEFT       RIGHT    RELTYPE    MAX
D          D        +          ?
A          A        +          ?

FEATURES:
NAME       TYPE

MTG:
TOPO
/D1
^/A1++A7
                  +D1/A1
                         +D3/A1+A2
```

(continues on next page)

(continued from previous page)

```
^+A2++A6
+D2/A1
+D4/A1+A2
^+A2++A5
```

Whorls and supra-numerary buds

Whorls and supra-numerary buds can be encoded in several ways. One possible solution is to use the

multiscale property a a MTG as illustrated in the following example.

```
CODE:␣
↪␣
↪␣
↪FORM-
↪A
CLASSES:
SYMBOL␣
↪␣
↪SCALE␣
↪␣
↪␣
↪DECOMPOSIT
↪␣
↪␣
↪INDEXATION
↪␣
↪DEFINITION

↪$␣
↪␣
↪␣
↪0␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪IMPLICIT
U␣
↪␣
↪␣
↪1␣
↪␣
↪␣
↪
↪<-
↪LINEAR␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪IMPLICIT
E␣
↪␣
↪␣
↪2␣
↪␣
↪␣
↪
↪<-
↪LINEAR␣
↪␣
↪␣
↪␣
↪FREE
```

(continues on next page)

(continued from previous page)

```
V␣
↪␣
↪␣
↪3␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪EXPLICIT
N␣
↪␣
↪␣
↪4␣
↪␣
↪␣
↪NONE␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪IMPLICIT
DESCRIPTION:
LEFT␣
↪␣
↪␣
↪␣
↪RIGHT␣
↪␣
↪␣
↪RELTYPE␣
↪MAX
U␣
↪␣
↪␣
↪U␣
↪␣
↪␣
↪+␣
↪␣
↪␣
↪?
↪
U␣
↪␣
↪␣
↪U␣
↪␣
↪␣
↪
↪<␣
↪␣
↪␣
↪1
```

(continues on next page)

(continued from previous page)

```
E
 
 
E
 
 

<
 
 
1
E
 
 
E
 
 
+
 
 
?

FEATURES:
NAME
 
 
 
TYPE
MTG:
TOPO
/
U90
^
/
E1

 
 
 
/
V1

 
 
 
 
 
 
 
/
N1+U91

 
 
 
 
 
 
 
/
N2+U91
```

(continues on next page)

(continued from previous page)

```
␣
↪␣
↪␣
↪␣
↪/
↪V2
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪/
↪N1+U91
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪/
↪N2+U91
␣
↪␣
↪␣
↪␣
↪/
↪V3
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪/
↪N1+U91
^
↪
↪<E2
␣
↪␣
↪␣
↪␣
↪/
↪V1
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪/
↪N1+U91
```

(continues on next page)

(continued from previous page)

```
␣
↪␣
↪␣
↪␣
↪/
↪V2
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪/
↪N1+U92
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪/
↪N2+U92
␣
↪␣
↪␣
↪␣
↪/
↪V3
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪/
↪N1+U92
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪/
↪N2+U92
^
↪
↪<E3␣
↪.
↪.
↪.
↪
```

(continues on next page)

(continued from previous page)

Entities E denote internodes. Each internode contains a whorl, whose elements are denoted by class V. Each V can itself be decomposed into several supranumerary positions, denoted by class N. Then on each position, a growth unit (class U) can be described. Note that within a whorl E, V positions are not connected to one another. They are simply considered as one part of the whorl. This is also true for supra-numerary positions.

Plant growth observation

Plant growth can be observed and described using MTGs. To this end, observation dates are recorded. If some entity is observed at several dates, the new values of its attributes at different dates are recorded on consecutive lines where the topological code of the entity is not repeated but rather replaced by a star symbol '*'.

```
CODE:
FORM-
A
CLASSES:
SYMBOL
SCALE
DECOMPOSIT
INDEXATION
DEFINITION
$
0
FREE
FREE
IMPLICIT
P
1
CONNECTED
FREE
IMPLICIT
U
2
NONE
FREE
```

(continues on next page)

(continued from previous page)

```
DESCRIPTION:
LEFT RIGHT RELTYPE MAX
U U < 1
U U + ?

FEATURES:
NAME TYPE
Date DD/MM/YY
MTG:
TOPO Date
/P1
^/U1<U2 08/06/00
```

(continues on next page)

(continued from previous page)

```
*␣
↪␣
↪␣
↪19/
↪06/
↪00
*␣
↪␣
↪␣
↪30/
↪06/
↪00
*␣
↪␣
↪␣
↪10/
↪07/
↪00
␣
↪␣
↪␣
↪␣
↪+U1
↪<U2␣
↪␣
↪19/
↪06/
↪00
␣
↪␣
↪␣
↪␣
↪*␣
↪␣
↪␣
↪30/
↪06/
↪00
␣
↪␣
↪␣
↪␣
↪*␣
↪␣
↪␣
↪10/
↪07/
↪00
^
↪
↪<U3␣
↪␣
↪␣
↪␣
↪19/
↪06/
↪00
```

(continues on next page)

(continued from previous page)

```
*
↪
↪
↪30/
↪06/
↪00

↪
↪
↪
↪+U1
↪<
↪<U3
↪
↪
↪
↪
↪19/
↪06/
↪00

↪
↪
↪
↪*
↪
↪
↪
↪
↪
↪
↪30/
↪06/
↪00

↪
↪
↪
↪*
↪
↪
↪
↪
↪
↪
↪10/
↪07/
↪00

↪
↪
↪
↪
↪<U4
↪
↪
↪
↪
↪30/
↪06/
↪00
```

(continues on next page)

(continued from previous page)

```
␣
↪␣
↪␣
↪␣
↪*␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪10/
↪07/
↪00
```

Branching units located on the bearer according their height from the basis

In some cases, it is useful to use the index of an entity label to record infor-

ma-
tion.

Here, the index of the entity is used to denote the position of an element is used to record the height of this position with respect to the basis of the corresponding axis.

```
CODE:
↪
↪
↪FORM-
↪A
CLASSES:
SYMBOL
↪
↪SCALE
↪
↪
↪DECOMPOSIT
↪
↪
↪INDEXATION
↪
↪DEFINITION

↪$
↪
↪
↪0
↪
↪
↪FREE
↪
↪
↪
↪FREE
↪
↪
↪
↪IMPLICIT
X
↪
↪
↪1
↪
↪
↪FREE
↪
↪
↪
↪FREE
↪
↪
↪
↪IMPLICIT
L
↪
↪
↪2
↪
↪
↪NONE
↪
↪
↪
↪FREE
↪
```

(continues on next page)

(continued from previous page)

```
DESCRIPTION:
LEFT 
↪ 
↪ 
↪ 
↪RIGHT 
↪ 
↪ 
↪RELTYPE 
↪MAX
X 
↪ 
↪ 
↪X 
↪ 
↪ 
↪+ 
↪ 
↪ 
↪?
↪
FEATURES:
NAME 
↪ 
↪ 
↪ 
↪TYPE
MTG:
TOPO 
↪ 
↪ 
↪ 
↪ 
↪ 
↪ 
↪ 
↪Alias
/
↪X90
 
↪ 
↪ 
↪ 
↪/
↪L50+X91
 
↪ 
↪ 
↪ 
↪/
↪L100+X91 
↪ 
↪ 
↪A91
 
↪ 
↪ 
↪ 
↪/
↪L123+X92
```

(continues on next page)

(continued from previous page)

```
(A91)␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪
↪#␣
↪Back␣
↪to␣
↪axis␣
↪borne␣
↪at␣
↪position␣
↪L100
␣
↪␣
↪␣
↪␣
↪/
↪L10+X92
␣
↪␣
↪␣
↪␣
↪/
↪L25+X92
␣
↪␣
↪␣
↪␣
↪.
↪.
↪.
↪
```

Description of a plant from the extremities

On some plants, it is easier to described branches starting from the bud of the stem on proceeding downward to the stem basis. This is the case for instance, for large trees where biological markers of growth, nodes, growth unit limits, sympodial module, etc., are more leagible near the branch extremities. Here follows a strategy to code the plant in such a case.

```
CODE: FORM-A
CLASSES:
SYMBOL SCALE DECOMPOSIT INDEXATION DEFINITION
$ 0 FREE FREE IMPLICIT
P 1 CONNECTED FREE IMPLICIT
```

(continues on next page)

(continued from previous page)

```
U␣
↪␣
↪␣
↪2␣
↪␣
↪␣
↪
↪<-
↪LINEAR␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪EXPLICIT
E␣
↪␣
↪␣
↪3␣
↪␣
↪␣
↪NONE␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪EXPLICIT
DESCRIPTION:
LEFT␣
↪␣
↪␣
↪␣
↪RIGHT␣
↪␣
↪␣
↪RELTYPE␣
↪MAX
U␣
↪␣
↪␣
↪U␣
↪␣
↪␣
↪+␣
↪␣
↪␣
↪?
↪
U␣
↪␣
↪␣
↪U␣
↪␣
↪␣
↪
↪<␣
↪␣
↪␣
↪1
```

(continues on next page)

(continued from previous page)

```
E
E
<
1
E
E
+
1
FEATURES:
NAME
TYPE
MTG:
TOPO
/
P1
^
/
U86

/
E2+U87
^

<U87
^

<U88
^

<U89

/
E10+U89

/
E4+U90
```

(continues on next page)

(continued from previous page)

```
/
E3+U90
/
E1+U90
^
<U90
/
E6+U90
/
E3+U90
/
E2+U91
/
E1+U91
^
<U91
/
E7+U91
# 
7th 
internode 
from 
the 
apex 
U91
/
E3+U92
#
```

(continues on next page)

(continued from previous page)

```
␣
↪␣
↪␣
↪␣
↪/
↪E2+U92␣
↪
↪#␣
↪2nd␣
↪internode␣
↪from␣
↪the␣
↪apex␣
↪U91
```

The entities of the stem must be ordered in the file bottom-up (cf. the firt column where growth units U have increasing indexes). However, the positions within a given growth unit is given from top down to the basis of this growth unit. In addition, if the user wants to enter the stem entities (here growth units) from the top down to the basis of the stem, (s)he can use a laptop computer and insert new growth units (say U90) before the ones already observed at the top (say U91).

A second solution consists of using a FORM-B code. Using this more specific code allows you to enter the entities of the stem from top to basis (see first column).

```
CODE:
FORM-B
CLASSES:
SYMBOL SCALE DECOMPOSIT
INDEXATION
DEFINITION
$ 0 FREE FREE IMPLICIT
P 1 CONNECTED FREE IMPLICIT
```

(continues on next page)

(continued from previous page)

```
U␣
↪␣
↪␣
↪2␣
↪␣
↪␣
↪
↪<-
↪LINEAR␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪EXPLICIT
E␣
↪␣
↪␣
↪3␣
↪␣
↪␣
↪NONE␣
↪␣
↪␣
↪␣
↪FREE␣
↪␣
↪␣
↪␣
↪EXPLICIT
DESCRIPTION:
LEFT␣
↪␣
↪␣
↪␣
↪RIGHT␣
↪␣
↪␣
↪RELTYPE␣
↪MAX
U␣
↪␣
↪␣
↪U␣
↪␣
↪␣
↪+␣
↪␣
↪␣
↪?
↪
U␣
↪␣
↪␣
↪U␣
↪␣
↪␣
↪
↪<␣
↪␣
↪␣
↪1
```

(continues on next page)

(continued from previous page)

```
E␣
↪␣
↪␣
↪E␣
↪␣
↪␣
↪
↪<␣
↪␣
↪␣
↪1
E␣
↪␣
↪␣
↪E␣
↪␣
↪␣
↪+␣
↪␣
↪␣
↪1
FEATURES:
NAME␣
↪␣
↪␣
↪␣
↪TYPE
MTG:
TOPO
/
↪P1
^
↪/
↪U91
␣
↪␣
↪␣
↪␣
↪/
↪E2+U92␣
↪
↪#␣
↪2nd␣
↪internode␣
↪from␣
↪the␣
↪apex␣
↪U91
␣
↪␣
↪␣
↪␣
↪/
↪E3+U92␣
↪
↪#␣
↪3rd␣
↪internode␣
↪from␣
↪the␣
↪apex␣
↪U91
```

(continues on next page)

(continued from previous page)

```
    /E7+U91   # 7th internode from the apex U91
^/U90
    /E1+U91
    /E2+U91
    /E3+U90
    /E6+U90
^<U89
    /E1+U90
    /E3+U90
    /E4+U90
```

(continues on next page)

(continued from previous page)

```
␣
↪␣
↪␣
↪␣
↪/
↪E10+U89
^
↪
↪<U88
^
↪
↪<U87
␣
↪␣
↪␣
↪␣
↪/
↪E7+U87
```

Reference Manual - STAT module 4.2 Dressing files

3.8.3 Dress Files (.drf)

The dressing data are the default data that are used to define the geometric models associated with geometric entities and to compute their geometric parameters when inference algorithms cannot be applied. These data are basically constant values (see the table below) and may be redefined in the dressing file. If no dressing file is defined, default (hard-coded) values are used (see table below). The dressing file .drf , if it exists in the current directory, is always used as a default dressing file.

The dressing data entries can be subdivided into 3 categories (any of these categories can be omitted).

Definition of basic geometric models associated with plant components

A graphic model can be associated with a component in the following way (all keywords are in boldface characters):

1. First, a set of all the basic geometric models of interest must be defined. This is done by specifying a file containing the geometric description of these models (for a definition of the syntax of geometric models, refer to the annexe section):

```
Geometr
= 
file1
geom
Geometr
= 
.
.
/
.
.
/
file2
geom
```

The effect of these lines is to load the geometric models that are defined in files file1.geom and in file ../../file2.geom. Each geometric model defined is these files is associated with a symbolic name. If the same symbolic name is found twice during the loading operation, an error is generated and should be corrected.

2. Any symbolic name (like internode) can then be associated with a component using the class of the component as follows:

```
Class I = inter
```

(continues on next page)

(continued from previous page)

where I corresponds to a class name. This means that all the vertices of class I will have a geometry defined by the geometric model internode. Note that class I does not necessarily correspond to a valid class of a MTG (however, it should be a alphabetic letter in a-z,A-Z).

Alternatively, to allow for ascendant compatibility with previous versions of AMAPmod, it is possible to directly refer to geometric models defined in .smb files. In this case, the set of geometric models corresponds to the files contained in directory SMBPath and a geometric model can be loaded in AMAPmod by identifying a smb file in this directiry. This is done as follows in the dressing file:

```
SMBPath = ../../databases/SMBFiles
SMBModel internode2 = nentn105
SMBModel leaf3 = oakleaf
```

Here, geometric models internode2 and are respectively associated with polygon files **nentn105.smb** and **oakleaf.smb** which are both located in directory **../../databases/SMBFiles**.

Like exposed above, SMB geometric models can then be associated with vertex classes:

```
Class J = internode2
Class F = leaf3
```

Then, global shapes can be defined for branches. This is done using the feature “category” defined for branches. The category of a branch is defined by the category of its first component. Note that the category may depend on the scale at which a branch is considered. For each category, the user can associate a 3 dimensional shape as a 3D bezier curve. The shape of the branch is then fit to the general shape associated with its category.

Assuming a set

of Bezier curves are specified in a file beziershapes.crv (for example), we can associate branch categories with the Bezier curves using the following notation:

```
BranchPatter
↪= 
↪.
↪.
↪/
↪Curves/
↪beziershap
↪crv
Form 
↪category 
↪= 
↪curve2
```

Note that the file beziershapes.crv is included, using a path relative to the directory where the .drf file itself is located. Alternatively, an absolute filename could be given. The structure of the file beziershapes.crv is discribed in section 4.4.

Definition of virtual elements

Components that don't appear in an MTG description can be added to a MTG (e.g. leaves, flowers or fruits). It is possible to define these new symbols as follows:

```
Geometry = file1.geom

SMBPath = SMBFiles
SMBModel leaf = feuil13

Class L = leaf
Class A = apple
```

(continues on next page)

(continued from previous page)

```
Class B = apricot_flower

LeafClass = L
FlowerClass = B
FruitClass = A
```

A symbol L (a character) is defined and is associated with geometric model leaf.

The two last lines associate respectively virtual leaf and fruit components with the geometric model associated with classes L and A.

Definition of defaults parameters

The value of default parameters used to compute geometric models can be changed in the dressing file. Here follows the complete list of these parameters illustrated on an example:

```
# Default geometric units (these quantities are used
# to divide every value of the corresponding type before use)

LengthUnit = 10
DiameterUnit = 100
AlphaUnit = 1

DefaultAlpha = 30
DefaultTeta = 0
DefaultPhi = 90
```

(continues on next page)

(continued from previous page)

```
DefaultPsi = 180

DefaultCateg = 3
DefaultTrunk = 0

Alpha = Relative
Phyllotaxy = 2/5

DefaultEdge = PLUS  # used for plantframe constructi

# Redefiniti of default values of the geometric models of

# components (here component S)

MinLength S = 1000
MinTopDiamet S = 20
MinBottomDia S = 20
```

(continues on next page)

(continued from previous page)

