HierPart Documentation

Release 0.1

Juan I. Perotti

November 19, 2015

Contents

CHAPTER 1

What is HierPart?

HierPart is a python package that implements the *hierarchical partition* data structure [1]. Furthermore, it can be used to compute the *hierarchical mutual information* between hierarchical partitions.

Hierarchical partitions can be used to represent the *hierarchical community structure* of complex networks. Therefore, the hierarchical mutual information can be used to compare *hierarchical community structures*. In other words, the hierarchical mutual information provides a generalization of the traditional approach, where the standard mutual information is used to compare node partitions, ie., community structures.

[1] Juan I. Perotti, Claudio J. Tessone, Guido Caldarelli, *The Hierarchical Mutual Information for the Comparison of Hierarchical Community Structures*, [arXiv:....] (2015)

CHAPTER 2

Table of Contents:

2.1 Installation

2.1.1 GitHub

Download or clone the package from github https://github.com/rayohauno/hierpart.git

2.2 Tutorial

The best way to learn to use **HierPart** is through a bunch of examples.

2.2.1 Example 1

The first example is about constructing a `HierarchicalPartition` object:

```
>>> from hierpart import HierarchicalPartition
>>>
>>> # A HierarchicalPartition object is created. It contains the elements 'a','b',...
>>> # which, in this case are strings. But, they can be numbers, or whatever other
>>> # thing that can be stored in a set container.
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>>
>>> # To build the hierarchy, lets start from the root.
>>> root=hp.root()
>>>
>>> # Lets add two children to the root. The elements the children will contain should
>>> # be specified. These elements should belong to the parent vertex, in this case,
>>> # the root. Otherwise, an error is raised.
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>>
>>> # Now we add children to the children, and so on...
>>> hp.add_child(_n1,['a'])
3
>>> n3=hp.add_child(n1,['b','c'])
>>> hp.add_child(n3,['b'])
5
>>> hp.add_child(n3,['c'])
6
```

This creates the following hierarchy:

2.2.2 Example 2

Two hierarchical partitions can be created, and compared with the hierarchical mutual information:

```
>>> import hierpart as hp
>>>
>>> from hierpart import HierarchicalPartition
>>> from hierpart import hierarchical_mutual_information
>>> # Lets create a HierarchicalPartition called "x"
>>> hpx=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rootx=hpx.root()
>>> n1x=hpx.add_child(rootx,['a','b','c'])
>>> n2x=hpx.add_child(rootx,['d','e','f'])
>>> dummy=hpx.add_child(n1x,['a'])
>>> n3x=hpx.add_child(n1x,['b','c'])
>>> dummy=hpx.add_child(n3x,['b'])
>>> dummy=hpx.add_child(n3x,['c'])
>>> # Lets create another slightly different HierarchicalPartition called "y"
>>> hpy=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rooty=hpy.root()
>>> n1y=hpy.add_child(rooty,['a','b','c'])
>>> n2y=hpy.add_child(rooty,['d','e','f'])
>>> dummy=hpy.add_child(n2y,['f'])
>>> n3y=hpy.add_child(n2y,['d','e'])
>>> dummy=hpy.add_child(n3y,['d'])
>>> dummy=hpy.add_child(n3y,['e'])
>>> # Lets see how they look...
>>> hpx.show()
0 ['a', 'b', 'c', 'd', 'e', 'f']
1 ['a', 'b', 'c']
2 ['d', 'e', 'f']
3 ['a']
4 ['b', 'c']
5 ['b']
6 ['c']
>>> hpy.show()
0 ['a', 'b', 'c', 'd', 'e', 'f']
1 ['a', 'b', 'c']
2 ['d', 'e', 'f']
3 ['f']
4 ['d', 'e']
5 ['d']
6 ['e']
>>> # Now we compare the hierarchies with themselves, and against each other, using the hierarchical
>>> print hierarchical_mutual_information(hpx,hpx)
1.24245332489
>>> print hierarchical_mutual_information(hpy,hpy)
1.24245332489
>>> print hierarchical_mutual_information(hpx,hpy)
0.69314718056
>>> # Now we repeat using the normalized hierarchical mutual information
>>> print normalized_hierarchical_mutual_information(hpx,hpx)
(1.0, 1.242453324894, 1.242453324894, 1.242453324894)
>>> print normalized_hierarchical_mutual_information(hpy,hpy)
(1.0, 1.242453324894, 1.242453324894, 1.242453324894)
```

```
>>> print normalized_hierarchical_mutual_information(hpx,hpy)
(0.55788589130225974, 0.69314718055994529, 1.242453324894, 1.242453324894)
```

2.3 Documentation

class hierpart.**HierarchicalPartition**(*elements*, *checks=True*)

This class implements the hierarchical partition data structure. It is able to contain any kind of element that can be contained in a set.

