Digraph3 Documentation

Release 3.6-2500+

Raymond Bisdorff

December 28, 2018

CONTENTS

Author Raymond Bisdorff, Emeritus Professor, University of Luxembourg FSTC - CSC/ILIAS

Version Revision: Python 3.6

CHAPTER

ONE

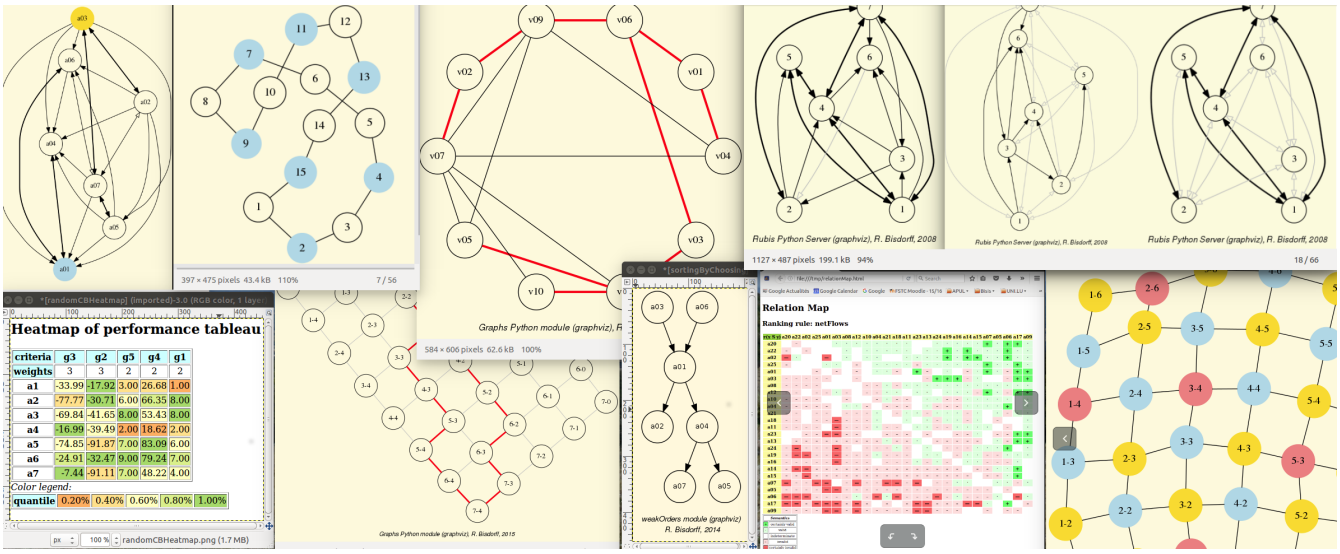
CONTENTS

CHAPTER

TWO

INTRODUCTION

This documentation, also available on the Read The Docs site: , describes the Python3 resources for implementing decision aid algorithms in the context of a bipolarly-valued outranking approach ([1],[2]). These computing resources are useful in the field of *Algorithmic Decision Theory* (https://www.algodec.org/) and more specifically in outranking based *Multiple Criteria Decision Aid* (MCDA).



Parts of the documentation:

The documentation contains, first, a set of tutorials introducing the main objects like digraphs, outranking digraphs and performance tableaux. There is also a tutorial provided on undirected graphs. Some tutorials are problem oriented and show how to compute the winner of an election, how to build a best choice recommendation, or how to linearly rank with multiple incommensurable ranking criteria. A last tutorial illustrates how to compute non isomorphic maximal independent sets in the n-cycle graph.

1

18. Bisdorff, L.C. Dias, P. Meyer, V. Mousseau and M. Pirlot (Eds.) (2015). *Evaluation and decision models with multiple criteria: Case studies*. Springer-Verlag Berlin Heidelberg, International Handbooks on Information Systems, ISBN 978-3-662-46815-9, 643 pages (downloadable content extract PDF file 401.4 kB).

2

18. Bisdorff (2013) "On Polarizing Outranking Relations with Large Performance Differences" *Journal of Multi-Criteria Decision Analysis* (Wiley) **20**:3-12 (Preprint PDF file 403.5kB)

2.1 Tutorials of the Digraph3 resources

Author Raymond Bisdorff, Emeritus Professor, University of Luxembourg FSTC/CSC

Version Revision: Python 3.6

Table of Contents

2.1.1 Working with the *Digraph3* software resources

Purpose

The basic idea of these Python3 modules is to make easy python interactive sessions or write short Python3 scripts for computing all kind of results from a bipolar valued digraph or graph. These include such features as maximal independent or irredundant choices, maximal dominant or absorbent choices, rankings, outrankings, linear ordering, etc. Most of the available computing resources are meant to illustrate the *Algorithmic Decision Theory* course given at the University of Luxembourg in the context of its Master in Information and Computer Science (MICS).

The Python development of these computing resources offers the advantage of an easy to write and maintain OOP source code as expected from a performing scripting language without loosing on efficiency in execution times compared to compiled languages such as C++ or Java.

Downloading of the Digraph3 resources

Using the Digraph3 modules is easy. You only need to have installed on your system the Python programming language of version 3.+ (readily available under Linux and Mac OS). Notice that, from Version 3.3 on, Python implements very efficiently the decimal class in C. Now, Decimal objects are mainly used in the Digraph3 characteristic valuation functions, which makes the recent python version much faster (more than twice as fast) when extensive digraph operations are performed.

Three download options are given:

1. Either (easiest under Linux or Mac OS-X), by using a git client:

```
..$ git clone https://github.com/rbisdorff/Digraph3
```

2. or a subversion client:

```
..$ svn co https://leopold-loewenheim.uni.lu/svn/repos/Digraph3
```

3. Or, with a browser access, download and extract the latest distribution tar.gz archive from this sourceforge page .

Starting a python3 session

You may start an interactive Python3 session in the `Digraph3` directory for exploring the classes and methods provided by the `digraphs` module. To do so, enter the `python3` commands following the session prompts marked with >>>. The lines without the prompt are output from the Python interpreter:

```
[\$HOME/Digraph3]\$ python3
Python 3.5.0 (default, Sep 23 2015, 13:39:18)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from digraphs import Digraph
>>> dg = Digraph('test/testdigraph')
>>> dg.save('tutorialDigraph')
*--- Saving digraph in file: <tutorialDigraph.py> ---*
>>>
```

`Digraph` object structure

All *digraphs.Digraph* object *dg* contains at least the following components:

1. A collection of digraph nodes called **actions** (decision actions): a list, set or (ordered) dictionary of nodes with 'name' and 'shortname' attributes,
2. A logical characteristic **valuationdomain**, a dictionary with three decimal entries: the minimum (-1.0, means certainly false), the median (0.0, means missing information) and the maximum characteristic value (+1.0, means certainly true),
3. The digraph **relation** : a double dictionary indexed by an oriented pair of actions (nodes) and carrying a characteristic value in the range of the previous valuation domain,
4. Its associated **gamma function** : a dictionary containing the direct successors, respectively predecessors of each action, automatically added by the object constructor,

5. Its associated **notGamma function** : a dictionary containing the actions that are not direct successors respectively predecessors of each action, automatically added by the object constructor.

See the reference manual of the *digraphs module*.

Permanent storage

The `dg.save('tutorialDigraph')` command stores the digraph *dg* in a file named `tutorialDigraph.py` with the following content:

```
# Saved digraph instance
actionset = {'1','2','3','4','5'}
valuationdomain = {'min': -1,
                   'med': 0,
                   'max': 1}
relation = {
'1': {'1':-1,'2':-1,'3':-1,'4':1,'5':-1},
'2': {'1':-1,'2':-1,'3':1,'4':-1,'5':-1},
'3': {'1':-1,'2':1,'3':-1,'4':-1,'5':1},
'4': {'1':1,'2':-1,'3':1,'4':-1,'5':1},
'5': {'1':1,'2':-1,'3':1,'4':-1,'5':-1}
}
```

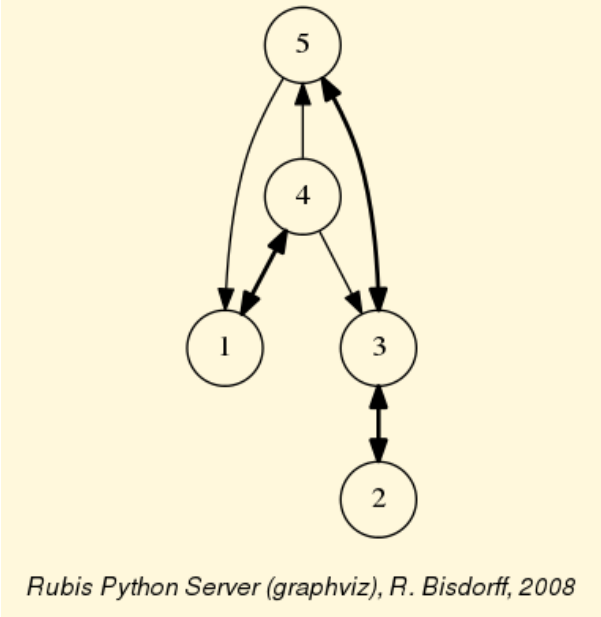
Inspecting a `Digraph` object

We may reload a previously saved `Digraph` instance from the file named `tutorialDigraph.py` with the `Digraph` class constructor and the *`digraphs.Digraph.showAll()`* method output reveals us that *dg* is a connected irreflexive digraph of order five evaluated in a valuation domain from -1 to 1.

```
>>> dg = Digraph('tutorialDigraph')
>>> dg.showAll()
*----- show details --------------*
Digraph          : tutorialDigraph
Actions          : ['1', '2', '3', '4', '5']
Valuation domain : {'med': Decimal('0'),
                    'max': Decimal('1'),
                    'min': Decimal('-1')}
* ---- Relation Table -----
  S   |  '1'      '2'      '3'      '4'      '5'
 -----|-------------------------------------------------------
  '1' |  -1.00   -1.00   -1.00   +1.00   -1.00
  '2' |  -1.00   -1.00   +1.00   -1.00   -1.00
  '3' |  -1.00   +1.00   -1.00   -1.00   +1.00
  '4' |  +1.00   -1.00   +1.00   -1.00   +1.00
  '5' |  +1.00   -1.00   +1.00   -1.00   -1.00
*--- Connected Components ---*
1: ['1', '2', '3', '4', '5']
```

The *`digraphs.Digraph.exportGraphViz()`* method generates in the current working directory a `tutorial.dot` file and a `tutorialdigraph.png` picture of the tutorial digraph *g*, if the graphviz tools are installed on your system.:

```
>>> dg.exportGraphViz('tutorialDigraph')
*---- exporting a dot file do GraphViz tools ---------*
Exporting to tutorialDigraph.dot
dot -Grankdir=BT -Tpng tutorialDigraph.dot -o tutorialDigraph.png
```



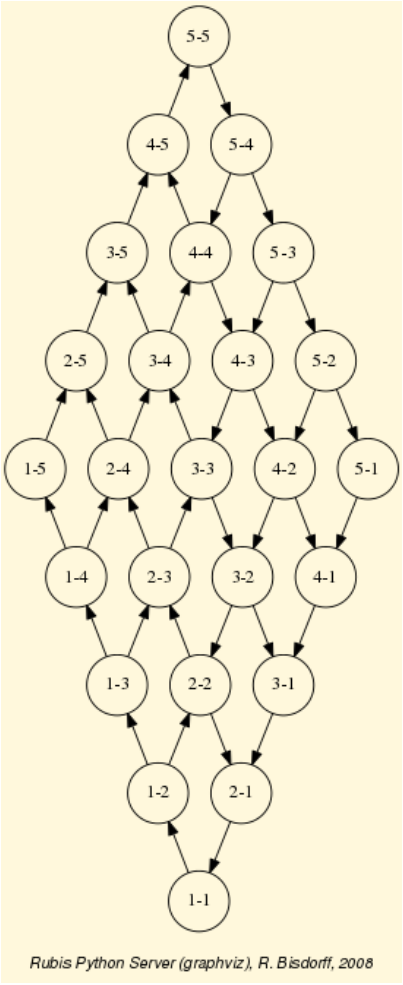
Some simple methods are easily applicable to this instantiated Digraph object *dg* , like the following *digraphs.Digraph.showStatistics()* method:

```
>>> dg.showStatistics()
*----- general statistics -------------*
for digraph             : <tutorialdigraph.py>
order                   :  5 nodes
size                    :  9 arcs
# undetermined          :  0 arcs
arc density             : 45.00
# components            :  1
                        :  [0, 1, 2, 3, 4]
outdegrees distribution :  [0, 2, 2, 1, 0]
indegrees distribution  :  [0, 2, 2, 1, 0]
degrees distribution    :  [0, 4, 4, 2, 0]
mean degree : 1.80
                                  :  [0, 1, 2, 3, 4, 'inf']
neighbourhood-depths distribution :  [0, 0, 2, 2, 1, 0]
mean neighborhood depth : 2.80
digraph diameter :  4
agglomeration distribution :
1 : 50.00
2 : 0.00
3 : 16.67
4 : 50.00
5 : 50.00
agglomeration coefficient : 33.33
>>> ...
```

Special classes

Some special classes of digraphs, like the *digraphs.CompleteDigraph*, the *digraphs.EmptyDigraph* or the oriented *digraphs.GridDigraph* class for instance, are readily available:

```
>>> from digraphs import GridDigraph
>>> grid = GridDigraph(n=5,m=5,hasMedianSplitOrientation=True)
>>> grid.exportGraphViz('tutorialGrid')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to tutorialGrid.dot
dot -Grankdir=BT -Tpng TutorialGrid.dot -o tutorialGrid.png
```



For more information about its resources, see the technical documentation of the *digraphs module* .

Back to *Tutorials of the Digraph3 resources*

2.1.2 Manipulating `Digraph` objects

- *Random digraph*
- *Graphviz drawings*
- *Asymmetric and symmetric parts*
- *Fusion by epistemic disjunction*
- *Dual, converse and codual digraphs*

- *Symmetric and transitive closures*
- *Strong components*
- *CSV storage*
- *Complete, empty and indeterminate digraphs*

Random digraph

We are starting this tutorial with generating a randomly [-1;1]-valued (*Normalized=True*) digraph of order 7, denoted *dg* and modelling a binary relation (*x S y*) defined on the set of nodes of *dg*. For this purpose, the `Digraph3` collection contains a `randomDigraphs` module providing a specific `digraphs.RandomValuationDigraph` constructor:

```
>>> from randomDigraphs import RandomValuationDigraph
>>> dg = RandomValuationDigraph(order=7,Normalized=True)
>>> dg.save('tutRandValDigraph')
```

With the `save()` method we may keep a backup version for future use of *dg* which will be stored in a file called *tutRandValDigraph.py* in the current working directory. The `Digraph` class now provides some generic methods for exploring a given `Digraph` object, like the `showShort()`, `showAll()`, `showRelationTable()` and the `showNeighborhoods()` methods:

```
>>> dg.showShort()
*----- show summary -------------*
Digraph          : randomValuationDigraph
*---- Actions ----*
['1', '2', '3', '4', '5', '6', '7']
*---- Characteristic valuation domain ----*
{'med': Decimal('0.0'), 'hasIntegerValuation': False,
'min': Decimal('-1.0'), 'max': Decimal('1.0')}
*--- Connected Components ---*
1: ['1', '2', '3', '4', '5', '6', '7']
>>> dg.showRelationTable(ReflexiveTerms=False)
* ---- Relation Table -----
r(xSy) |  '1'    '2'   '3'  '4'   '5'    '6'  '7'
-------|-------------------------------------------------------
'1'    |   -   -0.48  0.70  0.86  0.30  0.38  0.44
'2'    | -0.22   -   -0.38  0.50  0.80 -0.54  0.02
'3'    | -0.42  0.08   -    0.70 -0.56  0.84 -1.00
'4'    |  0.44 -0.40 -0.62   -    0.04  0.66  0.76
'5'    |  0.32 -0.48 -0.46  0.64   -   -0.22 -0.52
'6'    | -0.84  0.00 -0.40 -0.96 -0.18   -   -0.22
'7'    |  0.88  0.72  0.82  0.52 -0.84  0.04  -
>>> dg.showNeighborhoods()
Neighborhoods observed in digraph 'randomdomValuation'
Gamma     :
'1': in => {'5', '7', '4'}, out => {'5', '7', '6', '3', '4'}
'2': in => {'7', '3'}, out => {'5', '7', '4'}
'3': in => {'7', '1'}, out => {'6', '2', '4'}
'4': in => {'5', '7', '1', '2', '3'}, out => {'5', '7', '1', '6'}
'5': in => {'1', '2', '4'}, out => {'1', '4'}
'6': in => {'7', '1', '3', '4'}, out => set()
'7': in => {'1', '2', '4'}, out => {'1', '2', '3', '4', '6'}
 Not Gamma :
'1': in => {'6', '2', '3'}, out => {'2'}
```

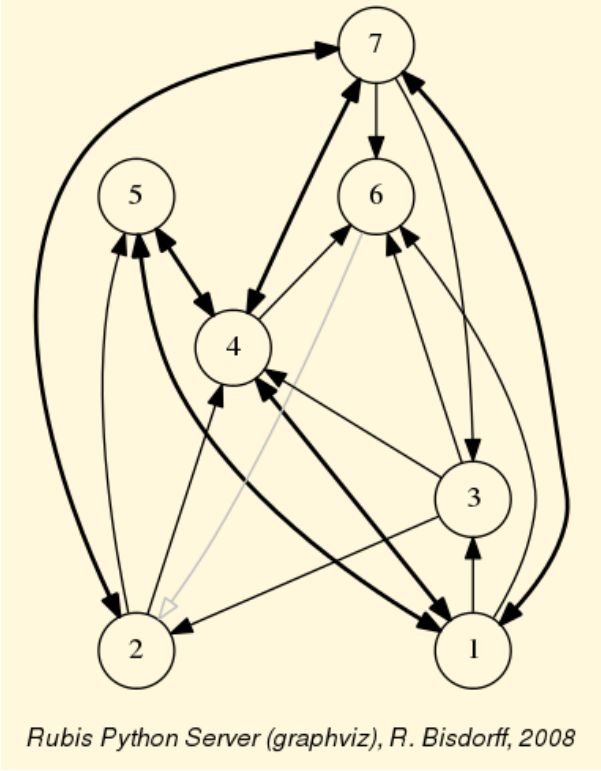
```
'2': in => {'5', '1', '4'}, out => {'1', '6', '3'}
'3': in => {'5', '6', '2', '4'}, out => {'5', '7', '1'}
'4': in => {'6'}, out => {'2', '3'}
'5': in => {'7', '6', '3'}, out => {'7', '6', '2', '3'}
'6': in => {'5', '2'}, out => {'5', '7', '1', '3', '4'}
'7': in => {'5', '6', '3'}, out => {'5'}
```

Warning: Notice that most Digraph class methods will ignore the reflexive couples by considering that the relation is indeterminate (the characteristic value $r(x\,S\,x)$ for all action x is put to the median, i.e. indeterminate, value) in this case.

Graphviz drawings

e may have an even better insight into the `Digraph` object *dg* by looking at a graphviz[1] drawing:

```
>>> dg.exportGraphViz('tutRandValDigraph')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to tutRandValDigraph.dot
dot -Grankdir=BT -Tpng tutRandValDigraph.dot -o tutRandValDigraph.png
```



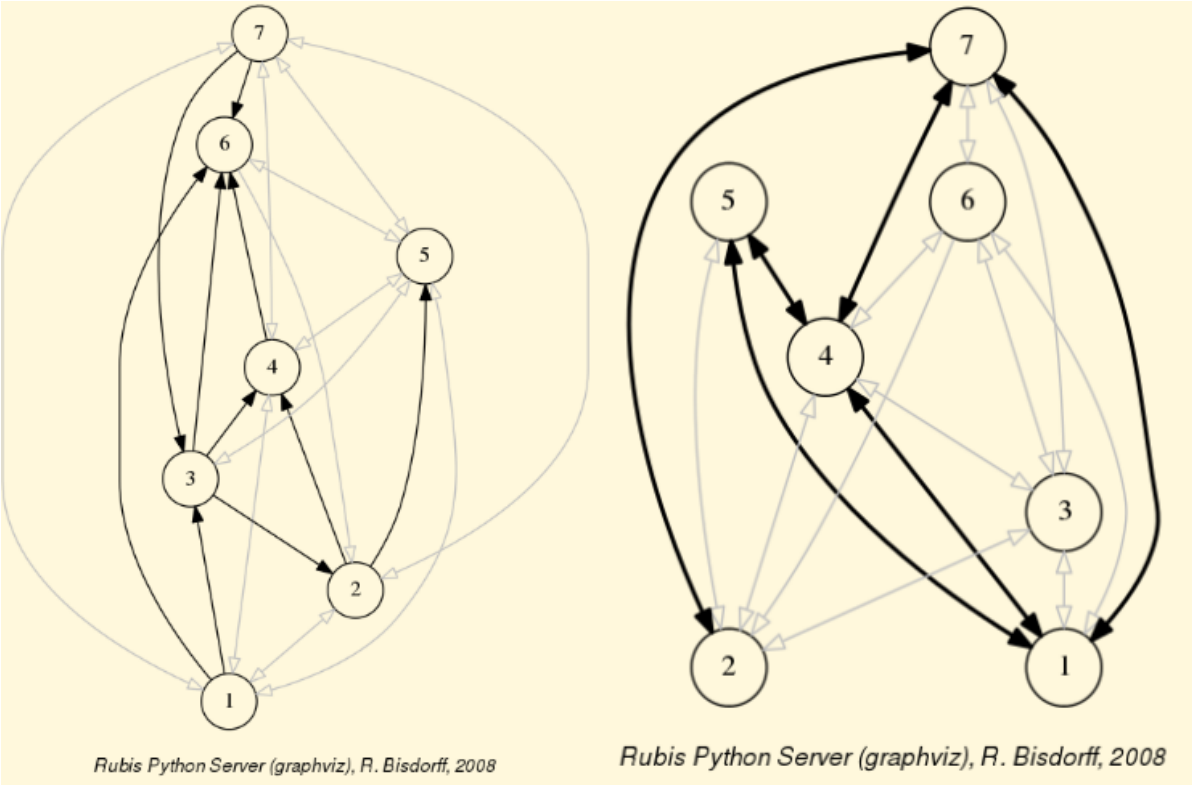
Double links are drawn in bold black with an arrowhead at each end, whereas single asymmetric links are drawn in black with an arrowhead showing the direction of the link. Notice the undetermined relational situation ($r(6\,S\,2) = 0.00$) observed between nodes '6' and '2'. The corresponding link is marked in gray with an open arrowhead in the drawing.

[1] The `exportGraphViz` method is depending on drawing tools from graphviz. On Linux Ubuntu or Debian you may try `sudo apt-get install graphviz` to install them. There are ready `dmg` installers for Mac OSX.

Asymmetric and symmetric parts

We may now extract both this symmetric as well as this asymmetric part of digraph *dg* with the help of two corresponding constructors:

```
>>> from digraphs import AsymmetricPartialDigraph, SymmetricPartialDigraph
>>> asymDg = AsymmetricPartialDigraph(dg)
>>> asymDg.exportGraphViz()
>>> symDG = SymmetricPartialDigraph(dg)
>>> symDg.exportGraphViz()
```



Note: Notice that the partial objects *asymDg* and *symDg* put to the indeterminate characteristic value all not-asymmetric, respectively not-symmetric links between nodes.

Here below, for illustration the source code of *relation* constructor of the *digraphs.AsymmetricPartialDigraph* class:

```
def _constructRelation(self):
    actions = self.actions
    Min = self.valuationdomain['min']
    Max = self.valuationdomain['max']
    Med = self.valuationdomain['med']
    relationIn = self.relation
    relationOut = {}
    for a in actions:
        relationOut[a] = {}
        for b in actions:
            if a != b:
                if relationIn[a][b] >= Med and relationIn[b][a] <= Med:
```

```
                    relationOut[a][b] = relationIn[a][b]
                elif relationIn[a][b] <= Med and relationIn[b][a] >= Med:
                    relationOut[a][b] = relationIn[a][b]
                else:
                    relationOut[a][b] = Med
            else:
                relationOut[a][b] = Med
    return relationOut
```

Fusion by epistemic disjunction

We may recover object *dg* from both partial objects *asymDg* and *symDg* with a **bipolar fusion** constructor, also called **epistemic disjunction**, available via the *digraphs.FusionDigraph* class:

```
>>> from digraphs import FusionDigraph
>>> fusDg = FusionDigraph(asymDg,symDg)
>>> fusDg.showRelationTable()
* ---- Relation Table -----
r(xSy) |  '1'    '2'   '3'  '4'   '5'    '6'  '7'
-------|-----------------------------------------------------
'1'    |  0.00 -0.48  0.70  0.86  0.30  0.38  0.44
'2'    | -0.22  0.00 -0.38  0.50  0.80 -0.54  0.02
'3'    | -0.42  0.08  0.00  0.70 -0.56  0.84 -1.00
'4'    |  0.44 -0.40 -0.62  0.00  0.04  0.66  0.76
'5'    |  0.32 -0.48 -0.46  0.64  0.00 -0.22 -0.52
'6'    | -0.84  0.00 -0.40 -0.96 -0.18  0.00 -0.22
'7'    |  0.88  0.72  0.82  0.52 -0.84  0.04  0.00
```

Dual, converse and codual digraphs

We may as readily compute the **dual**, the **converse** and the **codual** (dual and converse) of *dg*:

```
>>> from digraphs import DualDigraph, ConverseDigraph, CoDualDigraph
>>> ddg = DualDigraph(dg)
>>> ddg.showRelationTable()
-r(xSy) |  '1'    '2'   '3'  '4'   '5'    '6'  '7'
--------|---------------------------------------
'1 '    |  0.00  0.48 -0.70 -0.86 -0.30 -0.38 -0.44
'2'     |  0.22  0.00  0.38 -0.50  0.80  0.54 -0.02
'3'     |  0.42  0.08  0.00 -0.70  0.56 -0.84  1.00
'4'     | -0.44  0.40  0.62  0.00 -0.04 -0.66 -0.76
'5'     | -0.32  0.48  0.46 -0.64  0.00  0.22  0.52
'6'     |  0.84  0.00  0.40  0.96  0.18  0.00  0.22
'7'     |  0.88 -0.72 -0.82 -0.52  0.84 -0.04  0.00
>>> cdg = ConverseDigraph(dg)
>>> cdg.showRelationTable()
* ---- Relation Table -----
 r(ySx) |  '1'    '2'   '3'   '4'   '5'   '6'   '7'
--------|---------------------------------------
'1'     |  0.00 -0.22 -0.42  0.44  0.32 -0.84  0.88
'2'     | -0.48  0.00  0.08 -0.40 -0.48  0.00  0.72
'3'     |  0.70 -0.38  0.00 -0.62 -0.46 -0.40  0.82
'4'     |  0.86  0.50  0.70  0.00  0.64 -0.96  0.52
'5'     |  0.30  0.80 -0.56  0.04  0.00 -0.18 -0.84
'6'     |  0.38 -0.54  0.84  0.66 -0.22  0.00  0.04
```

```
'7'     |  0.44  0.02 -1.00  0.76 -0.52 -0.22  0.00
>>> cddg = CoDualDigraph(dg)
>>> cddg.showRelationTable()
* ---- Relation Table -----
-r(ySx) |  '1'    '2'   '3'   '4'   '5'   '6'   '7'
--------|-----------------------------------------------------
'1'     |  0.00  0.22  0.42 -0.44 -0.32  0.84 -0.88
'2'     |  0.48  0.00 -0.08  0.40  0.48  0.00 -0.72
'3'     | -0.70  0.38  0.00  0.62  0.46  0.40 -0.82
'4'     | -0.86 -0.50 -0.70  0.00 -0.64  0.96 -0.52
'5'     | -0.30 -0.80  0.56 -0.04  0.00  0.18  0.84
'6'     | -0.38  0.54 -0.84 -0.66  0.22  0.00 -0.04
'7'     | -0.44 -0.02  1.00 -0.76  0.52  0.22  0.00
```

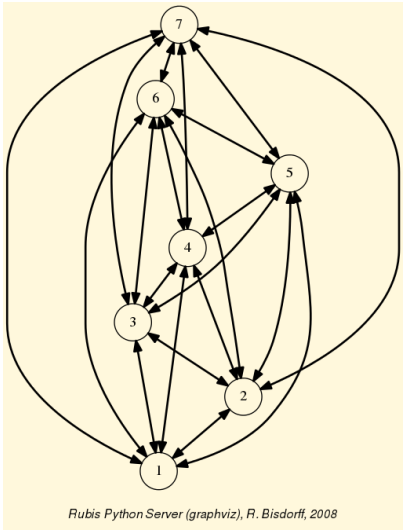
Computing the dual, respectively the converse, may also be done with prefixing the `__neg__` (`-`) or the `__invert__` (~) operator. The codual of a Digraph object may, hence, as well be computed with a **composition** (in either order) of both operations:

```
>>> ddg = -dg   # dual of dg
>>> cdg = ~dg   # converse of dg
>>> cddg = -(~dg) = ~(-dg)  # codual of dg
>>> cddg.showRelationTable()
* ---- Relation Table -----
-r(ySx) |  '1'    '2'   '3'   '4'   '5'   '6'   '7'
--------|-----------------------------------------------------
'1'     |  0.00  0.22  0.42 -0.44 -0.32  0.84 -0.88
'2'     |  0.48  0.00 -0.08  0.40  0.48  0.00 -0.72
'3'     | -0.70  0.38  0.00  0.62  0.46  0.40 -0.82
'4'     | -0.86 -0.50 -0.70  0.00 -0.64  0.96 -0.52
'5'     | -0.30 -0.80  0.56 -0.04  0.00  0.18  0.84
'6'     | -0.38  0.54 -0.84 -0.66  0.22  0.00 -0.04
'7'     | -0.44 -0.02  1.00 -0.76  0.52  0.22  0.00
```

Symmetric and transitive closures

Symmetric and transitive closure in-site constructors are also available. Note that it is a good idea, before going ahead with these in-site operations who irreversibly modify the original *dg* object, to previously make a backup version of *dg*. The simplest storage method, always provided by the generic `diggraphs.Digraph.save()`, writes out in a named file the python content of the Digraph object in string representation:

```
>>> dg.save('tutRandValDigraph')
>>> dg.closeSymmetric()
>>> dg.closeTransitive()
>>> dg.exportGraphViz('strongComponents')
```



Strong components

As the original digraph *dg* was connected (see above the result of the `dg.showShort()` command), both the symmetric and transitive closures operated together, will necessarily produce a single strong component, i.e. a complete digraph. We may sometimes wish to collapse all strong components in a given digraph and construct the so reduced digraph. Using the *digraphs.StrongComponentsCollapsedDigraph* constructor here will render a single hyper-node gathering all the original nodes :

```
>>> from digraphs import StrongComponentsCollapsedDigraph
>>> sc = StrongComponentsCollapsedDigraph(dg)
>>> sc.showAll()
*----- show detail -----*
Digraph           : tutRandValDigraph_Scc
*---- Actions ----*
['_7_1_2_6_5_3_4_']
* ---- Relation Table -----
  S      |  'Scc_1'
 -------|---------
'Scc_1' |  0.00
short         content
Scc_1         _7_1_2_6_5_3_4_
Neighborhoods:
  Gamma      :
'frozenset({'7', '1', '2', '6', '5', '3', '4'})': in => set(), out => set()
  Not Gamma :
'frozenset({'7', '1', '2', '6', '5', '3', '4'})': in => set(), out => set()
>>> ...
```

CSV storage

Sometimes it is required to exchange the graph valuation data in CSV format with a statistical package like R. For this purpose it is possible to export the digraph data into a CSV file. The valuation domain is hereby normalized by default to the range [-1,1] and the diagonal put by default to the minimal value -1:

```
>>> dg = Digraph('tutRandValDigraph')
>>> dg.saveCSV('tutRandValDigraph')
```

```
# content of file tutRandValDigraph.csv
"d","1","2","3","4","5","6","7"
"1",-1.0,0.48,-0.7,-0.86,-0.3,-0.38,-0.44
"2",0.22,-1.0,0.38,-0.5,-0.8,0.54,-0.02
"3",0.42,-0.08,-1.0,-0.7,0.56,-0.84,1.0
"4",-0.44,0.4,0.62,-1.0,-0.04,-0.66,-0.76
"5",-0.32,0.48,0.46,-0.64,-1.0,0.22,0.52
"6",0.84,0.0,0.4,0.96,0.18,-1.0,0.22
"7",-0.88,-0.72,-0.82,-0.52,0.84,-0.04,-1.0
```

It is possible to reload a Digraph instance from its previously saved CSV file content:

```
>>> dgcsv = CSVDigraph('tutRandValDigraph')
>>> dgcsv.showRelationTable(ReflexiveTerms=False)
* ---- Relation Table -----
r(xSy) |   '1'   '2'   '3'   '4'   '5'   '6'   '7'
-------|-----------------------------------------------------------
'1'    |   -   -0.48  0.70  0.86  0.30  0.38  0.44
'2'    | -0.22   -   -0.38  0.50  0.80 -0.54  0.02
'3'    | -0.42  0.08   -    0.70 -0.56  0.84 -1.00
'4'    |  0.44 -0.40 -0.62   -    0.04  0.66  0.76
'5'    |  0.32 -0.48 -0.46  0.64   -   -0.22 -0.52
'6'    | -0.84  0.00 -0.40 -0.96 -0.18   -   -0.22
'7'    |  0.88  0.72  0.82  0.52 -0.84  0.04   -
```

It is as well possible to show a colored version of the valued relation table in a system browser window tab:

```
>>> dgcsv.showHTMLRelationTable(tableTitle="Tutorial random digraph")
>>> ...
```

Tutorial random digraph

r(x S y)	1	2	3	4	5	6	7
1	0.00	-0.48	0.70	0.86	0.30	0.38	0.44
2	-0.22	0.00	-0.38	0.50	0.80	-0.54	0.02
3	-0.42	0.08	0.00	0.70	-0.56	0.84	-1.00
4	0.44	-0.40	-0.62	0.00	0.04	0.66	0.76
5	0.32	-0.48	-0.46	0.64	0.00	-0.22	-0.52
6	-0.84	0.00	-0.40	-0.96	-0.18	0.00	-0.22
7	0.88	0.72	0.82	0.52	-0.84	0.04	0.00

Positive arcs are shown in green and negative in red. Indeterminate -zero-valued- links, like the reflexive diagonal ones or the link between node *6* and node *2*, are shown in gray.

Complete, empty and indeterminate digraphs

Let us finally mention some special universal classes of digraphs that are readily available in the *digraphs* module, like the *digraphs.CompleteDigraph*, the *digraphs.EmptyDigraph* and the *digraphs.IndeterminateDigraph* classes, which put all characteristic values respectively to the *maximum*, the *minimum* or the median *indeterminate* characteristic value:

```
>>> from digraphs import CompleteDigraph, EmptyDigraph, IndeterminateDigraph
>>> help(CompleteDigraph)
Help on class CompleteDigraph in module digraphs:
class CompleteDigraph(Digraph)
 |  Parameters:
 |      order > 0; valuationdomain=(Min,Max).
 |  Specialization of the general Digraph class for generating
 |  temporary complete graphs of order 5 in {-1,0,1} by default.
 |  Method resolution order:
 |      CompleteDigraph
 |      Digraph
 |      builtins.object
...
>>> e = EmptyDigraph(order=5)
>>> e.showRelationTable()
* ---- Relation Table -----
  S   |  '1'      '2'      '3'      '4'      '5'
---- -|---------------------------------------
'1'   |  -1.00   -1.00   -1.00   -1.00   -1.00
'2'   |  -1.00   -1.00   -1.00   -1.00   -1.00
'3'   |  -1.00   -1.00   -1.00   -1.00   -1.00
'4'   |  -1.00   -1.00   -1.00   -1.00   -1.00
'5'   |  -1.00   -1.00   -1.00   -1.00   -1.00
>>> e.showNeighborhoods()
Neighborhoods:
  Gamma     :
'1': in => set(), out => set()
'2': in => set(), out => set()
'5': in => set(), out => set()
'3': in => set(), out => set()
'4': in => set(), out => set()
  Not Gamma :
'1': in => {'2', '4', '5', '3'}, out => {'2', '4', '5', '3'}
'2': in => {'1', '4', '5', '3'}, out => {'1', '4', '5', '3'}
'5': in => {'1', '2', '4', '3'}, out => {'1', '2', '4', '3'}
'3': in => {'1', '2', '4', '5'}, out => {'1', '2', '4', '5'}
'4': in => {'1', '2', '5', '3'}, out => {'1', '2', '5', '3'}
>>> i = IndeterminateDigraph()
* ---- Relation Table -----
  S   |  '1'      '2'      '3'      '4'      '5'
------|---------------------------------------
'1'   |  0.00     0.00     0.00     0.00     0.00
'2'   |  0.00     0.00     0.00     0.00     0.00
'3'   |  0.00     0.00     0.00     0.00     0.00
'4'   |  0.00     0.00     0.00     0.00     0.00
'5'   |  0.00     0.00     0.00     0.00     0.00
>>> i.showNeighborhoods()
Neighborhoods:
  Gamma     :
'1': in => set(), out => set()
'2': in => set(), out => set()
'5': in => set(), out => set()
'3': in => set(), out => set()
'4': in => set(), out => set()
  Not Gamma :
'1': in => set(), out => set()
'2': in => set(), out => set()
'5': in => set(), out => set()
```

```
'3': in => set(), out => set()
'4': in => set(), out => set()
```

Note: Notice the subtle difference between the neighborhoods of an *empty* and the neighborhoods of an *indeterminate* digraph instance. In the first kind, the neighborhoods are known to be completely *empty* whereas, in the latter, *nothing is known* about the actual neighborhoods of the nodes. These two cases illustrate why in the case of a bipolar valuation domain, we need both a *gamma* **and** a *notGamma* function.