```
↪#
↪Redefiniti
↪of
↪default
↪values
↪of
↪the
↪geometric
↪models
↪of

↪#
↪virtual
↪components

LeafLength
↪=
↪1
LeafTopDiame
↪=
↪2
LeafBottomDi
↪=
↪2
LeafAlpha
↪=
↪0
LeafBeta
↪=
↪0

FruitLength
↪=
↪1
FruitTopDiam
↪=
↪1
FruitBottomD
↪=
↪1
FruitAlpha
↪=
↪0
FruitBeta
↪=
↪0

FlowerLength
↪=
↪10
FlowerTopDia
↪=
↪5
FlowerBotton
↪=
↪5
```

(continues on next page)

(continued from previous page)

```
FlowerAlpha␣
↪=␣
↪180
FlowerBeta␣
↪=␣
↪0

DefaultTrunk
↪=␣
↪0
DefaultDista
↪=␣
↪1000
NbPlantsPerI
↪=␣
↪6


↪#␣
↪Colors␣
↪for␣
↪interpolat

MediumThresh
↪=␣
↪1
MediumThresh
↪=␣
↪0
MediumThresh
↪=␣
↪0
MinThreshold
↪=␣
↪0
MinThreshold
↪=␣
↪0
MinThreshold
↪=␣
↪1
MaxThreshold
↪=␣
↪0
MaxThreshold
↪=␣
↪1
MaxThreshold
↪=␣
↪0
```

Any of these key-words can

be omit- ted in the dress- ing file. If omit- ted, a pa- ram- e- ter takes a default value, hard-coded into AMAPmod. The default values are defined in the following table: i

Name of the parameter	Description
SMBPath	Plant where SMB files are recorded
LengthUnit	Unit used to divide all the length data
AlphaUnit	Unit used to divide all the insertion angle
AzimutUnit	Unit used to divide all the angles
DiametersUnit	Unit used to divide all the diameters
DefaultEdge	Type of edge used to reconstruct a connected MTG
DefaultAlpha	Default insertion angle (value in degrees with respect to the horizontal plane).
Phillotaxy	Phyllotaxic angle (given in degrees) or in number of turns over number of leaves for this number of turns
Alpha	Nature of the insertion angle.
DefaultTeta	Default first Euler angle
DefaultPhi	Default second Euler angle
DefaultPsi	Default third Euler angle
MinLength S	Default length for elements whose class is S.
MinTopDiameter S	Default top diameter for elements whose class is S.
MinBotDiameter S	Default bottom diameter for elements whose class is S.
DefaultTrunkCategory	Default category for elements of the plant trunk. The default category of the other axes is their (botanical
DefaultDistance	Distance between the trunk of two plants when several plants are vizualized at a time
NbPlantsPerLine	Number of plants per line when several plants are vizualized at a time
MediumThresholdGreen	Green component of the color used for the values equal to the MediumThreshold (see command Plot on a
MediumThresholdRed	Idem for the red component.
MediumThresholdBlue	Idem for the blue component.
MinThresholdGreen	Green component of the color used for the values equal to the MinThreshold (see command Plot on a PL
MinThresholdRed	Idem for the red component.
MinThresholdBlue	Idem for the blue component.
MaxThresholdGreen	Green component of the color used for the values equal to the MaxThreshold (see command Plot on a PL
MaxThresholdRed	Idem for the red component
MaxThresholdBlue	Idem for the blue component.
Whorl	Number of virtual symbols per node
LeafClass	Class used for a leaf
LeafLength	Length of the leaf
LeafTopDiameter	Top diameter of the leaf

Table 1 – continued from previous pa

Name of the parameter	Description
LeafBottomDiameter	Bottom diameter of the leaf
LeafAlpha	Insertion angle of a leaf
LeafBeta	Azimuthal angle of a leaf (w.r.t its carrier)
FruitClass	Class used for a fruit
FruitLength	Length of the fruit
FruitTopDiamter	Top diameter of the fruit
FruitBottomDiameter	Bottom diameter of the fruit
FruitAlpha	Insertion angle of a fruit
FruitBeta	Azimuthal angle of a fruit (w.r.t its carrier)
FlowerClass	Class used for a flower
FlowerLength	Length of the flower
FlowerTopDiameter	Top diameter of the flower
FlowerBottomDiameter	Bottom diameter of the flower
FlowerAlpha	Insertion angle of a flower
FlowerBeta	Azimuthal angle of a flower (w.r.t its carrier)

Example of dressing file

see aml example

3.8.4 Curve Files (.crv)

A curve file contains the specification of Bezier curves. It has the following general structure:

n
curve1
k_1
$x_1\ y_1\ z_1$
$\ldots$
$xk1yk1zk1$
curve2
$k2$
$x1y1z1$
$\ldots$
$xk2yk2zk2$
$\ldots$
curven
kn
$x1y1z1$
$\ldots$
$xknyknzkn$

where n, k1, kn, are integers and curve1, curve2, $\ldots$, curven are strings of characters. All coordinates are real numbers.

documentation status:: in

progress

Documentation adapted from the AMAPmod user manual version 1.8.

3.9 Lsyst and MTGs

Author
Thomas Cokelaer <Thomas.Coke

Contents

Let us start from the following L-system

```
angle = 20

context().turtle.setAngleIn

Axiom: X

def EndEach(ls

    print lstring

derivation length: 7
production:
X --> F[+X]F[-X]+X
F --> FF

homomorphism

F --> SetWidth(
5) F
```

(continues on next page)

(continued from previous page)

```
endlsystem
```

3.9.1 General usage

First, import some modules

```
import openalea.lpy as lpy
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import time
from openalea.plantgl.all import *
from openalea.mtg.io import lpy2mtg, mtg2lpy, axialtree2
mtg2axialt
from openalea.mtg.aml import *
```

(continues on next page)

(continued from previous page)

and then, read the lsystem:

```
>>> l = lpy.Lsystem('example.lpy')
```

execute it:

```
>>> tree = l.iterate()
F[+X]F[-X]+X...
```

and plot the results:

```
>>> l.plot(tree)
```

that you can save into

a PNG file as follows:

```
>>> Viewer.frameGL.saveImage('output.png', 'png')
```

3.9.2 Extra information from the lsystem

axiom

Get the axiom

into an axial tree object:

```
l.axiom
```

context

context gets the production rules, group, iteration number

```
>>> context = l.context()
>>> context.getGroup()
0
>>> context.getIterati
6
```

last iteration

If the Lsystem finished normally, the last iteration must be equal to the derivation length.

```
>>> l.getLastIte
6
>>> l.derivation
7
```

3.9.3 Activ the lsystem with makecurrent

Todo: what is this for ?

```
l.
↪makeCurren
```

3.9.4 Execu the lsys- tem

animate

In order to run the lsystem step by step with a plot refreshing at each step, use animate(), for which you may provide a minimal time step between each iteration.

```
l.
↪animate(st
```

where step is in seconds. Note that you may still set the animation to false using:

```
Viewer.animation(
```

iterate

Run all steps until the end:

```
>>> l.iterate()
```

or step by step:

```
>>> l.iterate(1)
F[+X]F[-X]+X
AxialTree(F[
X]+X)
>>> tree = l.iterate(1)
F[+X]F[-X]+X
>>> l.iterate(1,
1,
```

(continues on next page)

(continued from previous page)

```
FF[+F[+X]F[-
↪X]+X]FF[-
↪F[+X]F[-
↪X]+X]+F[+X
↪X]+X
F[+X]F[-
↪X]+X
FF[+F[+X]F[-
↪X]+X]FF[-
↪F[+X]F[-
↪X]+X]+F[+X
↪X]+X
True
```

Note: When using *iterate()* with 1 argument, the Lsystem is run from the beginning again. To keep track of a previous run, 3 arguments are required. In such case, the first is used only to keep track of the number of iteration, that is stored in l.getLastIterationNb(), the second argument is then the number of iteration required and the 3d argument is the axiom (i.e., the previous AxialTree output).

3.9.5 Trans the lstring/axia into MTG and vice-versa

```
>
↪>
↪>
↪␣
↪axialtree␣
↪=␣
↪l.
↪iterate()
```

lpy2mtg method

axialtree2mtg method

```
>
↪>
↪>
↪␣
↪axialtree␣
↪=␣
↪l.
↪iterate()
```

(continues on next page)

(continued from previous page)

```
F[+X]F[-X]+X...
>>> scales = {'F':1, 'X':1}
>>> mtg1 = axialtree2 scales, l.sceneInter None)
```

and come back to the original one:

```
>>> tree1 = mtg2axialt scales, None, axialtree)
```

(continues on next page)

(continued from previous page)

```
>
↪>
↪>
↪␣
↪assert␣
↪str(axialt
True
```

mtg2lpy and lpy2mtg method

```
>
↪>
↪>
↪␣
↪mtg2␣
↪=␣
↪lpy2mtg(ax
↪␣
↪l)
>
↪>
↪>
↪␣
↪tree2␣
↪=␣
↪lpy2mtg(mt
↪␣
↪l,
↪␣
↪axialtree)
>
↪>
↪>
↪␣
↪assert␣
↪str(axialt
True

>
↪>
↪>
↪␣
↪scene␣
↪=␣
↪l.
↪sceneInter
```

if no lsys-tem is avail-abe, you

may use generateScene(axialtr… using from openalea.lpy import generateScene

3.10 Bibl…

3.11 Clas… and Interfaces

Each data structure implement a set of specific interface. These interfaces define the name of the methods, their semantic, and sometime their complexity.

See openalea.mtg.mtg

3.12 Algo

The openalea.mtg.mtg package provides data structure as well as algorithms. This section introduces the reader to the main algorithms and shows simple examples.

CHAPTER 4

Reference

4.1 MTG - Multi-scale Tree Graph

4.1.1 Overview

`openalea.mtg.mtg.MTG` (*filename=''*, *has_date=False*)

A Multiscale Tree Graph (MTG) class.

MTGs describe tree structures at different levels of details, named scales. For example, a botanist can described plants at different scales :

- at scale 0, the whole scene.
- at scale 1, the individual plants.
- at scale 2, the axes of each plants.
- at scale 3, the growth units of each axis, and so on.

Each scale can have a label, e.g. :

- scale 1 : P(lant)
- scale 2 : A(xis)
- sclae 3 : U(nit of growth)

Compared to a classical tree, `complex()` can be seen as `parent()` and `components()` as `children()`. An element at `scale()` N belongs to a `complex()` at `scale()` N-1 and has `components()` at scale N+1:

- /P/A/U (decomposition is noted using "/")

Each scale is itself described as a tree or a forest (i.e. set of trees), e.g.:

- /P1/P2/P3
- A1+A2<A3
- ...

4.1.2 Iterating over vertices

MTG.root	Return the tree root.
MTG.vertices([scale])	Return a list of the vertices contained in an MTG.
MTG.nb_vertices([scale])	Returns the number of vertices.
MTG.parent(vtx_id)	Return the parent of *vtx_id*.
MTG.children(vtx_id)	returns a vertex iterator
MTG.nb_children(vtx_id)	returns the number of children
MTG.siblings(vtx_id)	returns an iterator of vtx_id siblings.
MTG.nb_siblings(vtx_id)	returns the number of siblings
MTG.roots([scale])	Returns a list of the roots of the tree graphs at a given scale.
MTG.complex(vtx_id)	Returns the complex of *vtx_id*.
MTG.components(vid)	returns the components of a vertex
MTG.nb_components(vid)	returns the number of components
MTG.complex_at_scale(vtx_id, scale)	Returns the complex of *vtx_id* at scale *scale*.
MTG.components_at_scale(vid, scale)	returns a vertex iterator

4.1.3 Adding and removing vertices

MTG.__init__([filename, has_date])	Create a new MTG object.
MTG.add_child(parent[, child])	Add a child to a parent.
MTG.insert_parent(vtx_id[, parent_id])	Insert parent_id between vtx_id and its actual parent.
MTG.insert_sibling(vtx_id1[, vtx_id2])	Insert a sibling of vtk_id1.
MTG.add_component(complex_id[, component_id])	Add a component at the end of the components
MTG.add_child_and_complex(parent[, child, ...])	Add a child at the end of children that belong to an other complex.
MTG.add_child_tree(parent, tree)	Add a tree after the children of the parent vertex.
MTG.clear()	Remove all vertices and edges from the MTG.

4.1.4 Some usefull functions

simple_tree(tree, vtx_id[, nb_children, ...])	Generate and add a regular tree to an existing one at a given vertex.
random_tree(mtg, root[, nb_children, ...])	Generate and add a random tree to an existing one.
random_mtg(tree, nb_scales)	Convert a tree into an MTG of *nb_scales*.
colored_tree(tree, colors)	Compute a mtg from a tree and the list of vertices to be quotiented.
display_tree(tree, vid[, tab, labels, edge_type])	Display a tree structure.
display_mtg(mtg, vid)	Display an MTG

4.1.5 All

class openalea.mtg.mtg.**MTG**(*filename=''*, *has_date=False*)

Bases: openalea.mtg.tree.PropertyTree

A Multiscale Tree Graph (MTG) class.

MTGs describe tree structures at different levels of details, named scales. For example, a botanist can described plants at different scales :

- at scale 0, the whole scene.
- at scale 1, the individual plants.
- at scale 2, the axes of each plants.
- at scale 3, the growth units of each axis, and so on.

Each scale can have a label, e.g. :

- scale 1 : P(lant)
- scale 2 : A(xis)
- sclae 3 : U(nit of growth)

Compared to a classical tree, `complex()` can be seen as `parent()` and `components()` as `children()`. An element at `scale()` N belongs to a `complex()` at `scale()` N-1 and has `components()` at scale N+1:

- /P/A/U (decomposition is noted using "/")

Each scale is itself described as a tree or a forest (i.e. set of trees), e.g.:

- /P1/P2/P3
- A1+A2<A3
- …

AlgHeight (*v1*, *v2*)

Algebraic value defining the number of components between two components.

This function is similar to function *Height(v1, v2)* : it returns the number of components between two components, at the same scale, but takes into account the order of vertices *v1* and *v2*.

The result is positive if *v1* is an ancestor of *v2*, and negative if *v2* is an ancestor of *v1*.

Usage

```
AlgHeight(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG.
- v2 (int) : vertex of the active MTG.

Returns int

If *v1* is not an ancestor of *v2* (or vise versa), or if *v1* and *v2* are not defined at the same scale, an error value None is returned.

See also:

`MTG()`, `Rank()`, `Order()`, `Height()`, `EdgeType()`, `AlgOrder()`, `AlgRank()`.

AlgOrder (*v1*, *v2*)

Algebraic value defining the relative order of one vertex with respect to another one.

This function is similar to function *Order(v1, v2)* : it returns the number of +-type edges between two components, at the same scale, but takes into account the order of vertices *v1* and *v2*.

The result is positive if *v1* is an ancestor of *v2*, and negative if *v2* is an ancestor of *v1*.

Usage

```
AlgOrder(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG.
- v2 (int) : vertex of the active MTG.

Returns int

If *v1* is not an ancestor of *v2* (or vise versa), or if *v1* and *v2* are not defined at the same scale, an error value None is returned.

See also:

`MTG()`, `Rank()`, `Order()`, `Height()`, `EdgeType()`, `AlgHeight()`, `AlgRank()`.

AlgRank(*v1*, *v2*)

Algebraic value defining the relative rank of one vertex with respect to another one.

This function is similar to function *Rank(v1, v2)* : it returns the number of <-type edges between two components, at the same scale, but takes into account the order of vertices *v1* and *v2*.

The result is positive if *v1* is an ancestor of *v2*, and negative if *v2* is an ancestor of *v1*.

Usage

```
AlgRank(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG.
- v2 (int) : vertex of the active MTG.

Returns int

If *v1* is not an ancestor of *v2* (or vise versa), or if *v1* and *v2* are not defined at the same scale, an error value None is returned.

See also:

`MTG()`, `Rank()`, `Order()`, `Height()`, `EdgeType()`, `AlgHeight()`, `AlgOrder()`.

Ancestors(*v*, *EdgeType='*'*, *RestrictedTo='NoRestriction'*, *ContainedIn=None*)

Array of all vertices which are ancestors of a given vertex

This function returns the array of vertices which are located before the vertex passed as an argument. These vertices are defined at the same scale as *v*. The array starts by *v*, then contains the vertices on the path from *v* back to the root (in this order) and finishes by the tree root.

Note: The anscestor array always contains at least the argument vertex *v*.

Usage

```
g.Ancestors(v)
```

Parameters

- v (int) : vertex of the active MTG

Optional Parameters

- RestrictedTo (str): cf. *Father*
- ContainedIn (int): cf. *Father*
- EdgeType (str): cf. *Father*

Returns list of vertices's id (int)

Examples

```
>>> v # prints vertex v
78
>>> g.Ancestors(v) # set of ancestors of v at the same scale
[78,45,32,10,4]
>>> list(reversed(g.Ancestors(v))) # To get the vertices in the order from
↪the root to the vertex v
[4,10,32,45,78]
```

See also:

MTG(), *Descendants()*.