Parameters

- **elements** (*<list>*) – The list of elements that are going to be contained by the HierarchicalPartition.
- **checks** (*<bool>*) – If True, different (slow) checks run througth the creation of the object, plus in some other methods. This is True by default.

Returns A list of nodes that are adjacent to n.

Return type HierarchicalPartition

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f']) # This is the line that creates the Hier
>>> # All the following lines of code are here for the doctest.
>>> # However, these extra lines of code are also useful for learning purposes.
>>> root=hp.root()
>>> print root
0
>>> print hp.node_elements(root)
['a', 'b', 'c', 'd', 'e', 'f']
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> print n1
1
>>> print n2
2
>>> print hp.node_elements(n1)
['a', 'b', 'c']
>>> print hp.node_elements(n2)
['d', 'e', 'f']
>>> hp.add_child(n1,['a'])
3
>>> n3=hp.add_child(n1,['b','c'])
>>> hp.add_child(n3,['b'])
5
>>> hp.add_child(n3,['c'])
6
>>> tree=hp.tree()
>>> print tree.edges()
[(0, 1), (0, 2), (1, 3), (1, 4), (4, 5), (4, 6)]
>>> for node in hp.nodes():
...     print node, hp.node_elements(node)
...
0 ['a', 'b', 'c', 'd', 'e', 'f']
```

```
1 ['a', 'b', 'c']
2 ['d', 'e', 'f']
3 ['a']
4 ['b', 'c']
5 ['b']
6 ['c']
```

__iter__()

Iterates over the nodes of the tree. The iterator goes from the largest node to the smallest node, where the size of the nodes is measured using the method **node_size()**.

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print [node for node in hp]
[0, 1, 2, 4, 3, 5, 6]
```

add_child(*parent*, *child_elements*)

To add a child to a given node of the tree.

Remarks: If **checks** is set to True (at the moment of the creation of the HierarchicalPartition object), then, the current method checks that the set of elements in the new child is contained by the set of elements of the parent node.

Parameters

- **parent** (*"node"*) – The parent node the new child will have.
- **child_elements** (<*list*>) – The elements that will be contained in the new child.

Returns The new child.

Return type "node"

Examples

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> print hp.node_elements(n1)
['a', 'b', 'c']
```

all_elements()

Returns a list of all elements in the tree.

Returns A list of all elements contained in the tree.

Return type <list>

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c']) # This line is to show that it doesn't count
>>> print hp.all_elements()
['a', 'b', 'c', 'd', 'e', 'f']
```

bfs_traversal()
It yields over the nodes of the tree, performing a BFS algorithm that starts from the root.

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print [node for node in hp.bfs_traversal()]
[0, 1, 2, 3, 4, 5, 6]
```

branching_factors(*no_leaves=True*)
Returns a list with all branching factors of the tree; ie., one for each node.

Parameters **no_leaves** (<*bool*>) – If True, the leaves of the tree are excluded from the list.

Returns A list containing the branching factors of all nodes in the tree.

Return type <list>

Examples

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print hp.branching_factors()
[2, 2, 2]
>>> print hp.branching_factors(no_leaves=False)
[2, 2, 0, 0, 2, 0, 0]
```

branching_factors_basic_stats(*no_leaves=True*)
Return *basic_stats* about the list of depths in the tree.

Parameters **no_leaves** (<*bool*>) – If True, the leaves of the tree are excluded of the computation.

Returns

- *(float,float,float,float,int)*
- *The returned tuple contains values that are computed out of the list of all depth values among all leaves in the tree.*
- *The returned values are* – 1. average branching factor of nodes of the tree. 2. minimum branching factor of nodes of the tree. 3. maximum branching factor of nodes of the tree. 4. std of the branching factor of the nodes of the tree. 5. number of nodes of the tree that have beeen considered.