Back to *Tutorials of the Digraph3 resources*

2.1.3 Computing the winner of an election

Linear voting profiles

The *votingProfiles module* provides resources for handling election results *[ADT-L2]*, like the `votingProfiles.LinearVotingProfile` class. We consider an election involving a finite set of candidates and finite set of weighted voters, who express their voting preferences in a complete linear ranking (without ties) of the candidates. The data is internally stored in two ordered dictionaries, one for the voters and another one for the candidates. The linear ballots are stored in a standard dictionary:

```
candidates = OrderedDict([('a1',...), ('a2',...), ('a3', ...), ...}
voters = OrderedDict([('v1',{'weight':10}), ('v2',{'weight':3}), ...}
## each voter specifies a linearly ranked list of candidates
## from the best to the worst (without ties
linearBallot = {
'v1' : ['a2','a3','a1', ...],
'v2' : ['a1','a2','a3', ...],
...
}
```

The module provides a `votingProfiles.RandomLinearVotingProfile` class for generating random instances of the `votingProfiles.LinearVotingProfile` class. In an interactive Python session we may obtain for the election of 3 candidates by 5 voters the following result:

```
>>> from votingProfiles import RandomLinearVotingProfile
>>> v = RandomLinearVotingProfile(numberOfVoters=5,
...                               numberOfCandidates=3
...                               votersWeights=[2,3,1,5,4])
>>> v.candidates
OrderedDict(['a1',{'name':'a1'}), ('a2',{'name':'a2'}), ('a3':{'name':'a3'})])
>>> v.voters
OrderedDict([('v1',{'weight': 2}), ('v2':{'weight': 3}),
 ('v3',{'weight': 1}), ('v4':{'weight': 5}),
```

```
 ('v5',{'weight': 4})])
>>> v.linearBallot
{'v4': ['a1', 'a3', 'a2'], 'v3': ['a1', 'a3', 'a2'], 'v1': ['a1', 'a2', 'a3'],
 'v5': ['a2', 'a3', 'a1'], 'v2': ['a3', 'a2', 'a1']}
 >>> ...
```

Notice that in this random example, the five voters are weighted (see Line 4). Their linear ballots can be viewed with the `showLinearBallots` method:

```
>>> v.showLinearBallots()
voters(weight)       candidates rankings
v1(2):       ['a2', 'a1', 'a3']
v2(3):       ['a3', 'a1', 'a2']
v3(1):       ['a1', 'a3', 'a2']
v4(5):       ['a1', 'a2', 'a3']
v5(4):       ['a3', 'a1', 'a2']
# voters: 15
>>> ...
```

Editing of the linear voting profile may be achieved by storing the data in a file, edit it, and reload it again:

```
>>> v.save('tutorialLinearVotingProfile')
*--- Saving linear profile in file: <tutorialLinearVotingProfile.py> ---*
>>> v = LinearVotingProfile('tutorialLinearVotingProfile')
```

Computing the winner

We may easily compute **uni-nominal votes**, i.e. how many times a candidate was ranked first, and see who is consequently the **simple majority** winner(s) in this election.

```
>>> v.computeUninominalVotes()
{'a2': 2, 'a1': 6, 'a3': 7}
>>> v.computeSimpleMajorityWinner()
['a3']
>>> ...
```

As we observe no absolute majority (8/15) of votes for any of the three candidate, we may look for the **instant runoff** winner instead (see *[ADT-L2]*):

```
>>> v.computeInstantRunoffWinner()
['a1']
>>> ...
```

We may also follow the *Chevalier de Borda*'s advice and, after a **rank analysis** of the linear ballots, compute the **Borda score** of each candidate and hence determine the **Borda winner(s)**:

```
>>> v.computeRankAnalysis()
{'a2': [2, 5, 8], 'a1': [6, 9, 0], 'a3': [7, 1, 7]}
>>> v.computeBordaScores()
{'a2': 36, 'a1': 24, 'a3': 30}
>>> v.computeBordaWinners()
['a1']
```

The Borda **rank analysis table** my be printed out with a corresponding `show` command:

```
>>> v.showRankAnalysisTable()
*----  Borda rank analysis tableau -----*
candi- | alternative-to-rank |      Borda
dates  |   1     2     3     | score  average
-------|-----------------------------------
 'a1'  |   6     9     0     |  24     1.60
 'a3'  |   7     1     7     |  30     2.00
 'a2'  |   2     5     8     |  36     2.40
>>> ...
```

The Condorcet winner

In our randomly generated election results, we are lucky: The instant runoff winner and the Borda winner both are candidate *a1*. However, we could also follow the *Marquis de Condorcet*'s advice, and compute the **majority margins** obtained by voting for each individual pair of candidates. For instance, candidate *a1* is ranked four times before and once behind candidate *a2*. Hence the majority margin *M(a1,a2)* is 4 - 1 = +3. These majority margins define on the set of candidates what we call the **Condorcet digraph**. The *votingProfiles.CondorcetDigraph* class (a specialization of the *digraphs.Digraph* class) is available for handling such pairwise majority margins:

```
>>> from votingProfiles import CondorcetDigraph
>>> cdg = CondorcetDigraph(v,hasIntegerValuation=True)
>>> cdg.showAll()
*----- show detail -------------*
Digraph          : rel_randLinearProfile
*---- Actions ----*
['a1', 'a2', 'a3']
*---- Characteristic valuation domain ----*
{'max': Decimal('15.0'), 'med': Decimal('0'),
 'min': Decimal('-15.0'), 'hasIntegerValuation': True}
* ---- majority margins -----
   M(x,y)    |  'a1'   'a2'  'a3'
  ----------|-----------------------------------------
    'a1'     |    0     11     1
    'a2'     |  -11      0    -1
    'a3'     |   -1      1     0
Valuation domain: [-15;+15]
```

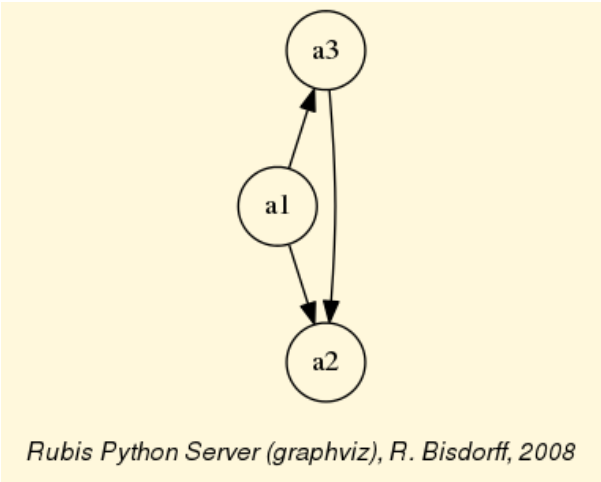
A candidate *x*, showing a positive majority margin *M(x,y)*, is beating candidate *y* with an absolute majority in a pairwise voting. Hence, a candidate showing only positive terms in her row in the Condorcet digraph relation table, beats all other candidates with absolute majority of votes. Condorcet recommends to declare this candidate (is always unique, why?) the winner of the election. Here we are lucky, it is again candidate *a1* who is hence the **Condorcet winner**:

```
>>> cdg.computeCondorcetWinner()
['a1']
```

By seeing the majority margins like a bipolarly-valued characteristic function for a global preference relation defined on the set of canditates, we may use all operational resources of the generic `Digraph` class (see *Working with the Digraph3 software resources*), and especially its `exportGraphViz` method[1], for visualizing an election result:

```
>>> cdg.exportGraphViz('tutorialLinearBallots')
*---- exporting a dot file for GraphViz tools ---------*
```

```
Exporting to tutorialLinearBallots.dot
dot -Grankdir=BT -Tpng tutorialLinearBallots.dot -o tutorialLinearBallots.png
```



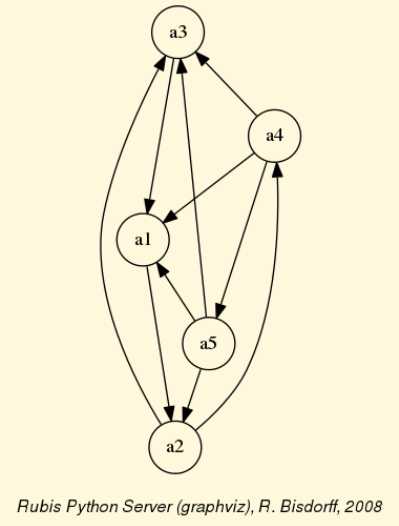
Cyclic social preferences

Usually, when aggregating linear ballots, there appear cyclic social preferences. Let us consider for instance the following linear voting profile and construct the corresponding Condorcet digraph:

```
>>> v.showLinearBallots()
voters(weight)       candidates rankings
v1(1):       ['a1', 'a3', 'a5', 'a2', 'a4']
v2(1):       ['a1', 'a2', 'a4', 'a3', 'a5']
v3(1):       ['a5', 'a2', 'a4', 'a3', 'a1']
v4(1):       ['a3', 'a4', 'a1', 'a5', 'a2']
v5(1):       ['a4', 'a2', 'a3', 'a5', 'a1']
v6(1):       ['a2', 'a4', 'a5', 'a1', 'a3']
v7(1):       ['a5', 'a4', 'a3', 'a1', 'a2']
v8(1):       ['a2', 'a4', 'a5', 'a1', 'a3']
v9(1):       ['a5', 'a3', 'a4', 'a1', 'a2']
>>> cdg = CondorcetDigraph(v)
>>> cdg.showRelationTable()
* ---- Relation Table -----
  S    |  'a1'   'a2'   'a3'   'a4'    'a5'
------|-------------------------------------
'a1'  |   -     0.11  -0.11  -0.56   -0.33
'a2'  | -0.11    -     0.11   0.11   -0.11
'a3'  |  0.11  -0.11    -    -0.33   -0.11
'a4'  |  0.56  -0.11   0.33    -      0.11
'a5'  |  0.33   0.11   0.11  -0.11     -
```

Now, we cannot find any completely positive row in the relation table. No one of the five candidates is beating all the others with an absolute majority of votes. There is no Condorcet winner anymore. In fact, when looking at a graphviz drawing of this Condorcet digraph, we may observe cyclic preferences, like ($a1 > a2 > a3 > a1$) for instance.

```
>>> cdg.exportGraphViz('cycles')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to cycles.dot
dot -Grankdir=BT -Tpng cycles.dot -o cycles.png
```



But, there may be many cycles appearing in a digraph, and, we may detect and enumerate all minimal chordless circuits in a Digraph instance with the `computeChordlessCircuits()` method:

```
>>> cdg.computeChordlessCircuits()
[(['a2', 'a3', 'a1'], frozenset({'a2', 'a3', 'a1'})),
 (['a2', 'a4', 'a5'], frozenset({'a2', 'a5', 'a4'})),
 (['a2', 'a4', 'a1'], frozenset({'a2', 'a1', 'a4'}))]
```

Condorcet's approach for determining the winner of an election is hence not decisive in all circumstances and we need to exploit more sophisticated approaches for finding the winner of the election on the basis of the majority margins of the given linear ballots (see *[BIS-2008]*).

Many more tools for exploiting voting results are available, see the technical documentation of the *votingProfiles module*.

Back to *Tutorials of the Digraph3 resources*

2.1.4 Working with the `outrankingDigraphs` module

- *Outranking digraph*
- *Browsing the performances*
- *Valuation semantics*
- *Pairwise comparisons*
- *Recoding the valuation*
- *Codual digraph*
- *XMCDA 2.0*

See also the technical documentation of the *outrankingDigraphs module*.

Outranking digraph

In this *Digraph3* module, the root *outrankingDigraphs.OutrankingDigraph* class provides a generic **outranking digraph model**. A given object of this class consists in:

1. a potential set of decision **actions** : a dictionary describing the potential decision actions or alternatives with 'name' and 'comment' attributes,
2. a coherent family of **criteria**: a dictionary of criteria functions used for measuring the performance of each potential decision action with respect to the preference dimension captured by each criterion,
3. the **evaluations**: a dictionary of performance evaluations for each decision action or alternative on each criterion function.
4. the digraph **valuationdomain**, a dictionary with three entries: the *minimum* (-100, means certainly no link), the *median* (0, means missing information) and the *maximum* characteristic value (+100, means certainly a link),
5. the **outranking relation** : a double dictionary defined on the Cartesian product of the set of decision alternatives capturing the credibility of the pairwise *outranking situation* computed on the basis of the performance differences observed between couples of decision alternatives on the given family if criteria functions.

With the help of the *outrankingDigraphs.RandomBipolarOutrankingDigraph* class (of type *outrankingDigraphs.BipolarOutrankingDigraph*) , let us generate for illustration a random bipolar outranking digraph consisting of 7 decision actions denoted *a01*, *a02*, ..., *a07*:

```
>>> from outrankingDigraphs import RandomBipolarOutrankingDigraph
>>> odg = RandomBipolarOutrankingDigraph()
>>> odg.showActions()
*----- show digraphs actions --------------*
key:  a01
name:       random decision action
comment:    RandomPerformanceTableau() generated.
key:  a02
name:       random decision action
comment:    RandomPerformanceTableau() generated.
...
...
key:  a07
name:       random decision action
comment:    RandomPerformanceTableau() generated.
>>> ...
```

In this example we consider furthermore a family of seven equisignificant cardinal criteria functions *g01*, *g02*, ..., *g07*, measuring the performance of each alternative on a rational scale from 0.0 to 100.00. In order to capture the evaluation's uncertainty and imprecision, each criterion function *g1* to *g7* admits three performance discrimination thresholds of 10, 20 and 80 pts for warranting respectively any indifference, preference and veto situations:

```
>>> odg.showCriteria()
*----  criteria -----*
g01 'digraphs.RandomPerformanceTableau() instance'
  Scale = [0.0, 100.0]
  Weight = 3.0
  Threshold pref : 20.00 + 0.00x ; percentile:  0.28
  Threshold ind : 10.00 + 0.00x ; percentile:  0.095
  Threshold veto : 80.00 + 0.00x ; percentile:  1.0
g02 'digraphs.RandomPerformanceTableau() instance'
  Scale = [0.0, 100.0]
  Weight = 3.0
```

```
  Threshold pref : 20.00 + 0.00x ; percentile:  0.33
  Threshold ind : 10.00 + 0.00x ; percentile:  0.19
  Threshold veto : 80.00 + 0.00x ; percentile:  0.95
...
...
g07 'digraphs.RandomPerformanceTableau() instance'
  Scale = [0.0, 100.0]
  Weight = 10.0
  Threshold pref : 20.00 + 0.00x ; percentile:  0.476
  Threshold ind : 10.00 + 0.00x ; percentile:  0.238
  Threshold veto : 80.00 + 0.00x ; percentile:  1.0
```

The performance evaluations of each decision alternative on each criterion are gathered in a *performance tableau*:

```
>>> odg.showPerformanceTableau()
*----  performance tableau -----*
criteria |  'a01'   'a02'   'a03'   'a04'   'a05'   'a06'   'a07'
---------|-------------------------------------------------------
  'g01'  |   9.6    48.8    21.7    37.3    81.9    48.7    87.7
  'g02'  |  90.9    11.8    96.6    41.0    34.0    53.9    46.3
  'g03'  |  97.8    46.4    83.3    30.9    61.5    85.4    82.5
  'g04'  |  40.5    43.6    53.2    17.5    38.6    21.5    67.6
  'g05'  |  33.0    40.7    96.4    55.1    46.2    58.1    52.6
  'g06'  |  47.6    19.0    92.7    55.3    51.7    26.6    40.4
  'g07'  |  41.2    64.0    87.7    71.6    57.8    59.3    34.7
>>> ...
```

Browsing the performances

We may visualize the same performance tableau in a two-colors setting in the default system browser with the command:

```
>>> odg.showHTMLPerformanceTableau()
>>> ...
```

Performance table

criterion	a01	a02	a03	a04	a05	a06	a07
g01	9.56	48.84	21.73	37.26	81.93	48.68	87.73
g02	90.94	11.79	96.56	41.03	33.96	53.90	46.27
g03	97.79	46.36	83.35	30.89	61.55	85.36	82.53
g04	40.53	43.61	53.22	17.50	38.65	21.51	67.62
g05	33.04	40.67	96.42	55.13	46.21	58.10	52.65
g06	47.57	19.00	92.65	55.32	51.70	26.64	40.39
g07	41.21	63.95	87.70	71.61	57.79	59.29	34.69

It is worthwhile noticing that *green* and *red* marked evaluations indicate *best*, respectively *worst*, performances of an alternative on a criterion. In this example, we may hence notice that alternative *a03* is in fact best performing on *four* out of *seven* criteria.

We may, furthermore, rank the alternatives on the basis of the weighted marginal quintiles and visualize the same performance tableau in an even more colorful and sorted setting:

```
>>> odg.showHTMLPerformanceHeatmap(quantiles=5,colorLevels=5)
>>> ...
```

Performance heatmap

criterion	g07	g06	g03	g04	g05	g02	g01
weight	10.00	7.00	6.00	5.00	3.00	3.00	3.00
a03	87.70	92.65	83.35	53.22	96.42	96.56	21.73
a04	71.61	55.32	30.89	17.50	55.13	41.03	37.26
a05	57.79	51.70	61.55	38.65	46.21	33.96	81.93
a07	34.69	40.39	82.53	67.62	52.65	46.27	87.73
a06	59.29	26.64	85.36	21.51	58.10	53.90	48.68
a01	41.21	47.57	97.79	40.53	33.04	90.94	9.56
a02	63.95	19.00	46.36	43.61	40.67	11.79	48.84

Color legend

quantile	0.2	0.4	0.6	0.8	1.0

There is no doubt that action *a03*, with a performance in the highest quintile in five out of seven criteria, appears definitely to be best performing. Action *a05* shows a more or less average performance on most criteria, whereas action *a02* appears to be the weakest alternative.

Valuation semantics

Considering the given performance tableau, the `outrankingDigraphs.BipolarOutrankingDigraph` class constructor computes the characteristic value $r(x\,S\,y)$ of a pairwise outranking relation "$x\,S\,y$" (see *[BIS-2013]*, *[ADT-L7]*) in a default valuation domain [-100.0,+100.0] with the median value 0.0 acting as indeterminate characteristic value. The semantics of r(x S y) are the following:

1. If $r(x\,S\,y) > 0.0$ it is more *True* than *False* that *x outranks y*, i.e. alternative x is at least as well performing than alternative y **and** there is no considerable negative performance difference observed in disfavour of x,
2. If $r(x\,S\,y) < 0.0$ it is more *False* than *True* that *x outranks y*, i.e. alternative x is **not** at least as well performing than alternative y **and** there is no considerable positive performance difference observed in favour of x,
3. If $r(x\,S\,y) = 0.0$ it is *indeterminate* whether *x outranks y or not*.

The resulting bipolarly valued outranking relation may be inspected with the following command:

```
>>> odg.showRelationTable()
* ---- Relation Table -----
r(x S y)|   'a01'   'a02'   'a03'   'a04'   'a05'   'a06'   'a07'
--------|-------------------------------------------------------
 'a01'  |   +0.00  +29.73  -29.73  +13.51  +48.65  +40.54  +48.65
 'a02'  |  +13.51   +0.00 -100.00  +37.84  +13.51  +43.24  -37.84
 'a03'  |  +83.78 +100.00   +0.00  +91.89  +83.78  +83.78  +70.27
```

```
 'a04'  |  +24.32  +48.65  -56.76   +0.00  +24.32  +51.35  +24.32
 'a05'  |  +51.35 +100.00  -70.27  +72.97   +0.00  +51.35  +32.43
 'a06'  |  +16.22  +72.97  -51.35  +35.14  +32.43   +0.00  +37.84
 'a07'  |  +67.57  +45.95  -24.32  +27.03  +27.03  +45.95   +0.00
>>> odg.valuationdomain
{'min': Decimal('-100.0'), 'max': Decimal('100.0'), 'med': Decimal('0.0')}
```

Pairwise comparisons

From above given semantics, we may consider that *a01* outranks *a02* ($r(a_{01}\,S\,a_{02}) > 0.0$), but not *a03* ($r(a_{01}\,S\,a_{03}) < 0.0$). In order to comprehend the characteristic values shown in the relation table above, we may furthermore have a look at the pairwise multiple criteria comparison between alternatives *a01* and *a02*:

```
>>> odg.showPairwiseComparison('a01','a02')
*------------  pairwise comparison ----*
Comparing actions : (a01, a02)
crit. wght.   g(x)  g(y)    diff        | ind     p    concord  |
------------------------------ ---------------------------------
g01    3.00   9.56  48.84  -39.28       | 10.00  20.00   -3.00  |
g02    3.00  90.94  11.79  +79.15       | 10.00  20.00   +3.00  |
g03    6.00  97.79  46.36  +51.43       | 10.00  20.00   +6.00  |
g04    5.00  40.53  43.61   -3.08       | 10.00  20.00   +5.00  |
g05    3.00  33.04  40.67   -7.63       | 10.00  20.00   +3.00  |
g06    7.00  47.57  19.00  +28.57       | 10.00  20.00   +7.00  |
g07   10.00  41.21  63.95  -22.74       | 10.00  20.00  -10.00  |
-------------------------------------------------------------------
Valuation in range: -37.00 to +37.00; global concordance: +11.00
```

The outranking valuation characteristic appears as **majority margin** resulting from the difference of the weights of the criteria in favor of the statement that alternative *a01* is at least well performing as alternative *a02*. No considerable performance difference being observed, no veto or counter-veto situation is triggered in this pairwise comparison. Such a case is, however, observed for instance when we pairwise compare the performances of alternatives *a03* and *a02*:

```
>>> odg.showPairwiseComparison('a03','a02')
*------------  pairwise comparison ----*
Comparing actions : (a03, a02)
crit.  wght.  g(x)  g(y)    diff        | ind     p    concord  |  v  veto/
↪counter-
-------------------------------------------------------------------------------
↪-
g01    3.00  21.73  48.84  -27.11       | 10.00  20.00   -3.00  |
g02    3.00  96.56  11.79  +84.77       | 10.00  20.00   +3.00  |  80.00  +1.00
g03    6.00  83.35  46.36  +36.99       | 10.00  20.00   +6.00  |
g04    5.00  53.22  43.61   +9.61       | 10.00  20.00   +5.00  |
g05    3.00  96.42  40.67  +55.75       | 10.00  20.00   +3.00  |
g06    7.00  92.65  19.00  +73.65       | 10.00  20.00   +7.00  |
g07   10.00  87.70  63.95  +23.75       | 10.00  20.00  +10.00  |
-------------------------------------------------------------------------------
↪-
 Valuation in range: -37.00 to +37.00; global concordance: +31.00
>>> ...
```

This time, we observe a considerable out-performance of *a03* against *a02* on criterion g02 (see second row in the relation table above). We therefore notice a positively polarized *certainly confirmed* outranking situation in this case *[BIS-2013]*.

Recoding the valuation

All outranking digraphs, being of root type *digraphs.Digraph*, inherit the methods available under this class. The characteristic valuation domain of an outranking digraph may be recoded with the `digraphs.Digraph.recodeValutaion()` method below to the integer range [-37,+37], i.e. plus or minus the global significance of the family of criteria considered in this example instance:

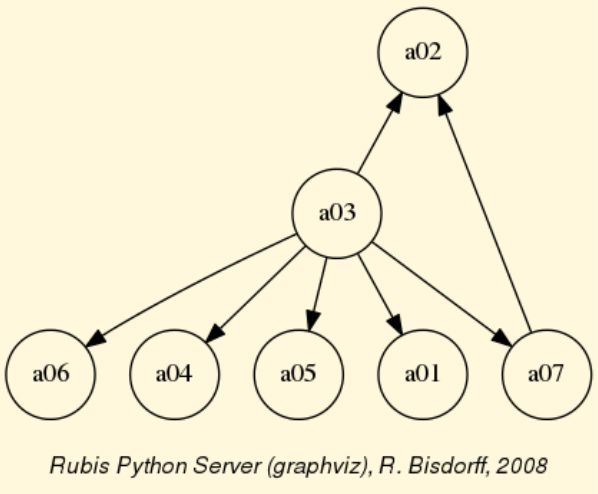
```
>>> odg.recodeValuation(-37,+37)
>>> odg.valuationdomain['hasIntegerValuation'] = True
>>> Digraph.showRelationTable(odg)
* ---- Relation Table -----
* ---- Relation Table -----
  S   | 'a01'   'a02'   'a03'  'a04'   'a05'   'a06'   'a07'
------|-------------------------------------------------------
'a01' |    0     +11     -11     +5     +17     +14     +17
'a02' |   +5       0     -37    +13      +5     +15     -14
'a03' |  +31     +37       0    +34     +31     +31     +26
'a04' |   +9     +18     -21      0      +9     +19      +9
'a05' |  +19     +37     -26    +27       0     +19     +12
'a06' |   +6     +27     -19    +13     +12       0     +14
'a07' |  +25     +17      -9     +9      +9     +17       0
Valuation domain:  {'hasIntegerValuation': True, 'min': Decimal('-37'),
                    'max': Decimal('37'), 'med': Decimal('0.000')}
>>> ...
```

Note: Notice that the reflexive self comparison characteristic $r(xSx)$ is set by default to the median indeterminate valuation value 0; the reflexive terms of binary relation being generally ignored in most of the `Digraph3` resources.

Codual digraph

From the theory (see *[BIS-2013]*, *[ADT-L7]*) we know that the bipolarly outranking relation is **weakly complete**, i.e. if $r(x\,S\,y) < 0.0$ then $r(y\,S\,x) >= 0.0$. From this property follows that the bipolarly valued outranking relation verifies the coduality principle: the dual (-) of the converse (~) of the outranking relation corresponds to its strict outranking part. We may visualize the codual (strict) outranking digraph with a graphviz drawing[1]:

```
>>> cdodg = -(~odg)
>>> cdodg.exportGraphViz('codualOdg')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to codualOdg.dot
dot -Grankdir=BT -Tpng codualOdg.dot -o codualOdg.png
>>> ...
```



It becomes readily clear now from the picture above that alternative *a03* strictly outranks in fact all the other alternatives. Hence, *a03* appears as **Condorcet winner** and may be recommended as *best decision action* in this illustrative preference modelling exercise.

XMCDA 2.0

As with all Digraph instances, it is possible to store permanently a copy of the outranking digraph *odg*. As its outranking relation is automatically generated by the *outrankingDigraphs.BipolarOutrankingDigraph* class constructor on the basis of a given performance tableau, it is sufficient to save only the latter. For this purpose we are using the XMCDA 2.00 XML encoding scheme of MCDA data, as provided by the Decision Deck Project (see https://www.decision-deck.org/):

```
>>> PerformanceTableau.saveXMCDA2(odg,'tutorialPerfTab')
*----- saving performance tableau in XMCDA 2.0 format  -------------*
File: tutorialPerfTab.xml saved !
>>> ...
```

The resulting XML file my be visualized in a browser window (other than Chrome or Chromium) with a corresponding XMCDA style sheet (see here). Hitting `Ctrl U` in Firefox will open a browser window showing the underlying xml encoded raw text. It is thus possible to easily edit and update as needed a given performance tableau instance. Reinstantiating again a corresponding updated *odg* object goes like follow:

```
>>> pt = XMCDA2PerformanceTableau('tutorialPerfTab')
>>> odg = BipolarOutrankingDigraph(pt)
>>> odg.showRelationTable()
* ---- Relation Table -----
  S    |  'a01'     'a02'   'a03'   'a04'   'a05'   'a06'   'a07'
------|-------------------------------------------------------------
'a01' |   +0.00   +29.73  -29.73  +13.51  +48.65  +40.54  +48.65
'a02' |   +13.51  +0.00  -100.00  +37.84  +13.51  +43.24  -37.84
'a03' |   +83.78  +100.00  +0.00  +91.89  +83.78  +83.78  +70.27
'a04' |   +24.32  +48.65  -56.76   +0.00  +24.32  +51.35  +24.32
'a05' |   +51.35  +100.00  -70.27  +72.97  +0.00  +51.35  +32.43
'a06' |   +16.22  +72.97  -51.35  +35.14  +32.43   +0.00  +37.84
'a07' |   +67.57  +45.95  -24.32  +27.03  +27.03  +45.95   +0.00
>>> ...
```

We recover the original bipolarly valued outranking characteristics, and we may restart again the preference modelling process.

Many more tools for exploiting bipolarly valued outranking digraphs are available in the Digraph3 resources (see the technical documentation of the *outrankingDigraphs module* and the *perfTabs module*).

Back to *Tutorials of the Digraph3 resources*

2.1.5 Generating random performance tableaux

Introduction

The `randomPerfTabs` module provides several constructors for random performance tableaux generators of different kind, mainly for the purpose of testing implemented methods and tools presented and discussed in the Algorithmic Decision Theory course at the University of Luxembourg. This tutorial concerns the four most useful generators:

1. The simplest model, called **RandomPerformanceTableau**, generates a set of *n* decision actions, a set of *m* real-valued performance criteria, ranging by default from 0.0 to 100.0, associated with default discrimination thresholds: 2.5 (ind.), 5.0 (pref.) and 60.0 (veto). The generated performances are Beta(2.2) distributed on each measurement scale.

2. One of the most useful random generator, called **RandomCBPerformanceTableau**, proposes two decision objectives, named *Costs* (to be minimized) respectively *Benefits* (to be maximized) model; its purpose being to generate more or less contradictory performances on these two, usually opposed, objectives. *Low costs* will randomly be coupled with *low benefits*, whereas *high costs* will randomly be coupled with high benefits.

3. Many multiple criteria decision problems concern three decision objectives which take into account *economical*, *societal* as well as *environmental* aspects. For this type of performance tableau model, we provide a specific generator, called **Random3ObjectivesPerformanceTableau**.

4. In order to study aggregation of linear orders, we provide a model called **RandomRankPerformanceTableau** which provides linearly ordered performances without ties on multiple criteria for a given number of decision actions.

Generating standard random performance tableaux

The `randomPerfTabs.RandomPerformanceTableau` class, the simplest of the kind, specializes the generic `refTabs.PerformanceTableau` class, and takes the following parameters:

- numberOfActions := nbr of decision actions.
- numberOfCriteria := number performance criteria.
- weightDistribution := 'random' (default) | 'fixed' | 'equisignificant'.

  If 'random', weights are uniformly selected randomly
  from the given weight scale;

If 'fixed', the weightScale must provided a corresponding weights distribution;

If 'equisignificant', all criterion weights are put to unity.

- weightScale := [Min,Max] (default =(1,numberOfCriteria).
- IntegerWeights := True (default) | False (normalized to proportions of 1.0).
- commonScale := [a,b]; common performance measuring scales (default = [0.0,100.0])
- commonThresholds := [(q0,q1),(p0,p1),(v0,v1)]; common indifference(q), preference (p) and considerable performance difference discrimination thresholds. For each threshold type *x* in *{q,p,v}*, the float x0 value represents a constant percentage of the common scale and the float x1 value a proportional value of the actual perforamance measure. Default values are [(2.5.0,0.0),(5.0,0.0),(60.0,0,0)].
- commonMode := common random distribution of random performance measurements:

  ('uniform',None,None), uniformly distributed float values on the given common scales' range [Min,Max].

  ('normal',*mu*,*sigma*), truncated Gaussian distribution, by default *mu* = (*b-a*)/2 and *sigma* = (*b-a*)/4.

  ('triangular',*mode*,*repartition*), generalized triangular distribution with a probability repartition parameter specifying the probability mass accumulated until the mode value. By deafult, *mode* = (*b-a*)/2 and *repartition* = 0.5.

  ('beta',None,(alpha,beta)), a beta generator with standard alpha=2 and beta=2 parameters.

- valueDigits := <integer>, precision of performance measurements (2 decimal digits by default).

Code example

```
>>> from randomPerfTabs import RandomPerformanceTableau
>>> t = RandomPerformanceTableau(numberOfActions=3,numberOfCriteria=1,seed=100)
>>> t.actions
    {'a1': {'comment': 'RandomPerformanceTableau() generated.', 'name': 'random␣
↪decision action'},
     'a2': {'comment': 'RandomPerformanceTableau() generated.', 'name': 'random␣
↪decision action'},
     'a3': {'comment': 'RandomPerformanceTableau() generated.', 'name': 'random␣
↪decision action'}}
>>> t.criteria
    {'g1': {'thresholds': {'ind' : (Decimal('10.0'), Decimal('0.0')),
                           'veto': (Decimal('80.0'), Decimal('0.0')),
                           'pref': (Decimal('20.0'), Decimal('0.0'))},
            'scale': [0.0, 100.0],
            'weight': Decimal('1'),
            'name': 'digraphs.RandomPerformanceTableau() instance',
            'comment': 'Arguments: ; weightDistribution=random;
                weightScale=(1, 1); commonMode=None'}}
>>> t.evaluation
    {'g01': {'a01': Decimal('45.95'),
             'a02': Decimal('95.17'),
             'a03': Decimal('17.47')
            }
    }
```

Generating random Cost-Benefit tableaux

We provide the `randomPerfTabs.RandomCBPerformanceTableau` class for generating random *Cost* versus *Benefit* organized performance tableaux following the directives below:

- We distinguish three types of decision actions: *cheap*, *neutral* and *expensive* ones with an equal proportion of 1/3. We also distinguish two types of weighted criteria: *cost* criteria to be *minimized*, and *benefit* criteria to be *maximized*; in the proportions 1/3 respectively 2/3.
- Random performances on each type of criteria are drawn, either from an ordinal scale [0;10], or from a cardinal scale [0.0;100.0], following a parametric triangular law of mode: 30% performance for cheap, 50% for neutral, and$70% performance for expensive decision actions, with constant probability repartition 0.5 on each side of the respective mode.
- Cost criteria use mostly cardinal scales (3/4), whereas benefit criteria use mostly ordinal scales (2/3).
- The sum of weights of the cost criteria by default equals the sum weights of the benefit criteria: weighDistribution = 'equiobjectives'.
- On cardinal criteria, both of cost or of benefit type, we observe following constant preference discrimination quantiles: 5% indifferent situations, 90% strict preference situations, and 5% veto situation.

Parameters

- If *numberOfActions* == None, a uniform random number between 10 and 31 of cheap, neutral or advantageous actions (equal 1/3 probability each type) actions is instantiated
- If *numberOfCriteria* == None, a uniform random number between 5 and 21 of cost or benefit criteria (1/3 respectively 2/3 probability) is instantiated
- *weightDistribution* = {'equiobjectives'|'fixed'|'random'|'equisignificant' (default = 'equisignificant')}
- default *weightScale* for 'random' weightDistribution is 1 - numberOfCriteria
- All cardinal criteria are evaluated with decimals between 0.0 and 100.0 whereas ordinal criteria are evaluated with integers between 0 and 10.
- commonThresholds is obsolete. Preference discrimination is specified as percentiles of concerned performance differences (see below).
- commonPercentiles = {'ind':5, 'pref':10, ['weakveto':90,] 'veto':95} are expressed in percents (reversed for vetoes) and only concern cardinal criteria.

Warning: Minimal number of decision actions required is 3 !

Example Python session:

```
>>> from randomPerfTabs import RandomCBPerformanceTableau
>>> t = RandomCBPerformanceTableau(
...           numberOfActions=7,
...           numberOfCriteria=5,
...           weightDistribution='equiobjectives',
...           commonPercentiles={'ind':5,'pref':10,'veto':95},
...           seed=100)
>>> t.showActions()
*----- show decision action --------------*
key:  a1
  short name: a1
  name:       random cheap decision action
key:  a2
  short name: a2
```

```
  name:       random neutral decision action
...
key:  a7
  short name: a7
  name:       random advantageous decision action
>>> t.showCriteria()
*----  criteria -----*
g1 'random ordinal benefit criterion'
  Scale = (0, 10)
  Weight = 0.167
g2 'random cardinal cost criterion'
  Scale = (0.0, 100.0)
  Weight = 0.250
  Threshold ind  :  1.76 + 0.00x ; percentile:  0.095
  Threshold pref :  2.16 + 0.00x ; percentile:  0.143
  Threshold veto : 73.19 + 0.00x ; percentile:  0.952
...
```

In the example above, we may notice the three types of decision actions (Lines 10-19), as well as the two types (Lines 22-25) of criteria with either an **ordinal** or a **cardinal** performance measuring scale. In the latter case, by default about 5% of the random performance differences will be below the **indifference** and 10% below the **preference discriminating threshold**. About 5% will be considered as **considerably large**. More statistics about the generated performances is available as follows:

```
>>> t.showStatistics()
*-------- Performance tableau summary statistics -------*
Instance name      : randomCBperftab
#Actions           : 7
#Criteria          : 5
*Statistics per Criterion*
Criterion name       : g1
  Criterion weight     : 2
  criterion scale    : 0.00 - 10.00
  mean evaluation    : 5.14
  standard deviation : 2.64
  maximal evaluation : 8.00
  quantile Q3 (x_75) : 8.00
  median evaluation  : 6.50
  quantile Q1 (x_25) : 3.50
  minimal evaluation : 1.00
  mean absolute difference      : 2.94
  standard difference deviation : 3.74
Criterion name       : g2
  Criterion weight     : 3
  criterion scale    : -100.00 - 0.00
  mean evaluation    : -49.32
  standard deviation : 27.59
  maximal evaluation : 0.00
  quantile Q3 (x_75) : -27.51
  median evaluation  : -35.98
  quantile Q1 (x_25) : -54.02
  minimal evaluation : -91.87
  mean absolute difference      : 28.72
  standard difference deviation : 39.02
...
```

A (potentially ranked) colored heat map with 5 color levels is also provided:

```
>>> t.showHTMLPerformanceHeatmap(colorLevels=5,Ranked=False)
```

Heatmap of performance tableau

criteria	g3	g2	g5	g4	g1
weights	3	3	2	2	2
a1	-33.99	-17.92	3.00	26.68	1.00
a2	-77.77	-30.71	6.00	66.35	8.00
a3	-69.84	-41.65	8.00	53.43	8.00
a4	-16.99	-39.49	2.00	18.62	2.00
a5	-74.85	-91.87	7.00	83.09	6.00
a6	-24.91	-32.47	9.00	79.24	7.00
a7	-7.44	-91.11	7.00	48.22	4.00

Color legend:

quantile	0.20%	0.40%	0.60%	0.80%	1.00%

Such a performance tableau may be stored and re-accessed in the XMCDA2 encoded format:

```
>>> t.saveXMCDA2('temp')
*----- saving performance tableau in XMCDA 2.0 format  ------------*
File: temp.xml saved !
>>> from perfTabs import XMCDA2PerformanceTableau
>>> t = XMCDA2PerformanceTableau('temp')
>>> ...
```

If needed for instance in an R session, a CSV version of the performance tableau may be created as follows:

```
>>> t.saveCSV('temp')
* --- Storing performance tableau in CSV format in file temp.csv
...$ less temp.csv
"actions","g1","g2","g3","g4","g5"
"a1",1.00,-17.92,-33.99,26.68,3.00
"a2",8.00,-30.71,-77.77,66.35,6.00
"a3",8.00,-41.65,-69.84,53.43,8.00
"a4",2.00,-39.49,-16.99,18.62,2.00
"a5",6.00,-91.87,-74.85,83.09,7.00
"a6",7.00,-32.47,-24.91,79.24,9.00
"a7",4.00,-91.11,-7.44,48.22,7.00
```

Back to *Tutorials of the Digraph3 resources*

Generating three objectives tableaux

We provide the *randomPerfTabs.Random3ObjectivesPerformanceTableau* class for generating random performance tableaux concerning three preferential decision objectives which take respectively into account *economical*, *societal* as well as *environmental* aspects.

Each decision action is qualified randomly as performing **weak** (-), **fair** (~) or **good** (+) on each of the three objectives.

Generator directives are the following:

- numberOfActions = 20 (default),
- numberOfCriteria = 13 (default),
- weightDistribution = 'equiobjectives' (default) | 'random' | 'equisignificant',
- weightScale = (1,numberOfCriteria): only used when random criterion weights are requested,
- integerWeights = True (default): False gives normlized rational weights,
- commonScale = (0.0,100.0),
- commonThresholds = [(5.0,0.0),(10.0,0.0),(60.0,0.0)]: Performance discrimination thresholds may be set for 'ind', 'pref' and 'veto',
- commonMode = ['triangular','variable',0.5]: random number generators of various other types ('uniform','beta') are available,
- valueDigits = 2 (default): evaluations are encoded as Decimals,
- missingProbability = 0.05 (default): random insertion of missing values with given probability,
- seed= None.

Note: If the mode of the **triangular** distribution is set to '*variable*', three modes at 0.3 (-), 0.5 (~), respectively 0.7 (+) of the common scale span are set at random for each coalition and action.

Warning: Minimal number of decision actions required is 3 !

Example Python session:

```
>>> from randomPerfTabs import Random3ObjectivesPerformanceTableau
>>> t = Random3ObjectivesPerformanceTableau(
          numberOfActions=31,
          numberOfCriteria=13,
          weightDistribution='equiobjectives',
          seed=120)
>>> t.showObjectives()
*------ show objectives -------"
Eco: Economical aspect
   g04 criterion of objective Eco 20
   g05 criterion of objective Eco 20
   g08 criterion of objective Eco 20
   g11 criterion of objective Eco 20
  Total weight: 80.00 (4 criteria)
Soc: Societal aspect
   g06 criterion of objective Soc 16
   g07 criterion of objective Soc 16
   g09 criterion of objective Soc 16
```

```
   g10 criterion of objective Soc 16
   g13 criterion of objective Soc 16
  Total weight: 80.00 (5 criteria)
Env: Environmental aspect
   g01 criterion of objective Env 20
   g02 criterion of objective Env 20
   g03 criterion of objective Env 20
   g12 criterion of objective Env 20
  Total weight: 80.00 (4 criteria)
```

In the example code above, we notice that 5 *equisignificant* criteria (g06, g07, g09, g10, g13) evaluate for instance the performance of the decision actions from the **societal** point of view. 4 *equisignificant* criteria do the same from the **economical**, respectively the **environmental** point of view. The *equiobjectives* directive results hence in a balanced total weight (80.00) for each decision objective.

```
>>> t.showActions()
key:  a01
  name:       random decision action Eco+ Soc- Env+
  profile:    {'Eco': 'good', 'Soc': 'weak', 'Env': 'good'}
key:  a02
...
key:  a26
  name:       random decision action Eco+ Soc+ Env-
  profile:    {'Eco': 'good', 'Soc': 'good', 'Env': 'weak'}
...
key:  a30
  name:       random decision action Eco- Soc- Env-
  profile:    {'Eco': 'weak', 'Soc': 'weak', 'Env': 'weak'}
...
```

Variable triangular modes (0.3, 0.5 or 0.7 of the span of the measure scale) for each objective result in different performance status for each decision action with respect to the three objectives. Action *a01* , for instance, will probably show *good* performances wrt the *economical* and environmental aspects, and *weak* performances wrt the *societal* aspect.

For testing purposes we provifde a special *perfTabs.PartialPerformanceTableau* class for extracting a **partial performance tableau** from a given tableau instance. In the example blow, we construct the partial performance tableaux corresponding to each on of the three decision objectives:

```
>>> from perfTabs import PartialPerformanceTableau
>>> teco = PartialPerformanceTableau(t,criteriaSubset=\
                          t.objectives['Eco']['criteria'])
>>> tsoc = PartialPerformanceTableau(t,criteriaSubset=\
                          t.objectives['Soc']['criteria'])
>>> tenv = PartialPerformanceTableau(t,criteriaSubset=\
                          t.objectives['Env']['criteria'])
```

One may thus compute a partial bipolar outranking digraph for each individual objective:

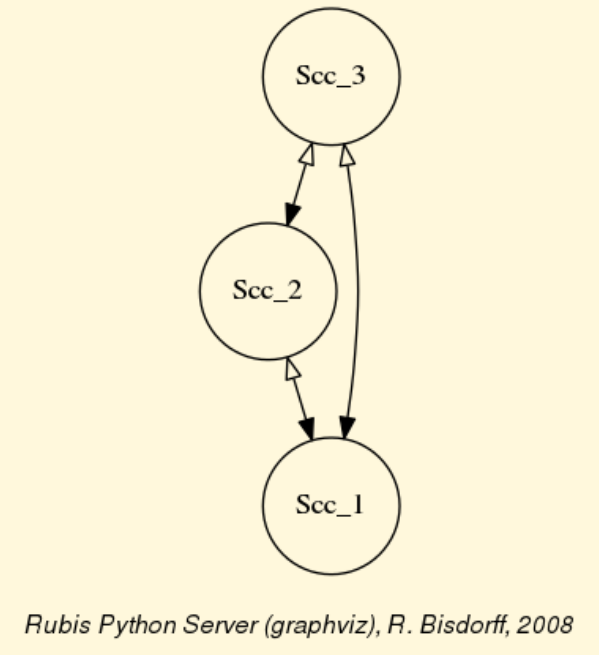
```
>>> from outrankingDigraphs import BipolarOutrankingDigraph
>>> geco = BipolarOutrankingDigraph(teco)
>>> gsoc = BipolarOutrankingDigraph(tsoc)
>>> genv = BipolarOutrankingDigraph(tenv)
```

The three partial digraphs: *geco*, *gsoc* and *genv*, hence model the preferences represented in each one of the partial performance tableaux. And, we may aggregate these three outranking digraphs with an epistemic fusion operator:

```
>>> from digraphs import FusionLDigraph
>>> gfus = FusionLDigraph([geco,gsoc,genv])
>>> gfus.strongComponents()
{frozenset({'a30'}),
 frozenset({'a10', 'a03', 'a19', 'a08', 'a07', 'a04', 'a21', 'a20',
            'a13', 'a23', 'a16', 'a12', 'a24', 'a02', 'a31', 'a29',
            'a05', 'a09', 'a28', 'a25', 'a17', 'a14', 'a15', 'a06',
            'a01', 'a27', 'a11', 'a18', 'a22'}),
 frozenset({'a26'})}
>>> from digraphs import StrongComponentsCollapsedDigraph
>>> scc = StrongComponentsCollapsedDigraph(gfus)
>>> scc.showActions()
*----- show digraphs actions --------------*
key:  frozenset({'a30'})
  short name: Scc_1
  name:       _a30_
  comment:    collapsed strong component
key:  frozenset({'a10', 'a03', 'a19', 'a08', 'a07', 'a04', 'a21', 'a20', 'a13',
                 'a23', 'a16', 'a12', 'a24', 'a02', 'a31', 'a29', 'a05', 'a09',
↪'a28', 'a25',
                 'a17', 'a14', 'a15', 'a06', 'a01', 'a27', 'a11', 'a18', 'a22'})
  short name: Scc_2
  name:       _a10_a03_a19_a08_a07_a04_a21_a20_a13_a23_a16_a12_a24_a02_a31_\
               a29_a05_a09_a28_a25_a17_a14_a15_a06_a01_a27_a11_a18_a22_
  comment:    collapsed strong component
key:  frozenset({'a26'})
  short name: Scc_3
  name:       _a26_
  comment:    collapsed strong component
```

A graphviz drawing illustrates the apparent preferential links between the strong components:

```
>>> scc.exportGraphViz('scFusionObjectives')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to scFusionObjectives.dot
dot -Grankdir=BT -Tpng scFusionObjectives.dot -o scFusionObjectives.png
```



Decision action *a26* (Eco+ Soc+ Env-) appears dominating the other decision alternatives, whereas decision action *a30* (Eco- Soc- Env-) appears to be dominated by all the others.