Axis(*v*, *Scale=-1*)

Array of vertices constituting a botanical axis

An axis is a maximal sequence of vertices connected by '<'-type edges. Axis return the array of vertices representing the botanical axis which the argument v belongs to. The optional argument enables the user to choose the scale at which the axis decomposition is required.

Usage

```
Axis(v)
Axis(v, Scale=s)
```

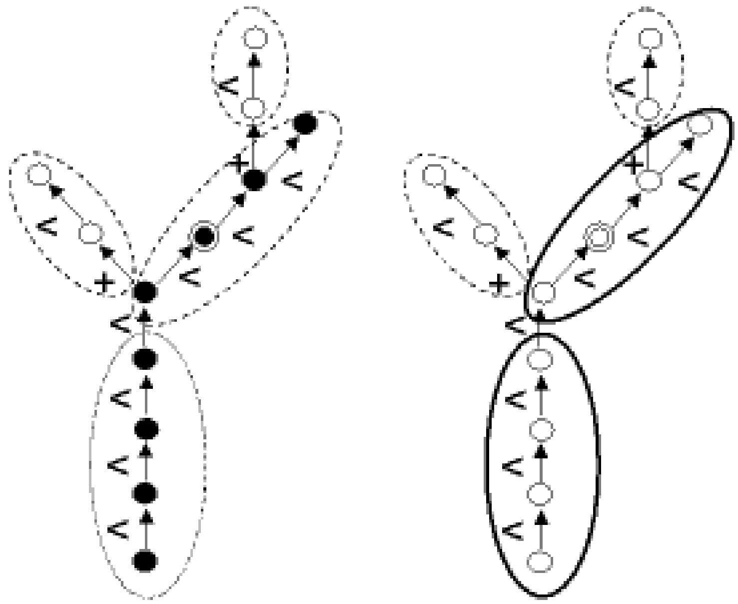
Parameters

- v (int) : Vertex of the active MTG

Optional Parameters

- Scale (str): scale at which the axis components are required.

Returns list of vertices ids



See also:

MTG(), *Path()*, *Ancestors()*.

Class (*vid*)

Class of a vertex.

The Class of a vertex are the first characters of the label. The label of a vertex is the string defined by the concatenation of the class and its index.

The label thus provides general information about a vertex and enable to encode the plant components.

The class_name may be not defined. Then, an empty string is returned.

Usage

```
>>> g.class_name(1)
```

Parameters

- *vid* (int)

Returns The class name of the vertex (str).

See also:

MTG(), *openalea.mtg.aml.Index()*, *openalea.mtg.aml.Class()*

ClassScale (*c*)

Scale at which appears a given class of vertex

Every vertex is associated with a unique class. Vertices from a given class only appear at a given scale which can be retrieved using this function.

Usage

```
ClassScale(c)
```

Parameters

- *c* (str) : symbol of the considered class

Returns int

See also:

MTG(), *Class()*, *Scale()*, *Index()*.

Complex (*v*, *Scale=-1*)

Complex of a vertex.

Returns the complex of *v*. The complex of a vertex *v* has a scale lower than *v* : *Scale(v)* - 1. In a MTG, every vertex except for the MTG root (cf. *MTGRoot*), has a uniq complex. None is returned if the argument is the MTG Root or if the vertex is undefined.

Usage

```
g.Complex(v)
g.Complex(v, Scale=2)
```

Parameters

- *v* (int) : vertex of the active MTG

Optional Parameters

- *Scale* (int) : scale of the complex

Returns Returns vertex's id (int)

Details When a scale different form Scale(v)-1 is specified using the optional parameter *Scale*, this scale must be lower than that of the vertex argument.

Todo: Complex(v, Scale=10) returns v why ? is this expected

See also:

MTG(), *Components()*.

ComponentRoots (*v*, *Scale=-1*)

Set of roots of the tree graphs that compose a vertex

In a MTG, a vertex may have be decomposed into components. Some of these components are connected to each other, while other are not. In the most general case, the components of a vertex are organized into several tree-graphs. This is for example the case of a MTG containing the description of several plants: the MTG root vertex can be decomposed into tree graphs (not connected) that represent the different plants. This function returns the set of roots of these tree graphs at scale *Scale(v)+1*. The order of these roots is not significant.

When a scale different from *Scale(v)+1* is specified using the optional argument *Scale()*, this scale must be greater than that of the vertex argument.

Usage

```
g.ComponentRoots(v)
g.ComponentRoots(v, Scale=s)
```

Parameters

- v (int) : vertex of the active MTG

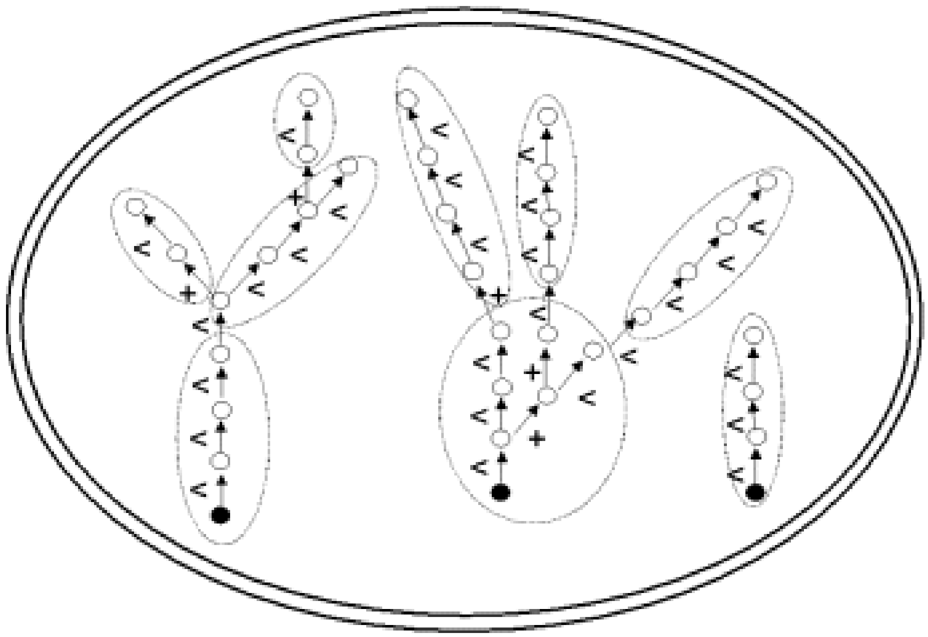
Optional Parameters

- Scale (str): scale of the component roots.

Returns list of vertices's id (int)

Examples

```
>>> v=g.MTGRoot() # global MTG root
0
>>> g.ComponentRoots(v) # set of first vertices at scale 1
[1,34,76,100,199,255]
>>> g.ComponentRoots(v, Scale=2) # set of first vertices at scale 2
[2,35,77,101,200,256]
```



See also:

MTG(), *Components()*, *Trunk()*.

Components (*v*, *Scale=-1*)

Set of components of a vertex.

The set of components of a vertex is returned as a list of vertices. If **s** defines the scale of **v**, components are defined at scale **s** + 1. The array is empty if the vertex has no components. The order of the components in the array is not significant.

When a scale is specified using optional argument :arg:Scale, it must be necessarily greater than the scale of the argument.

Usage

```
Components(v)
Components(v, Scale=2)
```

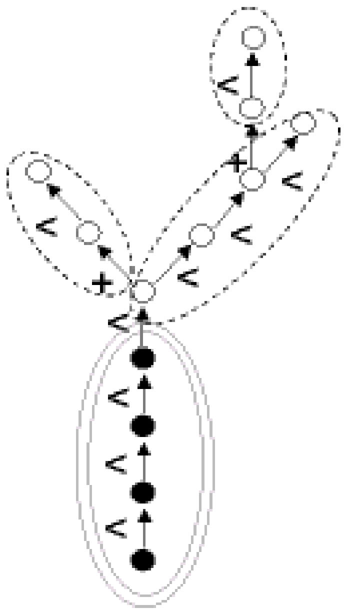
Parameters

- v (int) : vertex of the active MTG

Optional Parameters

- Scale (int) : scale of the components.

Returns list of int



See also:

MTG(), *Complex()*.

Defined (*vid*)

Test whether a given vertex belongs to the active MTG.

Usage

```
Defined(v)
```

Parameters

- v (int) : vertex of the active MTG

Returns True or False

See also:

MTG().

Descendants (*v*, *EdgeType='*'*, *RestrictedTo='NoRestriction'*, *ContainedIn=None*)

Set of vertices in the branching system borne by a vertex.

This function returns the set of descendants of its argument as an array of vertices. The array thus consists of all the vertices, at the same scale as *v*, that belong to the branching system starting at *v*. The order of the vertices in the array is not significant.

Note: The argument always belongs to the set of its descendants.

Usage

```
g.Descendants(v)
```

Parameters

- v (int) : vertex of the active MTG

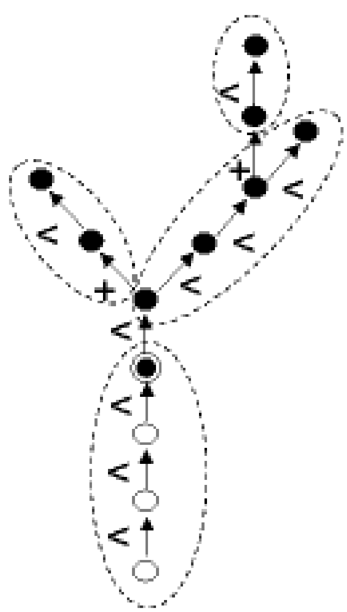
Optional Parameters

- RestrictedTo (str): cf. *Father*
- ContainedIn (int): cf. *Father*
- EdgeType (str): cf. *Father*

Returns list of int.

Examples

```
>>> v
78
>>> g.Sons(v) # set of sons of v
[78,99,101]
>>> g.Descendants(v) # set of descendants of v
[78,99,101,121,133,135,156,171,190]
```



See also:

MTG(), *Ancestors()*.

EdgeType (*v1*, *v2*)

Type of connection between two vertices.

Returns the symbol of the type of connection between two vertices (either < or +). If the vertices are not connected, None is returned.

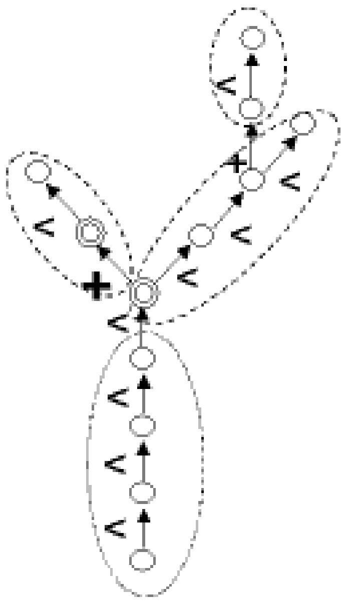
Usage

```
EdgeType(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG
- v2 (int) : vertex of the active MTG

Returns '<' (successor), '+' (branching) or *None*



See also:

MTG(), *Sons()*, *Father()*.

Extremities(*v*, *RestrictedTo='NoRestriction'*, *ContainedIn=None*)

Set of vertices that are the extremities of the branching system born by a given vertex.

This function returns the extremities of the branching system defined by the argument as a list of vertices. These vertices have the same scale as *v* and their order in the list is not signifiant. The result is always a non empty array.

Usage

```
Extremities(v)
```

Properties

- v (int) : vertex of the active MTG

Optional Parameters

- RestrictedTo (str): cf. *Father()*
- ContainedIn (int): cf. *Father()*

Returns list of vertices's id (int)

Examples

```
>>> g.Descendants(v)
[3, 45, 47, 78, 102]
```

(continues on next page)

(continued from previous page)

```
>>> g.Extremities(v)
[47, 102]
```

See also:

`MTG()`, `Descendants()`, `Root()`, `MTGRoot()`.

`Father`(*v*, *EdgeType='*'*, *RestrictedTo='NoRestriction'*, *ContainedIn=None*, *Scale=-1*)

Topological father of a given vertex.

Returns the topological father of a given vertex. And *None* if the father does not exist. If the argument is not a valid vertex, *None* is returned.

Usage

```
g.Father(v)
g.Father(v, EdgeType='<')
g.Father(v, RestrictedTo='SameComplex')
g.Father(v, ContainedIn=complex_id)
g.Father(v, Scale=s)
```

Parameters v (int) : vertex of the active MTG

Optional Parameters If no optional argument is specified, the function returns the topological father of the argument (vertex that bears or precedes to the vertex passed as an argument).

It may be usefull in some cases to consider that the function only applies to a subpart of the MTG (e.g. an axis).

The following options enables us to specify such restrictions:

- EdgeType (str) : filter on the type of edge that connect the vertex to its father.

  Values can be '<', '+', and '*'. Values '*' means both '<' and '+'. Only the vertex connected with the specified type of edge will be considered.

- RestrictedTo (str) : filter defining a subpart of the MTG where the father must be considered. If the father is actually outside this subpart, the result is *None*. Possible subparts are defined using keywords in ['SameComplex', 'SameAxis', 'NoRestriction'].

  For instance, if *RestrictedTo* is set to 'SameComplex', `Father(v)()` returns a defined vertex only if the father *f* of *v* existsin the MTG and if *v* and *f* have the same complex.

- ContainedIn (int) : filter defining a subpart of the MTG where the father must be considered. If the father is actually outside this subpart, the result is *None*.

  In this case, the subpart of the MTG is made of the vertices that composed *composite_id* (at any scale).

- Scale (int) : the scale of the considered father. Returns the vertex from scale *s* which either bears and precedes the argument.

  The scale *s* can be lower than the argument's (corresponding to a question such as 'which axis bears the internode?') or greater (e.g. 'which internodes bears this annual shoot?').

Returns the vertex id of the Father (int)

See also:

`MTG()`, `Defined()`, `Sons()`, `EdgeType()`, `Complex()`, `Components()`.

Height (*v1*, *v2=None*)
Number of components existing between two components in a tree graph.

The height of a vertex (*v2*) with respect to another vertex (*v1*) is the number of edges (of either type '+' or '<') that must be crossed when going from *v1* to *v2* in the graph.

This is a non-negative integer. When the function has only one argument *v1*, the height of *v1* correspond to the height of *v1 'with respect to the root of the branching system containing 'v1*.

Usage

```
Height(v1)
Height(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG
- v2 (int) : vertex of the active MTG

Returns int

Note: When the function takes two arguments, the order of the arguments is not important provided that one is an ancestor of the other. When the order is relevant, use function *AlgHeight*.

See also:

MTG(), Order(), *Rank()*, *EdgeType()*, *AlgHeight()*, *AlgHeight()*, *AlgOrder()*.

Index (*vid*)
Index of a vertex

The Index of a vertex is a feature always defined and independent of time (like the index). It is represented by a non negative integer. The label of a vertex is the string defined by the concatenation of its class and its index. The label thus provides general information about a vertex and enables us to encode the plant components.

Label (*vid*)
Label of a vertex.

Usage

```
>>> g.label(v)
```

Parameters

- *vid* (int) : vertex of the MTG

Returns The class and Index of the vertex (str).

See also:

MTG(), *index()*, *class_name()*

Path (*v1*, *v2*)
List of vertices defining the path between two vertices

This function returns the list of vertices defining the path between two vertices that are in an ancestor relationship. The vertex *v1* must be an ancestor of vertex *v2*. Otherwise, if both vertices are valid, then the empty array is returned and if at least one vertex is undefined, None is returned.

Usage

```
g.Path(v1, v2)
```

Parameters

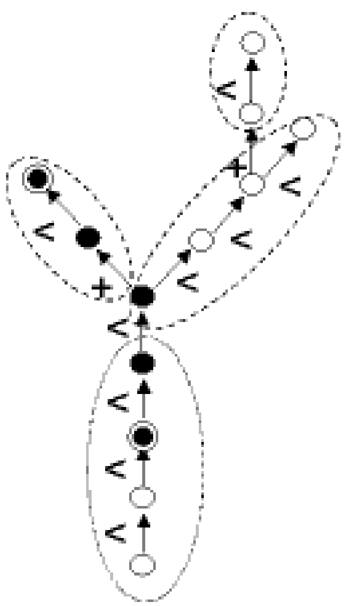
- *v1* (int) : vertex of the active MTG
- *v2* (int) : vertex of the active MTG

Returns list of vertices's id (int)

Examples

```
>>> v # print the value of v
78
>>> g.Ancestors(v)
[78,45,32,10,4]
>>> g.Path(10,v)
[10,32,45,78]
>>> g.Path(9,v) # 9 is not an ancestor of 78
[]
```

Note: *v1* can be equal to *v2*.



See also:

MTG(), *Axis()*, *Ancestors()*.

Predecessor(*v*, ***kwds*)

Father of a vertex connected to it by a '<' edge

This function is equivalent to Father(v, EdgeType-> '<'). It thus returns the father (at the same scale) of the argument if it is located in the same botanical. If it does not exist, None is returned.