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print hp.branching_factors_basic_stats()
(2.0, 2.0, 2.0, 0.0, 3)
>>> print hp.branching_factors_basic_stats(no_leaves=False)
(0.8571428571428571, 0.0, 2.0, 0.9897433186107869, 7)
```

checks()

Returns True or False, depending on how it was defined at the object creation.

Return type <bool>

consistency()

Checks the consistency of the tree.

Returns It returns True if the consistency is right. Otherwise, it returns False.

Return type <bool>

copy()

Copy the current <HierarchicalPartition> object into a new <HierarchicalPartition> object.

Returns A copy of the current tree.

Return type HierarchicalPartition

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
```

```
>>> hpc=hp.copy()
>>> print hpc.nodes()
[0, 1, 2, 3, 4, 5, 6]
>>> print hpc.edges()
[(0, 1), (0, 2), (1, 3), (1, 4), (4, 5), (4, 6)]
>>> for node in hpc.nodes(): assert hpc.node_elements(node)==hp.node_elements(node)
```

depths_basic_stats()

Return *basic_stats* about the list of depths of the leaves in the tree.

Returns

- *(float,float,float,float,int)*
- *The returned tuple contains values that are computed out of the list of all depth values among all leaves in the tree.*
- *The returned values are* – 1. average depth of the leaves of the tree. 2. minimum depth of the leaves of the tree. 3. maximum depth of the leaves of the tree. 4. std of the depths of the leaves of the tree. 5. number of leaves in the tree.

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print hp.depths_basic_stats()
(2.25, 1.0, 3.0, 0.82915619758884995, 4)
```

dfs_traversal()

It yields over the nodes of the tree, performing a DFS algorithm that starts from the root.

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print [node for node in hp.dfs_traversal()]
[0, 2, 1, 4, 6, 5, 3]
```

edges()

It returns the list of edges of the tree.

Returns The list of edges of the tree.

Return type <list>

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print hp.edges()
[(0, 1), (0, 2), (1, 3), (1, 4), (4, 5), (4, 6)]
```

leaves()

Returns a list with the leaves in the tree.

Returns A list of the nodes in the tree that are a leaf.

Return type <list>

max_depth()

Returns the depth of the node with the maximum depth of the tree.

Returns The largest value of the depth among all nodes in the tree.

Return type <int>

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root() # the root has depth 0
>>> n1=hp.add_child(root,['a','b','c']) # n1 and n2 have depth 1
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c']) # n3 has depth 2.
>>> dummy=hp.add_child(n3,['b']) # These children of n3 have depth 3.
>>> dummy=hp.add_child(n3,['c'])
>>> print hp.max_depth()
3
```

max_size()

Returns the size of the node with the largest size in the tree, aka, the root.

Comment: This function is created to check internal consistency, and also, for completion as min_size() also exist.

Returns The size of the largest node in the tree. The size of a node is measured as the number of elements the node contains. It should be the size of the root.

Return type <int>

min_size()

Returns the size of the node with the smallest size in the tree.

Returns The size of the smallest node in the tree. The size of a node is measured as the number of elements the node contains.

Return type <int>

node_branching_factor(*node*)

Returns the number of children a given node has.

Parameters **node** (*"node"*) – A given node of the tree.

Returns The number of children the node **node** has.

Return type <int>

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print hp.node_branching_factor(root)
2
>>> print hp.node_branching_factor(n1)
2
>>> print hp.node_branching_factor(n2)
0
```

node_children(*node*)

It yields the children of the node **node**.

Parameters **node** (*"node"*) – A node of the tree.

Returns yields over the children nodes of node **node**, if any.

Return type "node_iterator"

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> for child in hp.node_children(root):
...     print child
...
1
2
```

node_children_avrg_size(*node*, *weighted=True*)

Returns the average size of the children nodes of a given node **node**.

Parameters

- **node** (*"node"*) – This is the parent of the set of childern nodes, over which the average size is computed.
- **weighted** (<*bool*>) – If True, then, each term of the average is weighted by the weight **float(node_size(child))/node_size(parent)**.

Returns The average size of the children nodes of **node**.

Return type <float>

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy1=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy2=hp.add_child(n3,['b'])
>>> dummy3=hp.add_child(n3,['c'])
>>> print [hp.node_children_avrg_size(node) for node in hp.nodes()]
[3.0, 1.6666666666666665, 0.0, 0.0, 1.0, 0.0, 0.0]
>>> print [hp.node_children_avrg_size(node,weighted=False) for node in hp.nodes()]
[3.0, 1.5, 0.0, 0.0, 1.0, 0.0, 0.0]
```

node_depth(*node*)

Returns the depth at which a given node is.

Parameters **node** (*"node"*) – A node of the tree.

Returns The depth at which node **node** is in the tree.

Return type <int>

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print hp.node_depth(root)
0
>>> print hp.node_depth(n1)
1
>>> print hp.node_depth(n3)
2
```

node_elements(*node*)

The elements contained in node "node".