Generating random linearly ranked performances

Finally, we provide the `randomPerfTabs.RandomRankPerformanceTableau` class for generating multiple criteria ranked performances, ie on each criterion, all decision actions appear linearly ordered without ties.

This type of random performance tabeau is matching the `votingDigrahs.RandomLinearVotingProfile` class provided by the `votingProfiles` module.

Parameters:

- number of actions,
- number of performance criteria,
- weightDistribution := equisignificant | random (default, see above above)
- weightScale := (1, 1 | numberOfCriteria (default when random))
- integerWeights := Boolean (True = default)
- commonThresholds (default) := {

  'ind':(0,0),
  'pref':(1,0),
  'veto':(numberOfActions,0)
  } (default)

Back to *Tutorials of the Digraph3 resources*

2.1.6 Computing a best choice recommendation

See also the lecture 7 notes from the MICS Algorithmic Decision Theory course: *[ADT-L7]*.

What site to choose ?

A SME, specialized in printing and copy services, has to move into new offices, and its CEO has gathered seven **potential office sites**:

address	ID	Comment
Avenue de la liberté	A	High standing city center
Bonnevoie	B	Industrial environment
Cessange	C	Residential suburb location
Dommeldange	D	Industrial suburb environment
Esch-Belval	E	New and ambitious urbanization far from the city
Fentange	F	Out in the countryside
Avenue de la Gare	G	Main town shopping street

Three **decision objectives** are guiding the CEO's choice:

1. *minimize* the yearly costs induced by the moving,
2. *maximize* the future turnover of the SME,
3. *maximize* the new working conditions.

The decision consequences to take into account for evaluating the potential new office sites with respect to each of the three objectives are modelled by the following **family of criteria**:

Objective	ID	Name	Comment
Yearly costs	C	Costs	Annual rent, charges, and cleaning
Future turnover	St	Standing	Image and presentation
Future turnover	V	Visibility	Circulation of potential customers
Future turnover	Pr	Proximity	Distance from town center
Working conditions	W	Space	Working space
Working conditions	Cf	Comfort	Quality of office equipment
Working conditions	P	Parking	Available parking facilities

The evaluation of the seven potential sites on each criterion are gathered in the following **performance tableau**:

Criterion	weight	A	B	C	D	E	F	G
Costs	3.0	35.0K€	17.8K€	6.7K€	14.1K€	34.8K€	18.6K€	12.0K€
Stan	1.0	100	10	0	30	90	70	20
Visi	1.0	60	80	70	50	60	0	100
Prox	1.0	100	20	80	70	40	0	60
Wksp	1.0	75	30	0	55	100	0	50
Wkcf	1.0	0	100	10	30	60	80	50
Park	1.0	90	30	100	90	70	0	80

Except the *Costs* criterion, all other criteria admit for grading a qualitative satisfaction scale from 0% (worst) to 100% (best). We may thus notice that site *A* is the most expensive, but also 100% satisfying the *Proximity* as well as the *Standing* criterion. Whereas the site *C* is the cheapest one; providing however no satisfaction at all on both the *Standing* and the *Working Space* criteria.

All qualitative criteria, supporting their respective objective, are considered to be *equi-significant* (weights = 1.0). As a consequence, the three objectives are considered *equally important* (total weight = 3.0 each).

Concerning annual costs, we notice that the CEO is indifferent up to a performance difference of 1000€, and he actually prefers a site if there is at least a positive difference of 2500€. The grades observed on the six qualitative criteria (measured in percentages of satisfaction) are very subjective and rather imprecise. The CEO is hence indifferent up to a satisfaction difference of 10%, and he claims a significant preference when the satisfaction difference is at least of 20%. Furthermore, a satisfaction difference of 80% represents for him a *considerably large* performance difference, triggering a *veto* situation the case given (see *[BIS-2013]*).

In view of this performance tableau, what is now the office site we may recommend to the CEO as **best choice** ?

Performance tableau

The XMCDA 2.0 encoded version of this performance tableau is available for downloading here officeChoice.xml.

We may inspect the performance tableau data with the computing resources provided by the *perfTabs module* module.

```
>>> from perfTabs import *
>>> t = XMCDA2PerformanceTableau('officeChoice')
>>> help(t) # for discovering all the methods available
>>> t.showPerformanceTableau()
*----  performance tableau -----*
criteria |   weights |     'A'      'B'      'C'      'D'      'E'      'F'
↪    'G'
---------|-----------------------------------------------------------------
↪---------
'C'      |    3.00   | -35000.00 -17800.00 -6700.00 -14100.00 -34800.00 -18600.00
↪-12000.00
'Cf'     |    1.00   |      0.00    100.00    10.00     30.00     60.00     80.00
↪    50.00
'P'      |    1.00   |     90.00     30.00   100.00     90.00     70.00      0.00
↪    80.00
'Pr'     |    1.00   |    100.00     20.00    80.00     70.00     40.00      0.00
↪    60.00
'St'     |    1.00   |    100.00     10.00     0.00     30.00     90.00     70.00
↪    20.00
```

```
13 'V'      |    1.00    |      60.00      80.00     70.00      50.00      60.00       0.00␣
   ↪   100.00
14 'W'      |    1.00    |      75.00      30.00      0.00      55.00     100.00       0.00␣
   ↪    50.00
```

We thus recover all the input data. To measure the actual preference discrimination we observe on each criterion, we may use the `showCriteria` method:

```
1  >>> t.showCriteria()
2  *----  criteria -----*
3  C 'Costs'
4  Scale = (Decimal('0.00'), Decimal('50000.00'))
5  Weight = 0.333
6  Threshold ind : 1000.00 + 0.00x ; percentile:  0.095
7  Threshold pref : 2500.00 + 0.00x ; percentile:  0.143
8  Cf 'Comfort'
9  Scale = (Decimal('0.00'), Decimal('100.00'))
10 Weight = 0.111
11 Threshold ind : 10.00 + 0.00x ; percentile:  0.095
12 Threshold pref : 20.00 + 0.00x ; percentile:  0.286
13 Threshold veto : 80.00 + 0.00x ; percentile:  0.905
14 ...
```

On the *Costs* criterion, 9.5% of the performance differences are considered insignificant and 14.3% below the preference discrimination threshold (lines 6-7). On the qualitative *Comfort* criterion, we observe again 9.5% of insignificant performance differences (line 11). Due to the imprecision in the subjective grading, we notice here 28.6% of performance differences below the preference discrimination threshold (line 12). Furthermore, 100.0 - 90.5 = 9.5% of the performance differences are judged *considerably large* (line 13); 80% and more of satisfaction differences triggering in fact a veto situation. Same information is available for all the other criteria.

A colorful comparison of all the performances is shown by the **heat map** statistics, illustrating the respective quantile class of each performance. As the set of potential alternatives is tiny, we choose here a classification into performance quintiles:

```
>>> t.showHTMLPerformanceHeatmap(colorLevels=5)
```

Heatmap of performance tableau officeChoice.xml

criteria	C	W	V	St	Pr	P	Cf
weights	3.00	1.00	1.00	1.00	1.00	1.00	1.00
A	-35000.00	75.00	60.00	100.00	100.00	90.00	0.00
B	-17800.00	30.00	80.00	10.00	20.00	30.00	100.00
C	-6700.00	0.00	70.00	0.00	80.00	100.00	10.00
D	-14100.00	55.00	50.00	30.00	70.00	90.00	30.00
E	-34800.00	100.00	60.00	90.00	40.00	70.00	60.00
F	-18600.00	0.00	0.00	70.00	0.00	0.00	80.00
G	-12000.00	50.00	100.00	20.00	60.00	80.00	50.00

Color legend:

quantile	0.2	0.4	0.6	0.8	1.0

Site *A* shows extreme and contradictory performances: highest *Costs* and no *Working Comfort* on one hand, and total satisfaction with respect to *Standing*, *Proximity* and *Parking facilities* on the other hand. Similar, but opposite, situation is given for site *C*: unsatisfactory *Working Space*, no *Standing* and no *Working Comfort* on the one hand, and lowest *Costs*, best *Proximity* and *Parking facilities* on the other hand. Contrary to these contradictory alternatives, we observe two appealing compromise decision alternatives: sites *D* and *G*. Finally, site *F* is clearly the less satisfactory alternative of all.

Outranking digraph

To help now the CEO choosing the best site, we are going to compute pairwise outrankings (see *[BIS-2013]*) on the set of potential sites. For two sites *x* and *y*, the situation "*x* outranks *y*", denoted (*x* S *y*), is given if there is:

1. a **significant majority** of criteria concordantly supporting that site *x* is *at least as satisfactory as* site *y*, and
2. **no considerable** counter-performance observed on any discordant criterion.

The credibility of each pairwise outranking situation (see *[BIS-2013]*), denoted r(*x* S *y*), is measured in a bipolar significance valuation [-100.00, 100.00], where **positive** terms r(*x* S *y*) > 0.0 indicate a **validated**, and **negative** terms r(*x* S *y*) < 0.0 indicate a **non-validated** outrankings; whereas the **median** value r(*x* S *y*) = 0.0 represents an **indeterminate** situation.

For computing such a bipolar valued outranking digraph from the given performance tableau *t*, we use the `BipolarOutrankingDigraph` constructor from the *outrankingDigraphs module* module. The `Digraph.showHTMLRelationTable` method shows here the resulting bipolar-valued adjacency matrix in a system browser window:

```
>>> from outrankingDigraphs import BipolarOutrankingDigraph
>>> g = BipolarOutrankingDigraph(t)
>>> g.showHTMLRelationTable()
```

Valued Adjacency Matrix

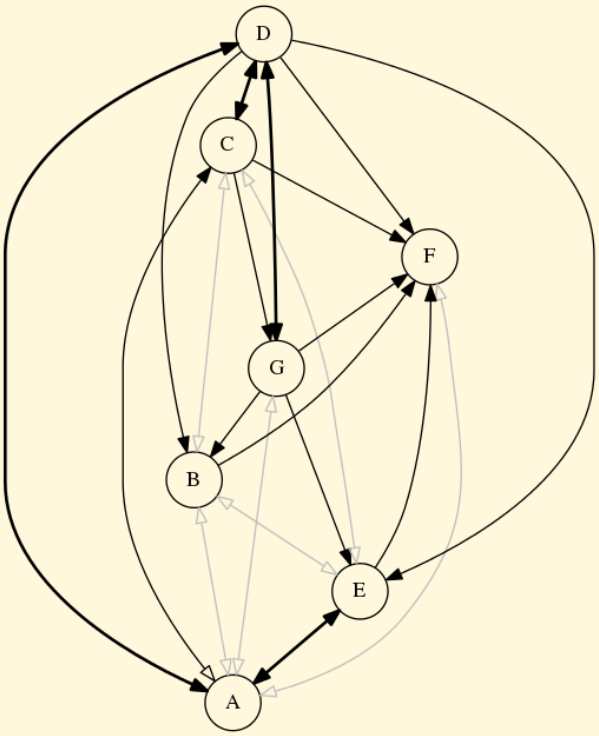
r(x S y)	A	B	C	D	E	F	G
A	0.00	0.00	100.00	11.11	55.56	0.00	0.00
B	0.00	0.00	0.00	-55.56	0.00	100.00	-55.56
C	0.00	0.00	0.00	33.33	0.00	100.00	11.11
D	33.33	55.56	11.11	0.00	33.33	100.00	22.22
E	55.56	0.00	0.00	-11.11	0.00	100.00	-11.11
F	0.00	-100.00	-100.00	-100.00	-100.00	0.00	-100.00
G	0.00	77.78	-11.11	100.00	55.56	100.00	0.00

We may notice that Alternative *D* is **positively outranking** all other potential office sites (a *Condorcet winner*). Yet, alternatives *A* (the most expensive) and *C* (the cheapest) are *not* outranked by any other site; they are in fact **weak** *Condorcet winners*.

```
>>> g.condorcetWinners()
['D']
>>> g.weakCondorcetWinners()
['A', 'C', 'D']
```

We may get even more insight in the apparent outranking situations when looking at the Condorcet digraph:

```
>>> g.exportGraphViz('officeChoice')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to officeChoice.dot
dot -Grankdir=BT -Tpng officeChoice.dot -o officeChoice.png
```



One may check that the outranking digraph *g* does not admit in fact a cyclic strict preference situation:

```
>>> g.computeChordlessCircuits()
[]
>>> g.showChordlessCircuits()
No circuits observed in this digraph.
*---- Chordless circuits ----*
0 circuits.
```

Rubis best choice

Following the Rubis outranking method (see *[BIS-2008]*), potential best choice recommendations are determined by the outranking pre-kernels (weakly independent and strictly outranking choices) of the chordless odd circuits augmented outranking digraph. As we observe no circuits here, we may directly compute the pre-kernels of *g*:

```
>>> g.showPreKernels()
*--- Computing preKernels ---*
Dominant preKernels :
['D']
   independence :  100.0
   dominance    :  11.111
   absorbency   :  -100.0
   covering     :  1.000
['B', 'E', 'C']
```

```
   independence :  0.00
   dominance    :  11.111
   absorbency   :  -100.0
   covering     :  0.500
['A', 'G']
   independence :  0.00
   dominance    :  55.556
   absorbency   :  0.00
   covering     :  0.700
Absorbent preKernels :
['F', 'A']
   independence :  0.00
   dominance    :  0.00
   absorbency   :  100.0
   covering     :  0.700
*----- statistics -----
graph name:  rel_officeChoice.xml
number of solutions
 dominant kernels :  3
 absorbent kernels:  1
cardinality frequency distributions
cardinality     :  [0, 1, 2, 3, 4, 5, 6, 7]
dominant kernel :  [0, 1, 1, 1, 0, 0, 0, 0]
absorbent kernel:  [0, 0, 1, 0, 0, 0, 0, 0]
Execution time  : 0.00018 sec.
Results in sets: dompreKernels and abspreKernels.
```

We notice three potential best choice recommendations: the Condorcet winner D (line 4), the triplet B, C and E (line 9), and finally the pair A and G (line 14). The Rubis best choice recommendation is given by the **most determined** pre-kernel; the one supported by the most significant criteria coalition. This result is shown with the following command:

```
>>> g.showRubisBestChoiceRecommendation(CoDual=False)
**********************
* --- Rubis best choice recommendation(s) (BCR) ---*
  (in decreasing order of determinateness)
Credibility domain:  {'min': -100.0, 'med': 0.0, 'max': 100.0}
 === >> potential BCR
* choice              : ['D']
  +-irredundancy      : 100.00
  independence        : 100.00
  dominance           : 11.11
  absorbency          : -100.00
  covering (%)        : 100.00
  determinateness (%) : 56,0
  characteristic vector = { 'D': 11.11, 'A': -11.11, 'B': -11.11,
         'C': -11.11, 'E': -11.11, 'F': -11.11, 'G': -11.11 }
 === >> potential BCR
* choice              : ['B', 'E', 'C']
  +-irredundancy      : 0.00
  independence        : 0.00
  dominance           : 11.11
  absorbency          : -100.00
  covering (%)        : 50.00
  determinateness (%) : 50.0
  - characteristic vector = { 'B': 0.00, 'E': 0.00, 'F': 0.00,
          'D': 0.00, 'A': 0.00, 'G': 0.00, 'C': 0.00 }
 === >> potential BCR
```

```
* choice              : ['A', 'G']
  +-irredundancy      : 0.00
  independence        : 0.00
  dominance           : 55.56
  absorbency          : 0.00
  covering (%)        : 70.00
  determinateness (%) : 50.0
  - characteristic vector = { 'B': 0.00, 'E': 0.00, 'F': 0.00,
           'D': 0.00, 'A': 0.00, 'G': 0.00, 'C': 0.00 }
 === >> potential worst choice
* choice              : ['A', 'F']
  +-irredundancy      : 0.00
  independence        : 0.00
  dominance           : 0.00
  absorbency          : 100.00
  covering (%)        : 30.00
  determinateness (%) : 50.0
  characteristic vector = { 'B': 0.00, 'E': 0.00, 'F': 0.00,
            'D': 0.00, 'A': 0.00, 'G': 0.00, 'C': 0.00 }
```

We notice in line 7 above that the most significantly supported best choice recommendation is indeed the Condorcet winner *D* with a majority of 56% of the criteria significance (see line 13). Both other recommendation candidates, as well as the worst choice candidate are not positively validated as best choices. They may or may not be considered so. Alternative *A*, with extreme contradictory performances, appears both, in a best and a worst choice recommendation (see lines 27 and 37) and seams hence not actually comparable to its competitors.

The same Rubis best choice recommendation, encoded in XMCDA 2.0 and presented in the default system browser, is provided by the *xmcda* module. In a python3 session working in the directory where the XMCDA encoded problem data is stored, we may proceed as follows:

```
>>> import xmcda
>>> xmcda.showXMCDARubisBestChoiveRecommendation(\
                            prolemFileName='officeChoice',\
                            valuationType='bipolar')
```

and, in a system browser window, browse the solution file.

The `valuationType` parameter allows to work:

- on the standard bipolar outranking digraph (valuationType = 'bipolar', default),
- on the normalized –[-1,1] valued– bipolar outranking digraph (valuationType = 'normalized'),
- on the robust –ordinal criteria weights– bipolar outranking digraph (valuationType = 'robust'),
- on the confident outranking digraph (valuationType = 'confident'),
- ignoring considerable performances differences (valuationType = 'noVeto').

One may as well use the Rubis XMCDA 2.0 Web services available at the Leopold-Loewenheim Apache Server of the University of Luxembourg:

```
>>> from outrankingDigraphs import RubisRestServer
>>> solver = RubisRestServer()
>>> solver.ping()
*************************************************
* This is the Leopold-Loewenheim Apache Server  *
* of the University of Luxembourg.              *
* Welcome to the Rubis XMCDA 2.0 Web service    *
```

```
* R. Bisdorff (c) 2009-2013                    *
* November 2013, version REST/D4 1.1           *
************************************************
```

We may submit the given performance tableau:

```
>>> t = XMCDA2PerformanceTableau('officeChoice')
>>> solver.submitProblem(t)
The problem submission was successful !
Server ticket: 1BYyGVwV866hSNZo
```

With the given ticket, saved in a text file in the working directory, we may request from the Rubis solver the corresponding best choice recommendation:

```
>>> solver.showSolution()
```

and, in a system browser window, browse again the solution file.

Here, we find confirmed again that alternative *D*, indeed, appears to be the most significant best choice candidate.

Yet, what about alternative *G*, the other good compromise best choice we have noticed from the performance heat map shown above?

Strictly best choice

When comparing the performances of alternatives *D* and *G* on a pairwise perspective, we notice that, with the given preference discrimination thresholds, alternative *G* is actually **certainly** *at least as good as* alternative *D* (r(*G* outranks *D*) = 100.0).

```
>>> g.showPairwiseComparison('G','D')
*------------  pairwise comparison ----*
Comparing actions : (G, D)
crit. wght.  g(x)      g(y)     diff.  |   ind     pref    concord   |
-------------------------------------------------------------------
C   3.00 -12000.00 -14100.00 +2100.00 | 1000.00 2500.00   +3.00     |
Cf  1.00     50.00     30.00   +20.00 |   10.00   20.00   +1.00     |
P   1.00     80.00     90.00   -10.00 |   10.00   20.00   +1.00     |
Pr  1.00     60.00     70.00   -10.00 |   10.00   20.00   +1.00     |
St  1.00     20.00     30.00   -10.00 |   10.00   20.00   +1.00     |
V   1.00    100.00     50.00   +50.00 |   10.00   20.00   +1.00     |
W   1.00     50.00     55.00    -5.00 |   10.00   20.00   +1.00     |
-------------------------------------------------------------------
Valuation in range: -9.00 to +9.00; global concordance: +9.00
```

However, we must as well notice that the cheapest alternative *C* is in fact **strictly outranking** alternative *G*:

```
>>> g.showPairwiseComparison('C','G')
*------------  pairwise comparison ----*
Comparing actions : (C, G)/(G, C)
crit. wght.   g(x)     g(y)      diff.  |   ind.   pref.     (C,G)/(G,C) |
-----------------------------------------------------------------------
C     3.00 -6700.00 -12000.00  +5300.00 | 1000.00 2500.00   +3.00/-3.00 |
Cf    1.00    10.00     50.00    -40.00 |   10.00   20.00   -1.00/+1.00 |
P     1.00   100.00     80.00    +20.00 |   10.00   20.00   +1.00/-1.00 |
Pr    1.00    80.00     60.00    +20.00 |   10.00   20.00   +1.00/-1.00 |
St    1.00     0.00     20.00    -20.00 |   10.00   20.00   -1.00/+1.00 |
V     1.00    70.00    100.00    -30.00 |   10.00   20.00   -1.00/+1.00 |
```

```
W     1.00     0.00     50.00    -50.00 |   10.00   20.00   -1.00/+1.00 |
-------------------------------------------------------------------------
Valuation in range: -9.00 to +9.00; global concordance: +1.00/-1.00
```

To model these *strict outranking* situations, we may compute the Rubis best choice recommendation on the **codual**, the converse (~) of the dual (-), of the outranking digraph instance *g* (see *[BIS-2013]*), as follows:

```
>>> g.showRubisBestChoiceRecommendation(CoDual=True,ChoiceVector=True)
* --- Rubis best choice recommendation(s) ---*
(in decreasing order of determinateness)
Credibility domain:  {'min':-100.0, 'max': 100.0', 'med':0.0'}
=== >> potential best choice(s)
* choice                : ['A', 'C', 'D']
  +-irredundancy        : 0.00
  independence          : 0.00
  dominance             : 11.11
  absorbency            : 0.00
  covering (%)          : 41.67
  determinateness (%)   : 53.17
  characteristic vector :
     { 'D': 11.11, 'A': 0.00, 'C': 0.00, 'G': 0.00,
       'B': -11.11, 'E': -11.11, 'F': -11.11 }
=== >> potential worst choice(s)
* choice                : ['A', 'F']
  +-irredundancy        : 0.00
  independence          : 0.00
  dominance             : -55.56
  absorbency            : 100.00
  covering (%)          : 0.00
  determinateness (%)   : 50.00
  characteristic vector :
     {'A': 0.00, 'B': 0.00, 'C': 0.00, 'D': 0.00,
      'E': 0.00, 'F': 0.00, 'G': 0.00, }
```

It is interesting to notice that the **strict best choice recommendation** consists in the set of weak Condorcet winners: 'A', 'C' and 'D' (see line 6). In the corresponding characteristic vector (see line 14-15), representing the bipolar credibility degree with which each alternative may indeed be considered a best choice (see *[BIS-2006]*), we find confirmed that alternative *D* is the only positively validated one, whereas both extreme alternatives - *A* (the most expensive) and *C* (the cheapest) - stay in an indeterminate situation. They may be potential best choice candidates besides *D*. Notice furthermore that compromise alternative *G*, while not actually included in the strict best choice recommendation, shows as well an indeterminate situation with respect to being or not a potential best choice candidate.

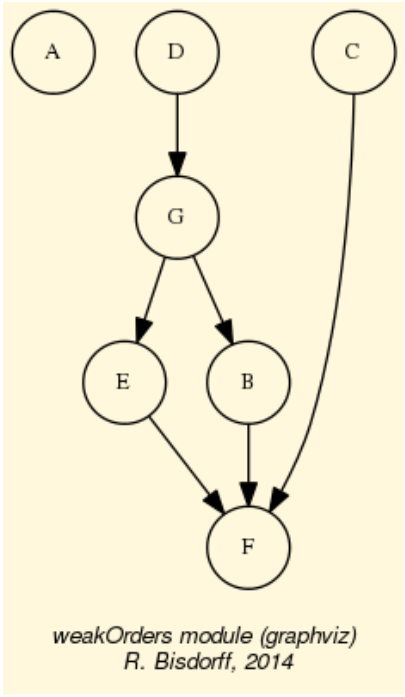
We may also notice (see line 17 and line 21) that both alternatives *A* and *F* are reported as certainly outranked choices, hence a **potential worst choice recommendation** . This confirms again the global incomparability status of alternative *A*.

Weakly ordering

To get a more complete insight in the overall strict outranking situations, we may use the *weakOrders.RankingByChoosingDigraph* constructor imported from the *weakOrders module* module, for computing a **ranking-by-choosing** result from the strict outranking digraph instance *gcd*:

```
>>> from weakOrders import RankingByChoosingDigraph
>>> rbc = RankingByChoosingDigraph(gcd)
Threading ...  ## multiprocessing if 2 cores are available
Exiting computing threads
```

```
>>> rbc.showRankingByChoosing()
Ranking by Choosing and Rejecting
1st ranked ['D'] (0.28)
   2nd ranked ['C', 'G'] (0.17)
   2nd last ranked ['B', 'C', 'E'] (0.22)
1st last ranked ['A', 'F'] (0.50)
>>> rbc.exportGraphViz('officeChoiceRanking')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to officeChoiceRanking.dot
0 { rank = same; A; C; D; }
1 { rank = same; G; }
2 { rank = same; E; B; }
3 { rank = same; F; }
dot -Grankdir=TB -Tpng officeChoiceRanking.dot -o officeChoiceRanking.png
```



In this **ranking-by-choosing** method, where we operate the epistemic fusion of iterated (strict) best and worst choices, compromise alternative *D* is indeed ranked before compromise alternative *G*. If the computing node supports multiple processor cores, best and worst choosing iterations are run in parallel. The overall partial ordering result shows again the important fact that the most expensive site *A*, and the cheapest site *C*, both appear incomparable with most of the other alternatives, as is apparent from the Hasse diagram (see above) of the ranking-by-choosing relation.

The best choice recommendation appears hence depending on the very importance the CEO is attaching to each of the three objectives he is considering. In the setting here, where he considers all three objectives to be **equally important** (minimize costs = 3.0, maximize turnover = 3.0, and maximize working conditions = 3.0), site *D* represents actually the best compromise. However, if *Costs* do not play much role, it would be perhaps better to decide to move to the most advantageous site *A*; or if, on the contrary, *Costs* do matter a lot, moving to the cheapest alternative *C* could definitely represent a more convincing recommendation.

It might be worth, as an **exercise**, to modify on the one hand this importance balance in the XMCDA data file by lowering the significance of the *Costs* criterion; all criteria are considered **equi-significant** (weight = 1.0) for instance. It may as well be opportune, on the other hand, to **rank** the importance of the three objectives as follows: *minimize costs* (weight = 9.0) **>** *maximize turnover* (weight = 3 x 2.0) **>** *maximize working conditions* (weight = 3 x 1.0). What will become the best choice recommendation under both working hypotheses?

Back to *Tutorials of the Digraph3 resources*

2.1.7 Ranking with multiple incommensurable criteria

The ranking problem

We need to rank without ties a set X of items (usually decision alternatives) that are evaluated on multiple incommensurable performance criteria; yet, for which we may know their pairwise valued outranking situation characteristics, i.e. r(x S y) for all x, y in X (see *[BIS-2013]*).

Unfortunately, the Condorcet digraph, associated with such a given outranking digraph, presents only exceptionally a linear ordering. Usually, pairwise majority comparisons do not render even a complete or, at least, a transitive partial outranking relation.

Let us consider a didactic outranking digraph generated from a random Cost-Benefit performance tableau concerning 9 decision alternatives evaluated on 13 performance criteria:

```
>>> from outrankingDigraphs import *
>>> t = RandomCBPerformanceTableau(numberOfActions=9,
...                                numberOfCriteria=13,seed=2)
>>> g = BipolarOutrankingDigraph(t,Normalized=True)
>>> g.showRelationTable(ReflexiveTerms=False)
* ---- Relation Table -----
 S   |  'a1'    'a2'    'a3'    'a4'    'a5'    'a6'    'a7'    'a8'    'a9'
-----|-------------------------------------------------------------------
'a1' |    -     +0.00   +0.24   +0.24   +0.00   +0.17   +0.26   +0.07   +0.00
'a2' |  +0.00    -      -0.50   +0.00   -0.13   +0.00   +0.00   -0.02   +0.00
'a3' |  +0.14   +0.50    -      +0.40   +0.36   +0.50   +0.71   +0.69   +1.00
'a4' |  +0.05   +0.00   -0.40    -      +0.00   +0.21   +0.26   -0.10   +0.10
'a5' |  +0.00   +0.36   -0.36   +0.00    -      +0.26   +0.00   +0.26   -1.00
'a6' |  -0.10   +0.00   -0.29   -0.07   +0.02    -      +0.24   +0.19   +0.04
'a7' |  -0.26   +0.00   -0.29   -0.02   +0.00   -0.10    -      +0.00   -1.00
'a8' |  -0.07   +0.33   -0.24   +0.10   +0.05   +0.29   +0.00    -      -0.02
'a9' |  +0.00   +0.00   -1.00   -0.10   +1.00   +0.33   +1.00   +0.02    -
```

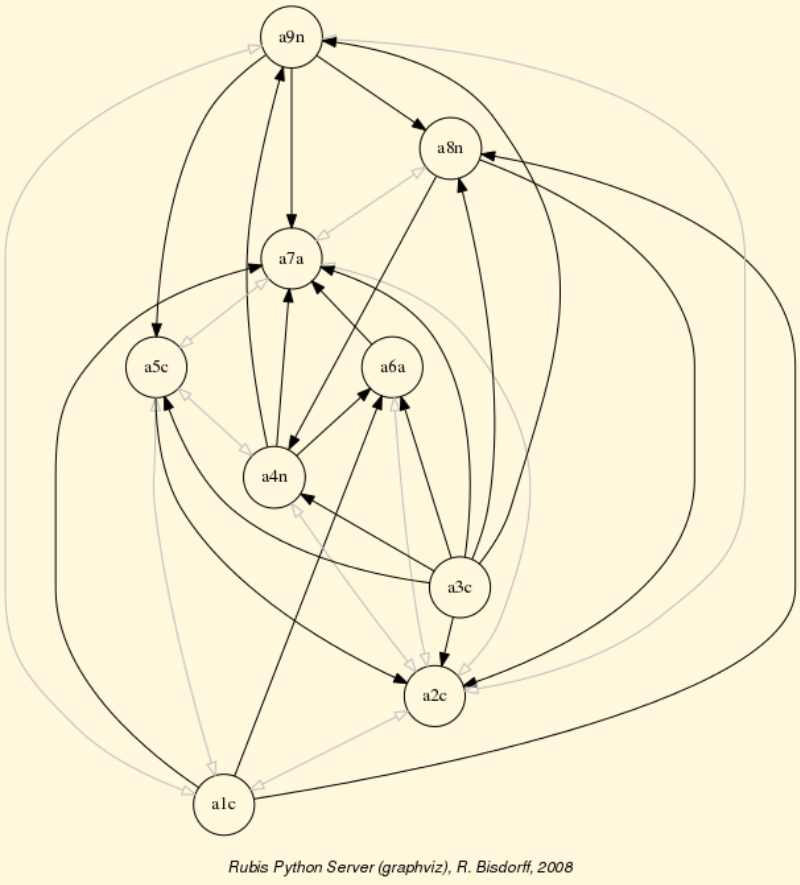
Some ranking rules will work on the associated Condorcet digraph, i.e. the strict median cut polarised digraph:

```
>>> c = PolarisedOutrankingDigraph(g,level=0,KeepValues=False,
...                                StrictCut=True)
>>> c.showRelationTable(ReflexiveTerms=False,IntegerValues=True)
```

```
* ---- Relation Table -----
 S   |  'a1' 'a2' 'a3' 'a4' 'a5' 'a6' 'a7' 'a8' 'a9',
-----|-------------------------------------------------------
'a1' |    -    0   +1   +1    0   +1   +1   +1    0
'a2' |    0    -   -1    0   -1    0    0   -1    0
'a3' |   +1   +1    -   +1   +1   +1   +1   +1   +1
'a4' |   +1    0   -1    -    0   +1   +1   -1   +1
'a5' |    0   +1   -1    0    -   +1    0   +1   -1
'a6' |   -1    0   -1   -1   +1    -   +1   +1   +1
'a7' |   -1    0   -1   -1    0   -1    -    0   -1
'a8' |   -1   +1   -1   +1   +1   +1    0    -   -1
'a9' |    0    0   -1   -1   +1   +1   +1   +1    -
```

To estimate how difficult this ranking problem may be, we can have a look at the corresponding *strict* outranking digraph graphviz drawing:

```
>>> gcd = ~(-g) # converse of the negation of g
>>> gcd.exportGraphViz('rankingTutorial')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to rankingTutorial.dot
dot -Grankdir=BT -Tpng rankingTutorial.dot -o rankingTutorial.png
```



The shown strict outranking relation is apparently not transitive: for instance, alternative *a9* outranks alternative *a5* and alternative *a5* outranks *a2*, however *a9* does not outrank *a2*. We may compute the transitivity degree of the outranking digraph, ie the ratio of the number of outranking arcs over the number of arcs of the transitive closue of the digraph *gcd*:

```
>>> gcd.computeTransitivityDegree()
Decimal('0.508')
```

The outranking relation is hence very far from being transitive; a serious problem when a linear ordering of the decision alternatives is looked for. Let us furthermore see if there are any cyclic outrankings:

```
>>> len(gcd.computeChordlessCircuits())
1
>>> gcd.showChordlessCircuits()
*---- Chordless circuits ----*
['a4', 'a9', 'a8'] , credibility : 0.024
```

There is one chordless circuit detected in the given strict outranking digraph *gcd*, namely *a4* outranks *a9*, the latter outranks *a8*, and *a8* outranks again *a4*. Any potential linear ordering of these three alternatives will, in fact, always contradict somehow the given outranking relation.

Several heuristic ranking rules have been proposed for constructing a linear ordering which is closest in some specific sense to a given outranking relation.

The Digraph3 resources provide some of the most common of these ranking rules, like Copeland's, Kemeny's, Slater's, Kohler and Tideman's ranking rules.

The Copeland ranking

Copeland's rule, the most intuitive one as it works well for any outranking relation which models in fact a linear order, computes for each alternative a score resulting from the difference between its crisp out-degree (number of validated (+1) crisp outranking situations) and its crisp in-degree (number of validated crisp (+1) outranked situations):

```
>>> from linearOrders import CopelandOrder
>>> cop = CopelandOrder(g,Debug=True)
Copeland score for a1 = +3  (5 - 3)
Copeland score for a2 = -3  (0 - 3)
Copeland score for a3 = +7  (8 - 1)
Copeland score for a4 = +1  (4 - 3)
Copeland score for a5 = -1  (3 - 4)
Copeland score for a6 = -2  (4 - 6)
Copeland score for a7 = -5  (0 - 5)
Copeland score for a8 = -1  (4 - 5)
Copeland score for a9 = +1  (4 - 3)
['a7', 'a2', 'a6', 'a8', 'a5', 'a9', 'a4', 'a1', 'a3']
>>> cop.showRanking()
['a3', 'a1', 'a4', 'a9', 'a5', 'a8', 'a6', 'a2', 'a7']
```

Alternative *a3* has the best score (+7), followed by alternative *a1* (+3). Alternatives *a4* and *a9* have the same score (+1); following the lexicographic rule, *a4* is hence ranked before *a9*. Same situation is observed for *a5* and *a8* with a score of -1.

Notice by the way that Copeland scores, as computed in the associated Condorcet relation table or similarly in the codual digraph drawing above, are in fact invariant under a codual - converse of the negation ~(-g) - transform of the outranking digraph.

Copeland's rule actually renders a linear order which is indeed highly correlated, in the ordinal Kendall sense (see *[BIS-2012]*), with the given pairwise outranking relation:

```
>>> corr = g.computeOrdinalCorrelation(cop)
>>> print("Fitness of Copeland's ranking: %.3f" % corr['correlation'])
Fitness of Copeland's ranking: 0.906
```

The Net-Flows ranking

The valued version of the Copeland rule, called **Net-Flows** rule, is working directly on the given valued outranking digraph g. For each alternative x we compute a net flow score that is the sum of the differences between the **outranking** characteristics $r(x\,S\,y)$ and the **outranked** characteristics r(y S x) for all pairs of alternatives where y is different from x:

```
>>> from linearOrders import NetFlowsOrder
>>> nf = NetFlowsOrder(g)
>>> nf.netFlows
[(Decimal('7.143'), 'a3'),
 (Decimal('2.155'), 'a9'),
 (Decimal('1.214'), 'a1'),
 (Decimal('-0.429'), 'a4'),
 (Decimal('-0.690'), 'a8'),
 (Decimal('-1.631'), 'a6'),
 (Decimal('-1.774'), 'a5'),
 (Decimal('-1.845'), 'a2'),
 (Decimal('-4.143'), 'a7')]
>>> nf.showRanking()
['a3', 'a9', 'a1', 'a4', 'a8', 'a6', 'a5', 'a2', 'a7']
>>> corr = g.computeOrdinalCorrelation(nf)
>>> print("Fitness of net flows ranking: %.3f" % corr['correlation'])
Fitness of net flows ranking: 0.828
```

The **Net-Flows** ranking is here, in this didactic example, not as much correlated with the given outranking relation as its crisp cousin ranking.

To appreciate the effective quality of both the Copeland and the Net-Flows rankings, it is useful to consider Kemeny's and Slater's optimal ranking rules.

Kemeny rankings

A **Kemeny** ranking is a linear order which is closest, in the sense of the ordinal Kendall distance (see *[BIS-2012]*), to the given valued outranking digraph g:

```
>>> from linearOrders import KemenyOrder
>>> ke = KemenyOrder(g,orderLimit=9) # default orderLimit is 7
>>> ke.showRanking()
['a1', 'a3', 'a4', 'a9', 'a5', 'a8', 'a2', 'a6', 'a7']
>>> corr = g.computeOrdinalCorrelation(ke)
>>> print("Fitness of Kemeny's ranking: %.3f" % corr['correlation'])
Fitness of Kemeny's ranking: 0.9175
```

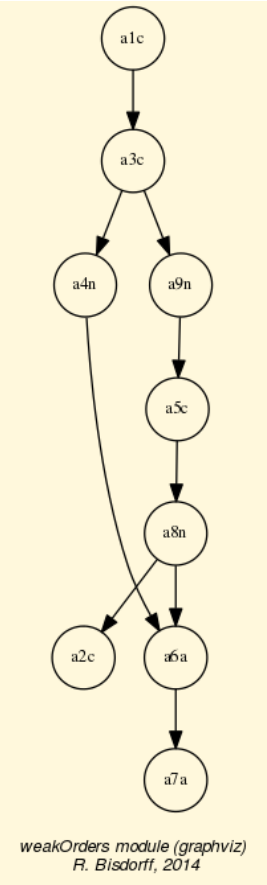
So, **+0.9175** is the highest possible ordinal correlation (fitness) any potential ranking can achieve with the given pairwise outranking relation. A Kemeny ranking may not be unique, and the first one discovered in a brute permutation trying computation, is retained. In our example we hence obtain seven optimal Kemeny rankings with a same maximal Kemeny index of 15.095:

```
>>> ke.maximalRankings
[['a1', 'a3', 'a4', 'a9', 'a5', 'a8', 'a2', 'a6', 'a7'],
['a1', 'a3', 'a4', 'a9', 'a5', 'a8', 'a6', 'a2', 'a7'],
['a1', 'a3', 'a4', 'a9', 'a5', 'a8', 'a6', 'a7', 'a2'],
['a1', 'a3', 'a9', 'a5', 'a8', 'a2', 'a4', 'a6', 'a7'],
['a1', 'a3', 'a9', 'a5', 'a8', 'a4', 'a2', 'a6', 'a7'],
['a1', 'a3', 'a9', 'a5', 'a8', 'a4', 'a6', 'a2', 'a7'],
['a1', 'a3', 'a9', 'a5', 'a8', 'a4', 'a6', 'a7', 'a2']]
```

```
>>> ke.maxKemenyIndex
Decimal('15.095')
```

We may visualize the partial order defined by the epistemic disjunction of these seven Kemeny rankings (see weakOrders module) as follows:

```
>>> from weakOrders import KemenyWeakOrder
>>> wke = KemenyWeakOrder(g,orderLimit=9)
>>> wke.exportGraphViz('tutorialKemeny')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to tutorialKemeny.dot
0 { rank = same; a1; }
1 { rank = same; a3; }
2 { rank = same; a4; a9; }
3 { rank = same; a5; }
4 { rank = same; a8; }
5 { rank = same; a2; a6; }
6 { rank = same; a7; }
dot -Grankdir=TB -Tpng tutorialKemeny.dot -o tutorialKemeny.png
```



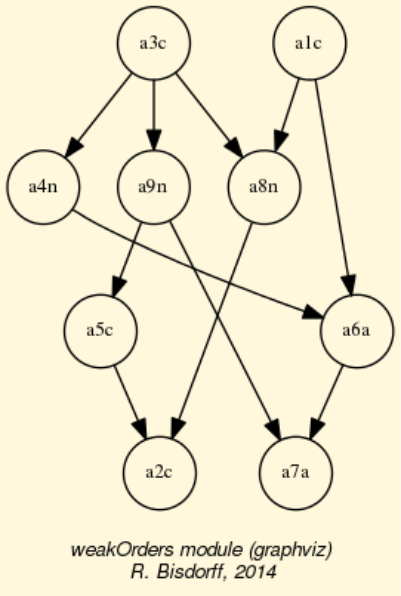
It is interesting to notice that all seven Kemeny rankings place alternative *a1* at rank 1 before alternative *a3*. This is precisely the only inversion that separates the Copeland ranking (see above) from being optimal in the Kemeny sense.