Usage

```
Predecessor(v)
```

Parameters

- v (int) : vertex of the active MTG

Optional Parameters

- RestrictedTo (str): cf. *Father*
- ContainedIn (int): cf. *Father*

Returns return the vertex id (int)

Examples

```
>>> Predecessor(v)
7
>>> Father(v, EdgeType='+')
>>> Father(v, EdgeType-> '<')
7
```

See also:

MTG(), *Father()*, *Successor()*.

Rank (*v1*, *v2=None*)
Rank of one vertex with respect to another one.

This function returns the number of consecutive '<'-type edges between two components, at the same scale, and does not take into account the order of vertices v1 and v2. The result is a non negative integer.

Usage

```
Rank(v1)
Rank(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG
- v2 (int) : vertex of the active MTG

Returns *int*

If v1 is not an ancestor of v2 (or vise versa) within the same botanical axis, or if v1 and v2 are not defined at the same scale, an error value Undef id returned.

See also:

MTG(), Order(), *Height()*, *EdgeType()*, *AlgRank()*, *AlgHeight()*, *AlgOrder()*.

Root (*v*, *RestrictedTo='*'*, *ContainedIn=None*)
Root of the branching system containing a vertex

This function is equivalent to Ancestors(v, EdgeType='<')[-1]. It thus returns the root of the branching system containing the argument. This function never returns None.

Usage

```
g.Root(v)
```

Parameters

- v (int) : vertex of the active MTG

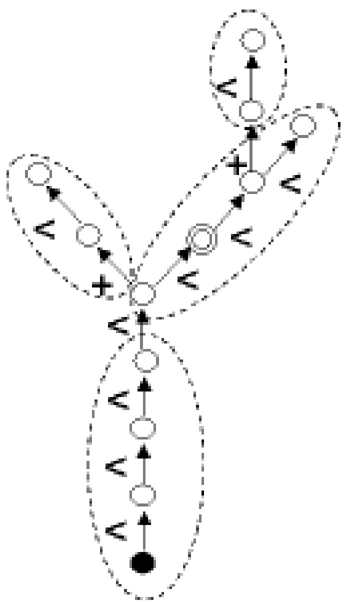
Optional Parameters

- RestrictedTo (str): cf. Father
- ContainedIn (int): cf. Father

Returns return vertex's id (int)

Examples

```
>>> g.Ancestors(v) # set of ancestors of v
[102,78,35,33,24,12]
>>> g.Root(v) # root of the branching system containing v
12
```



See also:

MTG(), *Extremities()*.

Scale (*vid*)

Returns the scale of a vertex.

All vertices should belong to a given scale.

Usage

```
g.scale(vid)
```

Parameters

- *vid* (int) - vertex identifier.

Returns The scale of the vertex. It is a positive int in [0,g.max_scale()].

Sons (*v, RestrictedTo='NoRestriction', EdgeType='*', Scale=-1, ContainedIn=None*)

Set of vertices born or preceded by a vertex

The set of sons of a given vertex is returned as an array of vertices. The order of the vertices in the array is not significant. The array can be empty if there are no son vertices.

Usage

```
g.Sons(v)
g.Sons(v, EdgeType= '+')
g.Sons(v, Scale= 3)
```

Parameters

- v (int) : vertex of the active MTG

Optional Parameters

- RestrictedTo (str) : cf. `Father()`
- ContainedIn (int) : cf. `Father()`
- EdgeType (str) : filter on the type of sons.
- Scale (int) : set the scale at which sons are considered.

Returns list(vid)

Details When the option EdgeType is applied, the function returns the set of sons that are connected to the argument with the specified type of relation.

Note: *Sons(v, EdgeType= '<')* is not equivalent to *Successor(v)*. The first function returns an array of vertices while the second function returns a vertex.

The returned vertices have the same scale as the argument. However, coarser or finer vertices can be obtained by specifying the optional argument *Scale* at which the sons are considered.

Examples

```
>>> g.Sons(v)
[3,45,47,78,102]
>>>  g.Sons(v, EdgeType= '+') # set of vertices borne by v
[3,45,47,102]
>>>  g.Sons(v, EdgeType= '<') # set of successors of v on the same axis
[78]
```

See also:

`MTG()`, `Father()`, `Successor()`, `Descendants()`.

Successor(*v*, *RestrictedTo='NoRestriction'*, *ContainedIn=None*)

Vertex that is connected to a given vertex by a '<' edge type (i.e. in the same botanical axis).

This function is equivalent to Sons(v, EdgeType='<')[0]. It returns the vertex that is connected to a given vertex by a '<' edge type (i.e. in the same botanical axis). If many such vertices exist, an arbitrary one is returned by the function. If no such vertex exists, None is returned.

Usage

```
g.Successor(v)
```

Parameters

- v1 (int) : vertex of the active MTG

Optional Parameters

- RestrictedTo (str): cf. Father
- ContainedIn (int): cf. Father

Returns Returns vertex's id (int)

Examples

```
>>> g.Sons(v)
[3, 45, 47, 78, 102]
>>> g.Sons(v, EdgeType='+') # set of vertices borne by v
[3, 45, 47, 102]
>>> g.Sons(v, EdgeType-> '<') # set of successors of v
[78]
>>> g.Successor(v)
78
```

See also:

MTG(), *Sons()*, *Predecessor()*.

Trunk (*v*, *Scale=-1*)

List of vertices constituting the bearing botanical axis of a branching system.

Trunk returns the list of vertices representing the botanical axis defined as the bearing axis of the whole branching system defined by *v*. The optional argument enables the user to choose the scale at which the trunk should be detailed.

Usage

```
Trunk(v)
Trunk(v, Scale= s)
```

Parameters

- *v* (int) : Vertex of the active MTG.

Optional Parameters

- *Scale* (str): scale at which the axis components are required.

Returns list of vertices ids

Todo: check the usage of the optional argument Scale

See also:

MTG(), *Path()*, *Ancestors()*, *Axis()*.

VtxList (*Scale=-1*)

Array of vertices contained in a MTG

The set of all vertices in the *MTG()* is returned as an array. Vertices from all scales are returned if no option is used. The order of the elements in this array is not significant.

Usage

```
>>> VtxList()
>>> VtxList(Scale=2)
```

Optional Parameters

- *Scale* (int): used to select components at a particular scale.

Returns

- list of vid

Background `MTGs()`

See also:

`MTG()`, `scale()`, `Class()`, `index()`.

add_child(*parent*, *child=None*, ***properties*)
Add a child to a parent. Child is appended to the parent's child list.

Parameters

- *parent* (int) - The parent identifier.
- ***child* (int or None) - The child identifier. If None,** an ID is generated.

Returns Identifier of the inserted vertex (child)

Returns Type int

add_child_and_complex(*parent*, *child=None*, *complex=None*, ***properties*)
Add a child at the end of children that belong to an other complex.

Parameters

- *parent*: The parent identifier.
- *child*: Set the child identifier to this value if defined.
- *complex*: Set the complex identifier to this value if defined.

Returns (vid, vid): child and complex ids.

add_child_tree(*parent*, *tree*)
Add a tree after the children of the parent vertex. Complexity have to be O(1) if tree == sub_tree()

Parameters

- **parent** – vertex identifier
- **tree** – a rooted tree

add_component(*complex_id*, *component_id=None*, ***properties*)
Add a component at the end of the components

Parameters

- *complex_id*: The complex identifier.
- *component_id*: Set the component identifier to this value if defined.

Returns The id of the new component or the component_id if given.

add_element(*parent_id*, *edge_type='/'*, *scale_id=None*)
Add an element to the graph, if vid is not provided create a new vid ??? .. warning: Not Implemented.

Parameters

- *parent_id* (int) - The id of the parent vertex
- ***edge_type* (str) - The type of relation:**
  - "/" : component (default)
  - "+" : branch
  - "<" : successor.
- ***scale_id* (int) - The id of the scale in which to** add the vertex.

Returns The vid of the created vertex

add_property (*property_name*)
Add a new map between vid and a data Do not fill this property for any vertex

backward_rewriting_traversal ()

children (*vtx_id*)
returns a vertex iterator

Parameters **vtx_id** – The vertex identifier.

Returns iter of vertex identifier

children_iter (*vtx_id*)
returns a vertex iterator

Parameters **vtx_id** – The vertex identifier.

Returns iter of vertex identifier

class_name (*vid*)
Class of a vertex.

The Class of a vertex are the first characters of the label. The label of a vertex is the string defined by the concatenation of the class and its index.

The label thus provides general information about a vertex and enable to encode the plant components.

The class_name may be not defined. Then, an empty string is returned.

Usage

```
>>> g.class_name(1)
```

Parameters

- *vid* (int)

Returns The class name of the vertex (str).

See also:

MTG(), *openalea.mtg.aml.Index()*, *openalea.mtg.aml.Class()*

clear ()
Remove all vertices and edges from the MTG.

This also removes all vertex properties. Don't change references to object such as internal dictionaries.

Example

```
>>> g.clear()
>>> g.nb_vertices()
0
>>> len(g)
0
```

clear_properties(*exclude=[]*)
Remove all the properties of the MTG.

Example

```
>>> g.clear_properties()
```

complex(*vtx_id*)
Returns the complex of *vtx_id*.

Parameters

- *vtx_id* (int) - The vertex identifier.

Returns complex identifier or None if vtx_id has no parent.

Return Type int

complex_at_scale(*vtx_id, scale*)
Returns the complex of *vtx_id* at scale *scale*.

Parameters

- *vtx_id*: The vertex identifier.
- *scale*: The scale identifier.

Returns vertex identifier

Returns Type int

component_roots(*vtx_id*)
Return the set of roots of the tree graphs that compose a vertex.

component_roots_at_scale(*vtx_id, scale*)
Return the list of roots of the tree graphs that compose a vertex.

component_roots_at_scale_iter(*vtx_id, scale*)
Return the set of roots of the tree graphs that compose a vertex.

component_roots_iter(*vtx_id*)
Return an iterator of the roots of the tree graphs that compose a vertex.

components(*vid*)
returns the components of a vertex

Parameters **vid** – The vertex identifier.

Returns list of vertex identifier

components_at_scale(*vid, scale*)
returns a vertex iterator

Parameters

- *vid*: The vertex identifier.

Returns iter of vertex identifier

components_at_scale_iter (*vid*, *scale*)
returns a vertex iterator

Parameters

- *vid*: The vertex identifier.

Returns iter of vertex identifier

components_iter (*vid*)
returns a vertex iterator

Parameters **vid** – The vertex identifier.

Returns iter of vertex identifier

copy ()
Return a copy of the graph.

Returns

- *g* (MTG) - A copy of the MTG

display (*max_scale=0*, *display_id=True*, *display_scale=False*, *nb_tab=12*, ***kwds*)
Print an MTG on the console.

Optional Parameters

- *max_scale*: do not print vertices of scale greater than max_scale
- *display_id*: display the vid of the vertices
- *display_scale*: display the scale of the vertices
- *nb_tab*: display the MTG using nb_tab columns

edge_type (*vid*)
Type of the edge between a vertex and its parent.

The different values are '<' for successor, and '+' for ramification.

edges (*scale=-1*)

Parameters

- *scale* (int) - Scale at which to iterate.

Returns Iterator on the edges of the MTG at a given scale or on all edges if scale < 0.

Returns Type iter

forward_rewriting_traversal ()

get_root ()
Return the tree root.

Returns vertex identifier

get_vertex_property (*vid*)
Returns all the properties defined on a vertex.

graph_properties ()
return a dict containing the graph properties/

Return type dict of `{property_name:data}`

has_vertex (*vid*)
Tests whether a vertex belongs to the graph.

Parameters

- *vid* (int) - vertex id to test

Returns Type bool

index(*vid*)
Index of a vertex

The Index of a vertex is a feature always defined and independent of time (like the index). It is represented by a non negative integer. The label of a vertex is the string defined by the concatenation of its class and its index. The label thus provides general information about a vertex and enables us to encode the plant components.

insert_parent(*vtx_id*, *parent_id=None*, ***properties*)
Insert parent_id between vtx_id and its actual parent. Inherit of the complex of the parent of vtx_id.

Parameters

- *vtx_id* (int): a vertex identifier
- *parent_id* (int): a vertex identifier

Returns Identifier of the inserted vertex (parent_id).

Returns Type int

insert_scale(*inf_scale=None*, *partition=None*, *default_label=None*, *preserve_order=True*)
Add a scale to MTG

Parameters

- *inf_scale* (int) - New scale is inserted between inf_scale and inf_scale-1
- *partition* (lambda v: bool) - Function defining new scale by quotienting vertices at inf_scale
- *default_label* (str) - default label of inserted vertices
- ***preserve_order* (bool) - True iif children at new scale are ordered consistently** with children at inf_scale

Returns MTG with inserted scale

Remark

- New scale is inserted in self as well.
- function partition should return True at roots of subtrees where partition changes

and False elsewhere.

insert_sibling(*vtx_id1*, *vtx_id2=None*, ***properties*)
Insert a sibling of vtk_id1. The vertex in inserted before vtx_id1.

Parameters

- *vtx_id1* (int) : a vertex identifier
- *vtx_id2* (int) : the vertex to insert

Returns Identifier of the inserted vertex (vtx_id2)

Returns Type int

insert_sibling_tree (*vid*, *tree*)
Insert a tree before the vid. vid and the root of the tree are siblings. Complexity have to be O(1) if tree comes from the actual tree (tree= self.sub_tree())

Parameters

- **vid** – vertex identifier
- **tree** – a rooted tree

is_leaf (*vtx_id*)
Test if *vtx_id* is a leaf.

Returns bool

is_valid ()
Tests the validity of the graph. Currently always returns True.

Returns Type bool

Todo Implement this function.

iter_edges (*scale=-1*)

Parameters

- *scale* (int) - Scale at which to iterate.

Returns Iterator on the edges of the MTG at a given scale or on all edges if scale < 0.

Returns Type iter

iteredges (*scale=-1*)
Iter on the edges of the tree.

label (*vid*)
Label of a vertex.

Usage

```
>>> g.label(v)
```

Parameters

- *vid* (int) : vertex of the MTG

Returns The class and Index of the vertex (str).

See also:

MTG(), *index()*, *class_name()*

max_scale ()
Return the max scale identifier.

By convention, the mtg contains scales in $[0, max\_scale]$.

Usage

```
>>> print g.max_scale()
```

Returns S, the maximum scale identifier.

Note: The complexity is $O(n)$.

See also:

scale(), *scales()*

nb_children (*vtx_id*)
returns the number of children

Parameters

- *vtx_id*: The vertex identifier.

Returns int

nb_components (*vid*)
returns the number of components

Parameters

- *vid*: The vertex identifier.

Returns int

nb_scales ()

Returns The number of scales defined in the mtg..

Returns Type int

Note: The complexity is $O(n)$.

nb_siblings (*vtx_id*)
returns the number of siblings

Returns int

nb_vertices (*scale=-1*)
Returns the number of vertices.

Usage

```
>>> g.nb_vertices()
100
>>> g.nb_vertices(scale=3)
68
```

Parameters

- *scale* (int) - Id of scale for which to count vertices.

Returns Number of vertices at *scale* or total number of vertices if scale < 0.

node (*vid, klass=None*)
Return a node associated to the vertex *vid*.

It allows to access to the properties with an object oriented interface.

Example

```
node = g.node(1)
print node.edge_type
print node.label
node.label = 'B'
print g.label(1)
```

(continues on next page)

(continued from previous page)

```
print node.parent
print list(node.children)
```

order (*v1*)
Order of a vertex in a graph.

The order of a vertex in a graph is the number of '+' edges crossed when going from *v1'to 'v2*.

If v2 is None, the order of v1 correspond to the order of v1 with respect to the root.

parent (*vtx_id*)
Return the parent of *vtx_id*.

Parameters

- *vtx_id*: The vertex identifier.

Returns vertex identifier

plot_property (*prop*, ***kwds*)
Plot properties of MTG using matplotlib

Example

```
>>> g.plot_property('length')
```

properties ()
Returns all the property maps contain in the graph.

property (*name*)
Returns the property map between the vid and the data. :returns: dict of {vid:data}

property_names ()
names of all property maps. Properties are defined only on vertices, even edge properties. return iter of names

property_names_iter ()
iter on names of all property maps. Properties are defined only on vertices, even edge properties. return iter of names

reindex (*mapping=None*, *copy=False*)
Assign a new identifier to each vertex.

This method assigns a new identifier to each vertex of the MTG. The mapping can be user defined or is implicit (*mapping*). This method modify the MTG in place or return a new MTG (*copy*).

Usage