Parameters **node** (*"node"*) – A node of the tree.

Returns A list of elements contained in node "node".

Return type <list>

Examples

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print hp.node_elements(root)
['a', 'b', 'c', 'd', 'e', 'f']
>>> print hp.node_elements(n1)
['a', 'b', 'c']
```

node_leaf(*node*)

Returns True if the node **node** is a leaf of the tree.

Parameters **node** (*"node"*) – A node of the tree.

Returns True if the node **node** is a leaf; else, returns False.

Return type <bool>

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print hp.node_leaf(root)
False
>>> print hp.node_leaf(n1)
False
>>> print hp.node_leaf(n2)
True
```

node_parent(*node*)

Return the parent node of **node**.

Parameters **node** (*"node"*) – A node in the graph

Returns The parent node of **node** if any. Otherwise, it returns None.

Return type "node"

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print hp.node_parent(n3)==n1
True
>>> print hp.node_parent(n3)==n2
False
>>> print hp.node_parent(root)==None
True
```

node_size(*node*)

The number of elements of a node of the tree.

Parameters **node** ("*node*") – A node of the tree.

Returns The number of elements the node **node** contains.

Return type <int>

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> print hp.node_size(root)
6
>>> print hp.node_size(n1)
3
>>> print hp.node_size(n2)
3
>>> print hp.node_size(n3)
2
```

nodes()

Returns a list of the node, ie., sub-communities, in the tree.

Returns A list of the nodes in the tree.

Return type <list>

nodes_at_depth(*depth*)

Returns a list of all the nodes in the tree that have a specified depth.

Comments: The list might be empty. The list not necessarily conform a partition of the full set of elements.

Parameters **depth** (*<int>*) – The depth at which the nodes to be returned should be.

Returns A list of the nodes at depth "depth".

Return type <list>

Examples

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy1=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy2=hp.add_child(n3,['b'])
>>> dummy3=hp.add_child(n3,['c'])
>>> print root, n1, n2, n3, dummy1, dummy2, dummy3
0 1 2 4 3 5 6
>>> for node in hp.nodes():
...     print node, hp.node_depth(node)
...
0 0
1 1
2 1
3 2
4 2
5 3
6 3
>>> print hp.nodes_at_depth(0)
[0]
>>> hp.nodes_at_depth(1)
[1, 2]
>>> hp.nodes_at_depth(2)
[3, 4]
>>> hp.nodes_at_depth(3)
[5, 6]
>>> hp.nodes_at_depth(4)
[]
```

num_edges()

Returns The number of edges, or links, in the tree of the hierarchical partition.

Return type <int>

num_nodes()

Returns The number of nodes (not elements) in the tree of the hierarchical partition.

Return type <int>

replica(*old_elements_2_new_elements*)

This method allows to replicate the current tree, into another tree, where the nodes change according to a predefined mapping.

What is this useful for? One possible usage is that of the randomization of a hierarchy.

Parameters **old_elements_2_new_elements** (*<dict>*) – It maps old elements, into new elements.

Returns The new hierarchical partition has elements with "renamed names" according to the specification in **old_elements_2_new_elements**.

Return type HierarchicalPartition

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> old_elements_2_new_elements={'a':'A', 'b':'B', 'c':'C', 'd':'D', 'e':'E', 'f':'F'}
>>> hpr=hp.replica(old_elements_2_new_elements)
>>> print hpr.all_elements()
['A', 'B', 'C', 'D', 'E', 'F']
>>> for node in hpr.nodes():
...     print node, hpr.node_elements(node)
...
0 ['A', 'B', 'C', 'D', 'E', 'F']
1 ['A', 'B', 'C']
2 ['D', 'E', 'F']
3 ['A']
4 ['B', 'C']
5 ['B']
6 ['C']
```

root()

Returns the root node of the tree.

Returns The returned node is the root of the tree.

Return type "node"

show()

Shows in the screen a list of the nodes, and their respective elements.

total_num_elements()

Returns the number of elements contained in the tree.