Slater rankings

The **Slater** ranking rule is similar to Kemeny's, but it is working, instead, on the associated crisp Condorcet digraph *c*. It renders here the following results:

```
>>> sl = KemenyOrder(c,orderLimit=9)
>>> len(sl.maximalRankings)
174
>>> sl.showRanking()
['a1', 'a3', 'a8', 'a4', 'a6', 'a9', 'a5', 'a2', 'a7']
>>> corr = g.computeOrdinalCorrelation(sl)
>>> print("Fitness of Slater's ranking: %.3f" % corr['correlation'])
Fitness of Slater's ranking: 0.844
>>> slw = KemenyWeakOrder(c,orderLimit=9)
>>> slw.exportGraphViz('tutorialSlater')
```

We notice that the first crisp Slater ranking is a rather good fit (+0.844), better apparently than the Net-Flows ranking. However, there are in fact 174 such potentially optimal Slater rankings. The corresponding epistemic disjunction gives the follwowing partial ordering:



What precise ranking result should we hence adopt ?

Kemeny's as well as Slater's ranking rules are furthermore computationally difficult problems and effective ranking results are only computable for tiny outranking digraphs (< 15 objects).

More efficient ranking heuristics, like the Copeland and the Net-Flows rules, are therefore needed in practice.

Kohler's ranking-by-choosing rule

Kohler's **ranking-by-choosing** rule can be formulated like this.

At step *r* (*r* goes from 1 to *n*) do the following:

1. Compute for each row of the valued outranking relation table (see above) the smallest value;
2. Select the row where this minimum is maximal. Ties are resolved in lexicographic order;
3. Put the selected decision alternative at rank *r*;

4. Delete the corresponding row and column from the relation table and restart until the table is empty.

```
>>> from linearOrders import KohlerOrder
>>> ko = KohlerOrder(g)
>>> ko.showRanking()
['a3', 'a1', 'a8', 'a4', 'a9', 'a6', 'a7', 'a5', 'a2']
>>> corr = g.computeOrdinalCorrelation(ko)
>>> print("Fitness of Kohler's ranking: %.3f" % corr['correlation'])
Fitness of Kohler's ranking: 0.868
```

Here, we find a better fitness (0.868) when compared with Slater's (0.844) or the Net-Flows result (0.828), but not as good as Copeland crisp rule's result (+0.906).

Tideman's Ranked-Pairs rule

A further ranking heuristic, the **Ranked-Pairs** rule, is based on a prudent incremental construction of linear orders that avoids on the fly any cycling outrankings. The ranking procedure may be formulated as follows:

1. Rank the ordered pairs (x, y) of alternatives in decreasing order of the outranking characteristic values $r(x\,S\,y)$;
2. Consider the pairs in that order (ties are resolved by a lexicographic rule):
   - if the next pair does not create a cycle with the pairs already blocked, block this pair;
   - if the next pair creates a cycle with the already blocked pairs, skip it.

In our didactic outranking example, we get the following result:

```
>>> from linearOrders import RankedPairsOrder
>>> rp = RankedPairsOrder(g,Debug=True)
next pair:  ('a3', 'a9') 1.00
added: (a3,a9) characteristic: 1.00 (1.0)
added: (a9,a3) characteristic: -1.00 (-1.0)
   ...
   ...
next pair:  ('a8', 'a4') 0.09523809523809523809523809524
Circuit detected !!
next pair:  ('a1', 'a8') 0.07142857142857142857142857143
added: (a1,a8) characteristic: 0.07 (1.0)
added: (a8,a1) characteristic: -0.07 (-1.0)
   ...
   ...
next pair:  ('a2', 'a4') 0.00
Circuit detected !!
next pair:  ('a2', 'a6') 0.00
added: (a2,a6) characteristic: 0.00 (1.0)
added: (a6,a2) characteristic: 0.00 (-1.0)
   ...
   ...
>>> rp.showRanking()
['a1', 'a3', 'a4', 'a9', 'a5', 'a8', 'a2', 'a6', 'a7']
```

The Ranked-Pairs rule actually renders one of the seven optimal Kemeny rankings as we may verify below:

```
>>> corr = g.computeOrdinalCorrelation(rp)
>>> print("Fitness of Tideman's ranking: %.3f" % corr['correlation'])
Fitness of Tideman's ranking: 0.918
```

Unfortunately, the Ranked-Pairs ranking rule is again not efficiently scalable to outranking digraphs of larger orders (> 100). For such outranking digraphs, with several hundred of alternatives, only the Copeland and the Net-Flows ranking rules, with a polynomial complexity of $O(n^2)$ where n is the order of the outranking digraph, remain in fact computationally efficient.

Ranking big performance tableaux

None of the previous ranking heuristics, using essentially only the information given by the outranking relation, are scalable for big outranking digraphs gathering millions of pairwise outranking situations. We may notice, however, that a given outranking digraph -the association of a set of decision alternatives and an outranking relation- is, following the methodological requirements of the outranking approach, necessarily associated with a corresponding performance tableau. And, we may use this underlying performance data for linearly decomposing big sets of decision alternatives into ordered quantiles equivalence classes. This decomposition will lead to a pre-ranked sparse outranking digraph.

In the coding example, we generate for instance, by using multiprocessing techniques, first, a cost benefit performance tableau of 100 decision alternatives and, secondly, we construct a **pre-ranked sparse outranking digraph** instance called *bg*. Notice bwt the *BigData* flag used here for generating a parcimonous performance tableau:

```
>>> from sparseOutrankingDigraphs import PreRankedOutrankingDigraph
>>> tp = RandomCBPerformanceTableau(numberOfActions=100,BigData=True,
...                                 Threading=MP,
...                                 seed=100)
>>> bg = PreRankedOutrankingDigraph(tp,quantiles=20,
...                                 LowerClosed=False,
...                                 minimalComponentSize=1,
...                                 Threading=True)
>>> print(bg)
*----- show short --------------*
Instance name     : randomCBperftab_mp
# Actions         : 100
# Criteria        : 7
Sorting by        : 10-Tiling
Ordering strategy : average
Ranking rule      : Copeland
# Components      : 20
Minimal order     : 1
Maximal order     : 20
Average order     : 5.0
fill rate         : 10.061%
----  Constructor run times (in sec.) ----
Total time        : 0.17790
QuantilesSorting  : 0.09019
Preordering       : 0.00043
Decomposing       : 0.08522
Ordering          : 0.00000
<class 'sparseOutrankingDigraphs.PreRankedOutrankingDigraph'> instance
```

The total run time of the *sparseOutrankingDigraphs.PreRankedOutrankingDigraph* constructor is less than a fifth of a second. The corresponding multiple criteria deciles sorting leads to 20 quantiles equivalence classes. The corresponding pre-ranked decomposition may be visualized as follows:

```
>>> bg.showDecomposition()
*--- quantiles decomposition in decreasing order---*
0. ]0.80-0.90] : [49, 10, 52]
1. ]0.70-0.90] : [45]
2. ]0.70-0.80] : [18, 84, 86, 79]
3. ]0.60-0.80] : [41, 70]
```

```
4. ]0.50-0.80] : [44]
5. ]0.60-0.70] : [2, 35, 68, 37, 7, 8, 75, 12, 80, 21, 55, 90, 30, 95]
6. ]0.50-0.70] : [19]
7. ]0.40-0.70] : [69]
8. ]0.50-0.60] : [96, 1, 66, 67, 38, 33, 72, 73, 71, 13, 77, 16, 82,
                  85, 22, 25, 88, 57, 87, 91]
9. ]0.30-0.70] : [42]
10. ]0.40-0.60] : [47]
11. ]0.30-0.60] : [0, 32, 48]
12. ]0.40-0.50] : [34, 5, 31, 83, 76, 78, 15, 51, 14, 54, 56, 27, 60,
                   29, 94, 63]
13. ]0.30-0.50] : [4, 50, 92, 39]
14. ]0.20-0.50] : [43]
15. ]0.30-0.40] : [97, 99, 36, 6, 89, 61, 93]
16. ]0.20-0.40] : [65, 20, 46, 62]
17. ]0.20-0.30] : [64, 81, 3, 53, 24, 40, 74, 28, 26, 58]
18. ]0.10-0.30] : [17, 98, 11]
19. ]0.10-0.20] : [9, 59, 23]
```

The best decile (]80%-90%]) gathers decision alternatives *49*, *10*, and *52*. Worst decile (]10%-20%]) gathers alternatives *9*, *59*, and *23*.

Each one of these 20 ordered components may now be locally ranked by using a suitable ranking rule. Best operational results, both in run times and quality, are more or less equally given with the Copeland and the NetFlows rules. The eventually obtained linear ordering (from the worst to best) is the following:

```
>>> print(bg.boostedOrder)
[59, 9, 23, 17, 11, 98, 26, 81, 40, 64, 3, 74,
28, 53, 24, 58, 65, 62, 46, 20, 93, 89, 97, 61,
99, 6, 36, 43, 4, 50, 39, 92, 94, 60, 14, 76, 63,
51, 56, 34, 5, 54, 27, 78, 15, 29, 31, 83, 32, 0,
48, 47, 42, 16, 1, 66, 72, 71, 38, 57, 33, 73, 88,
85, 82, 22, 96, 91, 67, 87, 13, 77, 25, 69, 19, 21,
95, 35, 80, 37, 7, 12, 68, 2, 90, 55, 30, 75, 8, 44,
41, 70, 79, 86, 84, 18, 45, 49, 10, 52]
```

Alternative *52* appears first ranked, whereas alternative *59* is last ranked. The quality of this ranking result may be assessed by computing its ordinal correlation with the corresponding standard outranking relation:

```
>>> g = BipolarOutrankingDigraph(tp,Normalized=True,Threading=True)
>>> g.computeOrderCorrelation(bg.boostedOrder)
{'correlation': 0.7485,
 'determination': 0.4173}
```

This ranking heuristic is readily scalable with ad hoc HPC tuning to several millions of decision alternatives (see *[BIS-2016]*).

Back to *Tutorials of the Digraph3 resources*

2.1.8 Rating with learned quantile norms

- *Rating performances with quantile norms*

Introduction

In this tutorial we address the problem of **rating multiple criteria performances** of a set of potential decision actions with respect to empirical order statistics, ie performance quantiles learned from historical performance data gathered from similar decision actions observed in the past (see *[CPSTAT-L5]*).

To illustrate the decision problem we face, consider for a moment that, in a given decision aid study, we observe, for instance in the Table below, the multi-criteria performances of two potential decision actions, named *a1007* and *a1008*, marked on 7 seven **incommensurable** preference criteria: a unique **costs** criterion *c1* (to **minimize**) and 6 **benefit** criteria *b1* to *b6* (to **maximize**).

Criterion	c1	b1	b2	b3	b4	b5	b6
weight	6	1	1	1	1	1	1
a1007	-96.9	70.6	9	82.0	5	34.0	8
a1008	-35.7	9.4	5	62.9	6	51.0	4

The performance on the cost criterion *c1* is measured on a cardinal negative scale from -100.00 (worst) to 0.0 (best). The performances on the benefit criteria *b1*, *b3* and *b5* are measured on a cardinal scale from 0.0 (worst) to 100.00 (best), wheras the performances on benefit criteria *b2*, *b4* and *b6* are measured on an ordinal scale from 0 (worst) to 10 (best). The importance (weight) of the costs criterion is equal to the importance (sum of weights) of the benefit criteria taken all together.

The non trivial decision problem we now face here, is to decide, how the multi-criteria performances of *a1007*, respectively *a1008*, may be rated (**excellent** ? **good** ?, or **fair** ?; perhaps even, **weak** ? or **very weak** ?) in an **order statistical sense**, when compared with all potential similar multi-criteria performances one has encountered, or may encounter in the future.

To solve this absolute rating decision problem, first, we need to estimate multi-criteria **performance quantiles** from historical records.

Incremental learning of historical performance quantiles

See also the technical documentation of the *performanceQuantiles module*.

Suppose that we see flying in random multiple criteria performances from a given model of random performance tableau (see the `randomPerfTabs` module). The question we address here is to estimate empirical performance quantiles on the basis of so far observed performance vectors. For this task, we are inspired by *[CHAM-2006]* and *[NR3-2007]*, who present an efficient algorithm for incrementally updating a quantile-binned cumulative density function (CDF) with newly observed CDFs.

The `performanceQuantiles.PerformanceQuantiles` class implements such a performance quantiles estimation based on a given performance tableau. Its main components are:

- An **objectives** and a **criteria** ordered dictionary from a valid performance tableau instance;
- A list **quantileFrequencies** of quantile frequencies like *quartiles* [0.0, 0.25, 05, 0.75,1.0], *quintiles* [0.0, 0.2, 0.4, 0.6, 0.8, 1.0] or *deciles* [0.0, 0.1, 0.2, ... 1.0] for instance;
- An ordered dictionary **limitingQuantiles** of so far estimated *lower* (default) or *upper* quantile class limits for each frequency per criterion;

- An ordered dictionary **historySizes** for keeping track of the number of evaluations seen so far per criterion. Missing data may make these sizes vary from criterion to criterion.

Example python session:

```
>>> from performanceQuantiles import PerformanceQuantiles
>>> from randomPerfTabs import RandomCBPerformanceTableau
>>> nbrActions=1000
>>> nbrCrit = 7
>>> seed = 105
>>> tp = RandomCBPerformanceTableau(numberOfActions=nbrActions,\
...                numberOfCriteria=nbrCrit,seed=seed)
>>> pq = PerformanceQuantiles(tp,\
...                numberOfBins = 'quartiles',\
...                LowerClosed=True,Debug=False)
>>> pq.__dict__.keys()
dict_keys(['objectives', 'LowerClosed', 'name',
'quantilesFrequencies', 'criteria', 'historySizes',
'limitingQuantiles', ... ])
```

The constructor parameter *numberOfBins* (see Lines 7-9 above), choosing the wished number of quantile frequencies, may be either **quartiles**, **quintiles** (5 bins), **deciles** (10 bins) , **dodeciles** (20 bins) or any other integer number of quantile bins. The quantile bins may be either **lower closed** (default) or **upper-closed**.

```
>>> # Printing out the estimated quartile limits
>>> pq.showLimitingQuantiles(ByObjectives=True)
*----  performance quantiles -----*
Costs
criteria  | weights |  '0.0'   '0.25'  '0.5'   '0.75'  '1.0'
 ---------|-------------------------------------------------
    'c1'  |   6     | -97.12  -69.69  -50.08  -28.95  -1.85
Benefits
criteria  | weights |  '0.0'   '0.25'  '0.5'   '0.75'  '1.0'
 ---------|-------------------------------------------------
    'b1'  |   1     |   2.11   27.92   48.76   68.94   98.69
    'b2'  |   1     |   0.00    3.00    5.00    7.00   10.00
    'b3'  |   1     |   1.08   30.41   50.57   69.01   97.23
    'b4'  |   1     |   0.00    3.00    5.00    7.00   10.00
    'b5'  |   1     |   1.84   29.77   50.62   70.14   96.40
    'b6'  |   1     |   0.00    3.00    5.00    7.00   10.00
```

Both objectives are equi-important; the weight (6) of the cost criterion balances the sum of weights (6) of the benefit criteria (see column 2). The preference direction of the cost criterion *c1* is negative; the lesser the costs the better it is, wheras all the benefit criteria *b1* to *b6* show positive preference directions, ie the higher the benefits the better it is. The columns entitled '0.0', resp. '1.0' show the quartile *Q0*, resp. *Q4*, ie the **worst**, resp. **best** performance observed so far on each criterion. Column '0.5' shows the **median** (*Q2*) observed on the criteria.

New decision actions with random multiple criteria performance vectors from the same random performance tableau model may now be generated with ad hoc random performance generators. We provide for experimental purpose, in the *randomPerfTabs* module, three such generators: one for the standard *randomPerfTabs.RandomPerformanceTableau* model, one the for the two objectives *randomPerfTabs.RandomCBPerformanceTableau* Cost-Benefit model, and one for the *randomPerfTabs.Random3ObjectivesPerformanceTableau* model with three objectives concerning respectively economic, environmental or social aspects. Given a set of 10 new decision actions with generated random performance evaluations, the so far estimated historical quantile limits may be updated as follows:

```
>>> # generate 100 new random decision actions
>>> from randomPerfTabs import RandomPerformanceGenerator
>>> rpg = RandomPerformanceGenerator(tp,seed=seed)
>>> newActions = rpg.randomActions(100)
>>> # Updating the quartile norms shown above
>>> pq.updateQuantiles(newActions,historySize=None)
```

Parameter *historySize* (see Line 6) of the *performanceQuantiles.PerformanceQuantiles.updateQuantiles()* method allows to **balance** the **new** evaluations against the **historical** ones. With **historySize = None** (the default setting), the balance in the example above is 1000/1100 (91%, weight of historical data) against 100/1100 (9%, weight of the new incoming observations). Putting **historySize = 0**, for instance, will ignore all historical data (0/100 against 100/100) and restart building the quantile estimation with solely the new incomping data. The updated quantile limits may be shown in a browser view:

```
>>> # showing the updated quantile limits in a browser view
>>> pq.showHTMLLimitingQuantiles(Transposed=True)
```

Performance quantiles

Sampling sizes between 994 and 1004.

criterion	0.00	0.25	0.50	0.75	1.00
b1	2.11	24.39	49.80	69.75	98.69
b2	0.00	5.05	6.57	7.84	10.00
b3	1.08	53.61	59.43	80.16	97.23
b4	0.00	5.10	5.89	6.52	10.00
b5	1.84	32.00	39.16	59.81	96.40
b6	0.00	3.96	4.41	7.65	10.00
c1	-97.12	-73.58	-59.89	-42.08	-1.85

Rating performances with quantile norms

For **absolute rating** of a newly given set of decision actions with the help of empirical performance quantiles estimated from historical data, we provide the *sortingDigraphs.NormedQuantilesRatingDigraph* class, a specialisation of the *sortingDigraphs.QuantilesSortingDigraph* class.

The constructor requires a valid *performanceQuantiles.PerformanceQuantiles* instance.

Note: It is important to notice that the *sortingDigraphs.NormedQuantilesRatingDigraph* class, contrary to the generic *outrankingDigraphs.OutrankingDigraph* class, does not inherit from the generic *perfTabs.PerformanceTableau* class, but instead from the *performanceQuantiles.PerformanceQuantiles* class. The **actions** in such a *sortingDigraphs.NormedQuantilesRatingDigraph* class instance contain not only the newly given decision actions, but also the historical quantile profiles obtained from a given *performanceQuantiles.PerformanceQuantiles* class instance, ie estimated quantile bins' performance limits from historical performance data.

We reconsider the `PerformanceQuantiles` object instance *pq* as computed in the previous section. Let *newActions* be a set of 10 random generated new decision actions of the same kind:

```
>>> from sortingDigraphs import NormedQuantilesRatingDigraph
>>> newActions = rpg.randomActions(10)
>>> nqr = NormedQuantilesRatingDigraph(pq,newActions,rankingRule='best')
>>> nqr
*-----  Object instance description ----------*
Instance class      : NormedQuantilesRatingDigraph
Instance name       : normedRatingDigraph
# Criteria          : 7
# Quantile classes  : 4
# New actions       : 10
Digraph Size        : 85
Determinateness     : 64.44%
Attributes: [
 'LowerClosed', 'actions', 'actionsRanking', 'categories', 'cdf',
 'completeRelation', 'concordanceRelation', 'criteria', 'criteriaCategoryLimits',
 'evaluation', 'gamma', 'hasNoVeto', 'historySizes', 'limitingQuantiles', 'name',
 'nbrThreads', 'newActions', 'notGamma', 'objectives', 'order', 'profileLimits',
 'profiles',
 'quantilesFrequencies', 'rankingCorrelation', 'rankingRule', 'rankingScores',
 'ratingCategories', 'relation', 'runTimes', 'valuationdomain']
*------  Constructor run times (in sec.) ------*
#Threads         : 1
Total time       : 0.54058
Data input       : 0.00191
Quantile classes : 0.00361
Compute profiles : 0.07990
Compute relation : 0.41333
Compute rating   : 0.01617
Compute sorting  : 0.00000
```

Data input to the *sortingDigraphs.NormedQuantilesRatingDigraph* class constructor (see Line 2-3) are a valid PerformanceQuantiles object *pq* and a compatible set *newActions* of new decision actions generated from the same random origin. Let us have a look at the digraph's nodes, here called **actions**. Among the 10 new incoming decision actions (see below) there are 3 advantageous (high benefits, but also high costs), 4 cheap (low costs, but also low benefits) and 4 neutral decision actions. Among the new decision actions shown in the performance tableau below, we recognize actions *a1007* and *a1008* we have mentioned in our introduction.

```
>>> nqr.showPerformanceTableau(actionsSubset=nqr.newActions)
*----  performance tableau -----*
criteria |  'a1001'  'a1002'  'a1003'  'a1004'  'a1005'  'a1006'  'a1007'  'a1008'
 ↪'a1009'  'a1010'
---------|-------------------------------------------------------------------------
 ↪--------------
   'c1'  |   -58.5    -70.9    -70.3    -76.7    -38.1    -45.5    -96.9    -35.7    -
 ↪79.1    -48.5
   'b1'  |    80.6     49.8     65.7     34.9     18.3     20.4     70.6      9.4
 ↪69.0     48.8
   'b2'  |     9        7        8        6        5        7        9        5
 ↪ 2        5
   'b3'  |    55.0     60.0     89.0     53.0     28.0     80.0     82.0     62.0
 ↪59.0     11.0
   'b4'  |     8        6        9        5        5        5        5        6
 ↪ 3        7
   'b5'  |    57.0     30.0     64.0     35.0     30.0     29.0     34.0     51.0
 ↪86.0     39.0
```

```
11    'b6'  |     4        4        4        3        2        7        8        4
↪ 9        5
```

The NormedQuantilesRatingDigraphdigraph instance's actions dictionary also contains the closed lower limits of the four quartile classes: *m1* = [0.0-0.25[, *m2* = [0.25-0.5[, *m3* =[0.5 - 0.75[, *m4* = [0.75 - 1.0[.

```
1  >>> nqr.showPerformanceTableau(actionsSubset=nqr.profiles)
2  *----  performance tableau -----*
3  criteria |  'm1'   'm2'   'm3'   'm4'
4  ---------|---------------------------------------
5     'c1'  | -97.1  -73.7  -60.5  -42.5,
6     'b1'  |   2.1   26.4   49.8   68.9
7     'b2'  |   0.0    4.0    5.0    6.8
8     'b3'  |   1.1   44.4   59.8   73.5
9     'b4'  |   0.0    3.4    4.0    6.1
10    'b5'  |   1.8   31.9   38.9   57.9
11    'b6'  |   0.0    2.2    2.8    5.7
```

The main time (0.4 out of 0.5 sec. , see Lines 21-27 of the object description above) is spent by the class constructor in computing a bipolar valued outranking relation on the extended actions set including both the new actions as well as the quartile class limits. In case of large volumes, ie many new decision actions and centile classes for instance, a multi-threading version may be used when multiple processing cores are available (see the technical description of the *sortingDigraphs.NormedQuantilesRatingDigraph* class).

The actual rating procedure will rely on a complete ranking of the new decision actions as well ass the quantile class limits obtained from the corresponding bipolar valued outranking digraph. Two efficient and scalable ranking rules, the **Copeland** and its valued version, the **Netflows** rule may be used for this purpose. The *rankingRule* parameter allows to choose one of both. With *rankingRule='best'* (see Line 2 above) the NormedQuantilesRatingDigraph constructor will choose the ranking rule that results in the highest ordinal correlation with the given outranking relation (see *[BIS-2012]*).

In this rating example, the Copeland rule appears to be the more appropriate ranking rule:

```
1  >>> print('Ranking rule        :', nqr.rankingRule)
2  Ranking rule        : Copeland
3  >>> print('Actions ranking     :', nqr.actionsRanking)
4  Actions ranking     : [
5  'm4', 'a1008', 'a1006', 'a1005', 'a1001', 'a1003', 'a1010',
6  'm3', 'a1002', 'm2', 'a1004', 'a1009', 'a1007', 'm1']
7  >>> print('Ranking correlation :', nqr.rankingCorrelation)
8  Ranking correlation : {
9   'determination': Decimal('0.544'),
10  'correlation': Decimal('0.966') }
```

We achieve here a linear ranking without ties (from best to worst) of the digraph's actions, ie including the new decision actions as well as the quartile limits *m1* to *m4*, which is very close in an ordinal sense (*tau* = 0.97) to the underlying valued outranking relation. With the *NetFlows* rule we would get the following slightly different ranking result:

```
1  >>> from linearOrders import NetFlowsOrder
2  >>> nf = NetFlowsOrder(nqr)
3  >>> nf.netFlowsRanking
4  ['m4', 'a1008', 'a1001', 'a1006', 'a1003', 'a1005', 'm3',
5   'a1010', 'a1002', 'a1007', 'a1009', 'm2', 'a1004', 'm1']
6  >>> nqr.computeOrderCorrelation(nf.netFlowsOrder)
7  {'determination': Decimal('0.544'),
8   'correlation': Decimal('0.892')}
```

which is, however, less correlated (*tau* = 0.89) with the underlying outranking relation.

The eventual rating procedure is based on the lower quantile limits, such that we may collect the quartile classes' contents in increasing order of the quartiles lower limits:

```
>>> print('Rating categories:', self.ratingCategories)
Rating categories: OrderedDict([
 ('m1', ['a1004', 'a1009', 'a1007']), ('m2', ['a1002']),
 ('m3', ['a1008', 'a1006', 'a1005', 'a1001', 'a1003', 'a1010']),('m4',[]) ])
```

We notice above that no decision action is rated in the highest quartile class [0.75 - 1.0]. Indeed, the rating result is shown in descending order as follows:

```
>>> nqr.showQuantilesRating()
[0.50 - 0.75[ ['a1008', 'a1006', 'a1005', 'a1001', 'a1003', 'a1010']
[0.25 - 0.50[ ['a1002']
[0.00 - 0.25[ ['a1004', 'a1009', 'a1007']
```

The same result may even more conviently be consulted in a browser view via a specialised heatmap format (see `perfTabs:PerformanceTableau.showHTMLPerformanceHeatmap()` method:

```
>>> nqr.showHTMLRatingHeatmap(pageTitle='Heatmap of Quantiles Rating',
↪Correlations=True)
```

Heat map of the ratings

Ranking rule: **Copeland**; *Ranking correlation*: **0.966**

criteria	c1	b4	b2	b5	b3	b6	b1
weights	6	1	1	1	1	1	1
tau(*)	0.80	0.45	0.25	0.24	0.20	0.16	-0.02
[0.75 - <[	-42.08	6.52	7.84	59.81	80.16	7.65	69.75
a1008c	-35.67	6.24	5.02	50.55	61.54	4.25	9.41
a1006n	-45.50	4.96	6.81	28.91	79.89	7.41	20.35
a1005c	-38.15	5.34	5.14	30.37	27.67	1.73	18.28
a1001a	-58.49	7.96	9.07	57.27	55.31	4.01	80.59
a1003n	-70.27	8.90	7.55	63.68	89.12	4.43	65.74
a1010n	-48.50	7.00	4.87	38.87	11.36	4.74	48.83
[0.50 - 0.75[	-59.89	5.89	6.57	39.16	59.43	4.41	49.80
a1002n	-70.86	6.17	7.47	30.26	59.79	3.94	49.84
[0.25 - 0.50[	-73.58	5.10	5.05	32.00	53.61	3.96	24.39
a1004a	-76.71	5.07	6.41	35.29	53.37	3.29	34.89
a1009n	-79.10	3.28	1.81	86.11	58.78	9.32	69.03
a1007a	-96.90	5.22	8.86	34.01	82.48	8.27	70.56
[0.00 - 0.25[	-97.12	0.00	0.00	1.84	1.08	0.00	2.11

Color legend:

quantile	20.00%	40.00%	60.00%	80.00%	100.00%

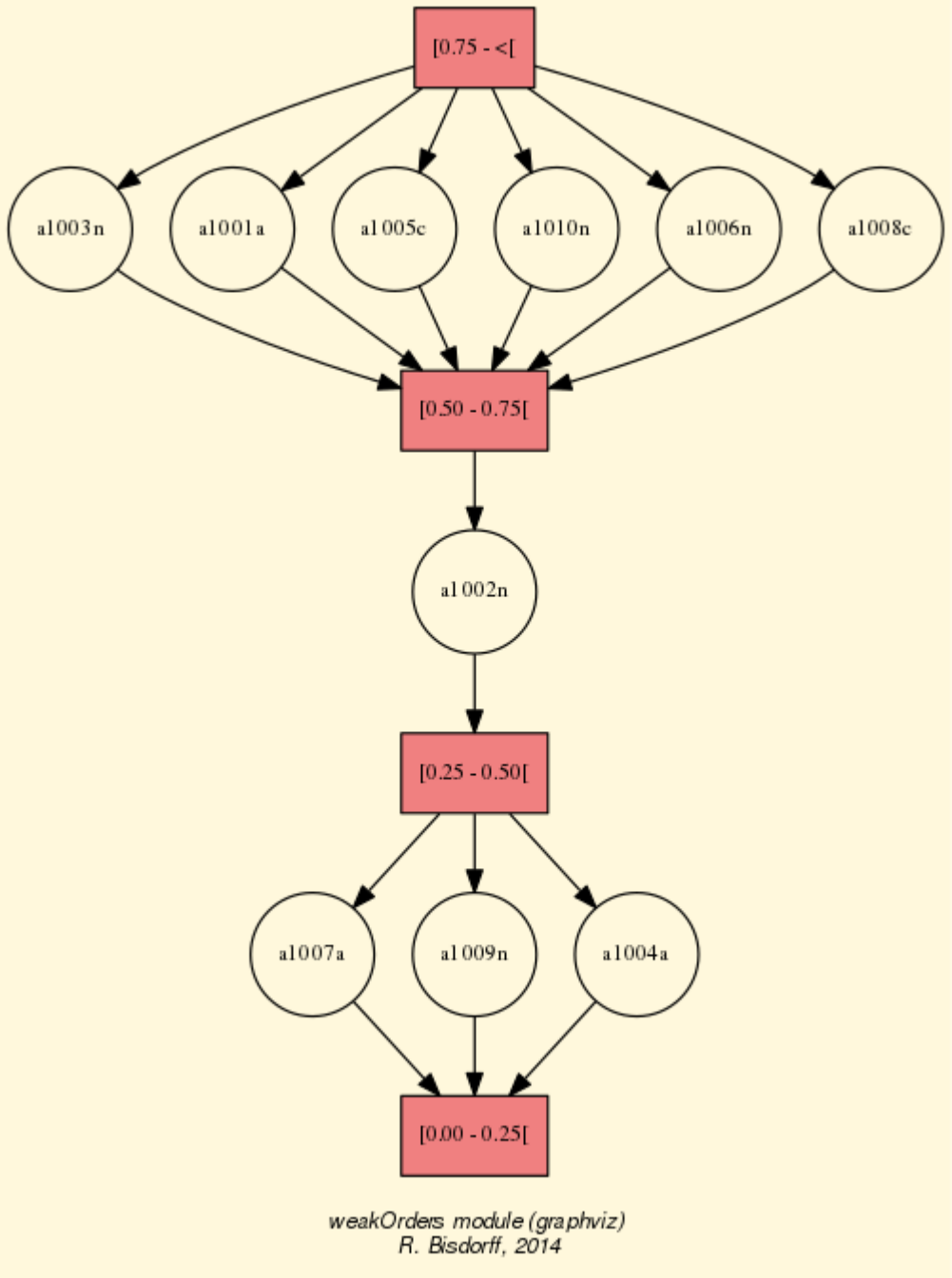
(*) tau: Ordinal (Kendall) correlation between marginal criterion and global ranking relation.
Ordinal (Kendall) correlation between global ranking and outranking relation: 0.97.

Due the fact that the importance weight (6) of the unique cost criterion *c1* is balancing the sum of the six benefit criteria (6) *b1* to *b6*, the marginal cost criteria ranking is highly correlated (*tau* = 0.80) with the proposed rating of the

new decision actions. Is is not a surprise than, that decision action **a1008c**, of **cheap** type (low costs, but several good benfits), appears first ranked in the third quartile class [0.50-0.75[. Whereas, action **a1007a**, of **advantageous** type (excellent benefits but also highest costs), appears worst ranked in the first quartile class [0.0 - 0.25[.

Using furthermore a specialised version of the *weakOrders.WeakOrder.exportGraphViz()* method allows drawing the rating result in a Hasse diagram format.

```
>>> nqr.exportRatingGraphViz()
 *---- exporting a dot file for GraphViz tools ---------*
  Exporting to quantilesRatingDigraph.dot
  dot -Grankdir=TB -Tpng quantilesRatingDigraph.dot -o quantilesRatingDigraph.png
```



A more precise rating result may be achieved when we use **deciles** instead of quartiles for estimating the historical cumulative density functions:

```
>>> pq1 = PerformanceQuantiles(tp, numberOfBins = 'deciles',\
...               LowerClosed=True,Debug=False)
...
>>> nqr1 = NormedQuantilesRatingDigraph(pq1,newActions,rankingRule='best')
>>> nqr1.showQuantilesRating()
*-------- Quantiles rating result ---------
 [0.70 - 0.80[ ['a1008']
 [0.60 - 0.70[ ['a1006', 'a1005']
 [0.50 - 0.60[ ['a1001', 'a1010', 'a1003']
 [0.30 - 0.40[ ['a1002']
 [0.20 - 0.30[ ['a1004']
 [0.10 - 0.20[ ['a1009', 'a1007']
```

Compared with the previous quartiles rating result, we notice that the six alternatives rated before into the third quartile class [0.50 - 0.75[, are now divided up: action *a1008* attains the 8th decile class [0.7 - 0.8[, actions *a1006* and *a1005* the 7th decile class [06 - 0.7[, and actions *a1001*, *a1010* and *a1003* only the 6th decile class [0.5 - 0.6[. Of the three lowest [0.0 - 0.25[rated actions (*a1004*, *a1009* and *a1007*), action *a1004* is now rated in the third decile class [0.2 - 0.3[, and *a1009* and *a1007* in the second decile class [0.1 - 0.2[.

A browser view may again more conveniently illustrate this preciser deciles rating result:

```
>>> nqr1.showHTMLRatingHeatmap(pageTitle='Heat map of the deciles rating',\
...                            colorLevels=5,Correlations=True)
```

Heat map of the deciles rating

Ranking rule: **Copeland**; *Ranking correlation*: **0.964**

criteria	c1	b4	b2	b5	b6	b3	b1
weights	6	1	1	1	1	1	1
tau(*)	0.84	0.58	0.41	0.39	0.39	0.38	0.24
[0.90 - <[	-36.05	8.00	8.97	81.04	8.87	86.56	73.89
[0.80 - 0.90[	-39.13	7.00	7.93	60.10	7.93	81.91	69.89
a1008c	-35.67	6.24	5.02	50.55	4.25	61.54	9.41
[0.70 - 0.80[	-45.80	6.24	7.51	53.00	7.00	65.40	67.84
a1006n	-45.50	4.96	6.81	28.91	7.41	79.89	20.35
a1005c	-38.15	5.34	5.14	30.37	1.73	27.67	18.28
[0.60 - 0.70[	-48.66	6.20	6.93	46.08	4.69	61.07	56.09
a1001a	-58.49	7.96	9.07	57.27	4.01	55.31	80.59
a1010n	-48.50	7.00	4.87	38.87	4.74	11.36	48.83
a1003n	-70.27	8.90	7.55	63.68	4.43	89.12	65.74
[0.50 - 0.60[	-60.50	5.93	6.58	38.82	4.41	59.57	49.76
a1002n	-70.86	6.17	7.47	30.26	3.94	59.79	49.84
[0.30 - 0.40[	-70.83	5.17	5.11	33.97	3.99	54.36	32.47
[0.40 - 0.50[	-69.55	5.31	5.83	35.42	4.22	57.54	41.97
a1004a	-76.71	5.07	6.41	35.29	3.29	53.37	34.89
[0.20 - 0.30[	-76.62	4.98	4.94	30.35	3.75	34.79	20.37
a1009n	-79.10	3.28	1.81	86.11	9.32	58.78	69.03
a1007a	-96.90	5.22	8.86	34.01	8.27	82.48	70.56
[0.10 - 0.20[	-80.13	3.78	2.00	29.94	2.00	18.21	16.10
[0.00 - 0.10[	-97.12	0.00	0.00	1.84	0.00	1.08	2.11

Color legend:

quantile	14.29%	28.57%	42.86%	57.14%	71.43%	85.71%	100.00%

(*) tau: Ordinal (Kendall) correlation between marginal criterion and global ranking relation.

In the case of industrial production monitoring problems, where large volumes of historical performance data may be available, it could become interesting to estimate even more precisely the marginal cumulative density functions with **dodeciles** or even **centiles**. Especially if **tail** rating results, ie distinguishing **very best**, or **very worst** multiple criteria performances, becomes a critical purpose. Similarly, the *historySize* parameter may be used for monitoring on the fly **unstable** random multiple criteria performance data.

Back to *Tutorials of the Digraph3 resources*

2.1.9 Working with the `graphs` module

- *The Berge mystery story: Who is the lier ?*

See also the technical documentation of the *graphs module*.

Structure of a `Graph` object

In the `graphs` module, the root `graphs.Graph` class provides a generic **simple graph model**, without loops and multiple links. A given object of this class consists in:

1. the graph **vertices** : a dictionary of vertices with 'name' and 'shortname' attributes,
2. the graph **valuationDomain** , a dictionary with three entries: the minimum (-1, means certainly no link), the median (0, means missing information) and the maximum characteristic value (+1, means certainly a link),
3. the graph **edges** : a dictionary with frozensets of pairs of vertices as entries carrying a characteristic value in the range of the previous valuation domain,
4. and its associated **gamma function** : a dictionary containing the direct neighbors of each vertice, automatically added by the object constructor.

See the technical documentation of the *graphs module*.