```
>>> g.reindex()
>>> g1 = g.reindex(copy=True)
>>> mymap = dict(zip(list(traversal.iter_mtg2(g,g.root)), range(len(g))))
>>> g2 = g.reindex(mapping=mymap, copy=True)
```

Optional Parameters

- *mapping* (dict): define a mapping between old and new vertex identifiers.
- *copy* (bool) : modify the object in place or return a new MTG.

Returns

- a MTG

Background `MTGs()`

See also:

sub_mtg()

remove_property(*property_name*)
Remove the property map called property_name from the graph.

remove_scale(*scale*)
Remove all the vertices at a given scale.

The upper and lower scale are then connected.

- scale : the scale that have to be removed

Returns

- **- g** (*the input MTG modified in place.*)
- **- results** (*a list of dict*) – all the vertices that have been removed

remove_tree(*vtx_id*)
Remove the sub tree rooted on *vtx_id*.

Returns bool

remove_vertex(*vid*, *reparent_child=False*)
Remove a specified vertex of the graph and remove all the edges attached to it.

Parameters

- *vid* (int) : the id of the vertex to remove
- *reparent_child* (bool) : reparent the children of *vid* to its parent.

Returns None

replace_parent(*vtx_id*, *new_parent_id*, ***properties*)
Change the parent of vtx_id to new_parent_id. The new parent of vtx_id is new_parent_id. vtx_id and new_parent_id must have the same scale.

This function do not change the edge_type between vtx_id and its parent.

Inherit of the complex of the parent of vtx_id.

Parameters

- *vtx_id* (int): a vertex identifier
- *new_parent_id* (int): a vertex identifier

Returns None

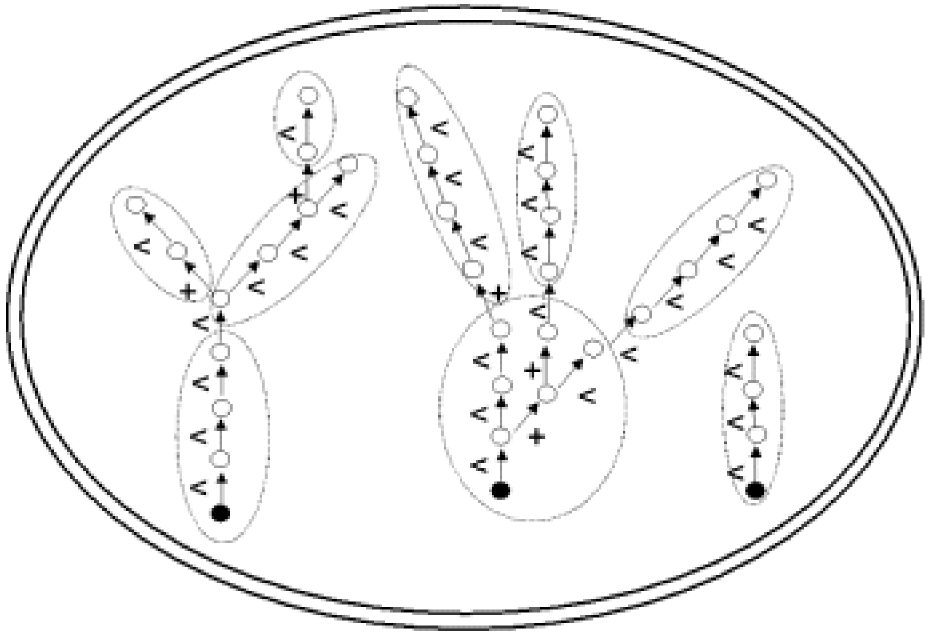
roots(*scale=0*)
Returns a list of the roots of the tree graphs at a given scale.

In an MTG, the MTG root vertex, namely the vertex *g.root*, can be decomposed into several, non-connected, tree graphs at a given scale. This is for example the case of an MTG containing the description of several plants.

Usage roots = g.roots(scale=g.max_scale()

Returns list on vertex identifiers of root vertices at a given *scale*.

Returns Type list of vid



roots_iter(*scale=0*)

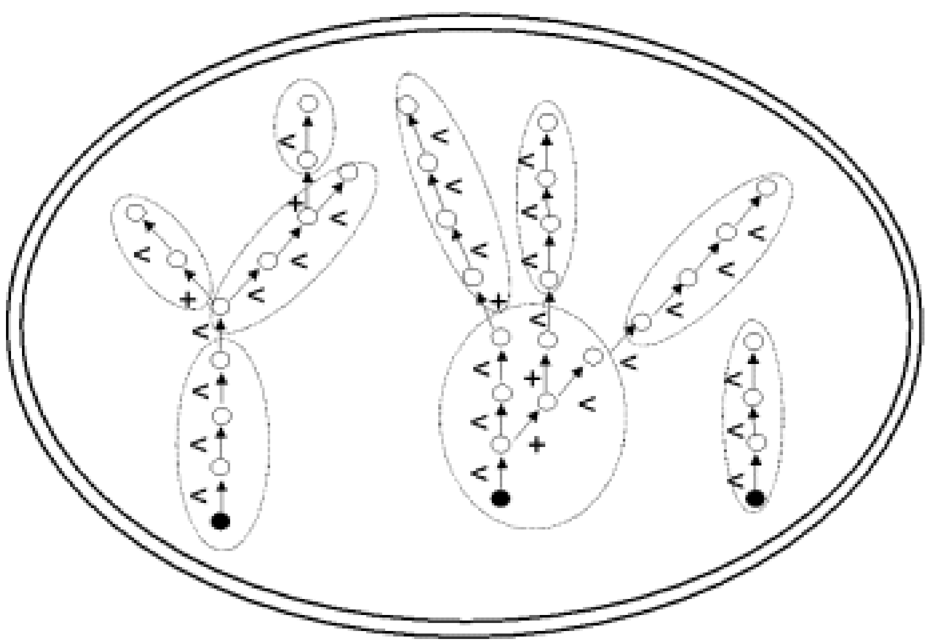
Returns an iterator of the roots of the tree graphs at a given scale.

In an MTG, the MTG root vertex, namely the vertex *g.root*, can be decomposed into several, non-connected, tree graphs at a given scale. This is for example the case of an MTG containing the description of several plants.

Usage roots = list(g.roots(scale=g.max_scale())

Returns iterator on vertex identifiers of root vertices at a given *scale*.

Returns Type iter



scale(*vid*)

Returns the scale of a vertex.

All vertices should belong to a given scale.

Usage

```
g.scale(vid)
```

Parameters

- *vid* (int) - vertex identifier.

Returns The scale of the vertex. It is a positive int in [0,g.max_scale()].

scales()

Return the different scales of the mtg.

Returns Iterator on scale identifiers (ints).

Note: The complexity is $O(n)$.

scales_iter()

Return the different scales of the mtg.

Returns Iterator on scale identifiers (ints).

Note: The complexity is $O(n)$.

set_root(*vtx_id*)

Set the tree root.

Parameters **vtx_id** – The vertex identifier.

siblings(*vtx_id*)

returns an iterator of vtx_id siblings. vtx_id is not include in siblings.

Parameters

- *vtx_id*: The vertex identifier.

Returns iter of vertex identifier

siblings_iter(*vtx_id*)

returns an iterator of vtx_id siblings. vtx_id is not include in siblings.

Parameters

- *vtx_id*: The vertex identifier.

Returns iter of vertex identifier

sub_mtg(*vtx_id*, *copy=True*)

Return the submtg rooted on *vtx_id*.

The induced sub mtg of the mtg are all the vertices which have vtx_id has a complex plus vtx_id.

Parameters

- *vtx_id*: A vertex of the original tree.

- *copy*: If True, return a new tree holding the subtree. If False, the subtree is created using the original tree by deleting all vertices not in the subtree.

Returns A sub mtg of the mtg. If copy=True, a new MTG is returned. Else the sub mtg is created inplace by modifying the original tree.

sub_tree(*vtx_id*, *copy=True*)
Return the subtree rooted on *vtx_id*.

The induced subtree of the tree has the vertices in the ancestors of vtx_id.

Parameters

- *vtx_id*: A vertex of the original tree.
- *copy*: If True, return a new tree holding the subtree. If False, the subtree is created using the original tree by deleting all vertices not in the subtree.

Returns A sub tree of the tree. If copy=True, a new Tree is returned. Else the subtree is created inplace by modifying the original tree.

vertices(*scale=-1*)
Return a list of the vertices contained in an MTG.

The set of all vertices in the MTG is returned. Vertices from all scales are returned if no scale is given. Otherwise, it returns only the vertices of the given scale. The order of the elements in this array is not significant.

Usage

```
g = MTG()
len(g) == len(list(g.vertices()))
for vid in g.vertices(scale=2):
    print g.class_name(vid)
```

Optional Parameters

- *scale* (int): used to select vertices at a given scale.

Returns Iterator on vertices at "scale" or on all vertices if scale < 0.

Returns Type list of vid

Background

See also:

children(), *components()*, *vertices_iter()*..

vertices_iter(*scale=-1*)
Return an iterator of the vertices contained in an MTG.

The set of all vertices in the MTG is returned. Vertices from all scales are returned if no scale is given. Otherwise, it returns only the vertices of the given scale. The order of the elements in this array is not significant.

Usage

```
g = MTG()
len(g) == len(list(g.vertices()))
for vid in g.vertices(scale=2):
    print g.class_name(vid)
```

Optional Parameters

- *scale* (int): used to select vertices at a given scale.

Returns Iterator on vertices at “scale” or on all vertices if scale < 0.

Returns Type iter of vid

Background

See also:

children(), *components()*.

root
Return the tree root.

Returns vertex identifier

openalea.mtg.mtg.**simple_tree**(*tree*, *vtx_id*, *nb_children=3*, *nb_vertices=20*)
Generate and add a regular tree to an existing one at a given vertex.

Add a regular tree at a given vertex id position *vtx_id*. The length of the sub_tree is *nb_vertices*. Each new vertex has at most *nb_children* children.

Parameters

- *tree*: the tree thaat will be modified
- *vtx_id* (id): vertex on which the sub tree will be added.
- *nb_children* (int) : number of children that are added to each vertex
- *nb_vertices* (int) : number of vertices to add

Returns The modified *tree*

Examples

```
g = MTG()
vid = g.add_component(g.root)
simple_tree(g, vid, nb_children=2, nb_vertices=20)
print len(g) # 22
```

See also:

random_tree(), *random_mtg()*

openalea.mtg.mtg.**random_tree**(*mtg*, *root*, *nb_children=3*, *nb_vertices=20*)
Generate and add a random tree to an existing one.

Add a random sub tree at a given vertex id position *root*. The length of the sub_tree is *nb_vertices*. The number of children for each vertex is sampled according to *nb_children* distribution. If nb_children is an interger, the random distribution is uniform between [1, nb_children]. Otherwise, you can give your own discrete distribution sampling function.

Parameters

- *mtg*: the mtg to modified
- *root* (id): vertex id on which the sub tree will be added.

Optional Parameters

- *nb_vertices*
- *nb_children* : an int or a discrete distribution sampling function.

Returns The last added vid.

Examples

```
g = MTG()
vid = g.add_component(g.root)
random_tree(g, vid, nb_children=2, nb_vertices=20)
print len(g) # 22
```

See also:

simple_tree(), *random_mtg()*

openalea.mtg.mtg.**random_mtg**(*tree*, *nb_scales*)

Convert a tree into an MTG of *nb_scales*.

Add a random sub tree at a given vertex id position *root*. The length of the sub_tree is *nb_vertices*. Each new vertex has at most *nb_children* children.

Parameters

- *mtg*: the mtg to modified
- *root* (id): vertex id on which the sub tree will be added.

Returns The last added vid.

Examples

```
g = MTG()
random_tree(g, g.root, nb_children=2, nb_vertices=20)
print len(g) # 21
```

See also:

simple_tree(), *random_tree()*

openalea.mtg.mtg.**colored_tree**(*tree*, *colors*)

Compute a mtg from a tree and the list of vertices to be quotiented.

Note: The tree has to be a real tree and not an MTG

Example

```
from random import randint, sample
g = MTG()
random_tree(g, g.root, nb_vertices=200)

# At each scale, define the vertices which will define a complex
nb_scales=4
colors = {}
colors[3] = g.vertices()
colors[2] = random.sample(colors[3], randint(1,len(g)))
colors[2].sort()
if g.root not in colors[2]:
    colors[2].insert(0, g.root)
colors[1] = [g.root]

g, mapping = colored_tree(g, colors)
```

`openalea.mtg.mtg.`**`display_tree`**(*tree*, *vid*, *tab=''*, *labels={}*, *edge_type={}*)

Display a tree structure.

`openalea.mtg.mtg.`**`display_mtg`**(*mtg*, *vid*)

Display an MTG

..todo:: Write doc.

Download the source file `../../src/mtg/mtg.py`.

4.2 High level reporting function compatible with AML

Interface to use the new MTG implementation with the old AMAPmod interface.

`openalea.mtg.aml.`**`Activate`**(*g*)

Activate a MTG already loaded into memory

All the functions of the MTG module use an implicit MTG argument which is defined as the active MTG.

This function activates a MTG already loaded into memory which thus becomes the implicit argument of all functions of module MTG.

Usage

```
>>> Activate(g)
```

Parameters

- *g*: MTG to be activated

Details When several MTGs are loaded into memory, only one is active at a time. By default, the active MTG is the last MTG loaded using function `MTG()`.

However, it is possible to activate an MTG already loaded using function `Activate()` The current active MTG can be identified using function `Active()`.

Background `MTG()`

See also:

`MTG()`

`openalea.mtg.aml.`**`Active`**()

Returns the active MTG.

If no MTG is loaded into memory, None is returned.

Usage

```
>>> Active()
```

Returns

- `MTG()`

Details When several MTGs are loaded into memory, only one is active at a time. By default, the active MTG is the last MTG loaded using function `MTG()`. However, it is possible to activate an MTG already loaded using function `Activate()`. The current active MTG can be identified using function `Active()`.

See also:

MTG(), *Activate()*.

openalea.mtg.aml.**AlgHeight**(*v1*, *v2*)

Algebraic value defining the number of components between two components.

This function is similar to function *Height(v1, v2)* : it returns the number of components between two components, at the same scale, but takes into account the order of vertices *v1* and *v2*.

The result is positive if *v1* is an ancestor of *v2*, and negative if *v2* is an ancestor of *v1*.

Usage

```
AlgHeight(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG.
- v2 (int) : vertex of the active MTG.

Returns int

If *v1* is not an ancestor of *v2* (or vise versa), or if *v1* and *v2* are not defined at the same scale, an error value None is returned.

See also:

MTG(), *Rank()*, *Order()*, *Height()*, *EdgeType()*, *AlgOrder()*, *AlgRank()*.

openalea.mtg.aml.**AlgOrder**(*v1*, *v2*)

Algebraic value defining the relative order of one vertex with respect to another one.

This function is similar to function *Order(v1, v2)* : it returns the number of +-type edges between two components, at the same scale, but takes into account the order of vertices *v1* and *v2*.

The result is positive if *v1* is an ancestor of *v2*, and negative if *v2* is an ancestor of *v1*.

Usage

```
AlgOrder(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG.
- v2 (int) : vertex of the active MTG.

Returns int

If *v1* is not an ancestor of *v2* (or vise versa), or if *v1* and *v2* are not defined at the same scale, an error value None is returned.

See also:

MTG(), *Rank()*, *Order()*, *Height()*, *EdgeType()*, *AlgHeight()*, *AlgRank()*.

openalea.mtg.aml.**AlgRank**(*v1*, *v2*)

Algebraic value defining the relative rank of one vertex with respect to another one.

This function is similar to function *Rank(v1, v2)* : it returns the number of <-type edges between two components, at the same scale, but takes into account the order of vertices *v1* and *v2*.

The result is positive if *v1* is an ancestor of *v2*, and negative if *v2* is an ancestor of *v1*.

Usage

```
AlgRank(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG.
- v2 (int) : vertex of the active MTG.

Returns int

If *v1* is not an ancestor of *v2* (or vise versa), or if *v1* and *v2* are not defined at the same scale, an error value None is returned.

See also:

MTG(), *Rank()*, *Order()*, *Height()*, *EdgeType()*, *AlgHeight()*, *AlgOrder()*.

openalea.mtg.aml.**Alpha**(*e1*, *e2*)

openalea.mtg.aml.**Ancestors**(*v*, *EdgeType='*'*, *RestrictedTo='NoRestriction'*, *ContainedIn=None*)

Array of all vertices which are ancestors of a given vertex

This function returns the array of vertices which are located before the vertex passed as an argument. These vertices are defined at the same scale as *v*. The array starts by *v*, then contains the vertices on the path from *v* back to the root (in this order) and finishes by the tree root.

Note: The anscestor array always contains at least the argument vertex *v*.

Usage

```
Ancestors(v)
```

Parameters

- v (int) : vertex of the active MTG

Optional Parameters

- RestrictedTo (str): cf. *Father*
- ContainedIn (int): cf. *Father*
- EdgeType (str): cf. *Father*

Returns list of vertices's id (int)

Examples

```
>>> v # prints vertex v
78
>>> Ancestors(v) # set of ancestors of v at the same scale
[78,45,32,10,4]
>>> list(reversed(Ancestors(v))) # To get the vertices in the order from the root␣
↪to the vertex v
[4,10,32,45,78]
```

See also:

MTG(), *Descendants()*.

openalea.mtg.aml.**Axis**(*v*, *Scale=-1*)

Array of vertices constituting a botanical axis

An axis is a maximal sequence of vertices connected by '<'-type edges. Axis return the array of vertices representing the botanical axis which the argument v belongs to. The optional argument enables the user to choose the scale at which the axis decomposition is required.