Returns The number of elements contained in the tree = size(root).

Return type <int>

Example

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c']) # This line is to show that it doesn't count
```

```
>>> print hp.total_num_elements()
6
```

tree()

The returned tree describes the topology of the hierarchical partition.

Returns A list of nodes that are adjacent to n.

Return type <networkx.DiGraph>

Examples

```
>>> from hierpart import HierarchicalPartition
>>> hp=HierarchicalPartition(['a','b','c','d','e','f'])
>>> root=hp.root()
>>> n1=hp.add_child(root,['a','b','c'])
>>> n2=hp.add_child(root,['d','e','f'])
>>> dummy=hp.add_child(n1,['a'])
>>> n3=hp.add_child(n1,['b','c'])
>>> dummy=hp.add_child(n3,['b'])
>>> dummy=hp.add_child(n3,['c'])
>>> tree=hp.tree()
>>> print tree.edges()
[(0, 1), (0, 2), (1, 3), (1, 4), (4, 5), (4, 6)]
```

hierpart.**save_hierarchical_partition**(*hier_part*, *fileout=None*, *fhw=None*)

It saves a HierarchicalPartition object into a file.

Parameters

- **hier_part** (*HierarchicalPartition*) – The tree to be saved.
- **fileout** (<*str*>) – The name (and path) of the file where the tree is saved.
- **fhw** (<*file-handler*>) – Should be a filehandler with writting privileges. This is optional to the use of **fileout**.

hierpart.**load_hierarchical_partition**(*filein*)

Load a Hierarchical Partition from file.

Parameters **filein** (<*str*>) – The filename (and path) to the file where a tree is stored.

Returns The loaded tree.

Return type HierarchicalPartition

hierpart.**sub_hierarchical_mutual_information**(*hierpart_x*, *hierpart_y*, *node_x*, *node_y*, *depth*, *show=False*)

Cumputes the hierarchical mutual information between two sub-trees. More specifically, it computes I(T_v ; T'_v'), where T and T' are <HierarchicalPartitions>, v is a node in T and v' is a node in T'. Also, T_v is the sub-tree obtained from T with v as root. The analogous for T'_v'.

Comments: This function is used recursivelly to compute I(T;T').

Parameters

- **hierpart_x** (*HierarchicalPartition*) – The hierarchical partition T.
- **hierpart_y** (*HierarchicalPartition*) – The hierarchical partition T'.
- **node_x** (*"node"*) – The node v. It should belong to T.

- **node_y** (*"node"*) – The node v'. It should belong to T'.
- **depth** (<*int*>) – The depth at which the nodes v and v' are. This variable is used for internal checks.
- **show** (<*bool*>) – If True, information is printed on the screen as the computation progress.

Returns The value I(T_v ; T'_v').

Return type <float>

Example

```
>>> from hierpart import HierarchicalPartition
>>> from hierpart import sub_hierarchical_mutual_information
>>> # Lets create a HierarchicalPartition called "x"
>>> hpx=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rootx=hpx.root()
>>> n1x=hpx.add_child(rootx,['a','b','c'])
>>> n2x=hpx.add_child(rootx,['d','e','f'])
>>> dummy=hpx.add_child(n1x,['a'])
>>> n3x=hpx.add_child(n1x,['b','c'])
>>> dummy=hpx.add_child(n3x,['b'])
>>> dummy=hpx.add_child(n3x,['c'])
>>> # Lets create another slightly different HierarchicalPartition called "y"
>>> hpy=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rooty=hpy.root()
>>> n1y=hpy.add_child(rooty,['a','b','c'])
>>> n2y=hpy.add_child(rooty,['d','e','f'])
>>> dummy=hpy.add_child(n2y,['f'])
>>> n3y=hpy.add_child(n2y,['d','e'])
>>> dummy=hpy.add_child(n3y,['d'])
>>> dummy=hpy.add_child(n3y,['e'])
>>> # Now lets compare sub-trees.
>>> print sub_hierarchical_mutual_information(hpx,hpy,rootx,rooty,0)
0.69314718056
>>> print sub_hierarchical_mutual_information(hpx,hpy,n1x,n1y,1)
0.0
```

hierpart.**hierarchical_mutual_information**(*hierpart_x*, *hierpart_y*, *show=False*)

Cumputes the hierarchical mutual information between two trees. More specifically, it computes I(T;T'), where T and T' are two <HierarchicalPartitions>.

Parameters

- **hierpart_x** (*HierarchicalPartition*) – The hierarchical partition T.
- **hierpart_y** (*HierarchicalPartition*) – The hierarchical partition T'.
- **show** (<*bool*>) – If True, information is printed on the screen as the computation progress.

Returns The value I(T;T').

Return type <float>

Example