Example Python3 session:

```
>>> from graphs import Graph
>>> g = Graph(numberOfVertices=7,edgeProbability=0.5)
>>> g.save(fileName='tutorialGraph')
```

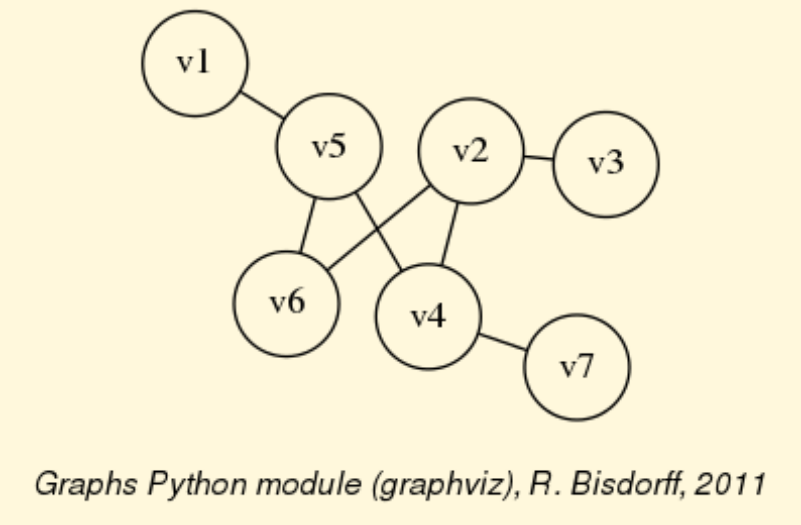
The saved Graph instance named `tutorialGraph.py` is encoded in python3 as follows:

```
# Graph instance saved in Python format
vertices = {
'v1': {'shortName': 'v1', 'name': 'random vertex'},
'v2': {'shortName': 'v2', 'name': 'random vertex'},
'v3': {'shortName': 'v3', 'name': 'random vertex'},
'v4': {'shortName': 'v4', 'name': 'random vertex'},
'v5': {'shortName': 'v5', 'name': 'random vertex'},
'v6': {'shortName': 'v6', 'name': 'random vertex'},
'v7': {'shortName': 'v7', 'name': 'random vertex'},
}
valuationDomain = {'min':-1,'med':0,'max':1}
edges = {
frozenset(['v1','v2']) : -1,
frozenset(['v1','v3']) : -1,
frozenset(['v1','v4']) : -1,
frozenset(['v1','v5']) : 1,
frozenset(['v1','v6']) : -1,
frozenset(['v1','v7']) : -1,
frozenset(['v2','v3']) : 1,
frozenset(['v2','v4']) : 1,
frozenset(['v2','v5']) : -1,
frozenset(['v2','v6']) : 1,
frozenset(['v2','v7']) : -1,
frozenset(['v3','v4']) : -1,
frozenset(['v3','v5']) : -1,
frozenset(['v3','v6']) : -1,
frozenset(['v3','v7']) : -1,
frozenset(['v4','v5']) : 1,
```

```
frozenset(['v4','v6']) : -1,
frozenset(['v4','v7']) : 1,
frozenset(['v5','v6']) : 1,
frozenset(['v5','v7']) : -1,
frozenset(['v6','v7']) : -1,
}
```

The stored graph can be recalled and plotted with the generic *graphs.Graph.exportGraphViz()*[1] method as follows:

```
>>> g = Graph('tutorialGraph')
>>> g.exportGraphViz()
*---- exporting a dot file for GraphViz tools ---------*
Exporting to tutorialGraph.dot
fdp -Tpng tutorialGraph.dot -o tutorialGraph.png
>>> ...
```



Properties, like the gamma function and vertex degrees and neighbourhooddepths may be shown with a *graphs.Graph.showShort()* method:

```
>>> g.showShort()
*---- short description of the graph ----*
Name              : 'tutorialGraph'
Vertices          :  ['v1', 'v2', 'v3', 'v4', 'v5', 'v6', 'v7']
Valuation domain  :  {'min': -1, 'med': 0, 'max': 1}
Gamma function    :
v1 -> ['v5']
v2 -> ['v6', 'v4', 'v3']
v3 -> ['v2']
v4 -> ['v5', 'v2', 'v7']
v5 -> ['v1', 'v6', 'v4']
v6 -> ['v2', 'v5']
v7 -> ['v4']
degrees       :  [0, 1, 2, 3, 4, 5, 6]
distribution  :  [0, 3, 1, 3, 0, 0, 0]
nbh depths    :  [0, 1, 2, 3, 4, 5, 6, 'inf.']
distribution  :  [0, 0, 1, 4, 2, 0, 0, 0]
>>> ...
```

A `Graph` instance corresponds bijectively to a symmetric `Digraph` instance and we may easily convert from one to the other with the *graphs.Graph.graph2Digraph()*, and vice versa with the *digraphs.Digraph.digraph2Graph()* method. Thus, all resources of the *digraphs.Digraph* class, suitable for symmetric digraphs, become readily available, and vice versa:

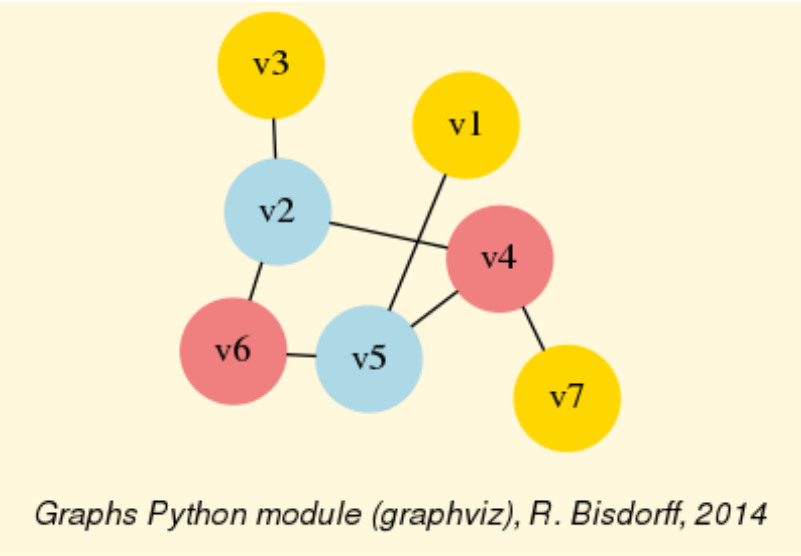
```
>>> dg = g.graph2Digraph()
>>> dg.showRelationTable(ndigits=0,ReflexiveTerms=False)
* ---- Relation Table -----
  S   |  'v1'  'v2'  'v3'  'v4'  'v5'  'v6'  'v7'
------|---------------------------------------
'v1' |    -    -1    -1    -1     1    -1    -1
'v2' |   -1     -     1     1    -1     1    -1
'v3' |   -1     1     -    -1    -1    -1    -1
'v4' |   -1     1    -1     -     1    -1     1
'v5' |    1    -1    -1     1     -     1    -1
'v6' |   -1     1    -1    -1     1     -    -1
'v7' |   -1    -1    -1     1    -1    -1     -
>>> g1 = dg.digraph2Graph()
>>> g1.showShort()
*---- short description of the graph ----*
Name             : 'tutorialGraph'
Vertices         :  ['v1', 'v2', 'v3', 'v4', 'v5', 'v6', 'v7']
Valuation domain :  {'med': 0, 'min': -1, 'max': 1}
Gamma function   :
v1 -> ['v5']
v2 -> ['v3', 'v6', 'v4']
v3 -> ['v2']
v4 -> ['v5', 'v7', 'v2']
v5 -> ['v6', 'v1', 'v4']
v6 -> ['v5', 'v2']
v7 -> ['v4']
degrees      :  [0, 1, 2, 3, 4, 5, 6]
distribution :  [0, 3, 1, 3, 0, 0, 0]
nbh depths   :  [0, 1, 2, 3, 4, 5, 6, 'inf.']
distribution :  [0, 0, 1, 4, 2, 0, 0, 0]
>>> ...
```

q-coloring of a graph

A 3-coloring of the tutorial graph g may for instance be computed and plotted with the *graphs.Q_Coloring* class as follows:

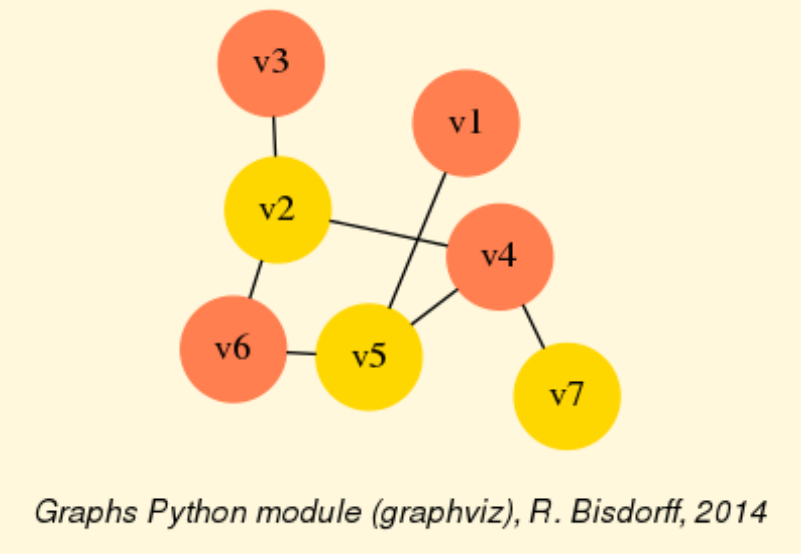
```
>>> from graphs import Q_Coloring
>>> qc = Q_Coloring(g)
Running a Gibbs Sampler for 42 step !
The q-coloring with 3 colors is feasible !!
>>> qc.showConfiguration()
v5 lightblue
v3 gold
v7 gold
v2 lightblue
v4 lightcoral
v1 gold
v6 lightcoral
>>> qc.exportGraphViz('tutorial-3-coloring')
*---- exporting a dot file for GraphViz tools ---------*
```

```
Exporting to tutorial-3-coloring.dot
fdp -Tpng tutorial-3-coloring.dot -o tutorial-3-coloring.png
```



Actually, with the given tutorial graph instance, a 2-coloring is already feasible:

```
>>> qc = Q_Coloring(g,colors=['gold','coral'])
Running a Gibbs Sampler for 42 step !
The q-coloring with 2 colors is feasible !!
>>> qc.showConfiguration()
v5 gold
v3 coral
v7 gold
v2 gold
v4 coral
v1 coral
v6 coral
>>> qc.exportGraphViz('tutorial-2-coloring')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to tutorial-2-coloring.dot
fdp -Tpng tutorial-2-coloring.dot -o tutorial-2-coloring.png
```



MIS and Clique enumeration

2-colorings define independent sets of vertices that are maximal in cardinality; for short called a **MIS**. Computing such MISs in a given `Graph` instance may be achieved by the *graphs.Graph.showMIS()* method;

```
>>> g = Graph('tutorialGraph')
>>> g.showMIS()
*---  Maximal Independent Sets ---*
['v2', 'v5', 'v7']
['v3', 'v5', 'v7']
['v1', 'v2', 'v7']
['v1', 'v3', 'v6', 'v7']
['v1', 'v3', 'v4', 'v6']
number of solutions:  5
cardinality distribution
card.:  [0, 1, 2, 3, 4, 5, 6, 7]
freq.:  [0, 0, 0, 3, 2, 0, 0, 0]
execution time: 0.00032 sec.
Results in self.misset
>>> g.misset
[frozenset({'v7', 'v2', 'v5'}),
 frozenset({'v3', 'v7', 'v5'}),
 frozenset({'v1', 'v2', 'v7'}),
 frozenset({'v1', 'v6', 'v7', 'v3'}),
 frozenset({'v1', 'v6', 'v4', 'v3'})]
```

A MIS in the dual of a graph instance g (its negation $-g$), corresponds to a maximal **clique**, ie a maximal complete subgraph in g. Maximal cliques may be directly enumerated with the *graphs.Graph.showCliques()* method:

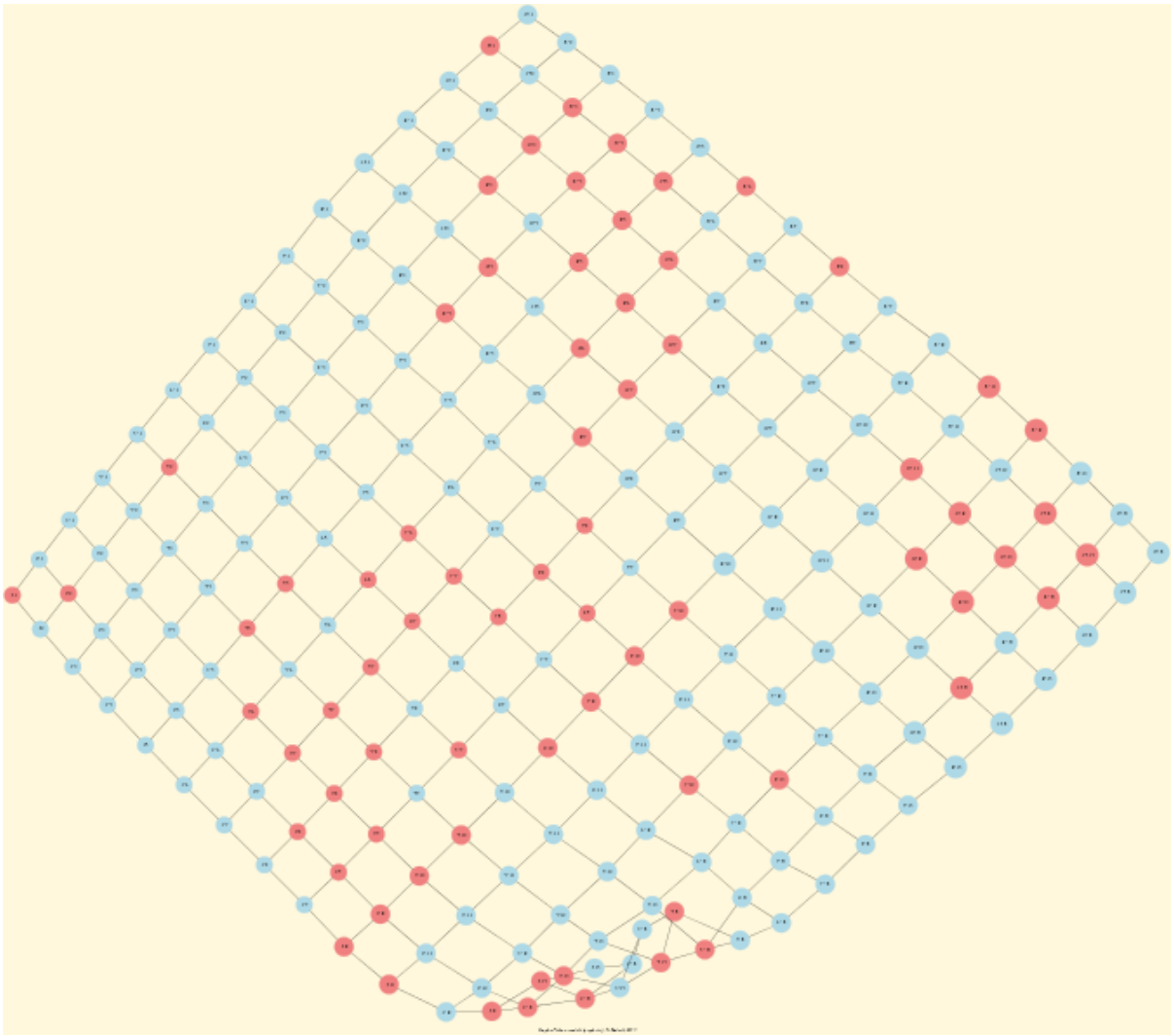
```
>>> g.showCliques()
*---  Maximal Cliques ---*
['v2', 'v3']
['v4', 'v7']
['v2', 'v4']
['v4', 'v5']
['v1', 'v5']
```

```
['v2', 'v6']
['v5', 'v6']
number of solutions:  7
cardinality distribution
card.:  [0, 1, 2, 3, 4, 5, 6, 7]
freq.:  [0, 0, 7, 0, 0, 0, 0, 0]
execution time: 0.00049 sec.
Results in self.cliques
>>> g.cliques
[frozenset({'v2', 'v3'}), frozenset({'v4', 'v7'}),
 frozenset({'v2', 'v4'}), frozenset({'v4', 'v5'}),
 frozenset({'v1', 'v5'}), frozenset({'v6', 'v2'}),
 frozenset({'v6', 'v5'})]
>>> ...
```

Grids and the Ising model

Special classes of graphs, like *n* x *m* **rectangular** or **triangular grids** (*graphs.GridGraph* and *graphs.IsingModel*) are available in the *graphs* module. For instance, we may use a Gibbs sampler again for simulating an **Ising Model** on such a grid:

```
>>> from graphs import GridGraph, IsingModel
>>> g = GridGraph(n=15,m=15)
>>> g.showShort()
*----- show short --------------*
Grid graph    :  grid-6-6
n             :  6
m             :  6
order         :  36
>>> im = IsingModel(g,beta=0.3,nSim=100000,Debug=False)
Running a Gibbs Sampler for 100000 step !
>>> im.exportGraphViz(colors=['lightblue','lightcoral'])
*---- exporting a dot file for GraphViz tools ---------*
Exporting to grid-15-15-ising.dot
fdp -Tpng grid-15-15-ising.dot -o grid-15-15-ising.png
```



Simulating Metropolis random walks

Finally, we provide the *graphs.MetropolisChain* class, a specialization of the *graphs.Graph* class, for implementing a generic **Metropolis MCMC** (Monte Carlo Markov Chain) sampler for simulating random walks on a given graph following a given probability `probs = {'v1': x, 'v2': y, ...}` for visiting each vertex (see lines 14-22).

```
1  >>> from graphs import MetropolisChain
2  >>> g = Graph(numberOfVertices=5,edgeProbability=0.5)
3  >>> g.showShort()
4  *---- short description of the graph ----*
5  Name             : 'randomGraph'
6  Vertices         :  ['v1', 'v2', 'v3', 'v4', 'v5']
7  Valuation domain :  {'max': 1, 'med': 0, 'min': -1}
8  Gamma function   :
9  v1 -> ['v2', 'v3', 'v4']
10 v2 -> ['v1', 'v4']
11 v3 -> ['v5', 'v1']
12 v4 -> ['v2', 'v5', 'v1']
13 v5 -> ['v3', 'v4']
```

```
>>> probs = {}  # initialize a potential stationary probability vector
>>> n = g.order # for instance: probs[v_i] = n-i/Sum(1:n) for i in 1:n
>>> i = 0
>>> verticesList = [x for x in g.vertices]
>>> verticesList.sort()
>>> for v in verticesList:
...     probs[v] = (n - i)/(n*(n+1)/2)
...     i += 1
>>> met = MetropolisChain(g,probs)
>>> frequency = met.checkSampling(verticesList[0],nSim=30000)
>>> for v in verticesList:
...     print(v,probs[v],frequency[v])
v1 0.3333 0.3343
v2 0.2666 0.2680
v3 0.2    0.2030
v4 0.1333 0.1311
v5 0.0666 0.0635
>>> met.showTransitionMatrix()
* ---- Transition Matrix -----
  Pij  | 'v1'    'v2'    'v3'    'v4'    'v5'
  -----|-------------------------------------
  'v1' |  0.23   0.33    0.30    0.13    0.00
  'v2' |  0.42   0.42    0.00    0.17    0.00
  'v3' |  0.50   0.00    0.33    0.00    0.17
  'v4' |  0.33   0.33    0.00    0.08    0.25
  'v5' |  0.00   0.00    0.50    0.50    0.00
```

The `checkSampling()` method (see line 23) generates a random walk of *nSim=30000* steps on the given graph and records by the way the observed relative frequency with which each vertex is passed by. In this example, the stationary transition probability distribution, shown by the `showTransitionMatrix()` method above (see lines 31-), is quite adequately simulated.

For more technical information and more code examples, look into the technical documentation of the *graphs module*. For the readers interested in algorithmic applications of Markov Chains we may recommend consulting O. Häggström's 2002 book: *[FMCAA]*.

The Berge mystery story: Who is the lier ?

Suppose that the file `berge.py` contains the following *graphs.Graph* instance data:

```
vertices = {
'A': {'name': 'Abe', 'shortName': 'A'},
'B': {'name': 'Burt', 'shortName': 'B'},
'C': {'name': 'Charlotte', 'shortName': 'C'},
'D': {'name': 'Desmond', 'shortName': 'D'},
'E': {'name': 'Eddie', 'shortName': 'E'},
'I': {'name': 'Ida', 'shortName': 'I'},
}
valuationDomain = {'min':-1,'med':0,'max':1}
edges = {
frozenset(['A','B']) : 1,
frozenset(['A','C']) : -1,
frozenset(['A','D']) : 1,
frozenset(['A','E']) : 1,
frozenset(['A','I']) : -1,
frozenset(['B','C']) : -1,
frozenset(['B','D']) : -1,
```

```
frozenset(['B','E']) : 1,
frozenset(['B','I']) : 1,
frozenset(['C','D']) : 1,
frozenset(['C','E']) : 1,
frozenset(['C','I']) : 1,
frozenset(['D','E']) : -1,
frozenset(['D','I']) : 1,
frozenset(['E','I']) : 1,
}
```

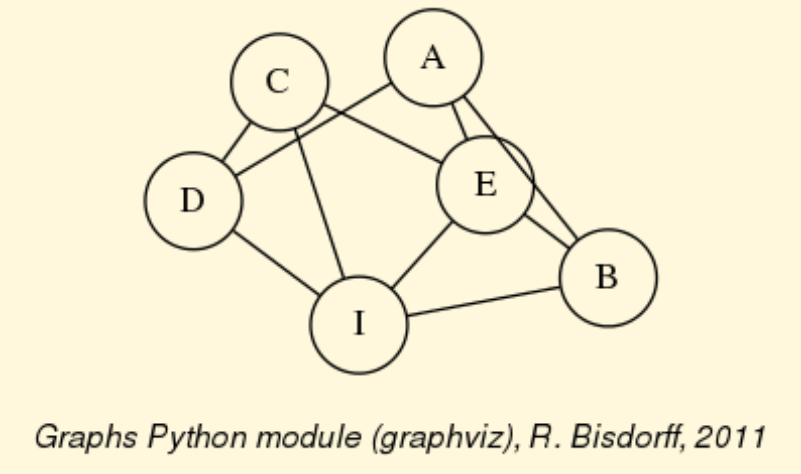
This data concerns the famous *Berge mystery story* (see Golumbic, M. C. Algorithmic Graph Theory and Perfect Graphs, *Annals of Discrete Mathematics* 57 p. 20) Six professors (labeled *A*, *B*, *C*, *D*, *E* and *I*) had been to the library on the day that a rare tractate was stolen. Each entered once, stayed for some time, and then left. If two professors were in the library at the same time, then at least one of them saw the other. Detectives questioned the professors and gathered the testimonies that *A* saw *B* and *E*; *B* saw *A* and *I*; *C* saw *D* and *I*; *D* saw *A* and *I*; *E* saw *B* and *I*; and *I* saw *C* and *E*. This data is gathered in the previous file, where each positive edge $\{x, y\}$ models the testimony that, either x saw y, or, y saw x.

Example Python3 session:

```
>>> from graphs import Graph
>>> g = Graph('berge')
>>> g.showShort()
*---- short description of the graph ----*
Name             : 'berge'
Vertices         :  ['A', 'B', 'C', 'D', 'E', 'I']
Valuation domain :  {'min': -1, 'med': 0, 'max': 1}
Gamma function   :
A -> ['D', 'B', 'E']
B -> ['E', 'I', 'A']
C -> ['E', 'D', 'I']
D -> ['C', 'I', 'A']
E -> ['C', 'B', 'I', 'A']
I -> ['C', 'E', 'B', 'D']
```

The graph data can be plotted as follows:

```
>>> g.exportGraphViz('berge1')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to berge1.dot
fdp -Tpng berge1.dot -o berge1.png
```

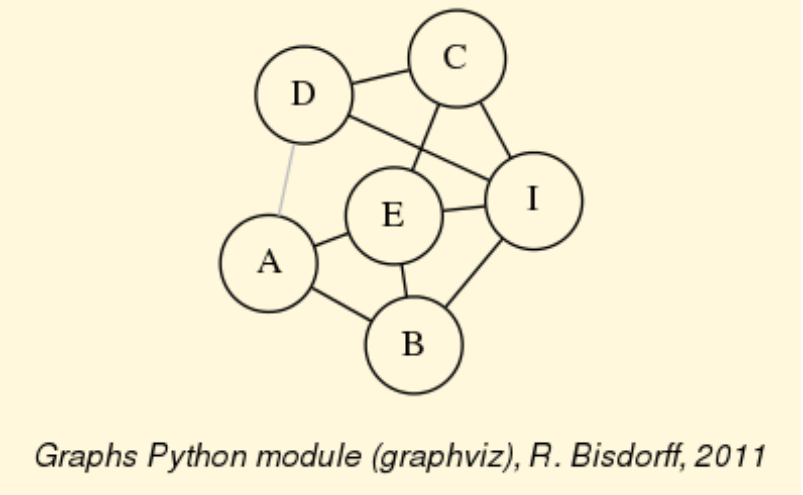


From graph theory we know that time interval intersection graphs must in fact be triangulated. The testimonies graph should therefore not contain any chordless cycles of four and more vertices. Now, the presence or not of chordless cycles may be checked as follows:

```
>>> g.computeChordlessCycles()
Chordless cycle certificate -->>>  ['D', 'C', 'E', 'A', 'D']
Chordless cycle certificate -->>>  ['D', 'I', 'E', 'A', 'D']
Chordless cycle certificate -->>>  ['D', 'I', 'B', 'A', 'D']
[(['D', 'C', 'E', 'A', 'D'], frozenset({'C', 'D', 'E', 'A'})),
(['D', 'I', 'E', 'A', 'D'], frozenset({'D', 'E', 'I', 'A'})),
(['D', 'I', 'B', 'A', 'D'], frozenset({'D', 'B', 'I', 'A'}))]
```

We see three intersection cycles of length 4, which is impossible to occur on the linear time line. Obviously one professor lied! And it is *D* ; if we put to doubt the testimony that he indeed saw *A*, we obtain a correctly triangulated graph:

```
>>> g.setEdgeValue( ('D','A'), 0)
>>> g.showShort()
*---- short description of the graph ----*
Name             : 'berge'
Vertices         :  ['A', 'B', 'C', 'D', 'E', 'I']
Valuation domain :  {'med': 0, 'min': -1, 'max': 1}
Gamma function   :
A -> ['B', 'E']
B -> ['A', 'I', 'E']
C -> ['I', 'E', 'D']
D -> ['I', 'C']
E -> ['A', 'I', 'B', 'C']
I -> ['B', 'E', 'D', 'C']
>>> g.computeChordlessCycles()
[]
>>> g.exportGraphViz('berge2')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to berge2.dot
fdp -Tpng berge2.dot -o berge2.png
```



Back to *Tutorials of the Digraph3 resources*

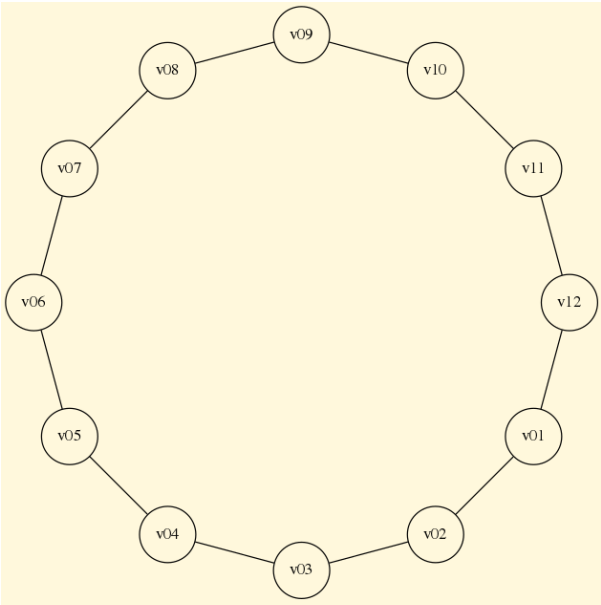
2.1.10 Computing the non isomorphic MISs of the n-cycle graph

Due to the public success of our common 2008 publication with Jean-Luc Marichal *[ISOMIS-08]* , we present in this last tutorial an example Python session for computing the **non isomorphic maximal independent sets** (MISs) from the 12-cycle graph, i.e. a *digraphs.CirculantDigraph* class instance of order 12 and symmetric circulants 1 and -1:

```
>>> from digraphs import CirculantDigraph
>>> c12 = CirculantDigraph(order=12,circulants=[1,-1])
>>> c12 # 12-cycle digraph instance
*------- Digraph instance description ------*
Instance class   : CirculantDigraph
Instance name    : c12
Digraph Order    : 12
Digraph Size     : 24
Valuation domain : [-1.0, 1.0]
Determinateness  : 100.000
Attributes       : ['name', 'order', 'circulants', 'actions',
                    'valuationdomain', 'relation', 'gamma',
                    'notGamma']
```

Such *n*-cycle graphs are also provided as undirected graph instances by the *graphs.CycleGraph* class:

```
>>> from graphs import CycleGraph
>>> cg12 = CycleGraph(order=12)
>>> cg12
*------- Graph instance description ------*
Instance class   : CycleGraph
Instance name    : cycleGraph
Graph Order      : 12
Graph Size       : 12
Valuation domain : [-1.0, 1.0]
Attributes       : ['name', 'order', 'vertices', 'valuationDomain',
                    'edges', 'size', 'gamma']
>>> cg12.exportGraphViz('cg12')
```



A non isomorphic MIS corresponds in fact to a set of isomorphic MISs, i.e. an orbit of MISs under the automorphism group of the 12-cycle graph. We are now first computing all maximal independent sets that are detectable in the 12-cycle digraph with the *digraphs.Digraph.showMIS()* method:

```
>>> c12.showMIS(withListing=False)
  *---  Maximal independent choices ---*
  number of solutions:  29
  cardinality distribution
  card.:  [0, 1, 2, 3, 4,  5,  6, 7, 8, 9, 10, 11, 12]
  freq.:  [0, 0, 0, 0, 3, 24,  2, 0, 0, 0,  0,  0,  0]
  Results in c12.misset
```

In the 12-cycle graph, we observe 29 labelled MISs: – 3 of cardinality 4, 24 of cardinality 5, and 2 of cardinality 6. In case of n-cycle graphs with n > 20, as the cardinality of the MISs becomes big, it is preferable to use the shell `perrinMIS` command compiled from C and installed[3] along with all the the Digraphs3 python modules for computing the set of MISs observed in the graph:

```
...$ echo 12 | /usr/local/bin/perrinMIS
# ------------------------------------ #
# Generating MIS set of Cn with the    #
# Perrin sequence algorithm.           #
# Temporary files used.                #
# even versus odd order optimized.     #
# RB December 2006                     #
# Current revision Dec 2018            #
# ------------------------------------ #
Input cycle order ? <-- 12
mis 1 : 100100100100
mis 2 : 010010010010
mis 3 : 001001001001
...
...
...
mis 27 : 001001010101
mis 28 : 101010101010
mis 29 : 010101010101
Cardinalities:
0 : 0
1 : 0
2 : 0
3 : 0
4 : 3
5 : 24
6 : 2
7 : 0
8 : 0
9 : 0
10 : 0
11 : 0
12 : 0
Total: 29
execution time: 0 sec. and 2 millisec.
```

Reading in the result of the `perrinMIS`, stored in a file called by default `curd.dat`, may be operated with the *digraphs.Digraph.readPerrinMisset()* method.

[3] The `perrinMIS` shell command my be installed system wide with the command `.../Digraph3$ make installPerrin` from the main Digraph3 directory. It is stored by default into `</usr/local/bin/>`. This may be changed with the `INSTALLDIR` flag. The command `.../Digraph3$ make installPerrinUser` installs it instead without sudo into the user's private `<$Home/.bin>` directory.

```
>>> c12.readPerrinMisset(file='curd.dat')
>>> c12.misset
{frozenset({'5', '7', '10', '1', '3'}),
 frozenset({'9', '11', '5', '2', '7'}),
 frozenset({'7', '2', '4', '10', '12'}),
 ...
 ...
 ...
 frozenset({'8', '4', '10', '1', '6'}),
 frozenset({'11', '4', '1', '9', '6'}),
 frozenset({'8', '2', '4', '10', '12', '6'})
}
```

For computing the corresponding non isomorphic MISs, we actually need the automorphism group of the c12-cycle graph. The *digraphs.Digraph* class therefore provides the *digraphs.Digraph.automorphismGenerators()* method which adds automorphism group generators to a *digraphs.Digraph* class instance with the help of the external shell <dreadnaut> command from the **nauty** software package[2].

```
>>> c12.automorphismGenerators()
...
  Permutations
  {'1': '1', '2': '12', '3': '11', '4': '10', '5':
   '9', '6': '8', '7': '7', '8': '6', '9': '5', '10':
   '4', '11': '3', '12': '2'}
  {'1': '2', '2': '1', '3': '12', '4': '11', '5': '10',
   '6': '9', '7': '8', '8': '7', '9': '6', '10': '5',
   '11': '4', '12': '3'}
>>> print('grpsize = ', c12.automorphismGroupSize)
  grpsize = 24
```

The 12-cycle graph automorphism group is generated with both the permutations above and has group size 24.

The command *digraphs.Digraph.showOrbits()* renders now the labelled representatives of each of the four orbits of isomorphic MISs observed in the 12-cycle graph (see Lines 7-10).

```
>>> c12.showOrbits(c12.misset,withListing=False)
...
  *---- Global result ----
  Number of MIS:  29
  Number of orbits :  4
  Labelled representatives and cardinality:
  1: ['2','4','6','8','10','12'], 2
  2: ['2','5','8','11'], 3
  3: ['2','4','6','9','11'], 12
  4: ['1','4','7','9','11'], 12
  Symmetry vector
  stabilizer size: [1, 2, 3, ..., 8, 9, ..., 12, 13, ...]
  frequency      : [0, 2, 0, ..., 1, 0, ...,  1,  0, ...]
```

The corresponding group stabilizers' sizes and frequencies – orbit 1 with 12 symmetry axes, orbit 2 with 8 symmetry axes, and orbits 3 and 4 both with one symmetry axis (see Lines 11-13), are illustrated in the corresponding unlabelled graphs of *Figure-1* below:

[2] Dependency: The :py:func:digraphs.Digraph.automorphismGenerators method uses the shell `dreadnaut` command from the nauty software package. See https://www3.cs.stonybrook.edu/~algorith/implement/nauty/implement.shtml . On Mac OS there exist dmg installers and on Ubuntu Linux or Debian, one may easily install it with:

```
...$ sudo apt-get install nauty
```

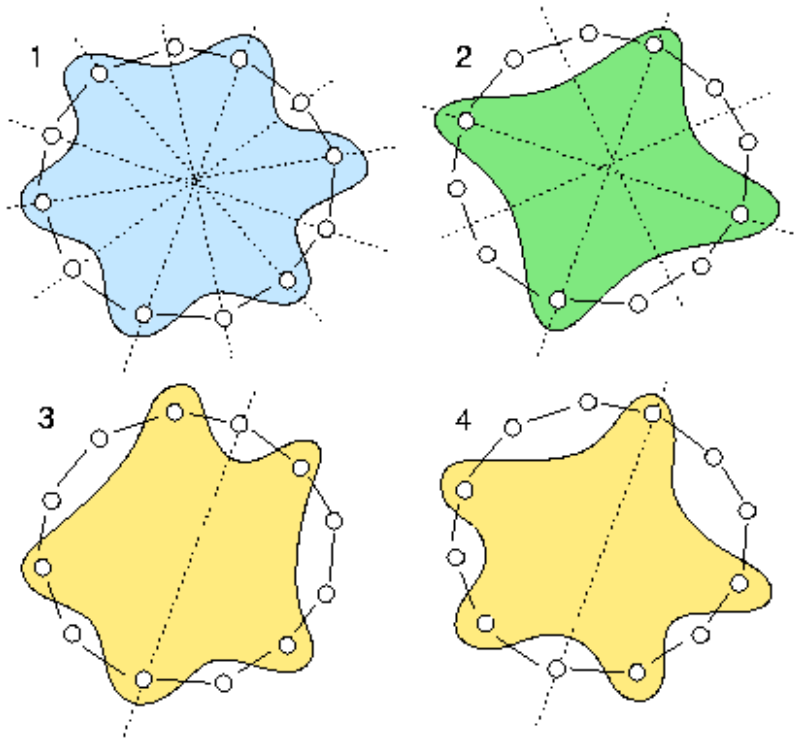


Figure-1: The symmetry axes of the four non isomorphic MISs of the 12-cycle graph:

The non isomorphic MISs in the 12-cycle graph represent in fact all the ways one may write the number 12 as the circular sum of '2's and '3's without distinguishing opposite directions of writing. The first orbit corresponds to writing six times a '2'; the second orbit corresponds to writing four times a '3'. The third and fourth orbit correspond to writing two times a '3' and three times a '2'. There are two non isomorphic ways to do this latter circular sum. Either separating the '3's by one and two '2's, or by zero and three '2's (see Bisdorff & Marichal *[ISOMIS-08]*).

2.1.11 Links and appendices

Documents

- Introduction
- Reference manual
- Tutorial

Indices and tables

- genindex
- modindex
- search

References

Footnotes

The second part concerns the reference manual of the proposed Python3 modules, classes and methods. The main generic root classes in this collection are the `digraphs.Digraph` class, the `perfTabs.PerformanceTableau` class and the `outrankingDigraphs.OutrankingDigraph` class. The technical documentation also provides links to the complete source code of all modules, classes and methods.

2.2 Technical Reference of the Digraph3 modules

Author Raymond Bisdorff, Emeritus Professor, University of Luxembourg FSTC/CSC

Version Revision: Python 3.6

Table of Contents

2.2.1 Installation

Dowloading the Digraph3 resources

Three download options are given:

1. Either (easiest under Linux or Mac OS-X), by using a git client:

```
..$ git clone https://github.com/rbisdorff/Digraph3
```

2. or, a subversion client:

```
..$ svn co https://leopold-loewenheim.uni.lu/svn/repos/Digraph3
```

3. Or, with a browser access, download and extract the latest distribution release tar.gz or zip archive from this sourceforge page:

```
https://sourceforge.net/projects/digraph3/
```

On Linux or Mac OS, ..$ cd to the extracted <Digraph3> directory:

```
../Digraph3$ make install
```

installs (with sudo !!) the digraphs module in the current running python environment. Pythhon 3.5 (or later) environment is recommended. Whereas:

```
../Digraph3$ make installVenv
```

installs the modules in an activated virtual python environment.

If the cyton C-compiled modules for Big Data applications are required, it is necessary to previously install the Cython package in the running Python environment:

```
...$pip3.5+ install cython
```

It is recommended to run a nose test suite:

```
../Digraph3$ make tests
```

in the ../test directory (python3 nose package required . . . $ pip3 install nose):

```
../Digraph3$ make verboseTests
```

runs a verbose (with stdout not captured) nose test suite:

```
../Digraph3$ make pTests
```

runs the nose test suite in multiple processing mode when the GNU parallel shell tool is installed and multiple cores are detected.

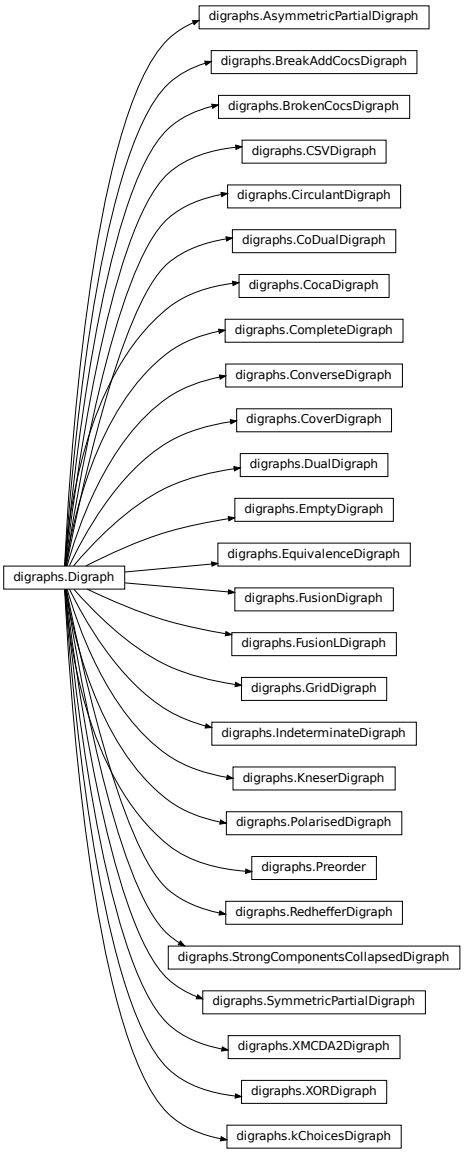
Dependencies: To be fully functional, the Digraph3 resources mainly need the graphviz tools and the R statistics resources to be installed. When exploring digraph isomorphisms, the nauty isomorphism testing program is required. Two specific criteria and actions clustering methods of the OutrankingDigraph class furthermore require the calmat matrix computing resource to be installed.

2.2.2 Organisation of the Digraph3 modules

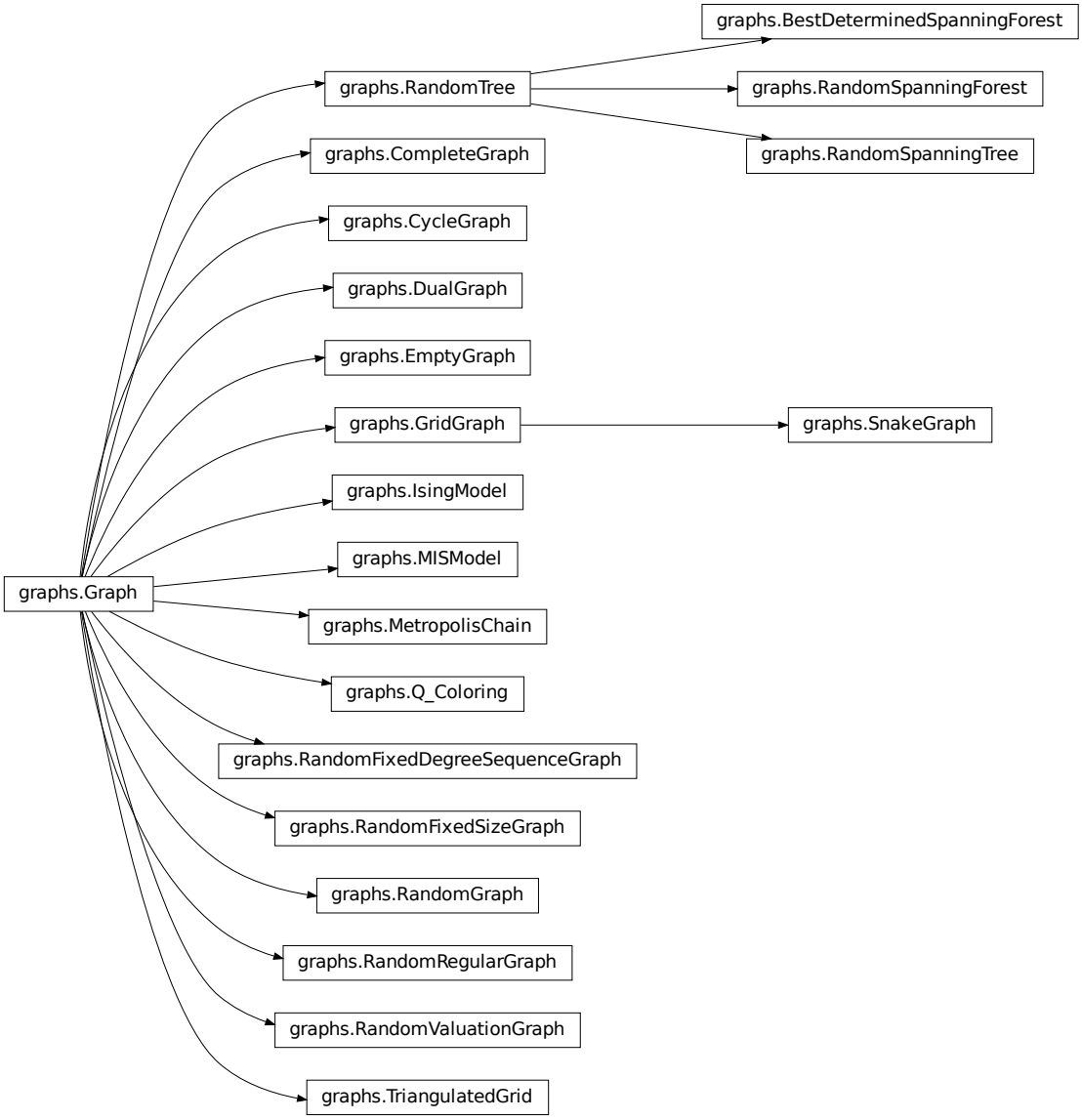
The Digraph3 source code is split into several interdependent modules of which the `digraphs` module is the master module.