Usage

```
Axis(v)
Axis(v, Scale=s)
```

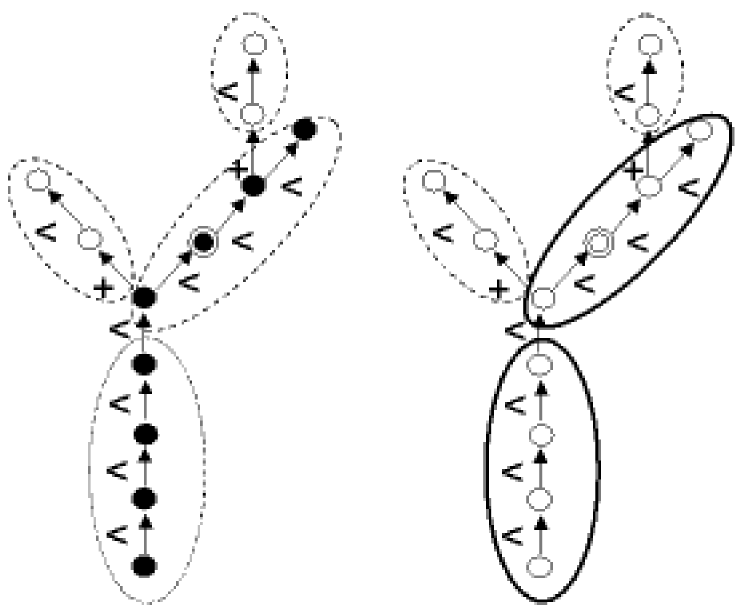
Parameters

- v (int) : Vertex of the active MTG

Optional Parameters

- Scale (str): scale at which the axis components are required.

Returns list of vertices ids



See also:

MTG(), *Path()*, *Ancestors()*.

openalea.mtg.aml.**Beta**(*e1*, *e2*)

openalea.mtg.aml.**BottomCoord**(*e1*, *e2*)

openalea.mtg.aml.**BottomDiameter**(*e1*, *e2*)

openalea.mtg.aml.**Class**(*vid*)

Class of a vertex

The *Class()* of a vertex is a feature always defined and independent of time (like the index). It is represented by an alphabetic character in upper or lower case (lower cases characters are considered different from upper cases). The label of a vertex is the string defined by the concatenation of its class and its index. The label thus provides general information about a vertex and enables us to encode the plant components.

Usage

```
>>> Class(v)
```

Parameters

- *vid* (int) : vertex of the active MTG

Returns The class of the vertex.

See also:

MTG(), *Index()*, *Scale()*.

`openalea.mtg.aml.`**`ClassScale`**`(`*c*`)`

Scale at which appears a given class of vertex

Every vertex is associated with a unique class. Vertices from a given class only appear at a given scale which can be retrieved using this function.

Usage

```
ClassScale(c)
```

Parameters

- *c* (str) : symbol of the considered class

Returns int

See also:

MTG(), *Class()*, *Scale()*, *Index()*.

`openalea.mtg.aml.`**`Complex`**`(`*v*, *Scale=-1*`)`

Complex of a vertex.

Returns the complex of *v*. The complex of a vertex *v* has a scale lower than *v* : *Scale(v)* - 1. In a MTG, every vertex except for the MTG root (cf. *MTGRoot*), has a uniq complex. None is returned if the argument is the MTG Root or if the vertex is undefined.

Usage

```
Complex(v)
Complex(v, Scale=2)
```

Parameters

- *v* (int) : vertex of the active MTG

Optional Parameters

- *Scale* (int) : scale of the complex

Returns Returns vertex's id (int)

Details When a scale different form Scale(v)-1 is specified using the optional parameter *Scale*, this scale must be lower than that of the vertex argument.

Todo: Complex(v, Scale=10) returns v why ? is this expected

See also:

MTG(), *Components()*.

openalea.mtg.aml.**ComponentRoots**(*v*, *Scale=-1*)

Set of roots of the tree graphs that compose a vertex

In a MTG, a vertex may have be decomposed into components. Some of these components are connected to each other, while other are not. In the most general case, the components of a vertex are organized into several tree-graphs. This is for example the case of a MTG containing the description of several plants: the MTG root vertex can be decomposed into tree graphs (not connected) that represent the different plants. This function returns the set of roots of these tree graphs at scale *Scale(v)+1*. The order of these roots is not significant.

When a scale different from *Scale(v)+1* is specified using the optional argument *Scale()*, this scale must be greater than that of the vertex argument.

Usage

```
ComponentRoots(v)
ComponentRoots(v, Scale=s)
```

Parameters

- v (int) : vertex of the active MTG

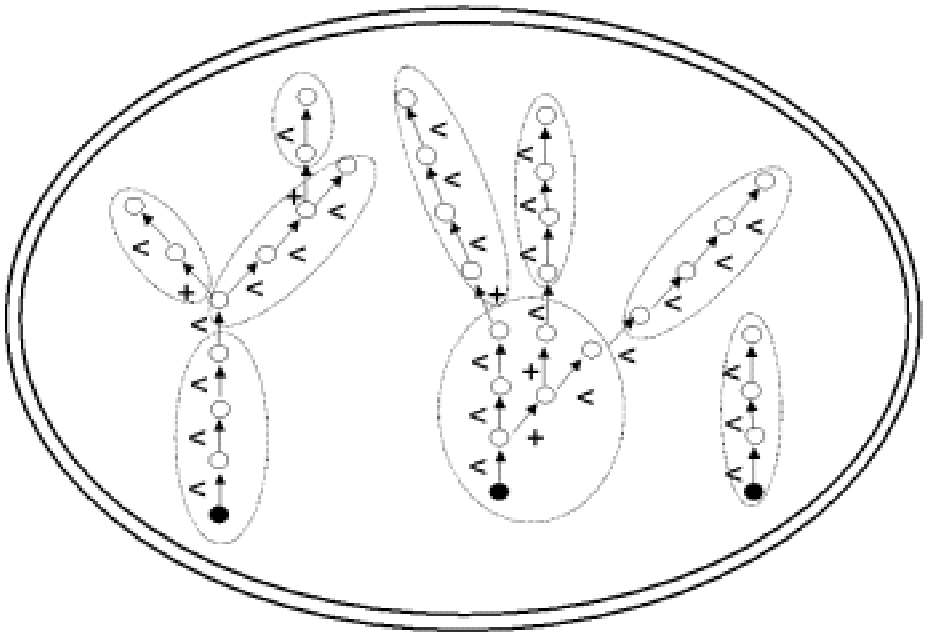
Optional Parameters

- Scale (str): scale of the component roots.

Returns list of vertices's id (int)

Examples

```
>>> v=MTGRoot() # global MTG root
0
>>> ComponentRoots(v) # set of first vertices at scale 1
[1,34,76,100,199,255]
>>> ComponentRoots(v, Scale=2) # set of first vertices at scale 2
[2,35,77,101,200,256]
```



See also:

MTG(), *Components()*, *Trunk()*.

`openalea.mtg.aml.`**`Components`**`(`*v*, *Scale=-1*`)`

Set of components of a vertex.

The set of components of a vertex is returned as a list of vertices. If **s** defines the scale of **v**, components are defined at scale **s** + 1. The array is empty if the vertex has no components. The order of the components in the array is not significant.

When a scale is specified using optional argument :arg:Scale, it must be necessarily greater than the scale of the argument.

Usage

```
Components(v)
Components(v, Scale=2)
```

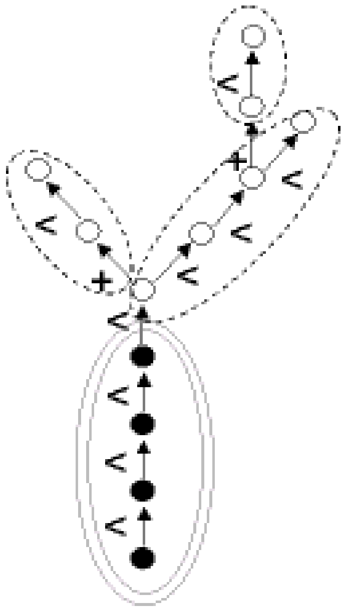
Parameters

- v (int) : vertex of the active MTG

Optional Parameters

- Scale (int) : scale of the components.

Returns list of int



See also:

MTG(), *Complex()*.

openalea.mtg.aml.**Coord**(*e1*, *e2*)

openalea.mtg.aml.**DateSample**(*e1*)

Array of observation dates of a vertex.

Returns the set of dates at which a given vertex (passed as an argument) has been observed as an array of ordered dates. Options can be specified to define a temporal window and the total list of observation dates will be truncated according to the corresponding temporal window.

Usage

```
DateSample(v)
DateSample(v, MinDate=d1, MaxDate=d2)
```

Parameters

- v (VTX) : vertex of the active MTG.

Optional Parameters

- MinDate (date) : defines a minimum date of interest.
- MaxDate (date) : defines a maximum date of interest.

Returns list of date

See also:

MTG(), *FirstDefinedFeature()*, *LastDefinedFeature()*, *PreviousDate()*, *NextDate()*.

openalea.mtg.aml.**Defined**(*vid*)

Test whether a given vertex belongs to the active MTG.

Usage

```
Defined(v)
```

Parameters

- v (int) : vertex of the active MTG

Returns True or False

See also:

MTG().

openalea.mtg.aml.**Descendants**(*v*, *EdgeType='*'*, *RestrictedTo='NoRestriction'*, *ContainedIn=None*)

Set of vertices in the branching system borne by a vertex.

This function returns the set of descendants of its argument as an array of vertices. The array thus consists of all the vertices, at the same scale as *v*, that belong to the branching system starting at *v*. The order of the vertices in the array is not significant.

Note: The argument always belongs to the set of its descendants.

Usage

```
Descendants(v)
```

Parameters

- v (int) : vertex of the active MTG

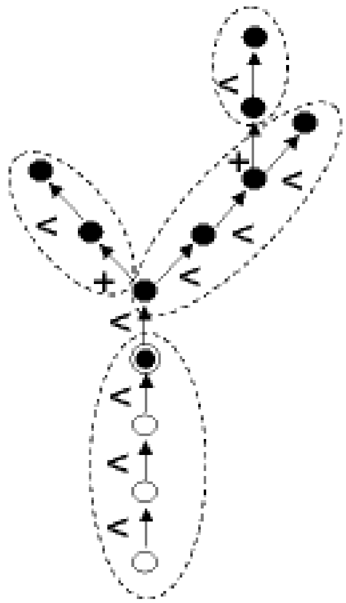
Optional Parameters

- RestrictedTo (str): cf. *Father*
- ContainedIn (int): cf. *Father*
- EdgeType (str): cf. *Father*

Returns list of int.

Examples

```
>>> v
78
>>> Sons(v) # set of sons of v
[78,99,101]
>>> Descendants(v) # set of descendants of v
[78,99,101,121,133,135,156,171,190]
```



See also:

MTG(), *Ancestors()*.

`openalea.mtg.aml.`**`DressingData`**`(`*e1*`)`

Use openalea.mtg.dresser.DressingData instead of this function

`openalea.mtg.aml.`**`EdgeType`**`(`*v1*, *v2*`)`

Type of connection between two vertices.

Returns the symbol of the type of connection between two vertices (either < or +). If the vertices are not connected, None is returned.

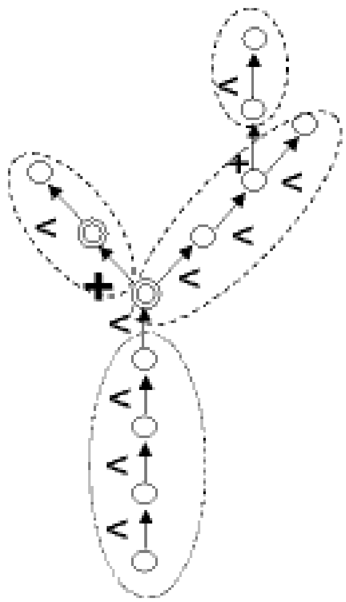
Usage

```
EdgeType(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG
- v2 (int) : vertex of the active MTG

Returns '<' (successor), '+' (branching) or *None*



See also:

MTG(), *Sons()*, *Father()*.

`openalea.mtg.aml.`**`Extremities`**(*v*, *RestrictedTo='NoRestriction'*, *ContainedIn=None*)

Set of vertices that are the extremities of the branching system born by a given vertex.

This function returns the extremities of the branching system defined by the argument as a list of vertices. These vertices have the same scale as *v* and their order in the list is not signifiant. The result is always a non empty array.

Usage

```
Extremities(v)
```

Properties

- v (int) : vertex of the active MTG

Optional Parameters

- RestrictedTo (str): cf. *Father()*
- ContainedIn (int): cf. *Father()*

Returns list of vertices's id (int)

Examples

```
>>> Descendants(v)
[3, 45, 47, 78, 102]
>>> Extremities(v)
[47, 102]
```

See also:

MTG(), *Descendants()*, *Root()*, *MTGRoot()*.

openalea.mtg.aml.**Father**(*v*, *EdgeType='*'*, *RestrictedTo='NoRestriction'*, *ContainedIn=None*, *Scale=-1*)

Topological father of a given vertex.

Returns the topological father of a given vertex. And *None* if the father does not exist. If the argument is not a valid vertex, *None* is returned.

Usage

```
Father(v)
Father(v, EdgeType='<')
Father(v, RestrictedTo='SameComplex')
Father(v, ContainedIn=complex_id)
Father(v, Scale=s)
```

Parameters v (int) : vertex of the active MTG

Optional Parameters If no optional argument is specified, the function returns the topological father of the argument (vertex that bears or precedes to the vertex passed as an argument).

It may be usefull in some cases to consider that the function only applies to a subpart of the MTG (e.g. an axis).

The following options enables us to specify such restrictions:

- EdgeType (str) : filter on the type of edge that connect the vertex to its father.

  Values can be '<', '+', and '*'. Values '*' means both '<' and '+'. Only the vertex connected with the specified type of edge will be considered.

- RestrictedTo (str) : filter defining a subpart of the MTG where the father must be considered. If the father is actually outside this subpart, the result is *None*. Possible subparts are defined using keywords in ['SameComplex', 'SameAxis', 'NoRestriction'].

  For instance, if *RestrictedTo* is set to 'SameComplex', `Father(v)()` returns a defined vertex only if the father *f* of *v* existsin the MTG and if *v* and *f* have the same complex.

- ContainedIn (int) : filter defining a subpart of the MTG where the father must be considered. If the father is actually outside this subpart, the result is *None*.

  In this case, the subpart of the MTG is made of the vertices that composed *composite_id* (at any scale).

- Scale (int) : the scale of the considered father. Returns the vertex from scale *s* which either bears and precedes the argument.

  The scale *s* can be lower than the argument's (corresponding to a question such as 'which axis bears the internode?') or greater (e.g. 'which internodes bears this annual shoot?').

Returns the vertex id of the Father (int)

See also:

MTG(), *Defined()*, *Sons()*, *EdgeType()*, *Complex()*, *Components()*.

openalea.mtg.aml.**Feature**(*vid*, *fname*, *date=None*)

Extracts the attributes of a vertex.

Returns the value of the attribute *fname* of a vertex in a *MTG*.

If the value of an attribute is not defined in the coding file, the value None is returned.

Usage

```
Feature(vid, fname)
Feature(vid, fname, date)
```

Parameters

- vid (int) : vertex of the active MTG.
- fname (str) : name of the attribute (as specified in the coding file).
- date (date) : (for a dynamic *MTG*) date at which the attribute of the vertex is considered.

Returns int, str, date or float

Details If for a given attribute, several values are available(corresponding to different dates), the date of interest must be specified as a third attribute.

This date must be a valid date appearing in the coding file for a considered vertex. Otherwise *None* is returned.

Background MTGs and Dynamic MTGs.

Todo: specify the format of *date*

See also:

MTG(), *Class()*, *Index()*, *Scale()*.

openalea.mtg.aml.**FirstDefinedFeature**(*e1*, *e2*)
Date of first observation of a vertex.

Returns the date *d* for which the attribute *fname* is defined for the first time on the vertex *v* passed as an argument. This date must be greater than the option *MinDate* and/or less than the maximum *MaxData* when specified. Otherwise the returned date is None.

Usage

```
FirstDefinedFeature(v, fname)
FirstDefinedFeature(v, fname, MinDate=d1, MaxDate=d2)
```

Properties

- v (int) : vertex of the active MTG
- fname (str) : name of the considered property

Optional Properties

- MinDate (date) : minimum date of interest.
- MaxData (date) : maximum date of interest.

Returns date

See also:

MTG(), *DateSample()*, *LastDefinedFeature()*, *PreviousDate()*, *NextDate()*.

openalea.mtg.aml.**Height**(*v1*, *v2=None*)
Number of components existing between two components in a tree graph.

The height of a vertex (*v2*) with respect to another vertex (*v1*) is the number of edges (of either type '+' or '<') that must be crossed when going from *v1* to *v2* in the graph.

This is a non-negative integer. When the function has only one argument *v1*, the height of *v1* correspond to the height of *v1 'with respect to the root of the branching system containing 'v1*.

Usage

```
Height(v1)
Height(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG
- v2 (int) : vertex of the active MTG

Returns int

Note: When the function takes two arguments, the order of the arguments is not important provided that one is an ancestor of the other. When the order is relevant, use function *AlgHeight*.