```
>>> from hierpart import HierarchicalPartition
>>> from hierpart import hierarchical_mutual_information
>>> # Lets create a HierarchicalPartition called "x"
>>> hpx=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rootx=hpx.root()
>>> n1x=hpx.add_child(rootx,['a','b','c'])
>>> n2x=hpx.add_child(rootx,['d','e','f'])
>>> dummy=hpx.add_child(n1x,['a'])
>>> n3x=hpx.add_child(n1x,['b','c'])
>>> dummy=hpx.add_child(n3x,['b'])
>>> dummy=hpx.add_child(n3x,['c'])
>>> # Lets create another slightly different HierarchicalPartition called "y"
>>> hpy=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rooty=hpy.root()
>>> n1y=hpy.add_child(rooty,['a','b','c'])
>>> n2y=hpy.add_child(rooty,['d','e','f'])
>>> dummy=hpy.add_child(n2y,['f'])
>>> n3y=hpy.add_child(n2y,['d','e'])
>>> dummy=hpy.add_child(n3y,['d'])
>>> dummy=hpy.add_child(n3y,['e'])
>>> # Lets see how they look...
>>> hpx.show()
0 ['a', 'b', 'c', 'd', 'e', 'f']
1 ['a', 'b', 'c']
2 ['d', 'e', 'f']
3 ['a']
4 ['b', 'c']
5 ['b']
6 ['c']
>>> hpy.show()
0 ['a', 'b', 'c', 'd', 'e', 'f']
1 ['a', 'b', 'c']
2 ['d', 'e', 'f']
3 ['f']
4 ['d', 'e']
5 ['d']
6 ['e']
>>> # Now we compare the hierarchies with themselves, and against each other.
>>> print hierarchical_mutual_information(hpx,hpx)
1.24245332489
>>> print hierarchical_mutual_information(hpy,hpy)
1.24245332489
>>> print hierarchical_mutual_information(hpx,hpy)
0.69314718056
```

hierpart.**normalized_hierarchical_mutual_information**(*hierpart_x*, *hierpart_y*, *show=False*, *norm='CS'*)

Computes the normalized hierarchical mutual information between two partitions. More specifically, it computes i(T;T') where T and T' are two <HierarchicalPartitions>.

Parameters

- **hierpart_x** (*<HierarchicalPartition>*) – The tree T.
- **hierpart_y** (*<HierarchicalPartition>*) – The tree T'.
- **show** (*<bool=False>*) – If True, then it shows useful information during the computation process.
- **norm** (*<str='CS'>*) – One of 'CS' (or Cauchy Schwarz), 'add' (or additive), 'max' (or using

the max function). The CS is defined as I(T,T')/sqrt(I(T,T)*I(T',T')). The add is defined as 2I(T,T')/(I(T,T)+I(T',T')). Finally, the max is defined as I(T,T')/max(I(T,T),I(T',T')).

Returns It returns i(T;T'), I(T;T'), I(T;T), I(T';T')

Return type (<float>,<float>,<float>,<float>)

Example