Basic modules

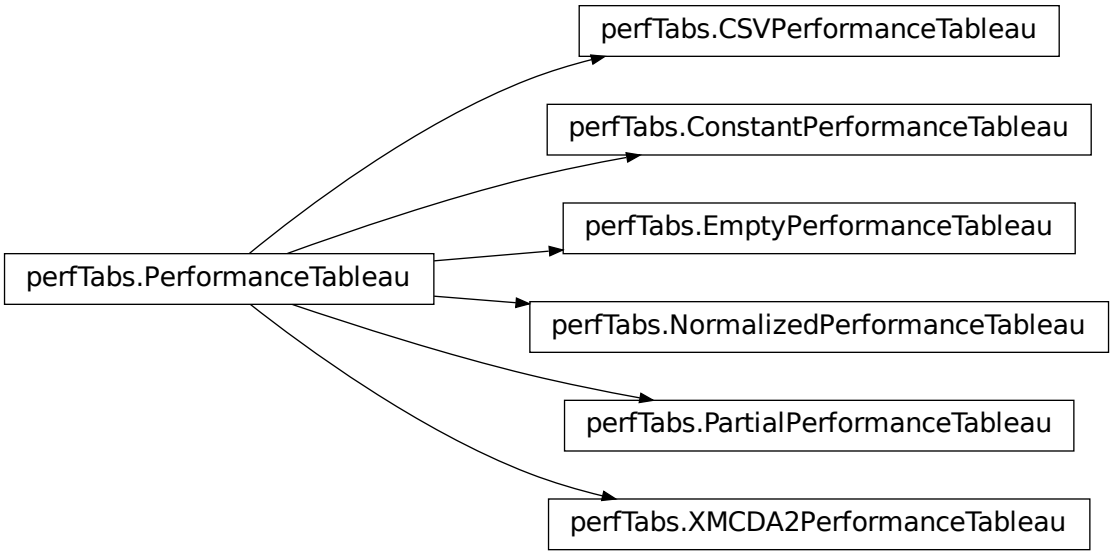
- *digraphs module* Main part of the Digraph3 source code with the root Digraph class.



- *graphs module* Resources for handling undirected graphs with the root Graph class and a brigde to the `digraphs` module resources.

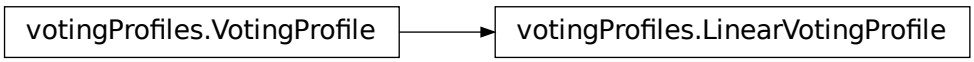


- *perfTabs module* Tools for handling multiple criteria performance tableaux with root PerformanceTableau class.



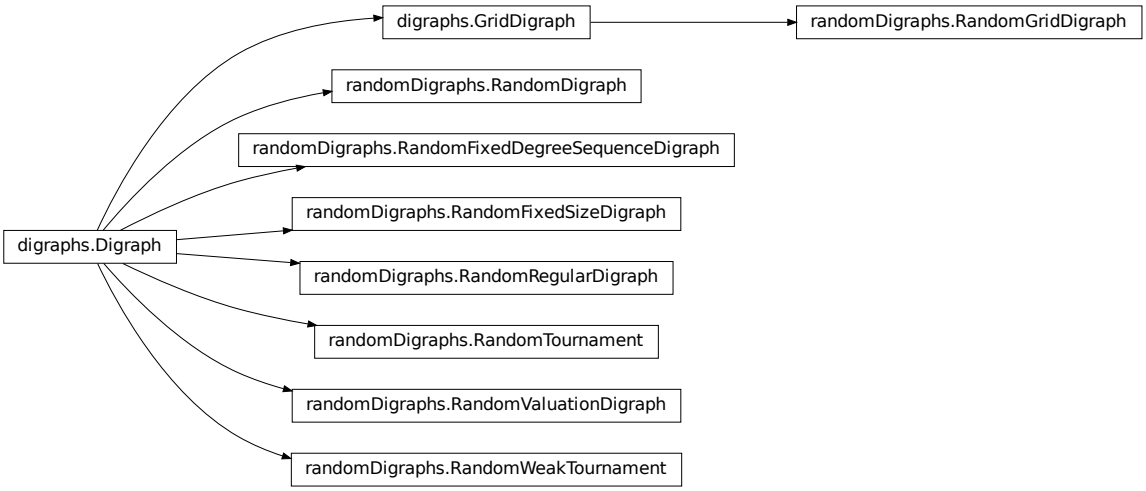
- *outrankingDigraphs module* Root module for handling outranking digraphs with the main BipolarOutrankingDigraph class and its specializations.

- *votingProfiles module* Classes and methods for handling voting ballots and computing election results with main LinearVotingProfile class.

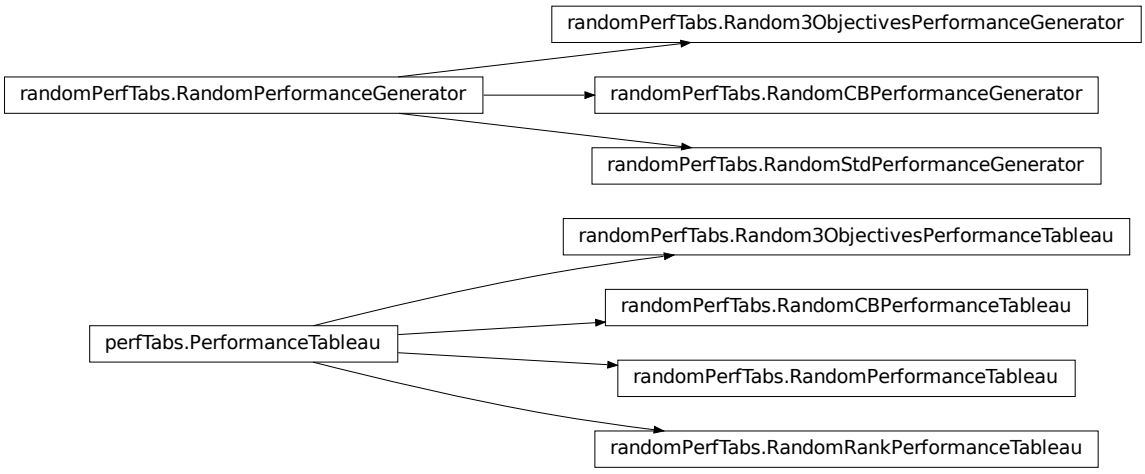


Various Random generators

- *randomDigraphs module* Various implemented random digraph models.



- *randomPerfTabs module* Various implemented random performance tableau models.



- *randomNumbers module* Additional random number generators, not available in the standard python random.py library.

Handling big data

- *performanceQuantiles module* Incremental representation of large performance tableaux via binned cumulated density functions per criteria. Depends on the `randomPerfTabs` module.

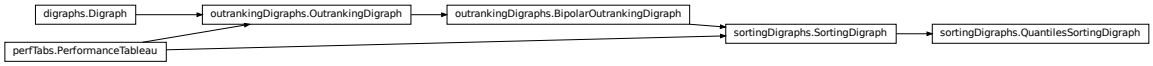
- *sparseOutrankingDigraphs module* Sparse implementation design for large bipolar outranking digraphs (order > 1000);

Cythonized modules

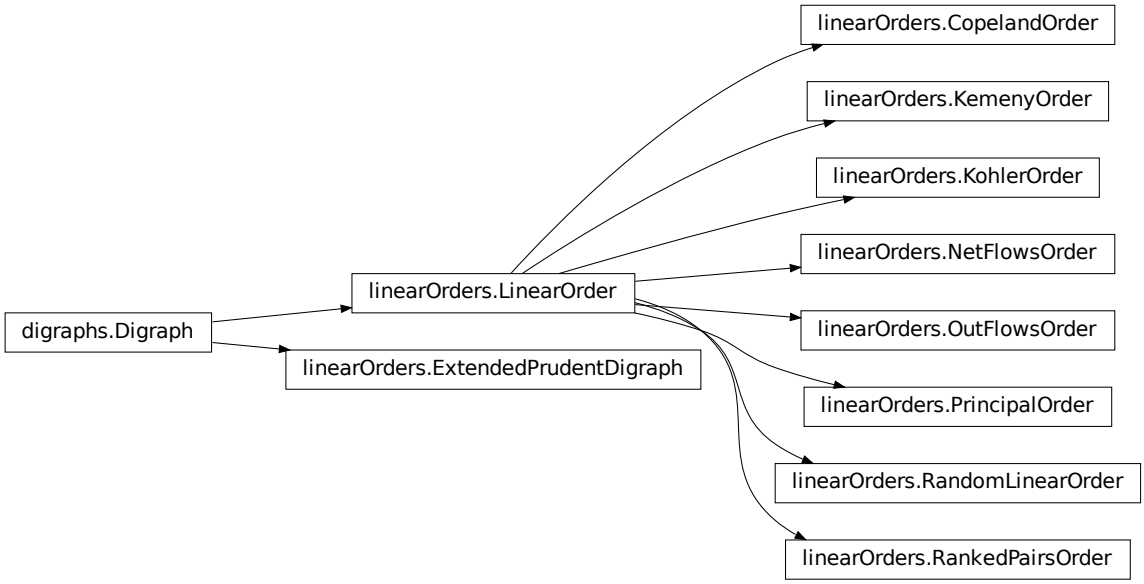
- *Cythonized modules for big digraphs* Cythonized C implementation for handling big performance tableaux and bipolar outranking digraphs (order > 1000).

Sorting, rating and ranking tools

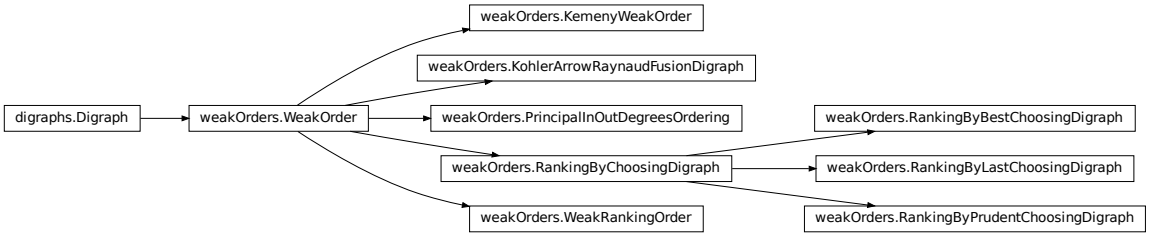
- *sortingDigraphs module* Additional tools for solving sorting problems with the main SortingDigraph class;



- *linearOrders module* Additional tools for solving linearly ranking problems with the root LinearOrder class;



- *weakOrders module* Additional tools for solving pre-ranking problems with root WeakOrder class.



Miscellaneous tools

- *digraphsTools module* Various generic methods and tools for handling digraphs.
- *xmcda module* Methods and tools for handling XMCDA encoded performance tableaux and digraphs.
- *arithmetics module* Some common methods and tools for computing with integer numbers.

Developping an outranking digraphs based decision support methodology is an ongoing research project of Raymond Bisdorff <https://leopold-loewenheim.uni.lu/bisdorff/>, University of Luxembourg.

2.2.3 digraphs module

A tutorial with coding examples is available here: *Working with the Digraph3 software resources*

Python3+ implementation of the digraphs module, root module of the Digraph3 resources.

class digraphs.**AsymmetricPartialDigraph**(*digraph*)

Bases: *digraphs.Digraph*

Renders the asymmetric part of a Digraph instance.

Note:

- The non asymmetric and the reflexive links are all put to the median indeterminate characteristic value!
- The constructor makes a deep copy of the given Digraph instance!

---

class digraphs.**BreakAddCocsDigraph**(*digraph=None*, *Cpp=False*, *Piping=False*, *Comments=False*, *Threading=False*, *nbrOfCPUs=1*)

Bases: *digraphs.Digraph*

Specialization of general Digraph class for instantiation of chordless odd circuits augmented digraphs.

Parameters:

- digraph: Stored or memory resident digraph instance.
- Cpp: using a C++/Agrum version of the Digraph.computeChordlessCircuits() method.
- Piping: using OS pipes for data in- and output between Python and C++.

A chordless odd circuit is added if the cumulated credibility of the circuit supporting arcs is larger or equal to the cumulated credibility of the converse arcs. Otherwise, the circuit is broken at the weakest asymmetric link, i.e. a link (x, y) with minimal difference between r(x S y) - r(y S x).

addCircuits(*Comments=False*)

Augmenting self with self.circuits.

closureChordlessOddCircuits(*Cpp=False*, *Piping=False*, *Comments=True*, *Debug=False*, *Threading=False*, *nbrOfCPUs=1*)

Closure of chordless odd circuits extraction.

showCircuits(*credibility=None*, *Debug=False*)

show methods for chordless odd circuits in CocaGraph

showComponents()

Shows the list of connected components of the digraph instance.

class digraphs.**BrokenCocsDigraph**(*digraph=None*, *Cpp=False*, *Piping=False*, *Comments=False*, *Threading=False*, *nbrOfCPUs=1*)

Bases: *digraphs.Digraph*

Specialization of general Digraph class for instantiation of chordless odd circuits broken digraphs.

Parameters:

- digraph: stored or memory resident digraph instance.
- Cpp: using a C++/Agrum version of the Digraph.computeChordlessCircuits() method.
- Piping: using OS pipes for data in- and output between Python and C++.

All chordless odd circuits are broken at the weakest asymmetric link, i.e. a link (x, y) with minimal difference between $r(xSy)$ and $r(ySx)$.

breakChordlessOddCircuits(*Cpp=False*, *Piping=False*, *Comments=True*, *Debug=False*, *Threading=False*, *nbrOfCPUs=1*)

Breaking of chordless odd circuits extraction.

breakCircuits(*Comments=False*)

Break all cricuits in self.circuits.

showComponents()

Shows the list of connected components of the digraph instance.

class digraphs.**CSVDigraph**(*fileName='temp'*, *valuationMin=-1*, *valuationMax=1*)

Bases: *digraphs.Digraph*

Specialization of the general Digraph class for reading stored csv formatted digraphs. Using the inbuilt module csv.

Param: fileName (without the extension .csv).

showAll()

class digraphs.**CirculantDigraph**(*order=7, valuationdomain={'max': Decimal('1.0'), 'min': Decimal('-1.0')}, circulants=[-1, 1]*)

Bases: *digraphs.Digraph*

Specialization of the general Digraph class for generating temporary circulant digraphs.

Parameters:

order > 0;
valuationdomain ={'min':m, 'max':M};
circulant connections = list of positive and/or negative circular shifts of value 1 to n.

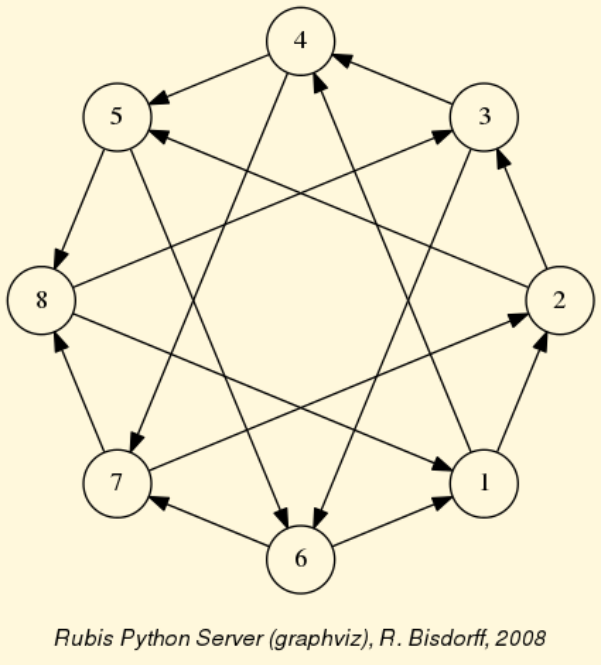
Default instantiation C_7:

order = 7,
valuationdomain = {'min':-1.0,'max':1.0},
circulants = [-1,1].

Example session:

```
>>> from digraphs import CirculantDigraph
>>> c8 = CirculantDigraph(order=8,circulants=[1,3])
>>> c8.exportGraphViz('c8')
*---- exporting a dot file dor GraphViz tools ---------*
Exporting to c8.dot
circo -Tpng c8.dot -o c8.png
# see below the graphviz drawing
>>> c8.showChordlessCircuits()
```

```
No circuits yet computed. Run computeChordlessCircuits()!
>>> c8.computeChordlessCircuits()
...
>>> c8.showChordlessCircuits()
*---- Chordless circuits ----*
['1', '4', '7', '8'] , credibility : 1.0
['1', '4', '5', '6'] , credibility : 1.0
['1', '4', '5', '8'] , credibility : 1.0
['1', '2', '3', '6'] , credibility : 1.0
['1', '2', '5', '6'] , credibility : 1.0
['1', '2', '5', '8'] , credibility : 1.0
['2', '3', '6', '7'] , credibility : 1.0
['2', '3', '4', '7'] , credibility : 1.0
['2', '5', '6', '7'] , credibility : 1.0
['3', '6', '7', '8'] , credibility : 1.0
['3', '4', '7', '8'] , credibility : 1.0
['3', '4', '5', '8'] , credibility : 1.0
12 circuits.
>>>
```



Rubis Python Server (graphviz), R. Bisdorff, 2008

showShort()

class digraphs.**CoDualDigraph**(*other*, *Debug=False*)

Bases: *digraphs.Digraph*

Instantiates the associated codual -converse of the negation- from a deep copy of a given Digraph instance called *other*.

Note: Instantiates *self* as other.__class__ ! And, deepcopies, the case given, the other.description, the other.criteria and the other.evaluation dictionaries into self.

---

class digraphs.**CocaDigraph**(*digraph=None*, *Cpp=False*, *Piping=False*, *Comments=False*)

Bases: *digraphs.Digraph*

Old CocaDigraph class without circuit breakings; all circuits and circuits of circuits are added as hyper-nodes.

> **Warning:** May sometimes give inconsistent results when an autranking digraph shows loads of chordless cuircuits. It is recommended in this case to use instead either the BrokenCocsDigraph class (preferred option) or the BreakAddCocsDigraph class.

Parameters:

- digraph: Stored or memory resident digraph instance.
- Cpp: using a C++/Agrum version of the Digraph.computeChordlessCircuits() method.
- Piping: using OS pipes for data in- and output between Python and C++.

Specialization of general Digraph class for instantiation of chordless odd circuits augmented digraphs.

addCircuits(*Comments=False*)
Augmenting self with self.circuits.

closureChordlessOddCircuits(*Cpp=False*, *Piping=False*, *Comments=False*)
Closure of chordless odd circuits extraction.

showCircuits(*credibility=None*)
show methods for chordless odd circuits in CocaGraph

showComponents()
Shows the list of connected components of the digraph instance.

class digraphs.**CompleteDigraph**(*order=5*, *valuationdomain=(-1.0*, *1.0)*)
Bases: *digraphs.Digraph*

Specialization of the general Digraph class for generating temporary complete graphs of order 5 in {-1,0,1} by default.

Parameters: order > 0; valuationdomain=(Min,Max).

class digraphs.**ConverseDigraph**(*other*)
Bases: *digraphs.Digraph*

Instantiates the associated converse or reciprocal version from a deep copy of a given Digraph called other.

Instantiates as other.__class__ !

Depp copies the case given the description, the criteria and the evaluation dictionary into self.

class digraphs.**CoverDigraph**(*other*, *Debug=False*)
Bases: *digraphs.Digraph*

Instantiates the associated cover relation -immediate neighbours- from a deep copy of a given Digraph called *other*. The Hasse diagram for instance is the cover relation of a transitive digraph.

Note: Instantiates as other.__class__ ! Copies the case given the other.description, the other.criteria and the other.evaluation dictionaries into self.

---

class digraphs.**Digraph**(*file=None*, *order=7*)
Bases: object

Genuine root class of all Digraph3 modules. See tutorial working with the digraphs module

All instances of the *digraphs.Digraph* class contain at least the following components:

1. A collection of digraph nodes called **actions** (decision alternatives): a list, set or (ordered) dictionary of nodes with 'name' and 'shortname' attributes,
2. A logical characteristic **valuationdomain**, a dictionary with three decimal entries: the minimum (-1.0, means certainly false), the median (0.0, means missing information) and the maximum characteristic value (+1.0, means certainly true),
3. The digraph **relation** : a double dictionary indexed by an oriented pair of actions (nodes) and carrying a characteristic value in the range of the previous valuation domain,
4. Its associated **gamma function** : a dictionary containing the direct successors, respectively predecessors of each action, automatically added by the object constructor,
5. Its associated **notGamma function** : a dictionary containing the actions that are not direct successors respectively predecessors of each action, automatically added by the object constructor.

A previously stored *digraphs.Digraph* instance may be reloaded with the *file* argument:

```
>>> from randomDigraphs import RandomValuationDigraph
>>> dg = RandomValuationDigraph(order=3,Normalized=True,seed=1)
>>> dg.save('testdigraph')
Saving digraph in file: <testdigraph.py>
>>> from digraphs import Digraph
>>> dg = Digraph(file='testdigraph') # without the .py extenseion
>>> dg.__dict__
{'name': 'testdigraph',
'actions': {'a1': {'name': 'random decision action', 'shortName': 'a1'},
            'a2': {'name': 'random decision action', 'shortName': 'a2'},
            'a3': {'name': 'random decision action', 'shortName': 'a3'}},
'valuationdomain': {'min': Decimal('-1.0'), 'med': Decimal('0.0'),
                        'max': Decimal('1.0'), 'hasIntegerValuation': False,},
'relation': {'a1': {'a1': Decimal('0.0'), 'a2': Decimal('-0.66'), 'a3': Decimal(
↪'0.44')},
             'a2': {'a1': Decimal('0.94'), 'a2': Decimal('0.0'), 'a3': Decimal('-
↪0.84')},
             'a3': {'a1': Decimal('-0.36'), 'a2': Decimal('-0.70'), 'a3': Decimal(
↪'0.0')}},
'order': 3,
'gamma': {'a1': ({'a3'}, {'a2'}), 'a2': ({'a1'}, set()), 'a3': (set(), {'a1'})},
'notGamma': {'a1': ({'a2'}, {'a3'}),
             'a2': ({'a3'}, {'a1', 'a3'}),
             'a3': ({'a1', 'a2'}, {'a2'})}}
>>>
```

MISgen(*S*, *I*)

generator of maximal independent choices (voir Byskov 2004):

- S ::= remaining nodes;
- I ::= current independent choice

Note: Inititalize: self.MISgen(self.actions.copy(),set())

---

absirred(*choice*)
Renders the crips -irredundance degree of a choice.

absirredundant(*U*)
Generates all -irredundant choices of a digraph.

absirredval(*choice*, *relation*)
Renders the valued -irredundance degree of a choice.

absirredx(*choice*, *x*)
Computes the crips -irredundance degree of node x in a choice.

abskernelrestrict(*choice*)
Parameter: prekernel Renders absorbent prekernel restricted relation.

absorb(*choice*)
Renders the absorbency degree of a choice.

absorbentChoices(*S*)
Generates all minimal absorbent choices of a bipolar valued digraph.

agglomerationDistribution()
Output: aggloCoeffDistribution, meanCoeff Renders the distribution of agglomeration coefficients.

aneighbors(*node*)
Renders the set of absorbed in-neighbors of a node.

automorphismGenerators()
Adds automorphism group generators to the digraph instance.

Note: Dependency: Uses the dreadnaut command from the nauty software package. See https://www3.cs.stonybrook.edu/~algorith/implement/nauty/implement.shtml

On Linux: ... $ sudo apt-get install nauty

---

averageCoveringIndex(*choice*, *direction='out'*)
Renders the average covering index of a given choice in a set of objects, ie the average number of choice members that cover each non selected object.

bestRanks()
renders best possible ranks from indegrees account

bipolarKCorrelation(*digraph*, *Debug=False*)
Renders the bipolar Kendall correlation between two bipolar valued digraphs computed from the average valuation of the XORDigraph(self,digraph) instance.

Warning: Obsolete! Is replaced by the self.computeBipolarCorrelation(other) Digraph method

bipolarKDistance(*digraph*, *Debug=False*)
Renders the bipolar crisp Kendall distance between two bipolar valued digraphs.

Warning: Obsolete! Is replaced by the self.computeBipolarCorrelation(other, MedianCut=True) Digraph method

chordlessPaths(*Pk*, *n2*, *Odd=False*, *Comments=False*, *Debug=False*)
New procedure from Agrum study April 2009 recursive chordless path extraction strating from path Pk = [n2, ..., n1] and ending in node n2. Optimized with marking of visited chordless P1s.

circuitAverageCredibility(*circ*)
Renders the average linking credibility of a COC.

circuitCredibilities(*circuit*, *Debug=False*)
Renders the average linking credibilities and the minimal link of a COC.

circuitMaxCredibility(*circ*)
Renders the minimal linking credibility of a COC.

circuitMinCredibility(*circ*)
Renders the minimal linking credibility of a COC.

closeSymmetric()
Produces the symmetric closure of self.relation.

closeTransitive(*Irreflexive=True*, *Reverse=False*)
Produces the transitive closure of self.relation.

collectcomps(*x*, *A*, *ncomp*)
Recursive subroutine of the components method.

components()
Renders the list of connected components.

computeAllDensities(*choice=None*)
parameter: choice in self renders six densitiy parameters: arc density, double arc density, single arc density, strict single arc density, absence arc density, strict absence arc densitiy.

computeArrowRaynaudOrder()
Renders a linear ordering from worst to best of the actions following Arrow&Raynaud's rule.

computeArrowRaynaudRanking()
renders a linear ranking from best to worst of the actions following Arrow&Raynaud's rule.

computeAverageValuation()
Computes the bipolar average correlation between self and the crisp complete digraph of same order of the irreflexive and determined arcs of the digraph

computeBadChoices(*Comments=False*)
Computes characteristic values for potentially bad choices.

Note: Returns a tuple with following content:

[(0)-determ,(1)degirred,(2)degi,(3)degd,(4)dega,(5)str(choice),(6)absvec]

---

computeBadPirlotChoices(*Comments=False*)
Characteristic values for potentially bad choices using the Pirlot's fixpoint algorithm.

computeBipolarCorrelation(*other*, *MedianCut=False*, *filterRelation=None*, *Debug=False*)
obsolete: dummy replacement for Digraph.computeOrdinalCorrelation method

computeChordlessCircuits(*Odd=False*, *Comments=False*, *Debug=False*)
Renders the set of all chordless odd circuits detected in a digraph. Result (possible empty list) stored in <self.circuitsList> holding a possibly empty list tuples with at position 0 the list of adjacent actions of the circuit and at position 1 the set of actions in the stored circuit.

computeChordlessCircuitsMP(*Odd=False*, *Threading=False*, *nbrOfCPUs=None*, *Comments=False*, *Debug=False*)
Multiprocessing version of computeChordlessCircuits().

Renders the set of all chordless odd circuits detected in a digraph. Result (possible empty list) stored in <self.circuitsList> holding a possibly empty list tuples with at position 0 the list of adjacent actions of the circuit and at position 1 the set of actions in the stored circuit. Inspired by Dias, Castonguay, Longo, Jradi, Algorithmica (2015).

Returns a possibly empty list of tuples (circuit,frozenset(circuit)).

If Odd == True, only circuits of odd length are retained in the result.

computeCoSize()
Renders the number of non validated non reflexive arcs

computeConcentrationIndex(*X*, *N*)
Renders the Gini concentration index of the X serie. N contains the partial frequencies. Based on the triangle summation formula.

computeConcentrationIndexTrapez(*X*, *N*)
Renders the Gini concentration index of the X serie. N contains the partial frequencies. Based on the triangles summation formula.

computeCondorcetLoosers()
Wrapper for condorcetLoosers().

computeCondorcetWinners()
Wrapper for condorcetWinners().

computeCopelandRanking()
Renders a linear ranking from best to worst of the actions following Copelands's rule.

computeCppChordlessCircuits(*Odd=False*, *Debug=False*)
python wrapper for the C++/Agrum based chordless circuits enumeration exchange arguments with external temporary files

computeCppInOutPipingChordlessCircuits(*Odd=False*, *Debug=False*)
python wrapper for the C++/Agrum based chordless circuits enumeration exchange arguments with external temporary files

computeCutLevelDensities(*choice*, *level*)
parameter: choice in self, robustness level renders three robust densitiy parameters: robust double arc density, robust single arc density, robust absence arc densitiy.

computeDensities(*choice*)
parameter: choice in self renders the four densitiy parameters: arc density, double arc density, single arc density, absence arc density.

computeDeterminateness()
Computes the Kendalll distance in % of self with the all median valued (indeterminate) digraph.

computeGoodChoiceVector(*ker*, *Comments=False*)

Characteristic values for potentially good choices.
[(0)-determ,(1)degirred,(2)degi,(3)degd,(4)dega,(5)str(choice),(6)domvec]

computeGoodChoices(*Comments=False*)
Computes characteristic values for potentially good choices.

..note:

```
Return a tuple with following content:

[(0)-determ,(1)degirred,(2)degi,(3)degd,(4)dega,(5)str(choice),(6)domvec,(7)cover]
```

computeGoodPirlotChoices(*Comments=False*)
Characteristic values for potentially good choices using the Pirlot fixpoint algorithm.

computeKemenyIndex(*otherRelation*)
renders the Kemeny index of the self.relation compared with a given crisp valued relation of a compatible other digraph (same nodes or actions).

computeKemenyOrder(*orderLimit=7*, *Debug=False*)
Renders a ordering from worst to best of the actions with maximal Kemeny index. Return a tuple: kemenyOrder (from worst to best), kemenyIndex

computeKemenyRanking(*isProbabilistic=False*, *orderLimit=7*, *seed=None*, *sampleSize=1000*, *Debug=False*)
Renders a ordering from worst to best of the actions with maximal Kemeny index.

Note: Returns a tuple: kemenyRanking (from best to worst), kemenyIndex.

---

computeKohlerOrder()

computeKohlerRanking()

computeMeanInDegree()
Renders the mean indegree of self. !!! self.size must be set previously !!!

computeMeanOutDegree()
Renders the mean degree of self. !!! self.size must be set previously !!!

computeMeanSymDegree()
Renders the mean degree of self. !!! self.size must be set previously !!!

computeMedianOutDegree()
Renders the median outdegree of self. !!! self.size must be set previously !!!

computeMedianSymDegree()
Renders the median symmetric degree of self. !!! self.size must be set previously !!!

computeMoreOrLessUnrelatedPairs()
Renders a list of more or less unrelated pairs.

computeNetFlowsOrder()
Renders an ordered list (from worst to best) of the actions following the net flows ranking rule.

computeNetFlowsRanking()
Renders an ordered list (from best to worst) of the actions following the net flows ranking rule.

computeODistance(*op2*, *comments=False*)
renders the squared normalized distance of two digraph valuations.

Note: op2 = digraphs of same order as self.

---

computeOrbit(*choice*, *withListing=False*)
renders the set of isomorph copies of a choice following the automorphism of the digraph self

computeOrderCorrelation(*order*, *Debug=False*)
Renders the ordinal correlation K of a digraph instance when compared with a given linear order (from worst to best) of its actions

K = sum_{x != y} [min(max(-self.relation(x,y)),other.relation(x,y), max(self.relation(x,y),-other.relation(x,y))]

K /= sum_{x!=y} [min(abs(self.relation(x,y),abs(other.relation(x,y))]

Note: Renders a dictionary with the key 'correlation' containing the actual bipolar correlation index and the key 'determination' containing the minimal determination level D of self and the other relation.

D = sum_{x != y} min(abs(self.relation(x,y)),abs(other.relation(x,y)) / n(n-1)

where n is the number of actions considered.

The correlation index with a completely indeterminate relation is by convention 0.0 at determination level 0.0 .

Warning: self must be a normalized outranking digraph instance !

computeOrdinalCorrelation(*other*, *MedianCut=False*, *filterRelation=None*, *Debug=False*)

Renders the bipolar correlation K of a self.relation when compared with a given compatible (same actions set)) digraph or a [-1,1] valued compatible relation (same actions set).

If MedianCut=True, the correlation is computed on the median polarized relations.

If filterRelation != None, the correlation is computed on the partial domain corresponding to the determined part of the filter relation.

Warning: Notice that the 'other' relation and/or the 'filterRelation', the case given, must both be normalized, ie [-1,1]-valued !

K = sum_{x != y} [min(max(-self.relation[x][y]),other.relation[x][y]), max(self.relation[x][y],-other.relation[x][y])]

K /= sum_{x!=y} [min(abs(self.relation[x][y]),abs(other.relation[x][y]))]

Note: Renders a tuple with at position 0 the actual bipolar correlation index and in position 1 the minimal determination level D of self and the other relation.

D = sum_{x != y} min(abs(self.relation[x][y]),abs(other.relation[x][y])) / n(n-1)

where n is the number of actions considered.

The correlation index with a completely indeterminate relation is by convention 0.0 at determination level 0.0 .

computeOrdinalCorrelationMP(*other*, *MedianCut=False*, *Threading=True*, *nbrOfCPUs=None*, *Comments=False*, *Debug=False*)

Multi processing version of the digraphs.computeOrdinalCorrelation() method.

Note: The relation filtering and the MedinaCut option are not implemented in the MP version.

computePairwiseClusterComparison(*K1*, *K2*, *Debug=False*)

Computes the pairwise cluster comparison credibility vector from bipolar-valued digraph g. with K1 and K2 disjoint lists of action keys from g actions disctionary. Returns the dictionary {'I': Decimal(),'P+':Decimal(),'P-':Decimal(),'R' :Decimal()} where one and only one item is strictly positive.

computePreKernels()

computing dominant and absorbent preKernels: Result in self.dompreKernels and self.abspreKernels

computePreRankingRelation(*preRanking*, *Normalized=True*, *Debug=False*)
Renders the bipolar-valued relation obtained from a given preRanking in decreasing levels (list of lists) result.

computePreorderRelation(*preorder*, *Normalized=True*, *Debug=False*)
Renders the bipolar-valued relation obtained from a given preordering in increasing levels (list of lists) result.

computePrincipalOrder(*plotFileName=None*, *Colwise=False*, *imageType=None*, *tempDir=None*, *Comments=False*, *Debug=False*)
Renders a ordered list of self.actions using the decreasing scores from the first rincipal eigenvector of the covariance of the valued outdegrees of self.

Note: The method, relying on writing and reading temporary files by default in a temporary directory is threading and multiprocessing safe ! (see Digraph.exportPrincipalImage method)

---

computePrudentBetaLevel(*Debug=False*)
computes alpha, ie the lowest valuation level, for which the bipolarly polarised digraph doesn't contain a chordless circuit.

computeRankedPairsOrder(*Cpp=False*, *Debug=False*)
Renders an actions ordering from the worst to the best obtained from the ranked pairs rule.

computeRankedPairsRanking()
Renders an actions ordering from the best to the worst obtained from the ranked pairs rule.

computeRankingByBestChoosing(*CoDual=False*, *CppAgrum=False*, *Debug=False*)
Computes a weak preordering of the self.actions by recursive best choice elagations.

Stores in self.rankingByBestChoosing['result'] a list of (P+,bestChoice) tuples where P+ gives the best choice complement outranking average valuation via the computePairwiseClusterComparison method.

If self.rankingByBestChoosing['CoDual'] is True, the ranking-by-choosing was computed on the codual of self.

computeRankingByBestChoosingRelation(*rankingByBestChoosing=None*, *Debug=False*)
Renders the bipolar-valued relation obtained from the self.rankingByBestChoosing result.

computeRankingByChoosing(*actionsSubset=None*, *CppAgrum=False*, *Debug=False*, *CoDual=False*)
Computes a weak preordring of the self.actions by iterating jointly best and worst choice elagations.

Stores in self.rankingByChoosing['result'] a list of ((P+,bestChoice),(P-,worstChoice)) pairs where P+ (resp. P-) gives the best (resp. worst) choice complement outranking (resp. outranked) average valuation via the computePairwiseClusterComparison method.

If self.rankingByChoosing['CoDual'] is True, the ranking-by-choosing was computed on the codual of self.

computeRankingByChoosingRelation(*rankingByChoosing=None*, *actionsSubset=None*, *Debug=False*)
Renders the bipolar-valued relation obtained from the self.rankingByChoosing result.

computeRankingByLastChoosing(*CoDual=False*, *CppAgrum=False*, *Debug=False*)
Computes a weak preordring of the self.actions by iterating worst choice elagations.

Stores in self.rankingByLastChoosing['result'] a list of (P-,worstChoice) pairs where P- gives the worst choice complement outranked average valuation via the computePairwiseClusterComparison method.

If self.rankingByChoosing['CoDual'] is True, the ranking-by-last-chossing was computed on the codual of self.

computeRankingByLastChoosingRelation(*rankingByLastChoosing=None*, *Debug=False*)
Renders the bipolar-valued relation obtained from the self.rankingByLastChoosing result.

computeRankingCorrelation(*ranking*, *Debug=False*)
Renders the ordinal correlation K of a digraph instance when compared with a given linear ranking of its actions

K = sum_{x != y} [min(max(-self.relation(x,y)),other.relation(x,y), max(self.relation(x,y),-other.relation(x,y))]

K /= sum_{x!=y} [min(abs(self.relation(x,y),abs(other.relation(x,y))]

Note: Renders a tuple with at position 0 the actual bipolar correlation index and in position 1 the minimal determination level D of self and the other relation.

D = sum_{x != y} min(abs(self.relation(x,y)),abs(other.relation(x,y)) / n(n-1)

where n is the number of actions considered.

The correlation index with a completely indeterminate relation is by convention 0.0 at determination level 0.0 .

---

computeRelationalStructure(*Debug=False*)
Renders the counted decomposition of the valued relations into the following type of links: gt '>', eq '=', lt '<', incomp '<>', leq '<=', geq '>=', indeterm '?'

computeRubisChoice(*CppAgrum=False*, *Comments=False*, *_OldCoca=False*, *BrokenCocs=True*, *Threading=False*, *nbrOfCPUs=1*)
Renders self.strictGoodChoices, self.nullChoices self.strictBadChoices, self.nonRobustChoices.

CppgArum = False (default | true : use C++/Agrum digraph library for computing chordless circuits in self.

Warning: Changes in site the outranking digraph by adding or braking chordless odd outranking circuits.

computeRubyChoice(*CppAgrum=False*, *Comments=False*, *_OldCoca=False*)
dummy for computeRubisChoice() old versions compatibility.

computeSingletonRanking(*Comments=False*, *Debug=False*)
Renders the sorted bipolar net determinatation of outrankingness minus outrankedness credibilities of all singleton choices.

res = ((netdet,singleton,dom,absorb)+)

computeSize()
Renders the number of validated non reflexive arcs

computeSizeTransitiveClosure()
Renders the size of the transitive closure of a digraph.

computeSlaterOrder(*isProbabilistic=False*, *seed=None*, *sampleSize=1000*, *Debug=False*)
Reversed return from computeSlaterRanking method.

computeSlaterRanking(*isProbabilistic=False*, *seed=None*, *sampleSize=1000*, *Debug=False*)
Renders a ranking of the actions with minimal Slater index. Return a tuple: slaterOrder, slaterIndex

computeTransitivityDegree()
Renders the transitivity degree of a digraph.

computeUnrelatedPairs()
Renders a list of more or less unrelated pairs.

computeValuationLevels(*choice=None*, *Debug=False*)
renders the symmetric closure of the apparent valuations levels of self in an increasingly ordered list. If parameter choice is given, the computation is limited to the actions of the choice.

computeValuationPercentages(*choice*, *percentiles*, *withValues=False*)
Parameters: choice and list of percentages. renders a series of quantiles of the characteristics valuation of the arcs in the digraph.

computeValuationPercentiles(*choice*, *percentages*, *withValues=False*)
Parameters: choice and list of percentages. renders a series of quantiles of the characteristics valuation of the arcs in the digraph.

computeValuationStatistics(*Sampling=False*, *Comments=False*)
Renders the mean and variance of the valuation of the non reflexive pairs.

computeWeakCondorcetLoosers()
Wrapper for weakCondorcetLoosers().

computeWeakCondorcetWinners()
Wrapper for weakCondorcetWinners().

computeupdown1(*s*, *S*)
Help method for show_MIS_HB2 method. fills self.newmisset, self.upmis, self.downmis.

computeupdown2(*s*, *S*)
Help method for show_MIS_HB1 method. Fills self.newmisset, self.upmis, self.downmis.

computeupdown2irred(*s*, *S*)
Help method for show_MIS_HB1 method. Fills self.newmisset, self.upmis, self.downmis.

condorcetLoosers()
Renders the set of decision actions x such that self.relation[x][y] < self.valuationdomain['med'] for all y != x.

condorcetWinners()
Renders the set of decision actions x such that self.relation[x][y] > self.valuationdomain['med'] for all y != x.

contra(*v*)
Parameter: choice. Renders the negation of a choice v characteristic's vector.

convertRelationToDecimal()
Converts the float valued self.relation in a decimal valued one.

convertValuationToDecimal()
Convert the float valuation limits to Decimals.

coveringIndex(*choice*, *direction='out'*)
Renders the covering index of a given choice in a set of objects, ie the minimum number of choice members that cover each non selected object.

crispKDistance(*digraph*, *Debug=False*)
Renders the crisp Kendall distance between two bipolar valued digraphs.

> **Warning:** Obsolete! Is replaced by the self.computeBipolarCorrelation(other, MedianCut=True) Digraph method

detectChordlessCircuits(*Comments=False*, *Debug=False*)
Detects a chordless circuit in a digraph. Returns a Boolean

detectChordlessPath(*Pk*, *n2*, *Comments=False*, *Debug=False*)
New procedure from Agrum study April 2009 recursive chordless path extraction starting from path Pk = [n2, ..., n1] and ending in node n2. Optimized with marking of visited chordless P1s.

detectCppChordlessCircuits(*Debug=False*)
python wrapper for the C++/Agrum based chordless circuits detection exchange arguments with external temporary files. Returns a boolean value

determinateness(*vec*, *inPercent=True*)
Renders the determinateness of a characteristic vector *vec* = [(r(x),x),(r(y),y), ...] of length *n* in valuationdomain [Min,Med,Max]:

result = sum_x(abs(r(x)-Med)) / (n*(Max-Med))

If inPercent, *result* shifted (+1) and reduced (/2) to [0,1] range.

diameter(*Oriented=False*)
Renders the (by default non-oriented) diameter of the digraph instance

digraph2Graph(*valuationDomain={'max': 1, 'med': 0, 'min': -1}*, *Debug=False*, *conjunctiveConversion=True*)
Convert a Digraph instance to a Graph instance.

dneighbors(*node*)
Renders the set of dominated out-neighbors of a node.

domin(*choice*)
Renders the dominance degree of a choice.

dominantChoices(*S*)
Generates all minimal dominant choices of a bipolar valued digraph.

Note: Initiate with S = self.actions,copy().

---

domirred(*choice*)
Renders the crips +irredundance degree of a choice.

domirredval(*choice*, *relation*)
Renders the valued +irredundance degree of a choice.

domirredx(*choice*, *x*)
Renders the crips +irredundance degree of node x in a choice.

domkernelrestrict(*choice*)
Parameter: prekernel Renders dominant prekernel restricted relation.

exportD3(*fileName='index'*, *Comments=True*)
This function was designed and implemented by Gary Cornelius, 2014 for his bachelor thesis at the University of Luxembourg. The thesis document with more explanations can be found here .