See also:

MTG(), *Order()*, *Rank()*, *EdgeType()*, *AlgHeight()*, *AlgHeight()*, *AlgOrder()*.

openalea.mtg.aml.**Index**(*vid*)

Index of a vertex

The *Index()* of a vertex is a feature always defined and independent of time (like the index). It is represented by a non negative integer. The label of a vertex is the string defined by the concatenation of its class and its index. The label thus provides general information about a vertex and enables us to encode the plant components.

Usage

```
>>> Index(v)
```

Parameters

- *vid* (int) : vertex of the active MTG

Returns int

See also:

MTG(), *Class()*, *Scale()*

openalea.mtg.aml.**Label**(*v*)

Label of a vertex

Usage

```
>>> Label(v) #doctest: +SKIP
```

Parameters

- *vid* (int) : vertex of the active MTG

Returns The class and Index of the vertex (str).

See also:

MTG(), *Index()*, *Class()*

`openalea.mtg.aml.`**`LastDefinedFeature`**`(`*e1*, *e2*`)`

Date of last observation of a given attribute of a vertex.

Returns the date *d* for which the attribute *fname* is defined for the last time on the vertex *v* passed as an argument. This date must be greater than the option *MinDate* and/or less than the maximum *MaxData* when specified. Otherwise the returned date is None.

Usage

```
FirstDefinedFeature(v, fname)
FirstDefinedFeature(v, fname, MinDate=d1, MaxDate=d2)
```

Properties

- v (int) : vertex of the active MTG
- fname (str) : name of the considered property

Optional Properties

- MinDate (date) : minimum date of interest.
- MaxData (date) : maximum date of interest.

Returns date

See also:

MTG(), *DateSample()*, *FirstDefinedFeature()*, *PreviousDate()*, *NextDate()*.

`openalea.mtg.aml.`**`Length`**`(`*e1*, *e2*`)`

`openalea.mtg.aml.`**`Location`**`(`*v*, *Scale=-1*, *ContainedIn=None*`)`

Vertex defining the father of a vertex with maximum scale.

If no options are supplied, this function returns the vertex defining the father of a vertex with maximum scale (cf. *Father()*). If it does not exist, None is returned. If a scale is specified, the function is equivalent to *Father(v, Scale=s)*.

Usage

```
Location(v)
Location(v, Scale=s)
Location(v, ContainedIn=complex_id)
```

Parameters

- v (int) : vertex of the active MTG.

Optional Parameters

- Scale (int) : scale at which the location is required.
- ContainedIn (int) : cf. *Father()*

Returns Returns vertex's id (int)

Examples

```
>>> Father(v, EdgeType='+')
7
>>> Complex(v)
4
>>> Components(7)
[9,19,23, 34, 77, 89]
>>> Location(v)
23
>>> Location(v, Scale= Scale(v)+1)
23
>>> Location(v, Scale= Scale(v))
7
>>> Location(v, Scale= Scale(v)-1)
4
```

See also:

MTG(), *Father()*.

openalea.mtg.aml.**MTG**(*filename*)
MTG constructor.

Builds a MTG from a coding file (text file) containing the description of one or several plants.

Usage

```
MTG(filename)
```

Parameters

- *filename* (str): name of the coding file describing the mtg

Returns If the parsing process succeeds, returns an object of type *MTG()*. Otherwise, an error is generated, and the formerly active *MTG* remains active.

Side Effect If the *MTG()* is built, the new *MTG()* becomes the active *MTG()* (i.e. the *MTG()* implicitly used by other functions such as *Father()*, *Sons()*, *VtxList()*, ...).

Details The parsing process is approximatively proportional to the number of components defined in the coding file.

Background MTG is an acronyme for Multiscale Tree Graph.

See also:

Activate() and all *openalea.mtg.aml* functions.

openalea.mtg.aml.**MTGRoot**()
Returns the root vertex of the MTG.

It is the only vertex at scale 0 (the coarsest scale).

Usage

```
>>> MTGRoot()
```

Returns

- vtx identifier

Details This vertex is the complex of all vertices from scale 1. It is a mean to refer to the entire database.

See also:

MTG(), *Complex()*, *Components()*, *Scale()*.

openalea.mtg.aml.**NextDate**(*e1*)

Next date at which a vertex has been observed after a specified date

Returns the first observation date at which the vertex has been observed starting at date d and proceeding forward in time. None is returned if it does not exists.

Usage

```
NextDate(v, d)
```

Parameters

- v (int) : vertex of the active MTG.
- d (date) : departure date.

Returns date

See also:

MTG(), *DateSample()*, *FirstDefinedFeature()*, *LastDefinedFeature()*, *PreviousDate()*.

openalea.mtg.aml.**Order**(*v1*, *v2=None*)

Order of a vertex in a graph.

The order of a vertex (*v2*) with respect to another vertex (*v1*) is the number of edges of either type '+' that must be crossed when going from *v1* to *v2* in the graph. This is thus a non negative integer which corresponds to the "botanical order".

When the function only has one argument *v1*, the order of *v1* correspond to the order of *v1* with respect to the root of the branching system containing *v1*.

Usage

```
Order(v1)
Order(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG
- v2 (int) : vertex of the active MTG

Returns int

Note: When the function takes two arguments, the order of the arguments is not important provided that one is an ancestor of the other. When the order is relevant, use function AlgOrder().

Warning: The value returned by function Order is 0 for trunks, 1 for branches etc. This might be different with some botanical conventions where 1 is the order of the trunk, 2 the order of branches, etc.

See also:

MTG(), *Rank()*, *Height()*, *EdgeType()*, *AlgOrder()*, *AlgRank()*, *AlgHeight()*.

openalea.mtg.aml.**PDir**(*e1*, *e2*)

openalea.mtg.aml.**Path**(*v1*, *v2*)

List of vertices defining the path between two vertices

This function returns the list of vertices defining the path between two vertices that are in an ancestor relationship. The vertex *v1* must be an ancestor of vertex *v2*. Otherwise, if both vertices are valid, then the empty array is returned and if at least one vertex is undefined, None is returned.

Usage

```
Path(v1, v2)
```

Parameters

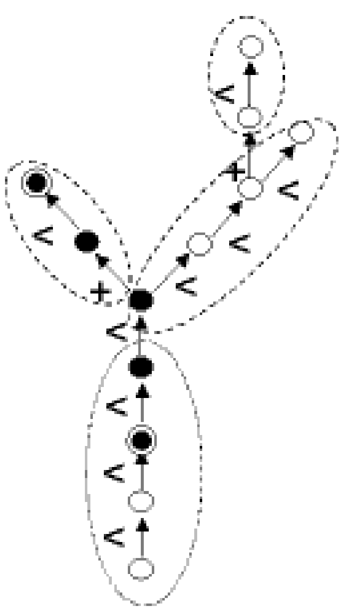
- *v1* (int) : vertex of the active MTG
- *v2* (int) : vertex of the active MTG

Returns list of vertices's id (int)

Examples

```
>>> v # print the value of v
78
>>> Ancestors(v)
[78,45,32,10,4]
>>> Path(10,v)
[10,32,45,78]
>>> Path(9,v) # 9 is not an ancestor of 78
[]
```

Note: *v1* can be equal to *v2*.



See also:

MTG(), *Axis()*, *Ancestors()*.

`openalea.mtg.aml.`**`PlantFrame`**`(e1)`
Use openalea.mtg.plantframe.PlantFrame insteead of this function

`openalea.mtg.aml.`**`Plot`**`(e1)`

`openalea.mtg.aml.`**`Predecessor`**`(v, **kwds)`
Father of a vertex connected to it by a '<' edge

This function is equivalent to Father(v, EdgeType-> '<'). It thus returns the father (at the same scale) of the argument if it is located in the same botanical. If it does not exist, None is returned.

Usage

```
Predecessor(v)
```

Parameters

- v (int) : vertex of the active MTG

Optional Parameters

- RestrictedTo (str): cf. *Father*
- ContainedIn (int): cf. *Father*

Returns return the vertex id (int)

Examples

```
>>> Predecessor(v)
7
>>> Father(v, EdgeType='+')
>>> Father(v, EdgeType-> '<')
7
```

See also:

MTG(), *Father()*, *Successor()*.

`openalea.mtg.aml.`**`PreviousDate`**`(e1)`
Previous date at which a vertex has been observed after a specified date.

Returns the first observation date at which the vertex has been observed starting at date d and proceeding backward in time. None is returned if it does not exists.

Usage

```
PreviousDate(v, d)
```

Parameters

- v (int) : vertex of the active MTG.
- d (date) : departure date.

Returns date

See also:

MTG(), *DateSample()*, *FirstDefinedFeature()*, *LastDefinedFeature()*, *NextDate()*.

openalea.mtg.aml.**Rank**(*v1*, *v2=None*)

Rank of one vertex with respect to another one.

This function returns the number of consecutive '<'-type edges between two components, at the same scale, and does not take into account the order of vertices v1 and v2. The result is a non negative integer.

Usage

```
Rank(v1)
Rank(v1, v2)
```

Parameters

- v1 (int) : vertex of the active MTG
- v2 (int) : vertex of the active MTG

Returns *int*

If v1 is not an ancestor of v2 (or vise versa) within the same botanical axis, or if v1 and v2 are not defined at the same scale, an error value Undef id returned.

See also:

MTG(), *Order()*, *Height()*, *EdgeType()*, *AlgRank()*, *AlgHeight()*, *AlgOrder()*.

openalea.mtg.aml.**RelBottomCoord**(*e1*, *e2*)

openalea.mtg.aml.**RelTopCoord**(*e1*, *e2*)

openalea.mtg.aml.**Root**(*v*, *RestrictedTo='*'*, *ContainedIn=None*)

Root of the branching system containing a vertex

This function is equivalent to Ancestors(v, EdgeType='<')[-1]. It thus returns the root of the branching system containing the argument. This function never returns None.

Usage

```
Root(v)
```

Parameters

- v (int) : vertex of the active MTG

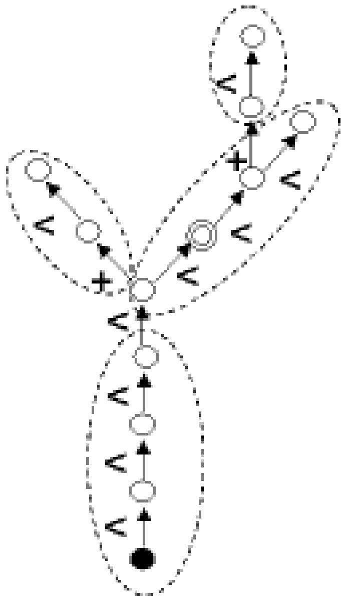
Optional Parameters

- RestrictedTo (str): cf. Father
- ContainedIn (int): cf. Father

Returns return vertex's id (int)

Examples

```
>>> Ancestors(v) # set of ancestors of v
[102,78,35,33,24,12]
>>> Root(v) # root of the branching system containing v
12
```



See also:

MTG(), *Extremities()*.

`openalea.mtg.aml.`**`SDir`**`(`*e1*, *e2*`)`

`openalea.mtg.aml.`**`Scale`**`(`*vid*`)`

Scale of a vertex

Returns the scale at which is defined the argument.

Usage

```
>>> Scale(vid)
```

Parameters

- *vid* (int) : vertex of the active MTG
- *vid* (PlantFrame) : a PlantFrame object computed on the active MTG
- *vid* (LineTree) : a LineTree computed on a PlantFrame representing the active MTG

Returns int

See also:

MTG(), *ClassScale()*, *Class()*, *Index()*.

`openalea.mtg.aml.`**`Sons`**`(`*v*, *RestrictedTo='NoRestriction'*, *EdgeType='*'*, *Scale=-1*, *ContainedIn=None*`)`

Set of vertices born or preceded by a vertex

The set of sons of a given vertex is returned as an array of vertices. The order of the vertices in the array is not significant. The array can be empty if there are no son vertices.

Usage

```
from openalea.mtg.aml import Sons
Sons(v)
Sons(v, EdgeType= '+')
Sons(v, Scale= 3)
```

Parameters

- v (int) : vertex of the active MTG

Optional Parameters

- RestrictedTo (str) : cf. *Father*
- ContainedIn (int) : cf. *Father*
- EdgeType (str) : filter on the type of sons.
- Scale (int) : set the scale at which sons are considered.

Returns list(vid)

Details When the option EdgeType is applied, the function returns the set of sons that are connected to the argument with the specified type of relation.

Note: *Sons(v, EdgeType= '<')* is not equivalent to *Successor(v)*. The first function returns an array of vertices while the second function returns a vertex.

The returned vertices have the same scale as the argument. However, coarser or finer vertices can be obtained by specifying the optional argument *Scale* at which the sons are considered.

Examples

```
>>> Sons(v)
[3,45,47,78,102]
>>>  Sons(v, EdgeType= '+') # set of vertices borne by v
[3,45,47,102]
>>>  Sons(v, EdgeType= '<') # set of successors of v on the same axis
[78]
```

See also:

MTG(), *Father()*, *Successor()*, *Descendants()*.

`openalea.mtg.aml.`**`Successor`**(*v*, *RestrictedTo='NoRestriction'*, *ContainedIn=None*)

Vertex that is connected to a given vertex by a '<' edge type (i.e. in the same botanical axis).

This function is equivalent to Sons(v, EdgeType='<')[0]. It returns the vertex that is connected to a given vertex by a '<' edge type (i.e. in the same botanical axis). If many such vertices exist, an arbitrary one is returned by the function. If no such vertex exists, None is returned.

Usage

```
Successor(v)
```

Parameters

- v1 (int) : vertex of the active MTG

Optional Parameters

- RestrictedTo (str): cf. Father
- ContainedIn (int): cf. Father

Returns Returns vertex's id (int)

Examples

```
>>> Sons(v)
[3, 45, 47, 78, 102]
>>> Sons(v, EdgeType='+') # set of vertices borne by v
[3, 45, 47, 102]
>>> Sons(v, EdgeType-> '<') # set of successors of v
[78]
>>> Successor(v)
78
```

See also:

MTG(), *Sons()*, *Predecessor()*.

openalea.mtg.aml.**TopCoord**(*e1*, *e2*)

openalea.mtg.aml.**TopDiameter**(*e1*, *e2*)

openalea.mtg.aml.**Trunk**(*v*, *Scale=-1*)

List of vertices constituting the bearing botanical axis of a branching system.

Trunk returns the list of vertices representing the botanical axis defined as the bearing axis of the whole branching system defined by *v*. The optional argument enables the user to choose the scale at which the trunk should be detailed.

Usage

```
Trunk(v)
Trunk(v, Scale= s)
```

Parameters

- *v* (int) : Vertex of the active MTG.

Optional Parameters

- *Scale* (str): scale at which the axis components are required.

Returns list of vertices ids

Todo: check the usage of the optional argument Scale

See also:

MTG(), *Path()*, *Ancestors()*, *Axis()*.

openalea.mtg.aml.**VirtualPattern**(*e1*)

openalea.mtg.aml.**VtxList**(*Scale=-1*)

Array of vertices contained in a MTG

The set of all vertices in the *MTG()* is returned as an array. Vertices from all scales are returned if no option is used. The order of the elements in this array is not significant.

Usage

```
>>> VtxList()
>>> VtxList(Scale=2)
```

Optional Parameters

- *Scale* (int): used to select components at a particular scale.

Returns

- list of vid

Background `MTGs()`

See also:

MTG(), *Scale()*, *Class()*, *Index()*.

Download the source file `../../src/mtg/aml.py`.

4.3 Reading and writing MTG

4.3.1 MTG

The MTG data structure can be read/write from/to a MTG file format. The functions *read_mtg()* , *write_mtg()* , *read_mtg_file()*.

`openalea.mtg.io.`**`read_mtg`**(*s*, *mtg=None*, *has_date=False*)

Create an MTG from its string representation in the MTG format.

Parameter

- s (string) - a multi-lines string

Return an MTG

Example

```
f = open('test.mtg')
txt = f.read()

g = read_mtg(txt)
```

See also:

read_mtg_file().

`openalea.mtg.io.`**`read_mtg_file`**(*fn*, *mtg=None*, *has_date=False*)

Create an MTG from a filename.

Usage

```
>>> g = read_mtg_file('test.mtg')
```

See also:

read_mtg().

`openalea.mtg.io.`**`write_mtg`**(*g*, *properties=[]*, *class_at_scale=None*, *nb_tab=None*, *display_id=False*)

Transform an MTG into a multi-line string in the MTG format.

This method build a generic header, then traverses the MTG and transform each vertex into a line with its label, topoloical relationship and specific *properties*.

Parameters

- *g* (MTG)
- ***properties* (list): a list of tuples associating a property name with its type.** Only these properties will be written in the out file.

Optional Parameters

- ***class_at_scale* (dict(name->int)): a map between a class name and its scale.** If *class_at_scale* is None, its value will be computed from *g*.
- *nb_tab* (int): the number of tabs used to write the code.
- *display_id* (bool): display the id for each vertex

Returns a list of strings.

Example