```
>>> from hierpart import HierarchicalPartition
>>> from hierpart import normalized_hierarchical_mutual_information
>>> # Lets create a HierarchicalPartition called "x"
>>> hpx=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rootx=hpx.root()
>>> n1x=hpx.add_child(rootx,['a','b','c'])
>>> n2x=hpx.add_child(rootx,['d','e','f'])
>>> dummy=hpx.add_child(n1x,['a'])
>>> n3x=hpx.add_child(n1x,['b','c'])
>>> dummy=hpx.add_child(n3x,['b'])
>>> dummy=hpx.add_child(n3x,['c'])
>>> # Lets create another slightly different HierarchicalPartition called "y"
>>> hpy=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rooty=hpy.root()
>>> n1y=hpy.add_child(rooty,['a','b','c'])
>>> n2y=hpy.add_child(rooty,['d','e','f'])
>>> dummy=hpy.add_child(n2y,['f'])
>>> n3y=hpy.add_child(n2y,['d','e'])
>>> dummy=hpy.add_child(n3y,['d'])
>>> dummy=hpy.add_child(n3y,['e'])
>>> # Lets see how they look...
>>> hpx.show()
0 ['a', 'b', 'c', 'd', 'e', 'f']
1 ['a', 'b', 'c']
2 ['d', 'e', 'f']
3 ['a']
4 ['b', 'c']
5 ['b']
6 ['c']
>>> hpy.show()
0 ['a', 'b', 'c', 'd', 'e', 'f']
1 ['a', 'b', 'c']
2 ['d', 'e', 'f']
3 ['f']
4 ['d', 'e']
5 ['d']
6 ['e']
>>> # Now we compare the hierarchies with themselves, and against each other.
>>> print normalized_hierarchical_mutual_information(hpx,hpx)
(1.0, 1.242453324894, 1.242453324894, 1.242453324894)
>>> print normalized_hierarchical_mutual_information(hpy,hpy)
(1.0, 1.242453324894, 1.242453324894, 1.242453324894)
>>> print normalized_hierarchical_mutual_information(hpx,hpy)
(0.55788589130225974, 0.69314718055994529, 1.242453324894, 1.242453324894)
>>> print normalized_hierarchical_mutual_information(hpx,hpy,norm='CS')
(0.55788589130225974, 0.69314718055994529, 1.242453324894, 1.242453324894)
>>> print normalized_hierarchical_mutual_information(hpx,hpy,norm='add')
(0.55788589130225974, 0.69314718055994529, 1.242453324894, 1.242453324894)
>>> print normalized_hierarchical_mutual_information(hpx,hpy,norm='max')
```

```
(0.55788589130225974, 0.69314718055994529, 1.242453324894, 1.242453324894)
>>>
>>> hpy=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rooty=hpy.root()
>>> n1y=hpy.add_child(rooty,['a','b','c'])
>>> n2y=hpy.add_child(rooty,['d','e','f'])
>>> dummy=hpy.add_child(n1y,['a'])
>>> n3y=hpy.add_child(n1y,['b','c'])
>>> dummy=hpy.add_child(n3y,['b'])
>>> dummy=hpy.add_child(n3y,['c'])
>>> dummy=hpy.add_child(n2y,['d'])
>>> n4y=hpy.add_child(n2y,['e','f'])
>>> dummy=hpy.add_child(n4y,['e'])
>>> dummy=hpy.add_child(n4y,['f'])
>>> print normalized_hierarchical_mutual_information(hpx,hpy)
(0.83272228480884947, 1.242453324894, 1.242453324894, 1.791759469228055)
>>> print normalized_hierarchical_mutual_information(hpx,hpy,norm='CS')
(0.83272228480884947, 1.242453324894, 1.242453324894, 1.791759469228055)
>>> print normalized_hierarchical_mutual_information(hpx,hpy,norm='add')
(0.81896255088035252, 1.242453324894, 1.242453324894, 1.791759469228055)
>>> print normalized_hierarchical_mutual_information(hpx,hpy,norm='max')
(0.6934264036172707, 1.242453324894, 1.242453324894, 1.791759469228055)
```

hierpart.**example_fig1b1c**()

It reproduces the computations corresponding to Figs. 1a and 1b in the manuscript.

Example

```
>>> # First check; to compute:
>>> # I(Delta_{v_{Omega}};Delta_{v'_{Omega}}|{a,b,c,d,e,f}) = 0.693...
>>> # [TODO] CHECK THIS BY HAND
>>>
>>> hpx=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rootx=hpx.root()
>>> n1x=hpx.add_child(rootx,['a','b','c'])
>>> n2x=hpx.add_child(rootx,['d','e','f'])
>>>
>>> hpy=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rooty=hpy.root()
>>> n1y=hpy.add_child(rooty,['a'])
>>> n2y=hpy.add_child(rooty,['b','c'])
>>> n3y=hpy.add_child(rooty,['d','e','f'])
>>>
>>> hierarchical_mutual_information(hpx,hpy,True)
# elements(node_x) ['a', 'b', 'c', 'd', 'e', 'f']
# elements(node_y) ['a', 'b', 'c', 'd', 'e', 'f']
# partition(node_x) a,b,c;d,e,f
# partition(node_y) a;b,c;d,e,f
# Sx 0.69314718056
# Sy 1.01140426471
# Sxy 1.01140426471
# Sx+Sy-Sxy 0.69314718056
# second_term_xy 0.0
# ret_val 0.69314718056
0.6931471805599454
>>>
>>> del hpx
```