Parameters:

- fileName, name of the generated html file, default = None (graph name as defined in python);
- Comments, True = default;

The idea of the project was to find a way that allows you to easily get details about certain nodes or edges of a directed graph in a dynamic format. Therefore this function allows you to export a html file together with all the needed libraries, including the D3 Library which we use for graph generation and the physics between nodes, which attracts or pushes nodes away from each other.

Features of our graph include i.e.

- A way to only inspect a node and it's neighbours
- Dynamic draging and freezing of the graph
- Export of a newly created general graph

You can find the list of fututres in the Section below which is arranged according to the graph type.

If the graph is an outrankingdigraphs:

- Nodes can be dragged and only the name and comment can be edited.

- Edges can be inspected but not edited for this purpose a special json array containing all possible pairwiseComparisions is generated.

If the graph is a general graph:

- Nodes can be dragged, added, removed and edited.
- Edges can be added, removed, inverted and edited. But edges cannot be inspected.
- The pairwiseComparisions key leads to an empty array {}.

In both cases, undefined edges can be hidden and reappear after a simple reload.(right click - reload)

The generated files:

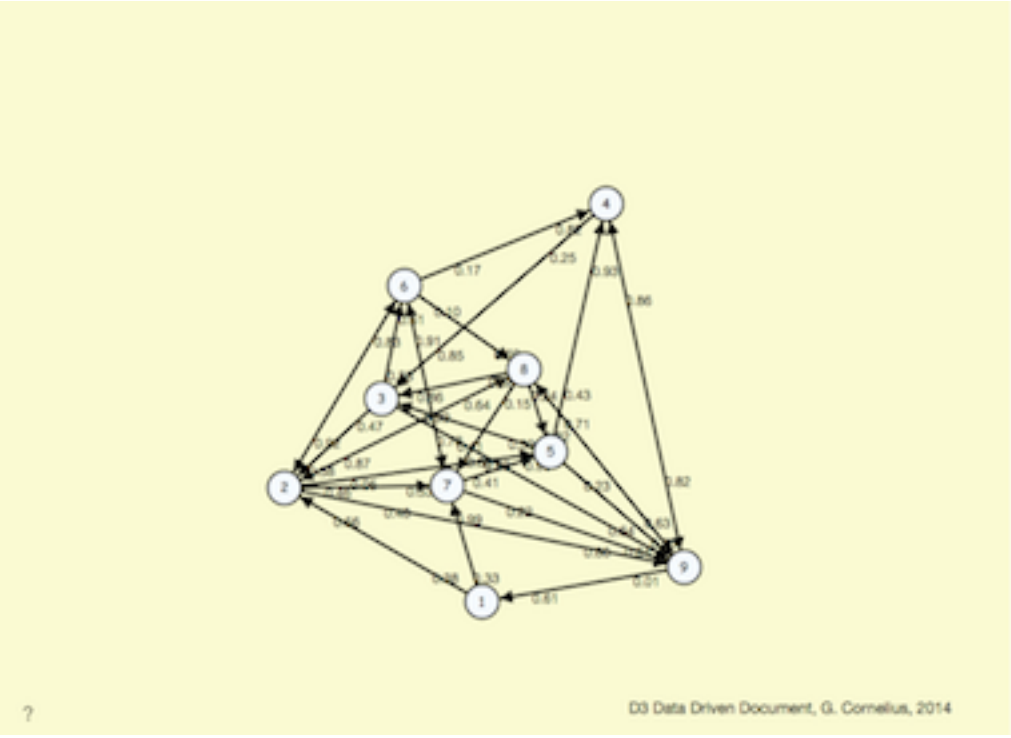
- d3.v3.js, contains the D3 Data-driven Documents source code, containing one small addition that we made in order to be able to easyly import links with a different formatself.
- digraph3lib.js, contains our library. This file contains everything that we need from import of an XMCDA2 file, visualization of the graph to export of the changed graph.
- d3export.json, usually named after the python graph name followed by a ticket number if the file is already present. It is the JSON file that is exported with the format "{"xmcda2": "some xml","pairwiseComparisions":"{"a01": "some html",...}"}.

Example 1:

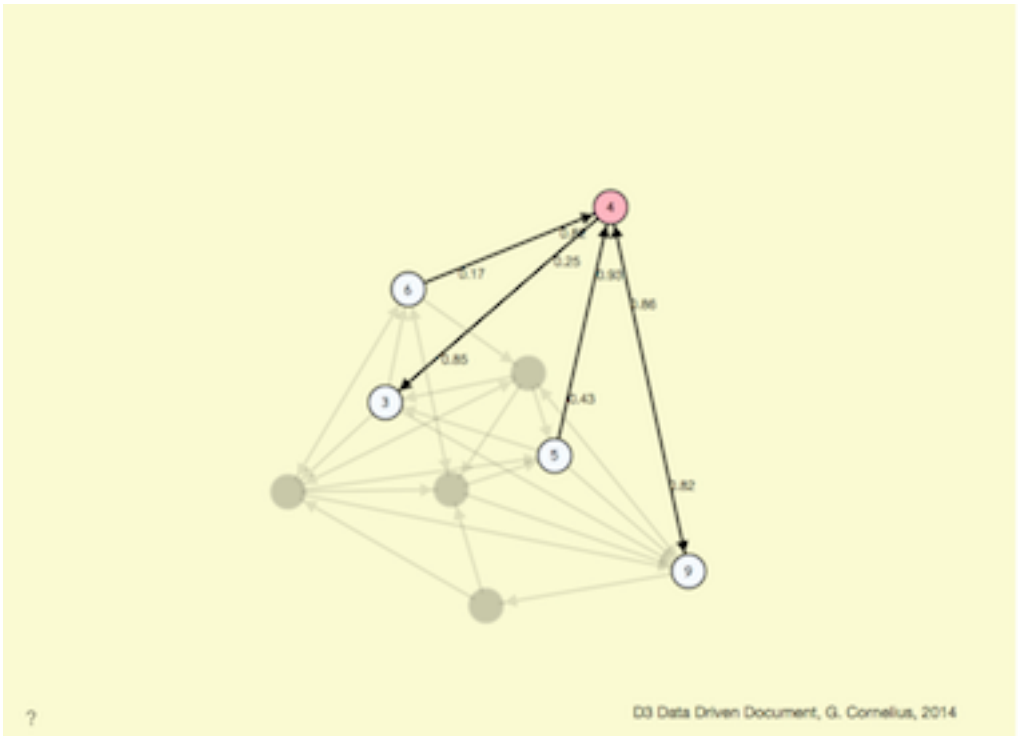
python3 session:

```
>>> from digraphs import RandomValuationDigraph
>>> dg = RandomValuationDigraph(order=5,Normalized=True)
>>> dg.exportD3()
or
>> dg.showInteractiveGraph()
```

Main Screen:



Inspect function:



Note:

If you want to use the automatic load in Chrome, try using the command: "python -m SimpleHTTPServer" and then access the index.html via "http://0.0.0.0:8000/index.html". In order to load the CSS an active internet connection is needed!

exportGraphViz(*fileName=None*, *actionsSubset=None*, *bestChoice=set()*, *worstChoice=set()*, *noSilent=True*, *graphType='png'*, *graphSize='7,7'*, *relation=None*)
export GraphViz dot file for graph drawing filtering.

exportPrincipalImage(*Reduced=False*, *Colwise=False*, *plotFileName=None*, *Type='png'*, *TempDir='.'*, *Comments=False*)
Export as PNG (default) or PDF the principal projection of the valued relation using the three principal eigen vectors.

Warning: The method, writing and reading temporary files: tempCol.r and rotationCol.csv, resp. tempRow.r and rotationRow.csv, by default in the working directory (./), is hence not safe for multiprocessing programs, unless a temporary dirctory is provided

flatChoice(*ch*, *Debug=False*)
Converts set or list ch recursively to a flat list of items.

forcedBestSingleChoice()
Renders the set of most determined outranking singletons in self.

gammaSets()
Renders the dictionary of neighborhoods {node: (dx,ax)} with set *dx* gathering the dominated, and set *ax* gathering the absorbed neighborhood.

generateAbsPreKernels()
Generate all absorbent prekernels from independent choices generator.

generateDomPreKernels()
Generate all dominant prekernels from independent choices generator.

graphDetermination(*Normalized=True*)
Output: average normalized (by default) arc determination:

averageDeterm = (sum_(x,y) [abs(relf-relation[x][y] - Med)] / n) / [(Max-Med) if Normalized],

where Med = self.valuationdomain['med'] and Max = self.valuationdomain['max'].

htmlRelationMap(*tableTitle='Relation Map', relationName='r(x R y)', actionsSubset=None, rankingRule='Copeland', symbols=['+', '·', ' ', '-', '_'], Colored=True, ContentCentered=True*)
renders the relation map in actions X actions html table format.

htmlRelationTable(*tableTitle='Valued Relation Table', relationName='r(x R y)', hasIntegerValues=False, actionsSubset=None, isColored=False*)
renders the relation valuation in actions X actions html table format.

inDegrees()
renders the median cut indegrees

inDegreesDistribution()
Renders the distribution of indegrees.

independentChoices(*U*)
Generator for all independent choices with neighborhoods of a bipolar valued digraph:

Note:

- Initiate with U = self.singletons().
- Yields [(independent choice, domnb, absnb, indnb)].

---

inner_prod(*v1*, *v2*)
Parameters: two choice characteristic vectors Renders the inner product of two characteristic vetors.

intstab(*choice*)
Computes the independence degree of a choice.

irreflex(*mat*)
Puts diagonal entries of mat to valuationdomain['min']

isComplete(*Debug=False*)
checks the completeness property of self.relation by checking for the absence of a link between two actions!!

> **Warning:** The reflexive links are ignored !!

isCyclic(*Debug=False*)
checks the cyclicity of self.relation by checking for a reflexive loop in its transitive closure-

> **Warning:** self.relation is supposed to be irreflexive !

isWeaklyComplete(*Debug=False*)
checks the weakly completeness property of self.relation by checking for the absence of a link between two actions!!

> **Warning:** The reflexive links are ignored !!

iterateRankingByChoosing(*Odd=False*, *CoDual=False*, *Comments=True*, *Debug=False*, *Limited=None*)
Renders a ranking by choosing result when progressively eliminating all chordless (odd only) circuits with rising valuation cut levels.

Parameters CoDual = False (default)/True Limited = proportion (in [0,1]) * (max - med) valuationdomain

kChoices(*A*, *k*)
Renders all choices of length k from set A

matmult2(*m*, *v*)
Parameters: digraph relation and choice characteristic vector matrix multiply vector by inner production

meanDegree()
Renders the mean degree of self. !!! self.size must be set previously !!!

meanLength(*Oriented=False*)
Renders the (by default non-oriented) mean neighbourhoor depth of self. !!! self.order must be set previously !!!

minimalChoices(*S*)
Generates all dominant or absorbent choices of a bipolar valued digraph.

minimalValuationLevelForCircuitsElimination(*Odd=True*, *Debug=False*, *Comments=False*)
renders the minimal valuation level <lambda> that eliminates all self.circuitsList stored odd chordless circuits from self.

> **Warning:** The <lambda> level polarised may still contain newly appearing chordless odd circuits !

neighbourhoodCollection(*Oriented=False*, *Potential=False*)
Renders the neighbourhood.

neighbourhoodDepthDistribution(*Oriented=False*)
Renders the distribtion of neighbourhood depths.

notGammaSets()
Renders the dictionary of neighborhoods {node: (dx,ax)} with set *dx* gathering the not dominated, and set *ax* gathering the not absorbed neighborhood.

notaneighbors(*node*)
Renders the set of absorbed not in-neighbors of a node.

notdneighbors(*node*)
Renders the set of not dominated out-neighbors of a node.

omax(*L*, *Debug=False*)
Epistemic disjunction for bipolar outranking characteristics computation.

omin(*L*, *Debug=False*)
Epistemic conjunction for bipolar outranking characteristics computation.

outDegrees()
renders the median cut outdegrees

outDegreesDistribution()
Renders the distribution of outdegrees.

plusirredundant(*U*)
Generates all +irredundant choices of a digraph.

powerset(*U*)
Generates all subsets of a set.

readPerrinMisset(*file='curd.dat'*)
read method for 0-1-char-coded MISs by default from the perrinMIS.c curd.dat result file.

readabsvector(*x*, *relation*)
Parameter: action x absorbent in vector.

readdomvector(*x*, *relation*)
Parameter: action x dominant out vector.

recodeValuation(*newMin=-1.0*, *newMax=1.0*, *Debug=False*)
Recodes the characteristic valuation domain according to the parameters given.

Note: Default values gives a normalized valuation domain

---

relationFct(*x*, *y*)
wrapper for self.relation dictionary access to ensure interoperability with the sparse and big outranking digraph implementation model.

save(*fileName='tempdigraph'*, *option=None*, *DecimalValuation=True*, *decDigits=2*)
Persistent storage of a Digraph class instance in the form of a python source code file

saveCSV(*fileName='tempdigraph'*, *Normalized=False*, *Dual=False*, *Converse=False*, *Diagonal=False*, *Debug=False*)
Persistent storage of a Digraph class instance in the form of a csv file.

saveXMCDA(*fileName='temp'*, *relationName='R'*, *category='random'*, *subcategory='valued'*, *author='digraphs Module (RB)'*, *reference='saved from Python'*, *valuationType='standard'*, *servingD3=False*)
save digraph in XMCDA format.

saveXMCDA2(*fileName='temp'*, *fileExt='xmcda2'*, *Comments=True*, *relationName='R'*, *relationType='binary'*, *category='random'*, *subcategory='valued'*, *author='digraphs Module (RB)'*, *reference='saved from Python'*, *valuationType='standard'*, *digits=2*, *servingD3=False*)
save digraph in XMCDA format.

saveXML(*name='temp'*, *category='general'*, *subcategory='general'*, *author='digraphs Module (RB)'*, *reference='saved from Python'*)
save digraph in XML format.

savedre(*name='temp'*)
save digraph in nauty format.

sharp(*x*, *y*)
Paramaters: choice characteristic values. Renders the sharpest of two characteristic values x and y.

sharpvec(*v*, *w*)
Paramaters: choice characteristic vectors. Renders the sharpest of two characteristic vectors v and w.

showActions()
presentation methods for digraphs actions

showAll()
Detailed show method for genuine digraphs.

showAutomorphismGenerators()
Renders the generators of the automorphism group.

showBadChoices(*Recompute=True*)
Characteristic values for potentially bad choices.

showChoiceVector(*ch*, *ChoiceVector=True*)
Show procedure for annotated bipolar choices.

showChordlessCircuits()
show methods for (chordless) circuits in a Digraph. Dummy for showCircuits().

showCircuits()
show methods for circuits observed in a Digraph instance.

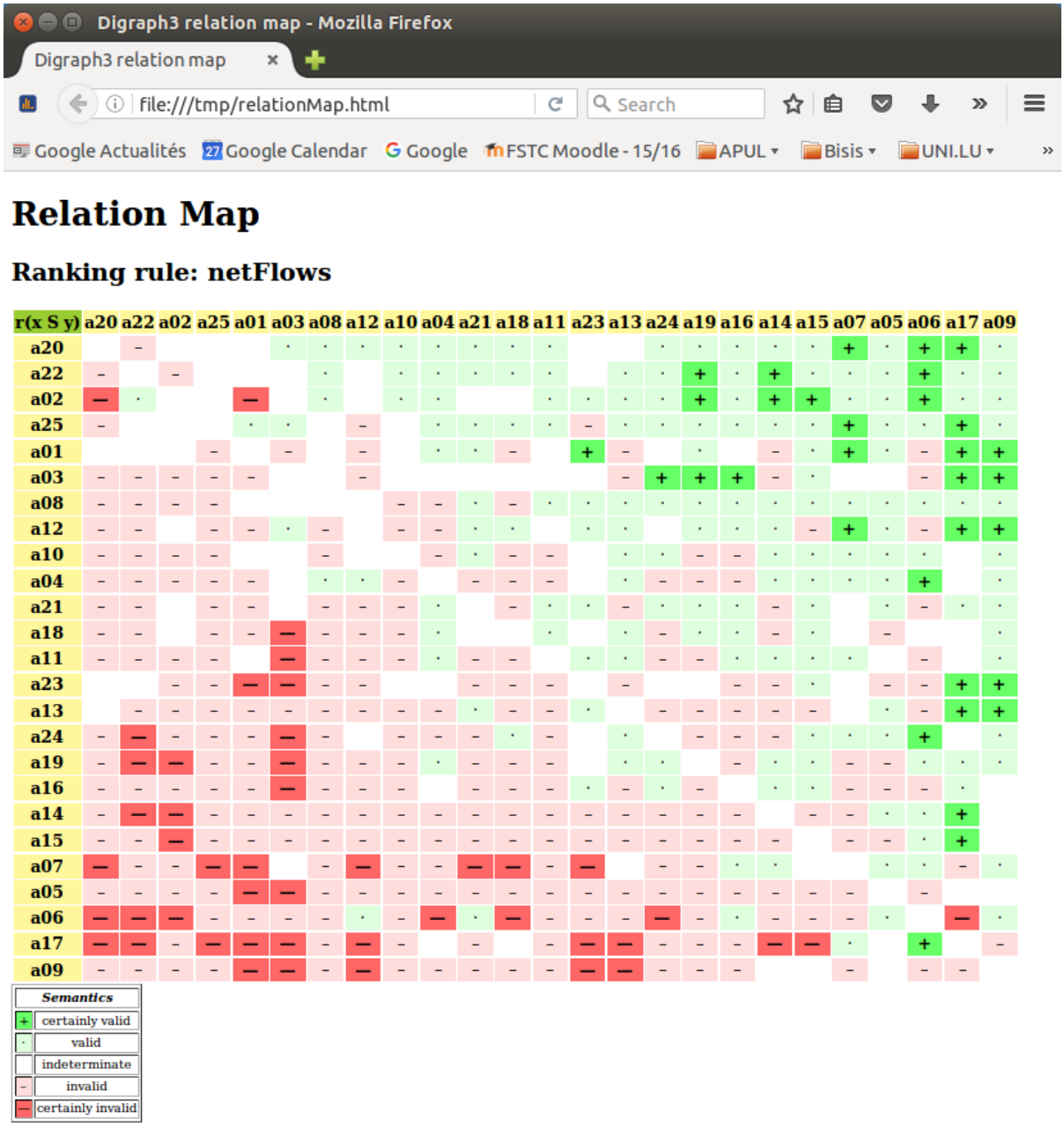
showComponents()
Shows the list of connected components of the digraph instance.

showGoodChoices(*Recompute=True*)
Characteristic values for potentially good choices.

showHTMLRelationMap(*actionsList=None, rankingRule='Copeland', Colored=True, tableTitle='Relation Map', relationName='r(x S y)', symbols=['+', '·', ' ', '–', '&#151']*)
Launches a browser window with the colored relation map of self. See corresponding Digraph.showRelationMap() method.

Example:

```
>>> from outrankingDigraphs import *
>>> t = RandomCBPerformanceTableau(numberOfActions=25,seed=1)
>>> g = BipolarOutrankingDigraph(t,Normalized=True)
>>> gcd = ~(-g)  # strict outranking relation
>>> gcd.showHTMLRelationMap(rankingRule="netFlows")
```



r(x S y)	a20	a22	a02	a25	a01	a03	a08	a12	a10	a04	a21	a18	a11	a23	a13	a24	a19	a16	a14	a15	a07	a05	a06	a17	a09
a20		-				·	·	·	·	·	·	·	·			·	·	·	·	·	+	·	+	+	·
a22	-		-				·		·	·	·	·	·		·	·	+	·	+	·	·	·	+	·	·
a02	—	·			—		·		·	·			·	·	·	·	+	·	+	+	·	·	+	·	·
a25	-				·	·		-		·	·	·	·	-	·	·	·	·	·	·	+	·	·	+	·
a01				-		-		-		·	·	-		+	-		·		-	·	+	·	-	+	+
a03	-	-	-	-	-			-							-	+	+	+	-	·			-	+	+
a08	-	-	-	-					-	-	·	-	·	·	·	·	·	·	·	·	·	·	·	·	·
a12	-	-		-	-	·	-		-	-	·	·		·	·		·	·	·	-	+	·	-	+	+
a10	-	-	-	-			-			-	·	-	-		·	·	-	-	·	·	·	·	·		·
a04	-	-	-	-	-		·	·	-		-	-	-		·	-	-	-	·	·	·	·	+		·
a21	-	-		-	-		-	-	-	·		-	·	·	-	·	·	·	-	·		·	-	·	·
a18	-	-		-	-	—	-	-	-	·			·		·	-	·	·	-	·		-			·
a11	-	-	-	-		—	-	-	-	·	-	-		·	·	-	-	·	·	·	·		-		·
a23			-	-	—	—	-	-			-	-	-		-			-	-	·		-	-	+	+
a13		-	-	-	-	-	-	-	-	-	·	-	-	·		-	-	-	-	-		·	-	+	+
a24	-	—	-	-	-	—	-		-	-	-	·	-		·		-	-	-	·	·	·	+		·
a19	-	—	—	-	-	—	-	-	-	·	-	-	-		·	·		-	·	·	-	-	·	·	·
a16	-	-	-	-	-	—	-	-	-		-	-	-	·	-	·	-		·	·	-	-	-	·	
a14	-	—	—	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-	-	·	·	+	
a15	-	-	—	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-	-	·	+	
a07	—	-	-	—	—		-	—	-	-	—	—	-	—		-	-	·	·			·	·	-	·
a05	-	-	-	-	—	—	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		
a06	—	—	—	-	-	-	-	·	-	—	·	—	-	-	-	—	-	·	-	-	-	·		—	·
a17	—	—	-	—	—	—	-	—	-		-		-	—	—	-	-	-	—	—	·		+		-
a09	-	-	-	-	—	—	-	—	-	-	-	-	-	—	—	-	-	-			-		-	-	

showHTMLRelationTable (*actionsList=None*, *IntegerValues=False*, *Colored=True*, *tableTitle='Valued Adjacency Matrix'*, *relationName='r(x S y)'*)

Launches a browser window with the colored relation table of self.

showInteractiveGraph ()

Save the graph and all needed files for the visualization of an interactive graph generated by the exportD3() function. For best experience make sure to use Firefox, because other browser restrict the loading of local files.

showMIS (*withListing=True*)

Prints all maximal independent choices: Result in self.misset.

showMIS_AH (*withListing=True*)

Prints all MIS using the Hertz method.

Result saved in self.hertzmisset.

showMIS_HB2 (*withListing=True*)
Prints all MIS using the Hertz-Bisdorff method.

Result saved in self.newmisset.

showMIS_RB (*withListing=True*)
Prints all MIS using the Bisdorff method.

Result saved in self.newmisset.

showMIS_UD (*withListing=True*)
Prints all MIS using the Hertz-Bisdorff method.

Result saved in self.newmisset.

showMaxAbsIrred (*withListing=True*)

Computing maximal -irredundant choices: Result in self.absirset.

showMaxDomIrred (*withListing=True*)

Computing maximal +irredundant choices: Result in self.domirset.

showMinAbs (*withListing=True*)

Prints minimal absorbent choices: Result in self.absset.

showMinDom (*withListing=True*)

Prints all minimal dominant choices: Result in self.domset.

showNeighborhoods ()
Lists the gamma and the notGamma function of self.

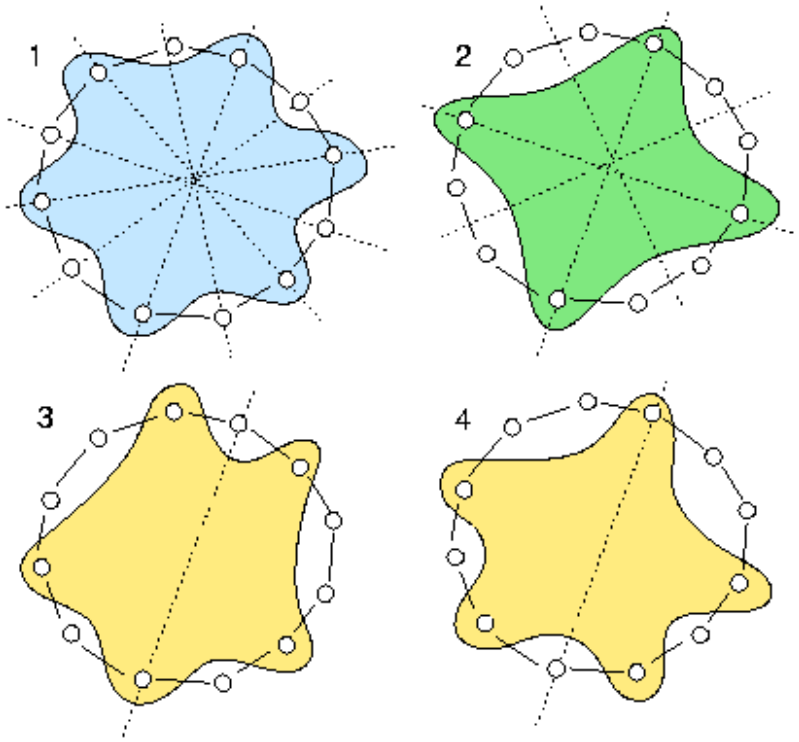
showOrbits (*InChoices*, *withListing=True*)
Prints the orbits of Choices along the automorphisms of the Digraph instance.

Example Python session for computing the non isomorphic MISs from the 12-cycle graph:

```
>>> from digraphs import *
>>> c12 = CirculantDigraph(order=12,circulants=[1,-1])
>>> c12.automorphismGenerators()
...
  Permutations
  {'1': '1', '2': '12', '3': '11', '4': '10', '5':
   '9', '6': '8', '7': '7', '8': '6', '9': '5', '10':
   '4', '11': '3', '12': '2'}
  {'1': '2', '2': '1', '3': '12', '4': '11', '5': '10',
   '6': '9', '7': '8', '8': '7', '9': '6', '10': '5',
   '11': '4', '12': '3'}
  Reflections {}
>>> print('grpsize = ', c12.automorphismGroupSize)
  grpsize = 24
>>> c12.showMIS(withListing=False)
  *---  Maximal independent choices ---*
  number of solutions:  29
  cardinality distribution
  card.:  [0, 1, 2, 3, 4,  5,  6, 7, 8, 9, 10, 11, 12]
  freq.:  [0, 0, 0, 0, 3, 24,  2, 0, 0, 0,  0,  0,  0]
  Results in c12.misset
>>> c12.showOrbits(c12.misset,withListing=False)
...
  *---- Global result ----
  Number of MIS:  29
```

```
Number of orbits :  4
Labelled representatives:
1: ['2','4','6','8','10','12']
2: ['2','5','8','11']
3: ['2','4','6','9','11']
4: ['1','4','7','9','11']
Symmetry vector
stabilizer size: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...]
frequency      : [0, 2, 0, 0, 0, 0, 0, 1, 0,  0,  0,  1, ...]
```

Figure: The symmetry axes of the non isomorphic MISs of the 12-cycle:



Reference: R. Bisdorff and J.L. Marichal (2008). Counting non-isomorphic maximal independent sets of the n-cycle graph. *Journal of Integer Sequences*, Vol. 11 Article 08.5.7 (openly accessible here)

showOrbitsFromFile(*InFile*, *withListing=True*)
Prints the orbits of Choices along the automorphisms of the digraph self by reading in the 0-1 misset file format. See the `digraphs.Digraph.readPerrinMisset()` method.

showPreKernels(*withListing=True*)

Printing dominant and absorbent preKernels: Result in self.dompreKernels and self.abspreKernels

showRankingByBestChoosing(*rankingByBestChoosing=None*)
A show method for self.rankinByBestChoosing result.

> **Warning:** The self.computeRankingByBestChoosing(CoDual=False/True) method instantiating the self.rankingByBestChoosing slot is pre-required !

showRankingByChoosing(*rankingByChoosing=None*)
A show method for self.rankinByChoosing result.

> **Warning:** The self.computeRankingByChoosing(CoDual=False/True) method instantiating the self.rankingByChoosing slot is pre-required !

showRankingByLastChoosing(*rankingByLastChoosing=None*, *Debug=None*)
A show method for self.rankinByChoosing result.

> **Warning:** The self.computeRankingByLastChoosing(CoDual=False/True) method instantiating the self.rankingByChoosing slot is pre-required !

showRelation()
prints the relation valuation in ##.## format.

showRelationMap(*symbols=None*, *rankingRule='Copeland'*)
Prints on the console, in text map format, the location of certainly validated and certainly invalidated outranking situations.

By default, symbols = {'max':'T','positive': '+', 'median': ' ', 'negative': '-', 'min': '_'}

The default ordering of the output is following the Copeland ranking rule from best to worst actions. Further available ranking rules are net flows (rankingRule="netFlows"), Kohler's (rankingRule="kohler"), and Tideman's ranked pairs rule (rankingRule="rankedPairs").

Example:

```
>>> from outrankingDigraphs import *
>>> t = RandomCBPerformanceTableau(numberOfActions=25,seed=1)
>>> g = BipolarOutrankingDigraph(t,Normalized=True)
>>> gcd = ~(-g)  # strict outranking relation
>>> gcd.showRelationMap(rankingRule="netFlows")
 -   ++++++++  +++++T+TT+
- -    + +++++ ++T+T+++T++
_+  _ + ++  ++++T+TT++T++
-   ++ - ++++-++++++T++T+
   - - - ++- T- + -+T+-TT
-----  -      -TTT-+  -TT
----    --+-++++++++++++
-- --+- --++ ++ +++-T+-TT
----  -  -+-- ++--+++++ +
----- ++- --- +---++++T +
-- -- ---+ -++-+++-+ +-++
-- --_---+  + +-++-+ -  +
---- _---+-- ++--++++ - +
  --__--  --- -  --+ --TT
 ---------+--+ ----- +-TT
-_---_- ---+- + ---+++T +
-__--_---+--- ++ -++--+++
-----_--- ---+-+- ++---+
-__--------------- --++T
--_---------------- --+T
_--__ -_--__-_ --++  ++-+
----__--------------- -
___----+-_+_---_-+---+ _+
__-___-_- - -__---__+ T -
----__-_-----__---  - --
Ranking rule: netFlows
>>>
```

showRelationTable(*Sorted=True*, *IntegerValues=False*, *actionsSubset=None*, *relation=None*, *ndigits=2*, *ReflexiveTerms=True*)
prints the relation valuation in actions X actions table format.

showRubisBestChoiceRecommendation(*Comments=False*, *ChoiceVector=False*, *CoDual=True*, *Debug=False*, *_OldCoca=False*, *BrokenCocs=True*, *Cpp=False*)
Renders the RuBis best choice recommendation.

Note: Computes by default the Rubis best choice recommendation on the corresponding strict (codual) outranking digraph.

In case of chordless circuits, if supporting arcs are more credible than the reversed negating arcs, we collapse the circuits into hyper nodes. Inversely, if supporting arcs are not more credible than the reversed negating arcs, we brake the circuits on their weakest arc.

Usage example:

```
>>> from outrankingDigraphs import *
>>> t = Random3ObjectivesPerformanceTableau(seed=5)
>>> g = BipolarOutrankingDigraph(t)
>>> g.showRubisBestChoiceRecommendation()
**********************
RuBis Best Choice Recommendation (BCR)
(in decreasing order of determinateness)
Credibility domain:  [-100.0, 100.0]
=== >> potential vest choices
* choice              : ['a04', 'a14', 'a19', 'a20']
   +-irredundancy      : 1.19
   independence        : 1.19
   dominance           : 4.76
   absorbency          : -59.52
   covering (%)        : 75.00
   determinateness (%) : 57.86
   - most credible action(s) = { 'a14': 23.81, 'a19': 11.90, 'a04': 2.38, 'a20':␣
↪1.19, }
=== >> potential worst choices
* choice              : ['a03', 'a12', 'a17']
   +-irredundancy      : 4.76
  independence         : 4.76
  dominance            : -76.19
  absorbency           : 4.76
  covering (%)         : 0.00
  determinateness (%)  : 65.39
  - most credible action(s) = { 'a03': 38.10, 'a12': 13.10, 'a17': 4.76, }
Execution time: 0.024 seconds
**************************
```

showRubyChoice(*Comments=False*, *_OldCoca=True*)
dummy for showRubisBestChoiceRecommendation() older versions compatibility

showShort()
concise presentation method for genuine digraphs.

showSingletonRanking(*Comments=True*, *Debug=False*)
Calls self.computeSingletonRanking(comments=True,Debug = False). Renders and prints the sorted bipolar net determinatation of outrankingness minus outrankedness credibilities of all singleton choices.

res = ((netdet,sigleton,dom,absorb)+)

showStatistics()
Computes digraph statistics like order, size and arc-density.

showdre()
Shows relation in nauty format.

singletons()
list of singletons and neighborhoods [(singx1, +nx1, -nx1, not(+nx1 or -nx1)),. . . .]

sizeSubGraph(*choice*)
Output: (size, undeterm,arcDensity). Renders the arc density of the induced subgraph.

strongComponents(*setPotential=False*)
Renders the set of strong components of self.

symDegreesDistribution()
Renders the distribution of symmetric degrees.

topologicalSort(*Debug=False*)
If self is acyclic, adds topological sort number to each node of self and renders ordered list of nodes. Otherwise renders None. Source: M. Golumbic Algorithmic Graph heory and Perfect Graphs, Annals Of Discrete Mathematics 57 2nd Ed. , Elsevier 2004, Algorithm 2.4 p.44.

weakAneighbors(*node*)
Renders the set of absorbed in-neighbors of a node.

weakCondorcetLoosers()
Renders the set of decision actions x such that self.relation[x][y] <= self.valuationdomain['med'] for all y != x.

weakCondorcetWinners()
Renders the set of decision actions x such that self.relation[x][y] >= self.valuationdomain['med'] for all y != x.

weakDneighbors(*node*)
Renders the set of dominated out-neighbors of a node.

weakGammaSets()
Renders the dictionary of neighborhoods {node: (dx,ax)}

worstRanks()
renders worst possible ranks from outdegrees account

zoomValuation(*zoomFactor=1.0*)
Zooms in or out, depending on the value of the zoomFactor provided, the bipolar valuation of a digraph.

class digraphs.**DualDigraph**(*other*)
Bases: *digraphs.Digraph*

Instantiates the dual (= negated valuation) Digraph object from a deep copy of a given other Digraph instance.

The relation constructor returns the dual of self.relation with generic formula: relationOut[a][b] = Max - self.relation[a][b] + Min where Max (resp. Min) equals valuation maximum (resp. minimum).

Note: In a bipolar valuation, the dual operator correspond to a simple changing of signs.

---

class digraphs.**EmptyDigraph**(*order=5*, *valuationdomain=(-1.0*, *1.0)*)
Bases: *digraphs.Digraph*

Parameters: order > 0 (default=5); valuationdomain =(Min,Max).

Specialization of the general Digraph class for generating temporary empty graphs of given order in {-1,0,1}.

class digraphs.**EquivalenceDigraph**(*d1*, *d2*, *Debug=False*)
Bases: *digraphs.Digraph*

Instantiates the logical equivalence digraph of two bipolar digraphs d1 and d2 of same order. Returns None if d1 and d2 are of different order

computeCorrelation()
Renders the global bipolar correlation index resulting from the pairwise equivalence valuations.

class digraphs.**FusionDigraph**(*dg1*, *dg2*, *operator='o-min'*)
Bases: *digraphs.Digraph*

Instantiates the epistemic fusion of two given Digraph instances called dg1 and dg2.

Parameter:

- operator = "o-min" | "o-max" (epistemic conjunctive or disjunctive fusion)

class digraphs.**FusionLDigraph**(*L*, *operator='o-min'*)
Bases: *digraphs.Digraph*

Instantiates the epistemic fusion a list L of Digraph instances.

Parameter:

- operator = "o-min" | "o-max" (epistemic conjunctive or dijunctive fusion)

class digraphs.**GridDigraph**(*n=5*, *m=5*, *valuationdomain={'max': 1.0*, *'min': -1.0}*, *hasRandomOrientation=False*, *hasMedianSplitOrientation=False*)
Bases: *digraphs.Digraph*

Specialization of the general Digraph class for generating temporary Grid digraphs of dimension n times m.

Parameters: n,m > 0; valuationdomain ={'min':m, 'max':M}.

Default instantiation (5 times 5 Grid Digraph): n = 5, m=5, valuationdomain = {'min':-1.0,'max':1.0}.

Randomly orientable with hasRandomOrientation=True (default=False).

showShort()

class digraphs.**IndeterminateDigraph**(*other=None*, *order=5*, *valuationdomain=(-1.0*, *1.0)*)
Bases: *digraphs.Digraph*

Parameters: order > 0; valuationdomain =(Min,Max). Specialization of the general Digraph class for generating temporary empty graphs of order 5 in {-1,0,1}.

class digraphs.**KneserDigraph**(*n=5*, *j=2*, *valuationdomain={'max': 1.0*, *'min': -1.0}*)
Bases: *digraphs.Digraph*

Specialization of the general Digraph class for generating temporary Kneser digraphs

Parameters:

n > 0; n > j > 0;
valuationdomain ={'min':m, 'max':M}.

Default instantiation as Petersen graph: n = 5, j = 2, valuationdomain = {'min':-1.0,'max':1.0}.

showShort()

class digraphs.**PolarisedDigraph**(*digraph=None*, *level=None*, *KeepValues=True*, *AlphaCut=False*, *StrictCut=False*)
Bases: *digraphs.Digraph*

Renders the polarised valuation of a Digraph class instance:

Parameters:

- If level = None, a default 75% cut level (0.5 in a normalized [-1,+1] valuation domain) is used.
- If KeepValues = False, the polarisation results in a three valued crisp result.
- If AlphaCut = True a genuine one-sided True-oriented cut is operated.
- If StrictCut = True, the cut level value is excluded resulting in an open polarised valuation domain. By default the polarised valuation domain is closed and the complementary indeterminate domain is open.

class digraphs.**Preorder**(*other*, *direction='best'*, *ranking=None*)
Bases: *digraphs.Digraph*

Instantiates the associated preorder from a given Digraph called other.

Instantiates as other.__class__ !

Copies the case given the description, the criteria and the evaluation dictionary into self.

class digraphs.**RedhefferDigraph**(*order=5*, *valuationdomain=(-1.0*, *1.0)*)
Bases: *digraphs.Digraph*

Specialization of the general Digraph class for generating temporary Redheffer digraphs.

https://en.wikipedia.org/wiki/Redheffer_matrix

Parameters: order > 0; valuationdomain=(Min,Max).

class digraphs.**StrongComponentsCollapsedDigraph**(*digraph=None*)
Bases: *digraphs.Digraph*

Reduction of Digraph object to its strong components.

showComponents()
Shows the list of connected components of the digraph instance.

class digraphs.**SymmetricPartialDigraph**(*digraph*)
Bases: *digraphs.Digraph*

Renders the symmetric part of a Digraph instance.

Note:

- The not symmetric and the reflexive links are all put to the median indeterminate characteristics value!.
- The constructor makes a deep copy of the given Digraph instance!

---

class digraphs.**XMCDA2Digraph**(*fileName='temp'*)
Bases: *digraphs.Digraph*

Specialization of the general Digraph class for reading stored XMCDA-2.0 formatted digraphs. Using the inbuilt module xml.etree (for Python 2.5+).

Param: fileName (without the extension .xmcda).

showAll()

class digraphs.**XORDigraph**(*d1*, *d2*, *Debug=False*)
Bases: *digraphs.Digraph*

Instantiates the XOR digraph of two bipolar digraphs d1 and d2 of same order.

class digraphs.**kChoicesDigraph**(*digraph=None*, *k=3*)
Bases: *digraphs.Digraph*

Specialization of general Digraph class for instantiation a digraph of all k-choices collapsed actions.

Parameters:

digraph := Stored or memory resident digraph instance
k := cardinality of the choices

Back to the *Installation*

2.2.4 randomDigraphs module

class randomDigraphs.**RandomDigraph**(*order=9*, *arcProbability=0.5*, *hasIntegerValuation=True*, *Bipolar=True*, *seed=None*)

Bases: *digraphs.Digraph*

Specialization of the general Digraph class for generating temporary crisp (irreflexive) random crisp digraphs.

Parameters:

- order (default = 10);
- arc_probability (in [0.,1.], default=0.5)
- If Bipolar=True, valuation domain = {-1,1} otherwise = {0,1}
- Is seed != None, the random generator is seeded

class randomDigraphs.**RandomFixedDegreeSequenceDigraph**(*order=7, degreeSequence=[3, 3, 2, 2, 1, 1, 0], seed=None*)

Bases: *digraphs.Digraph*

Specialization of the general Digraph class for generating temporary random crisp graphs (symmetric digraphs) with a fixed sequence of degrees.

Parameters: order=n and degreeSequence=[degree_1, . . . ,degree_n]>

> **Warning:** The implementation is not guaranteeing a uniform choice among all potential valid graph instances.

class randomDigraphs.**RandomFixedSizeDigraph**(*order=7*, *size=14*, *seed=None*)

Bases: *digraphs.Digraph*

Generates a random crisp digraph with a fixed size, by instantiating a fixed numbers of arcs from random choices in the set of potential oriented pairs of nodes numbered from 1 to order.

class randomDigraphs.**RandomGridDigraph**(*n=5*, *m=5*, *valuationdomain={'max': 1.0, 'min': -1.0}*, *seed=None*, *Debug=False*)

Bases: *digraphs.GridDigraph*

Specialization of the general Digraph class for generating temporary randomly oriented Grid digraphs of dimension n time m (default 5x5).

Parameters:

- n,m > 0;
- valuationdomain ={'min':-1 (default),'max': 1 (default)}.

class randomDigraphs.**RandomRegularDigraph**(*order=7*, *degree=2*, *seed=None*)

Bases: *digraphs.Digraph*

Parameters: order and degree.