```
# Export all the properties defined in `g`.
# We consider that all the properties are real numbers.

properties = [(p, 'REAL') for p in g.property_names() if p not in ['edge_type',
↪'index', 'label']]
mtg_lines = write_mtg(g, properties)

# Write the result into a file example.mtg

filename = 'example.mtg'
f = open(filename, 'w')
f.write(mtg_lines)
f.close()
```

4.3.2 LPy

The two functions `lpy2mtg()` and `mtg2lpy()` allow to convert the MTG data-structure into lpy and vise-versa. It ease the communication between the two modules. Each structure are traversed and the properties are copied. Properties can be any pyton object.

`openalea.mtg.io.`**`lpy2mtg`**(*axial_tree*, *lsystem*, *scene=None*)

`openalea.mtg.io.`**`mtg2lpy`**(*g*, *lsystem*, *axial_tree=None*)

Create an AxialTree from a MTG with scales.

Usage

```
tree = mtg2lpy(g,lsystem)
```

Parameters

- *g*: The mtg which have been generated by an LSystem.
- ***lsystem*: A lsystem object containing various information.** The *lsystem* is only used to retrieve the context and the parameters associated with each module name.

Optional Parameters

- ***axial_tree*: an empty axial tree.** It is used to avoid complex import in the code.

Return axial tree

See also:

mtg2axialtree()

4.3.3 AxialTree

openalea.mtg.io.**axialtree2mtg**(*tree*, *scale*, *scene*, *parameters=None*)

Create an MTG from an AxialTree.

Tha axial tree has been generated by LPy. It contains both modules with parameters. The geometry is provided by the scene. The shape ids are the same that the module ids in the axial tree. For each module name in the axial tree, a *scale* and a list of parameters should be defined. The *scale* dict allow to add a module at a given scale in the MTG. The *parameters* dict map for each module name a list of parameter name that are added to the MTG.

Parameters

- *tree*: The axial tree generated by the L-system
- *scale*: A dict containing the scale for each symbol name.
- *scene*: The scene containing the geometry.
- *parameters*: list of parameter names for each module.

Return mtg

Example

```
tree # axial tree
scales = {}
scales['P'] = 1
scales['A'] = 2
scales['GU'] = 3

params ={}
params['P'] = []
params['A'] = ['length', 'radius']
params['GU'] = ['nb_flower']

g = axialtree2mtg(tree, scales, scene, params)
```

See also:

mtg2axialtree(), *lpy2mtg()*, *mtg2lpy()*

openalea.mtg.io.**mtg2axialtree**(*g*, *parameters=None*, *axial_tree=None*)

Create a MTG from an AxialTree with scales.

Parameters

- *axial_tree*: The axial tree managed by the L-system. Use an empty AxialTree if you do not want to concatenate this axial_tree with previous results.
- *parameters*: list of parameter names for each module.

Return mtg

Example

```
params = dict()
params ['P'] = []
params['A'] = ['length', radius']
```

(continues on next page)

(continued from previous page)

```
params['GU']=['nb_flower']
tree = mtg2axialtree(g, params)
```

See also:

axialtree2mtg(), *mtg2lpy()*

4.3.4 Cpfg

openalea.mtg.io.**read_lsystem_string**(*string*, *symbol_at_scale*, *functional_symbol={}*, *mtg=None*)

Read a string generated by a lsystem.

Parameters

- *string*: The lsystem string representing the axial tree.
- *symbol_at_scale*: A dict containing the scale for each symbol name.

Optional parameters

- ***functional_symbol*: A dict containing a function for specific symbols.** The args of the function have to be coherent with those in the string. The return type of the functions have to be a dictionary of properties: dict(name, value)

Return MTG object

4.3.5 Mss

openalea.mtg.io.**mtg2mss**(*name*, *mtg*, *scene*, *envelop_type='CvxHull'*)

Convert an MTG into the multi-scale structure implemented by fractalysis.

Parameters

- *name*: name of the structure
- *mtg*: the mtg to convert
- *scene*: the scene containing the geometry
- *envelop_type*: algorithm used to fit the geometry.between scales.

Returns mss data structure.

Download the source file `../../src/mtg/io.py`.

4.4 Traversal methods on tree and MTG

Tree and MTG Traversals

class openalea.mtg.traversal.**Visitor**

Bases: `object`

Used during a tree traversal.

post_order(*vtx_id*)

pre_order(*vtx_id*)

openalea.mtg.traversal.**iter_mtg**(*mtg*, *vtx_id*)

Iterate on an MTG by traversiong *vtx_id* and all its components.

This function traverse a complex before its components and a parent before its children.

Usage

```
for vid in iter_mtg(g,g.root):
    print vid
```

Parameters

- *mtg*: the multi-scale graph
- *vtx_id*: the root of the sub-mtg which is traversed.

Returns iter of vid.

Traverse all the vertices contained in the sub_mtg defined by *vtx_id*.

See also:

iter_mtg2(), *iter_mtg_with_filter()*, *iter_mtg2_with_filter()*.

Note: Do not use this function. Use *iter_mtg2()* instead. If several trees belong to *vtx_id*, only the first one will be traversed.

Note: This is a recursive implementation. It can be problematic for large MTG with lots of scales (e.g. >40).

openalea.mtg.traversal.**iter_mtg2**(*mtg*, *vtx_id*)

Iterate on an MTG by traversiong *vtx_id* and all its components.

This function traverse a complex before its components and a parent before its children.

Usage

```
for vid in iter_mtg2(g,g.root):
    print vid
```

Parameters

- *mtg*: the multi-scale graph
- *vtx_id*: the root of the sub-mtg which is traversed.

Returns iter of vid.

Traverse all the vertices contained in the sub_mtg defined by *vtx_id*.

See also:

iter_mtg(), *iter_mtg_with_filter()*, *iter_mtg2_with_filter()*

Note: Use this function instead of *iter_mtg()*

openalea.mtg.traversal.**iter_mtg2_with_filter**(*mtg*, *vtx_id*, *pre_order_filter=None*, *post_order_visitor=None*)

Iterate on an MTG by traversiong *vtx_id* and all its components.

If defined, apply the two visitor functions before and after having visited all the successor of a vertex.

This function traverse a complex before its components and a parent before its children.

Usage

```
def pre_order_visitor(vid):
    print vid
    return True
def post_order_visitor(vid):
    print vid
for vid in iter_mtg_with_filter(g,g.root, pre_order_visitor, post_order_visitor):
    pass
```

Parameters

- *mtg*: the multi-scale graph
- *vtx_id*: the root of the sub-mtg which is traversed.

Optional Parameters

- ***pre_order_visitor*: function called before traversing the children or components.** This function returns a boolean. If False, the sub-mtg rooted on the vertex is skipped.
- *post_order_visitor* : function called after the traversal of all the children and components.

Returns iter of vid.

Traverse all the vertices contained in the sub_mtg defined by *vtx_id*.

See also:

iter_mtg(), *iter_mtg2()*, *iter_mtg2_with_filter()*

Note: Use this function instead of *iter_mtg_with_filter()*

openalea.mtg.traversal.**iter_mtg_with_filter**(*mtg*, *vtx_id*, *pre_order_filter=None*, *post_order_visitor=None*)

Iterate on an MTG by traversiong *vtx_id* and all its components.

If defined, apply the two visitor functions before and after having visited all the successor of a vertex.

This function traverse a complex before its components and a parent before its children.

Usage

```
def pre_order_visitor(vid):
    print vid
    return True
def post_order_visitor(vid):
    print vid
for vid in iter_mtg_with_filter(g,g.root, pre_order_visitor, post_order_visitor):
    pass
```

Parameters

- *mtg*: the multi-scale graph
- *vtx_id*: the root of the sub-mtg which is traversed.

Optional Parameters

- ***pre_order_visitor*: function called before traversing the children or components.** This function returns a boolean. If False, the sub-mtg rooted on the vertex is skipped.
- *post_order_visitor* : function called after the traversal of all the children and components.

Returns iter of vid.

Traverse all the vertices contained in the sub_mtg defined by *vtx_id*.

See also:

`iter_mtg()`, `iter_mtg2()`, `iter_mtg2_with_filter()`

Note: Do not use this function. Instead use `iter_mtg2_with_filter()`

`openalea.mtg.traversal.`**`iter_scale`**(*g*, *vtx_id*, *visited*)
Internal method used by `iter_mtg()` and `iter_mtg_with_visitor()`.

Warning: Do not use. This function may be removed in other version.

`openalea.mtg.traversal.`**`iter_scale2`**(*g*, *vtx_id*, *complex_id*, *visited*)
Internal method used by `iter_mtg()` and `iter_mtg_with_visitor()`.

Warning: Do not use. This function may be removed in other version.

`openalea.mtg.traversal.`**`post_order`**(*tree*, *vtx_id*, *complex=None*, *visitor_filter=None*)
Traverse a tree in a postfix way. (from leaves to root) This is a recursive implementation

`openalea.mtg.traversal.`**`post_order2`**(*tree*, *vtx_id*, *complex=None*, *pre_order_filter=None*, *post_order_visitor=None*)
Traverse a tree in a postfix way. (from leaves to root)

Same algorithm than post_order. The goal is to replace the post_order implementation.

`openalea.mtg.traversal.`**`pre_order`**(*tree*, *vtx_id*, *complex=None*, *visitor_filter=None*)
Traverse a tree in a prefix way. (root then children)

This is a non recursive implementation.

`openalea.mtg.traversal.`**`pre_order2`**(*tree*, *vtx_id*, *complex=None*, *visitor_filter=None*)
Traverse a tree in a prefix way. (root then children)

This is an iterative implementation.

`openalea.mtg.traversal.`**`pre_order2_with_filter`**(*tree*, *vtx_id*, *complex=None*, *pre_order_filter=None*, *post_order_visitor=None*)
Same algorithm than pre_order2. The goal is to replace the pre_order2 implementation.

The problem is for the pre_order filter when it is also a visitor

openalea.mtg.traversal.**pre_order_in_scale**(*tree*, *vtx_id*, *visitor_filter=None*)
Traverse a tree in a prefix way. (root then children)

This is a non recursive implementation.

openalea.mtg.traversal.**pre_order_with_filter**(*tree*, *vtx_id*, *pre_order_filter=None*, *post_order_visitor=None*)
Traverse a tree in a prefix way. (root then children)

This is an iterative implementation.

TODO: make the naming and the arguments more consistent and user friendly. pre_order_filter is a functor which has to return a boolean. If the return value is False, the vertex is not visited. Otherelse, some computation can be done.

The post_order_visitor is used to execute, store, compute a function when the tree rooted on the vertex has been visited.

openalea.mtg.traversal.**topological_sort**(*g*, *vtx_id*, *visited=None*)
Topological sort of a directed acyclic graph.

This is not a fully recursive implementation.

openalea.mtg.traversal.**traverse_tree**(*tree*, *vtx_id*, *visitor*)
Traverse a tree in a prefix or postfix way.

We call a visitor for each vertex. This is usefull for printing, computing or storing vertices in a specific order.

See boost.graph.

Download the source file `../../src/mtg/traversal.py`.

4.5 Common algorithms

Implementation of a set of algorithms for the MTG datastructure

openalea.mtg.algo.**alg_height**(*g*, *v1*, *v2=None*)

openalea.mtg.algo.**alg_order**(*g*, *v1*, *v2=None*)

openalea.mtg.algo.**alg_rank**(*g*, *v1*, *v2=None*)

openalea.mtg.algo.**ancestors**(*g*, *vid*, ***kwds*)
Return the vertices from vid to the root.

Parameters

- *g*: a tree or an MTG
- *vid*: a vertex id which belongs to *g*

Returns an iterator from *vid* to the root of the tree.

openalea.mtg.algo.**axis**(*g*, *vtx_id*, *scale=-1*, ***kwds*)
TODO: see aml doc

openalea.mtg.algo.**descendants**(*g*, *vtx_id*, *scale=-1*, ***kwds*)
TODO: see aml doc

openalea.mtg.algo.**edge_type**(*g*, *v*)

openalea.mtg.algo.**extremities**(*g*, *vid*, ***kwds*)
TODO see aml doc Implement the method more efficiently…

`openalea.mtg.algo.`**`father`**(*g*, *vid*, *scale=-1*, ***kwds*)
See aml.Father function.

`openalea.mtg.algo.`**`full_ancestors`**(*g*, *v1*, ***kwds*)
Return the vertices from v1 to the root.

`openalea.mtg.algo.`**`height`**(*g*, *v1*, *v2=None*)

`openalea.mtg.algo.`**`heights`**(*g*, *scale=-1*)
Compute the order of all vertices at scale *scale*.

If scale == -1, the compute the order for vertices at the finer scale.

`openalea.mtg.algo.`**`local_axis`**(*g*, *vtx_id*, *scale=-1*, ***kwds*)
Return a sequence of vertices connected by '<' edges. The first element of the sequence is vtx_id.

`openalea.mtg.algo.`**`location`**(*g*, *vid*, ***kwds*)
TODO: see doc aml.Location.

`openalea.mtg.algo.`**`lookForCommonAncestor`**(*g*, *commonAncestors*, *currentNode*)

`openalea.mtg.algo.`**`lowestCommonAncestor`**(*g*, *nodes*)
LCA algorithm

`openalea.mtg.algo.`**`order`**(*g*, *v1*, *v2=None*)

`openalea.mtg.algo.`**`orders`**(*g*, *scale=-1*)
Compute the order of all vertices at scale *scale*.

If scale == -1, the compute the order for vertices at the finer scale.

`openalea.mtg.algo.`**`path`**(*g*, *vid1*, *vid2=None*)
Compute the vertices between v1 and v2. If v2 is None, return the path between v1 and the root. Otherelse, return the path between v1 and v2. If the graph is oriented from v1 to v2, sign is positive. Else, sign is negative.

`openalea.mtg.algo.`**`predecessor`**(*g*, *vid*, ***kwds*)

`openalea.mtg.algo.`**`rank`**(*g*, *v1*, *v2=None*)

`openalea.mtg.algo.`**`root`**(*g*, *vid*, *RestrictedTo='NoRestriction'*, *ContainedIn=None*)
TODO: see aml.Root doc string.

`openalea.mtg.algo.`**`sons`**(*g*, *vid*, ***kwds*)
TODO: see doc aml.sons.

`openalea.mtg.algo.`**`split`**(*g*, *scale=1*)
Split at scale.

`openalea.mtg.algo.`**`successor`**(*g*, *vid*, ***kwds*)
TODO: see aml.Successor doc string.

`openalea.mtg.algo.`**`topological_path`**(*g*, *v1*, *v2=None*, *edge=None*)

`openalea.mtg.algo.`**`trunk`**(*g*, *vtx_id*, *scale=-1*, ***kwds*)

`openalea.mtg.algo.`**`union`**(*g1*, *g2*, *vid1=None*, *vid2=None*, *edge_type='<'*)
Return the union of the MTGs g1 and g2.

Parameters

- g1, g2 (MTG) : An MTG graph
- vid1 : the anchor vertex identid=fier that belong to *g1*
- vid2 : the root of the sub_mtg that belong to *g2* which will be added to g1.
- edge_type (str) : the type of the edge which will connect vid1 to vid2

`openalea.mtg.algo.`**`vertex_at_scale`**`(`*g*, *vtx_id*, *scale*`)`

Download the source file `../../src/mtg/algo.py`.

4.6 Graphical representation of MTG

4.6.1 PlantFrame

`openalea.mtg.`**`PlantFrame`**

4.6.2 DressingData

4.6.3 3D Plot

Plot a PlantFrame.

Download the source files `../../src/mtg/plantframe/plantframe.py`, `../../src/mtg/plantframe/dresser.py`, `../../src/mtg/plantframe/turtle.py`,

4.7 utilities (plots)

Different utilities such as plot2D, plot3D, and so on…

`openalea.mtg.util.`**`mtg_plot`**`(`*g*, *scales=1*`)`

`openalea.mtg.util.`**`plot2d`**`(`*g*, *image_name*, *scale=None*, *orientation=90*`)`

Compute an image of the tree via graphviz.

Parameters

- *g* (int) : an MTG object
- *image_name* (str) : output filename e.g. test.png

Optional parameters

- *scale* (int): represents the MTG's scale to look at (default max)
- *orientation* (int): orientation angle (default 90)

`openalea.mtg.util.`**`plot3d`**`(`*g*, *scale=None*`)`

Compute a 3d view of the MTG in a simple way:

- sphere for the nodes and thin cylinder for the edges.

`openalea.mtg.util.`**`plot_nx`**`(`*g*, **args*, ***kwds*`)`

Download the source file `../../src/mtg/util.py`.

CHAPTER 5

Credits

5.1 Lead Developer

- Christophe Pradal, <christophe pradal __at__ cirad fr>

5.2 Contributors

- Christophe Godin, <christophe godin __at__ inria fr> (Concepts, Design, Previous implementation, Documentation)
- Thomas Cokelaer <thomas cokelaer __at__ isp fr> (Docmentation)

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

O

Index

A

B

C

N

O

P

R

S

T

U

V

W