```
>>> del rootx
>>> del n1x
>>> del n2x
>>> del hpy
>>> del rooty
>>> del n1y
>>> del n2y
>>> del n3y
>>>
>>> # Second check; to compute:
>>> # I(T;T') = 0.693...
>>> # I also checked this by hand.
>>>
>>> hpx=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rootx=hpx.root()
>>> n1x=hpx.add_child(rootx,['a','b','c'])
>>> n2x=hpx.add_child(rootx,['d','e','f'])
>>> n3x=hpx.add_child(n1x,['a'])
>>> n4x=hpx.add_child(n1x,['b','c'])
>>>
>>> hpy=HierarchicalPartition(['a','b','c','d','e','f'])
>>> rooty=hpy.root()
>>> n1y=hpy.add_child(rooty,['a'])
>>> n2y=hpy.add_child(rooty,['b','c'])
>>> n3y=hpy.add_child(rooty,['d','e','f'])
>>>
>>> hierarchical_mutual_information(hpx,hpy,show=True)
# elements(node_x) ['a', 'b', 'c', 'd', 'e', 'f']
# elements(node_y) ['a', 'b', 'c', 'd', 'e', 'f']
# partition(node_x) a,b,c;d,e,f
# partition(node_y) a;b,c;d,e,f
# Sx 0.69314718056
# Sy 1.01140426471
# Sxy 1.01140426471
# Sx+Sy-Sxy 0.69314718056
# second_term_xy 0.0
# ret_val 0.69314718056
0.6931471805599454
>>>
>>> # Third check; to compute:
>>> # I(T;T) = 1.242...
>>> # I also checked this by hand.
>>>
>>> hierarchical_mutual_information(hpx,hpx,show=True)
# elements(node_x) ['a', 'b', 'c', 'd', 'e', 'f']
# elements(node_y) ['a', 'b', 'c', 'd', 'e', 'f']
# partition(node_x) a,b,c;d,e,f
# partition(node_y) a,b,c;d,e,f
# Sx 0.69314718056
# Sy 0.69314718056
# Sxy 0.69314718056
# Sx+Sy-Sxy 0.69314718056
# second_term_xy 0.318257084147
# ret_val 1.01140426471
1.0114042647073518
>>>
>>> # Fourth check; to compute:
>>> # I(T';T') = 1.011...
```

```
>>> # I also checked this by hand.
>>>
>>> hierarchical_mutual_information(hpy,hpy,show=True)
# elements(node_x) ['a', 'b', 'c', 'd', 'e', 'f']
# elements(node_y) ['a', 'b', 'c', 'd', 'e', 'f']
# partition(node_x) a;b,c;d,e,f
# partition(node_y) a;b,c;d,e,f
# Sx 1.01140426471
# Sy 1.01140426471
# Sxy 1.01140426471
# Sx+Sy-Sxy 1.01140426471
# second_term_xy 0.0
# ret_val 1.01140426471
1.0114042647073518
>>>
>>> # Fifth check; to compute:
>>> # i(T;T') = 0.685...
>>>
>>> normalized_hierarchical_mutual_information(hpx,hpy,norm='CS')
(0.68533147896158653, 0.6931471805599454, 1.0114042647073518, 1.0114042647073518)
>>> # Remember, this means i(T;T'), I(T;T'), I(T;T), I(T';T')
```

2.4 Glossary

In the glossary, equivalent terms are writen in the same entry of a list. Also, particular cases are enlisted in sub-lists.

- hierarchical partition, tree, hierarchical community structure
- **sub-communitiy, module**
  - root
- elements, nodes
- **hierarchical mutual information**
  - normalized hierarchical mutual information
  - self hierarchical mutual information

A brief description.

The terms *hierarchical partitions*, *trees* or *hierarchical communities*, although, different mathematical objects, for practical purposes, here are used as equivalent terms.

The term *node* should not be confused with the *module* of a *tree*. In some sense, a tree is a network, and as such, it has nodes. However, here, the word *node* is reserved to the elements contained in a hierarchical community structure. To refer to the "nodes" of a tree, it is used instead the term *module*, or equivalents.

CHAPTER 3

References

h

Symbols

A

B

C

D

E

H

L

M

N

R

S

T