Specialization of Digraph class for random regular symmetric instances.

class randomDigraphs.**RandomTournament**(*order=10, ndigits=2, isCrisp=True, valuationDomain=[-1, 1], seed=None*)

Bases: *digraphs.Digraph*

Specialization of the general Digraph class for generating temporary weak tournaments

Parameter:

- order = n > 0
- If valuationDomain = None, valuation is normalized (in [-1.0,1.0])
- If is Crips = True, valuation is polarized to min and max values

class randomDigraphs.**RandomValuationDigraph**(*order=9, ndigits=2, Normalized=True, hasIntegerValuation=False, seed=None*)

Bases: *digraphs.Digraph*

Specialization of the general Digraph class for generating temporary uniformly valuated random digraphs.

Parameters:

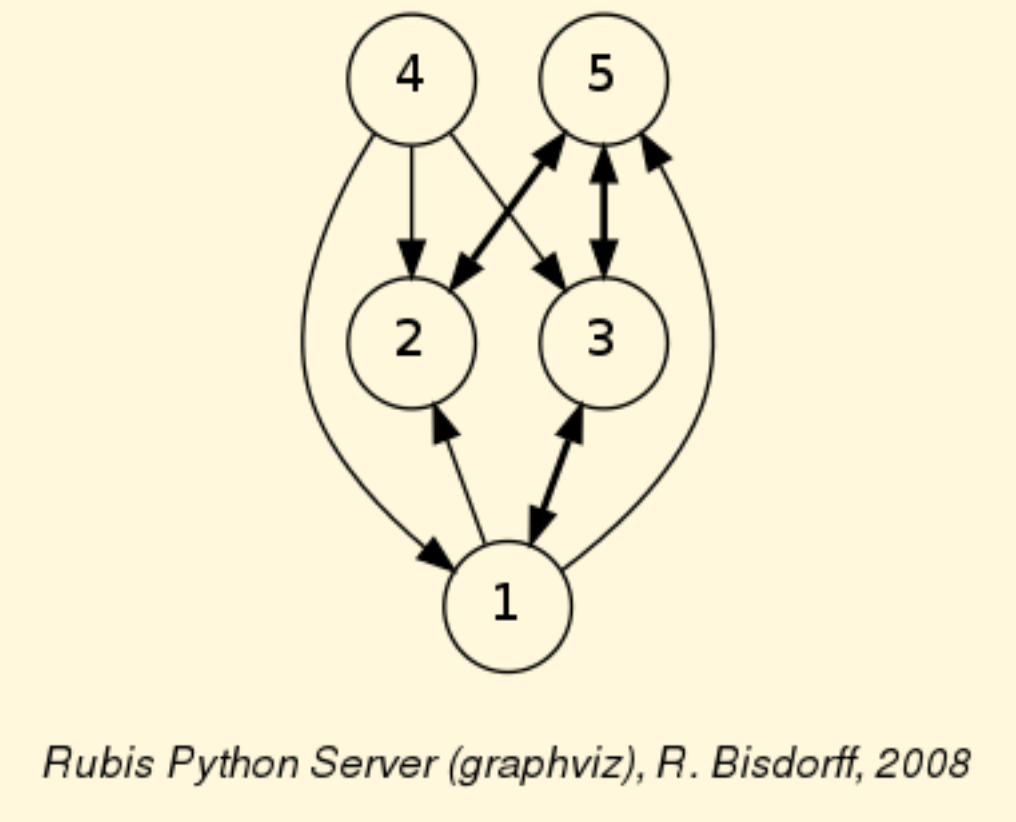
- order > 0, number of arcs;
- ndigits > 0, number of digits if hasIntegerValuation = True; Otherwise, decimal precision.
- Normalized = True (r in [-1,1], r in [0,1] if False/default);
- hasIntegerValuation = False (default)
- If seed != none, the random generator is seeded

Example python3 session:

```
>>> from digraphs import RandomValuationDigraph
>>> dg = RandomValuationDigraph(order=5,Normalized=True)
>>> dg.showAll()
*----- show detail -------------*
Digraph          : randomValuationDigraph
*---- Actions ----*
['1', '2', '3', '4', '5']
*---- Characteristic valuation domain ----*
{'max': Decimal('1.0'), 'min': Decimal('-1.0'),
 'med': Decimal('0.0'), 'hasIntegerValuation': False}
* ---- Relation Table -----
  S   |  '1'    '2'    '3'    '4'     '5'
 -----|-----------------------------------
  '1' |  0.00   0.28   0.46  -0.66   0.90
  '2' | -0.08   0.00  -0.46  -0.42   0.52
  '3' |  0.84  -0.10   0.00  -0.54   0.58
  '4' |  0.90   0.88   0.90   0.00  -0.38
  '5' | -0.50   0.64   0.42  -0.94   0.00
*--- Connected Components ---*
1: ['1', '2', '3', '4', '5']
Neighborhoods:
  Gamma     :
'4': in => set(), out => {'1', '2', '3'}
'5': in => {'1', '2', '3'}, out => {'2', '3'}
'1': in => {'4', '3'}, out => {'5', '2', '3'}
'2': in => {'4', '5', '1'}, out => {'5'}
'3': in => {'4', '5', '1'}, out => {'5', '1'}
  Not Gamma :
```

```
'4': in => {'5', '1', '2', '3'}, out => {'5'}
'5': in => {'4'}, out => {'4', '1'}
'1': in => {'5', '2'}, out => {'4'}
'2': in => {'3'}, out => {'4', '1', '3'}
'3': in => {'2'}, out => {'4', '2'}
```

```
>>> dg.exportGraphViz()
```



class randomDigraphs.**RandomWeakTournament**(*order=10*, *ndigits=2*, *hasIntegerValuation=False*, *weaknessDegree=0.25*, *seed=None*, *Comments=False*)

Bases: *digraphs.Digraph*

Specialization of the general Digraph class for generating temporary bipolar-valued weak tournaments

Parameters:

- order = n > 0
- weaknessDegree in [0.0,1.0]: proportion of indeterminate links (default = 0.25)
- If hasIntegerValuation = True, valuation domain = [-pow(10,ndigits); + pow(10,ndigits)] else valuation domain = [-1.0,1.0]
- If seed != None, the random number generator is seeded

Back to the *Installation*

2.2.5 graphs module

A tutorial with coding examples is available here: *Working with the graphs module*

Digraph3 graphs.py module Python3.3+ computing resources Copyright (C) 2011-2015 Raymond Bisdorff

> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

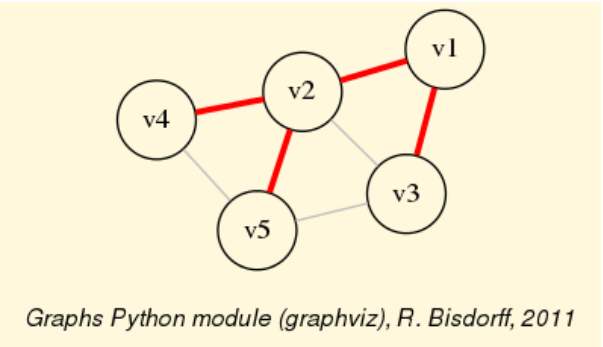
class graphs.**BestDeterminedSpanningForest**(*g*, *seed=None*, *Debug=False*)

Bases: *graphs.RandomTree*

Constructing the most determined spanning tree (or forest if not connected) using Kruskal's greedy algorithm on the dual valuation.

Example Python session:

```
>>> from graphs import *
>>> g = RandomValuationGraph(seed=2)
>>> g.showShort()
*---- short description of the graph ----*
Name             : 'randomGraph'
Vertices         :  ['v1', 'v2', 'v3', 'v4', 'v5']
Valuation domain :  {'med': Decimal('0'), 'min': Decimal('-1'), 'max':␣
↪Decimal('1')}
Gamma function   :
v1 -> ['v2', 'v3']
v2 -> ['v4', 'v1', 'v5', 'v3']
v3 -> ['v1', 'v5', 'v2']
v4 -> ['v5', 'v2']
v5 -> ['v4', 'v2', 'v3']
>>> mt = BestDeterminedSpanningForest(g)
>>> mt.exportGraphViz('spanningTree',withSpanningTree=True)
*---- exporting a dot file for GraphViz tools ---------*
Exporting to spanningTree.dot
[['v4', 'v2', 'v1', 'v3', 'v1', 'v2', 'v5', 'v2', 'v4']]
neato -Tpng spanningTree.dot -o spanningTree.png
```



Graphs Python module (graphviz), R. Bisdorff, 2011

class graphs.**CompleteGraph**(*order=5*, *seed=None*)

Bases: *graphs.Graph*

Instances of complete graphs bipolarly valuated in {-1,0,+1}. Each vertex x is positively linked to all the other vertices (edges[{x,y}] = +1)

Parameter:

- order (positive integer)

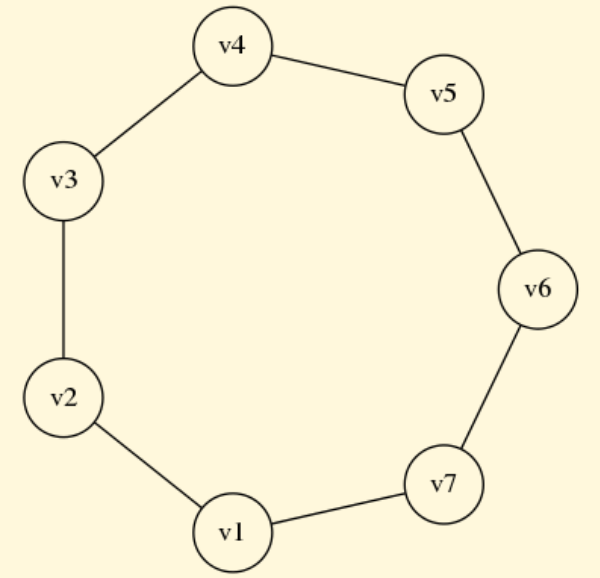
class graphs.**CycleGraph**(*order=5*, *seed=None*, *Debug=False*)
Bases: *graphs.Graph*

Instances of cycle graph characterized in [-1,1].

Parameter:

- order (positive integer)

Example of 7-cycle graph instance:



Graphs Python module (graphviz), R. Bisdorff, 2011

class graphs.**DualGraph**(*other*)
Bases: *graphs.Graph*

Instantiates the dual Graph object of a given other Graph instance.

The relation constructor returns the dual of self.relation with formula: relationOut[a][b] = Max - self.relation[a][b] + Min where Max (resp. Min) equals valuation maximum (resp. minimum).

class graphs.**EmptyGraph**(*order=5*, *seed=None*)
Bases: *graphs.Graph*

Intantiates graph of given order without any positively valued edge.

Parameter:

- order (positive integer)

class graphs.**Graph**(*fileName=None*, *Empty=False*, *numberOfVertices=7*, *edgeProbability=0.5*)
Bases: object

In the *graphs* module, the root *graphs.Graph* class provides a generic graph model. A given object consists in:

1. a vertices dictionary
2. a characteristic valuation domain, {-1,0,+1} by default
3. an edges dictionary, characterising each edge in the given valuation domain
4. a gamma function dictionary, holding the neighborhood vertices of each vertex

General structure:

```
vertices = {'v1': {'name': ...,'shortName': ...},
            'v2': {'name': ...,'shortName': ...},
            'v3': {'name': ...,'shortName': ...},
            ... }
valuationDomain = {'min': -1, 'med': 0, 'max': 1}
edges = {frozenset({'v1','v2'}): 1,
         frozenset({'v1','v3'}): 1,
         frozenset({'v2','v3'}): -1,
           ...}
## links from each vertex to its neighbors
gamma = {'v1': {'v2',v3'}, 'v2': {'v1'}, 'v3': {'v1'}, ... }
```

Example python3 session:

```
>>> from graphs import Graph
>>> g = Graph(numberOfVertices=5,edgeProbability=0.5) # random instance
>>> g.showShort()
*----- show short --------------*
*---- short description of the graph ----*
Name              :  'random'
Vertices          :  ['v1', 'v2', 'v3', 'v4', 'v5']
Valuation domain  :  {'med': 0, 'max': 1, 'min': -1}
Gamma function    :
v1 -> ['v4']
v2 -> []
v3 -> ['v4']
v4 -> ['v1', 'v3']
v5 -> []
```

computeChordlessCycles(*Cycle3=False*, *Comments=False*, *Debug=False*)

Renders the set of all chordless cycles observed in a Graph intance. Inspired from Dias, Castonguay, Longo & Jradi, Algorithmica 2015.

Note: By default, a chordless cycle must have at least length 4.If the Cycle3 flag is set to True, the cyclicly closed triplets will be inserted as 3-cycles in the result.

---

computeCliques(*Comments=False*)

Computes all cliques, ie maximal complete subgraphs in self:

Note:

- Computes the maximal independent vertex sets in the dual of self.
- Result is stored in self.cliques.

---

computeComponents()
Computes the connected components of a graph instance. Returns a partition of the vertices as a list

computeDegreeDistribution(*Comments=False*)
Renders the distribution of vertex degrees.

computeDiameter(*Oriented=False*)
Renders the diameter (maximal neighbourhood depth) of the digraph instance.

Note: The diameter of a disconnected graph is considered to be *infinite* (results in a value -1) !

---

computeMIS(*Comments=False*)
Prints all maximal independent vertex sets:

Note:

- Result is stored in self.misset !

---

computeNeighbourhoodDepth(*vertex*, *Debug=False*)
Renders the distribtion of neighbourhood depths.

computeNeighbourhoodDepthDistribution(*Comments=False*, *Debug=False*)
Renders the distribtion of neighbourhood depths.

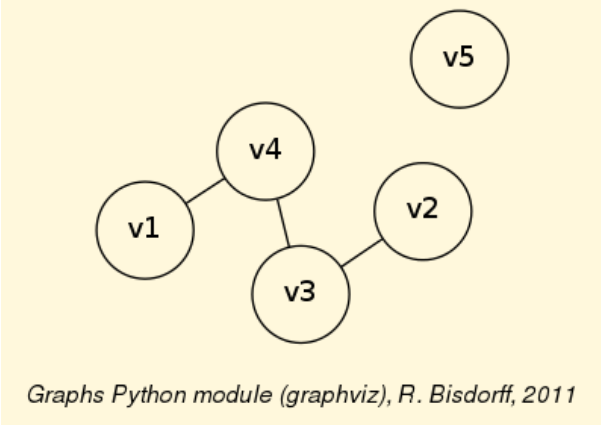
computeSize()
Renders the number of positively characterised edges of this graph instance (result is stored in self.size).

depthFirstSearch(*Debug=False*)
Depth first search through a graph in lexicographical order of the vertex keys.

exportGraphViz(*fileName=None*, *verticesSubset=None*, *noSilent=True*, *graphType='png'*, *graphSize='7, 7'*, *withSpanningTree=False*, *layout=None*, *arcColor='black'*, *lineWidth=1*)
Exports GraphViz dot file for graph drawing filtering.

Example:

```
>>> g = Graph(numberOfVertices=5,edgeProbability=0.3)
>>> g.exportGraphViz('randomGraph'))
```



gammaSets(*Debug=False*)
renders the gamma function as dictionary

generateIndependent(*U*)
Generator for all independent vertices sets with neighborhoods of a graph instance:

Note:

- Initiate with U = self._singletons().
- Yields [independent set, covered set, all vertices - covered set)].
- If independent set == (all vertices - covered set), the given independent set is maximal !

---

graph2Digraph()
Converts a Graph object into a symmetric Digraph object.

isConnected()
Cheks if self is a connected graph instance.

isTree()
Checks if self is a tree by verifing the required number of edges: order-1; and the existence of leaves.

randomDepthFirstSearch(*seed=None*, *Debug=False*)
Depth first search through a graph in random order of the vertex keys.

Note: The resulting spanning tree (or forest) is by far not uniformly selected among all possible trees. Spanning stars will indeed be much less probably selected then streight walks !

---

recodeValuation(*newMin=-1*, *newMax=1*, *Debug=False*)
Recodes the characteristic valuation domain according to the parameters given.

Note: Default values gives a normalized valuation domain

---

save(*fileName='tempGraph'*, *Debug=False*)
Persistent storage of a Graph class instance in the form of a python source code file.

setEdgeValue(*edge*, *value*, *Comments=False*)
Wrapper for updating the charactreistic valuation of a Graph instance. The egde parameter consists in a pair of vertices; edge = ('v1','v2') for instance. The new value must be in the limits of the valuation domain.

showCliques()

showMIS()

showMore()
Generic show method for Graph instances.

showShort()
Generic show method for Graph instances.

class graphs.**GridGraph**(*n=5*, *m=5*, *valuationMin=-1*, *valuationMax=1*)
Bases: *graphs.Graph*

Specialization of the general Graph class for generating temporary Grid graphs of dimension n times m.

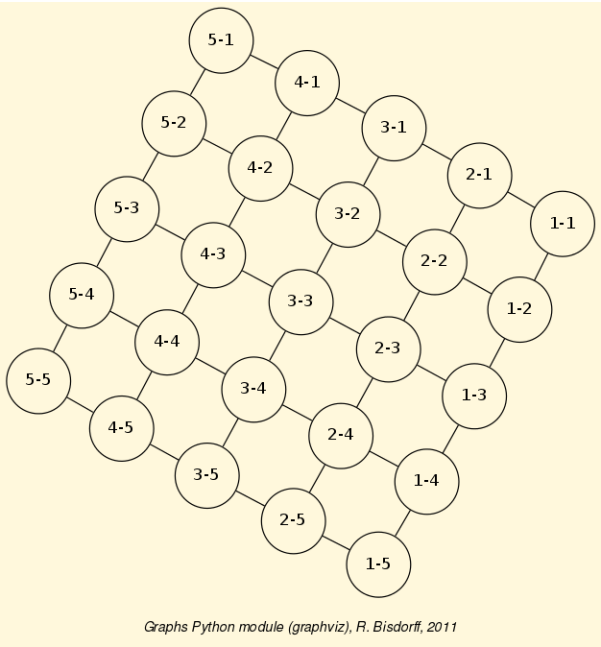
Parameters:

- n,m > 0
- valuationDomain ={'min':-1, 'med':0, 'max':+1}

Default instantiation (5 times 5 Grid Digraph):

- n = 5,
- m=5,
- valuationDomain = {'min':-1.0,'max':1.0}.

Example of 5x5 GridGraph instance:



showShort()

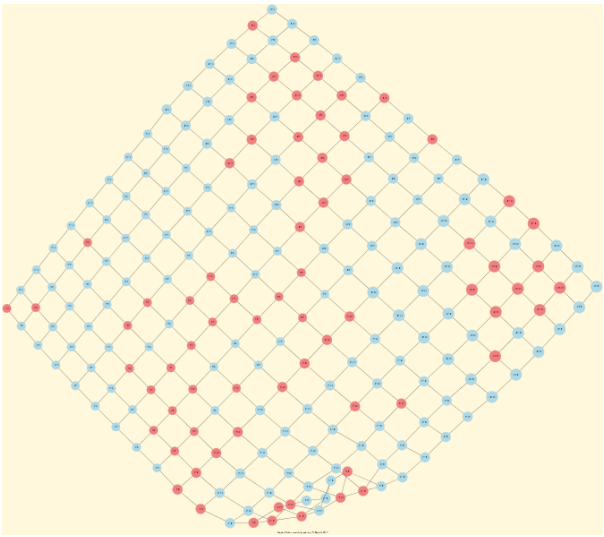
class graphs.**IsingModel**(*g*, *beta=0*, *nSim=None*, *Debug=False*)

Bases: *graphs.Graph*

Specialisation of a Gibbs Sampler for the Ising model

Example:

```
>>> from graphs import GridGraph, IsingModel
>>> g = GridGraph(n=15,m=15)
>>> g.showShort()
*----- show short --------------*
Grid graph    :  grid-6-6
n             :  6
m             :  6
order         :  36
>>> im = IsingModel(g,beta=0.3,nSim=100000,Debug=False)
Running a Gibbs Sampler for 100000 step !
>>> im.exportGraphViz(colors=['lightblue','lightcoral'])
*---- exporting a dot file for GraphViz tools ---------*
Exporting to grid-15-15-ising.dot
fdp -Tpng grid-15-15-ising.dot -o grid-15-15-ising.png
```



computeSpinEnergy()
Spin energy H(c) of a spin configuration is H(c) = -sum_{{x,y} in self.edges}[spin_c(x)*spin_c(y)]

exportGraphViz(*fileName=None, noSilent=True, graphType='png', graphSize='7,7', edgeColor='black', colors=['gold', 'lightblue']*)
Exports GraphViz dot file for Ising models drawing filtering.

generateSpinConfiguration(*beta=0*, *nSim=None*, *Debug=False*)

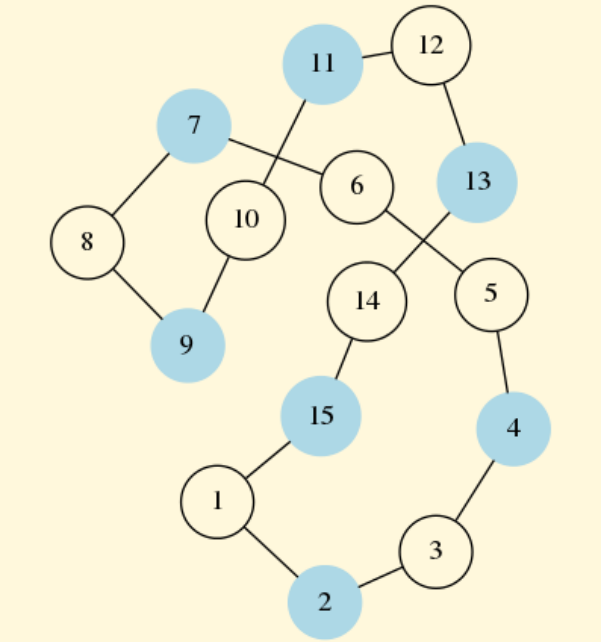
class graphs.**MISModel**(*g*, *nSim=None*, *maxIter=20*, *seed=None*, *Debug=False*)
Bases: *graphs.Graph*

Specialisation of a Gibbs Sampler for the hard code model, that is a random MIS generator.

Example:

```
>>> from graphs import MISModel
>>> from digraphs import CirculantDigraph
>>> dg = CirculantDigraph(order=15)
>>> g = dg.digraph2Graph()
>>> g.showShort()
*---- short description of the graph ----*
Name             : 'c15'
Vertices         :  ['1', '10', '11', '12', '13', '14',
                     '15', '2', '3', '4', '5', '6', '7',
                     '8', '9']
Valuation domain :  {'med': 0, 'min': -1, 'max': 1}
Gamma function   :
1 -> ['2', '15']
10 -> ['11', '9']
11 -> ['10', '12']
12 -> ['13', '11']
13 -> ['12', '14']
14 -> ['15', '13']
15 -> ['1', '14']
2 -> ['1', '3']
3 -> ['2', '4']
4 -> ['3', '5']
5 -> ['6', '4']
6 -> ['7', '5']
7 -> ['6', '8']
```

```
8 -> ['7', '9']
9 -> ['10', '8']
>>> mis = MISModel(g)
Running a Gibbs Sampler for 1050 step !
>>> mis.checkMIS()
{'2','4','7','9','11','13','15'}  is maximal !
>>> mis.exportGraphViz()
*---- exporting a dot file for GraphViz tools ---------*
Exporting to c15-mis.dot
fdp -Tpng c15-mis.dot -o c15-mis.png
```



Graphs Python module (graphviz), R. Bisdorff, 2014

checkMIS(*Comments=True*)
Verify maximality of independent set.

Note: Returns three sets: an independent choice, the covered vertices, and the remaining uncovered vertices. When the last set is empty, the independent choice is maximal.

---

exportGraphViz(*fileName=None*, *noSilent=True*, *graphType='png'*, *graphSize='7, 7'*, *misColor='lightblue'*)
Exports GraphViz dot file for MIS models drawing filtering.

generateMIS(*Reset=True*, *nSim=None*, *seed=None*, *Comments=True*, *Debug=False*)

class graphs.**MetropolisChain**(*g*, *probs=None*)
Bases: *graphs.Graph*

Specialisation of the graph class for implementing a generic Metropolis Markov Chain Monte Carlo sampler with a given probability distribution probs = {'v1': x, 'v2': y, ... }

Usage example:

```
>>> from graphs import *
>>> g = Graph(numberOfVertices=5,edgeProbability=0.5)
>>> g.showShort()
*---- short description of the graph ----*
Name             : 'randomGraph'
Vertices         :  ['v1', 'v2', 'v3', 'v4', 'v5']
Valuation domain :  {'max': 1, 'med': 0, 'min': -1}
Gamma function   :
v1 -> ['v2', 'v3', 'v4']
v2 -> ['v1', 'v4']
v3 -> ['v5', 'v1']
v4 -> ['v2', 'v5', 'v1']
v5 -> ['v3', 'v4']
>>> probs = {}
>>> n = g.order
>>> i = 0
>>> verticesList = [x for x in g.vertices]
>>> verticesList.sort()
>>> for v in verticesList:
...     probs[v] = (n - i)/(n*(n+1)/2)
...     i += 1
>>> met = MetropolisChain(g,probs)
>>> frequency = met.checkSampling(verticesList[0],nSim=30000)
>>> for v in verticesList:
...     print(v,probs[v],frequency[v])
v1 0.3333 0.3343
v2 0.2666 0.2680
v3 0.2    0.2030
v4 0.1333 0.1311
v5 0.0666 0.0635
>>> met.showTransitionMatrix()
* ---- Transition Matrix -----
  Pij  | 'v1'    'v2'    'v3'    'v4'    'v5'
  -----|---------------------------------------
  'v1' |  0.23   0.33    0.30    0.13    0.00
  'v2' |  0.42   0.42    0.00    0.17    0.00
  'v3' |  0.50   0.00    0.33    0.00    0.17
  'v4' |  0.33   0.33    0.00    0.08    0.25
  'v5' |  0.00   0.00    0.50    0.50    0.00
```

MCMCtransition(*si*, *Debug=False*)

checkSampling(*si*, *nSim*)

computeTransitionMatrix()

saveCSVTransition(*fileName='transition'*, *Debug=False*)
Persistent storage of the transition matrix in the form of a csv file.

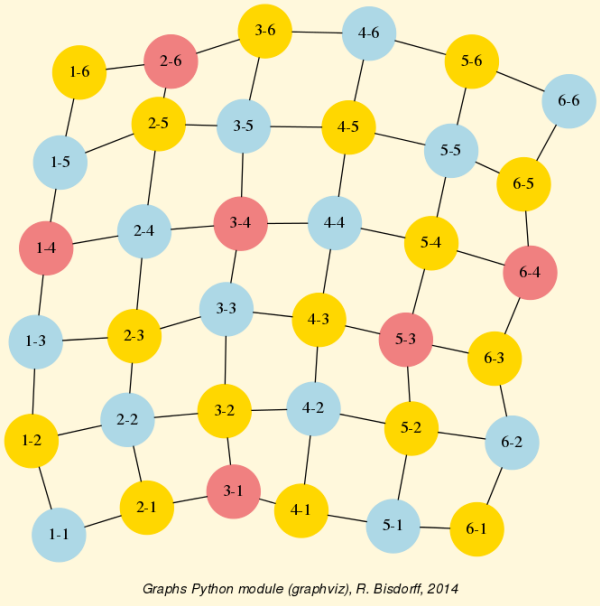
showTransitionMatrix(*Sorted=True*, *IntegerValues=False*, *vertices=None*, *relation=None*, *ndigits=2*, *ReflexiveTerms=True*)
Prints on stdout the transition probabilities in vertices X vertices table format.

class graphs.**Q_Coloring**(*g, colors=['gold', 'lightcoral', 'lightblue'], nSim=None, maxIter=20, seed=None, Comments=True, Debug=False*)
Bases: *graphs.Graph*

Generate a q-coloring of a Graph instance via a Gibbs MCMC sampler in nSim simulation steps (default = len(graph.edges)).

Example 3-coloring of a grid 6x6 :

```
>>> from graphs import *
>>> g = GridGraph(n=6,m=6)
>>> g.showShort()
>>> g.exportGraphViz()
*----- show short --------------*
Grid graph    :  grid-6-6
n             :  6
m             :  6
order         :  36
>>> qc = Q_Coloring(g,colors=['gold','lightblue','lightcoral'])
Running a Gibbs Sampler for 630 step !
>>> qc.checkFeasibility()
The q-coloring with 3 colors is feasible !!
>>> qc.exportGraphViz()
*---- exporting a dot file for GraphViz tools ---------*
Exporting to grid-6-6-qcoloring.dot
fdp -Tpng grid-6-6-qcoloring.dot -o grid-6-6-qcoloring.png
```



checkFeasibility(*Comments=True*, *Debug=False*)

exportGraphViz(*fileName=None*, *noSilent=True*, *graphType='png'*, *graphSize='7, 7'*, *layout=None*)

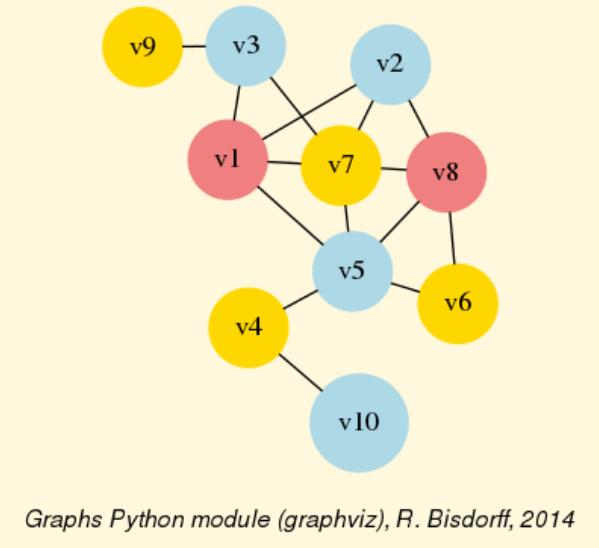
Exports GraphViz dot file for q-coloring drawing filtering.

The graph drawing layout is depending on the graph type, but can be forced to either 'fdp', 'circo' or 'neato' with the layout parameter.

Example:

```
>>> g = Graph(numberOfVertices=10,edgeProbability=0.4)
>>> g.showShort()
*---- short description of the graph ----*
Name : 'randomGraph'
Vertices :  ['v1','v10','v2','v3','v4','v5','v6','v7','v8','v9']
Valuation domain :  {'max': 1, 'min': -1, 'med': 0}
Gamma function   :
```

```
v1 -> ['v7', 'v2', 'v3', 'v5']
v10 -> ['v4']
v2 -> ['v1', 'v7', 'v8']
v3 -> ['v1', 'v7', 'v9']
v4 -> ['v5', 'v10']
v5 -> ['v6', 'v7', 'v1', 'v8', 'v4']
v6 -> ['v5', 'v8']
v7 -> ['v1', 'v5', 'v8', 'v2', 'v3']
v8 -> ['v6', 'v7', 'v2', 'v5']
v9 -> ['v3']
>>> qc = Q_Coloring(g,nSim=1000)
Running a Gibbs Sampler for 1000 step !
>>> qc.checkFeasibility()
The q-coloring with 3 colors is feasible !!
>>> qc.exportGraphViz()
*---- exporting a dot file for GraphViz tools ---------*
Exporting to randomGraph-qcoloring.dot
fdp -Tpng randomGraph-qcoloring.dot -o randomGraph-qcoloring.png
```



generateFeasibleConfiguration(*Reset=True*, *nSim=None*, *seed=None*, *Debug=False*)

showConfiguration()

class graphs.**RandomFixedDegreeSequenceGraph**(*order=7*, *degreeSequence=[3, 3, 2, 2, 1, 1, 0]*, *seed=None*)

Bases: *graphs.Graph*

Specialization of the general Graph class for generating temporary random graphs with a fixed sequence of degrees.

> **Warning:** The implementation is not guaranteeing a uniform choice among all potential valid graph instances.

class graphs.**RandomFixedSizeGraph**(*order=7*, *size=14*, *seed=None*, *Debug=False*)

Bases: *graphs.Graph*

Generates a random graph with a fixed size (number of edges), by instantiating a fixed numbers of arcs from random choices in the set of potential pairs of vertices numbered from 1 to order.

class graphs.**RandomGraph**(*order=5*, *edgeProbability=0.4*, *seed=None*)
Bases: *graphs.Graph*

Random instances of the Graph class

Parameters:

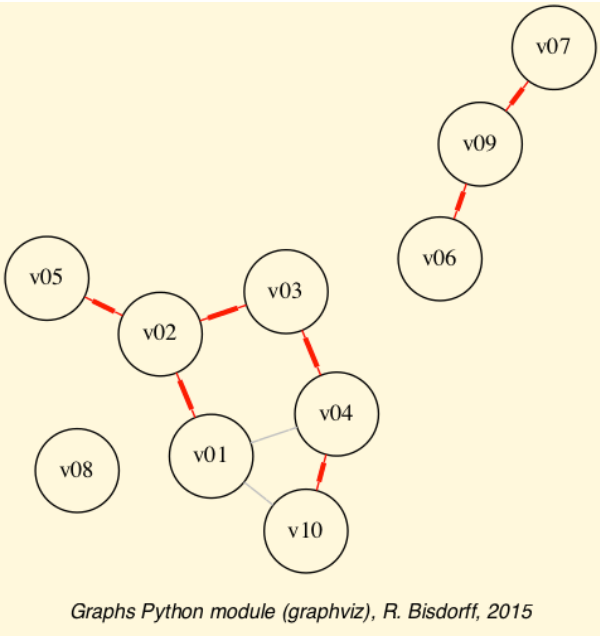
- order (positive integer)
- edgeProbability (in [0,1])

class graphs.**RandomRegularGraph**(*order=7*, *degree=2*, *seed=None*)
Bases: *graphs.Graph*

Specialization of the general Graph class for generating temporary random regular graphs of fixed degrees.

class graphs.**RandomSpanningForest**(*g*, *seed=None*, *Debug=False*)
Bases: *graphs.RandomTree*

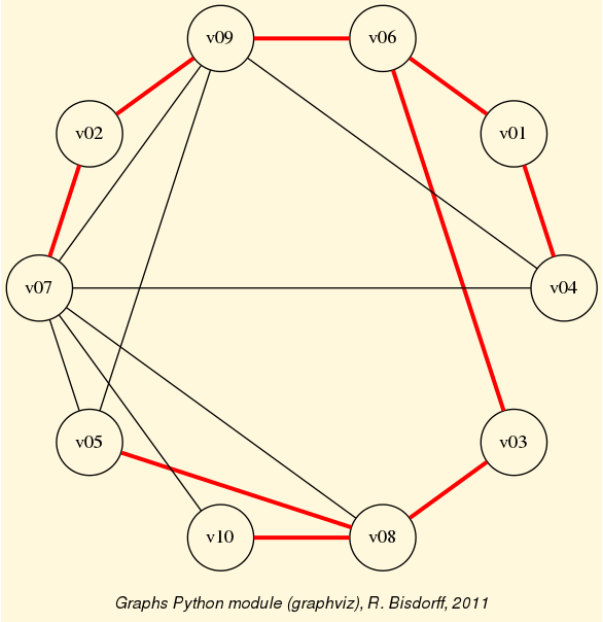
Random instance of a spanning forest (one or more trees) generated from a random depth first search graph g traversal.



class graphs.**RandomSpanningTree**(*g*, *seed=None*, *Debug=False*)
Bases: *graphs.RandomTree*

Uniform random instance of a spanning tree generated with Wilson's algorithm from a connected Graph g instance.

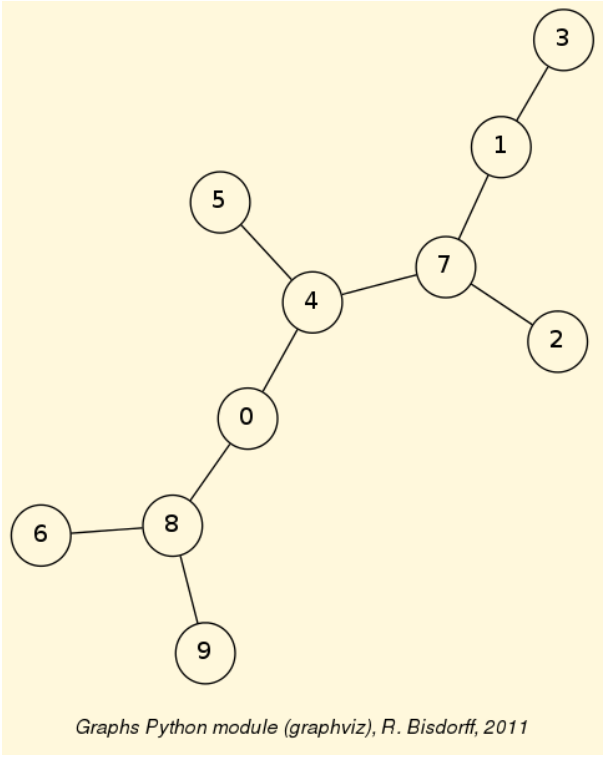
Note: Wilson's algorithm only works for connecte graphs.



class graphs.**RandomTree**(*order=None*, *vertices=None*, *prueferCode=None*, *seed=None*, *Debug=False*)

Bases: *graphs.Graph*

Instance of a tree generated from a random (or a given) Prüfer code.



tree2Pruefer(*vertices=None*, *Debug=False*)

Renders the Pruefer code of a given tree.

class graphs.**RandomValuationGraph**(*order=5*, *ndigits=2*, *seed=None*)

Bases: *graphs.Graph*

Specialization of the genuine Graph class for generating temporary randomly valuated graphs in the range [-1.0;1.0].

Parameter:

- order (positive integer)
- ndigits (decimal precision)

class graphs.**SnakeGraph**(*p*, *q*)

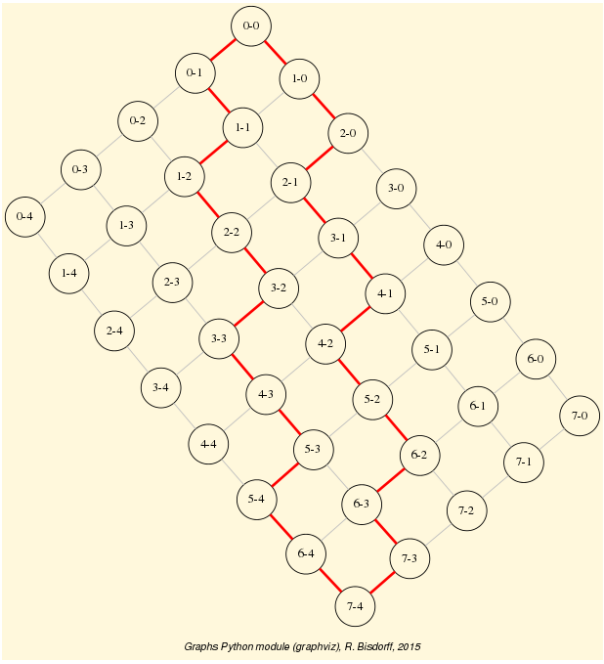
Bases: *graphs.GridGraph*

Snake graphs S(p/q) are made up of all the integer grid squares between the lower and upper Christofel paths of the rational number p/q, where p and q are two coprime integers such that 0 <= p <= q, i.e. p/q gives an irreducible ratio between 0 and 1.

Reference: M. Aigner, Markov's Theorem and 100 Years of the Uniqueness Conjecture, Springer, 2013, p. 141-149

S(4/7) snake graph instance:

```
>>> from graphs import SnakeGraph
>>> s4_7 = SnakeGraph(p=4,q=7)
>>> s4_7.showShort()
*---- short description of the snake graph ----*
Name              : 'snakeGraph'
Rational p/q      : 4/7
Christoffel words:
Upper word        :  BBABABA
Lower word        :  ABABABB
>>> s4_7.exportGraphViz('4_7_snake',lineWidth=3,arcColor='red')
```



showShort(*WithVertices=False*)

Show method for SnakeGraph instances.

class graphs.**TriangulatedGrid**(*n=5*, *m=5*, *valuationMin=-1*, *valuationMax=1*)

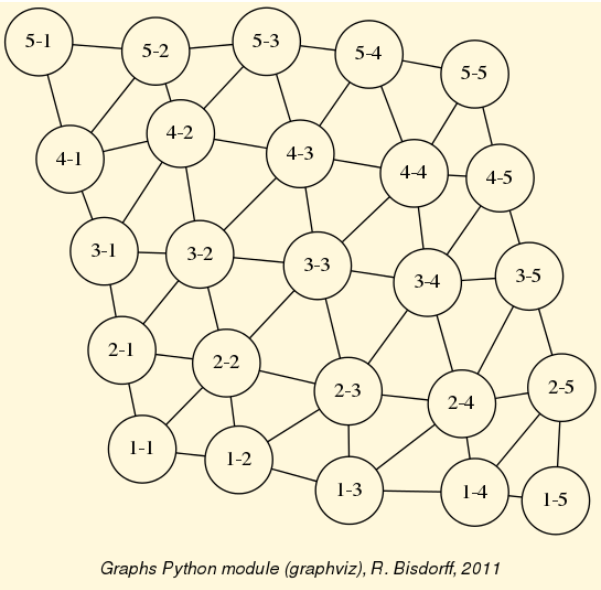
Bases: *graphs.Graph*

Specialization of the general Graph class for generating temporary triangulated grids of dimension n times m.

Parameters:

- n,m > 0
- valuationDomain = {'min':m, 'max':M}

Example of 5x5 triangulated grid instance:



Graphs Python module (graphviz), R. Bisdorff, 2011

showShort()

Back to the *Installation*

2.2.6 perfTabs module

class perfTabs.**CSVPerformanceTableau**(*fileName='temp'*, *Debug=True*)
Bases: *perfTabs.PerformanceTableau*

Reading stored CSV encoded actions x criteria PerformanceTableau instances, Using the inbuilt module csv.

Param: fileName (without the extension .csv).

class perfTabs.**ConstantPerformanceTableau**(*inPerfTab*, *actionsSubset=None*, *criteriaSubset=None*, *position=0.5*)
Bases: *perfTabs.PerformanceTableau*

Constructor for (partially) constant performance tableaux.

Parameter:

- *actionsSubset* selects the actions to be set at equal constant performances,
- *criteriaSubset* select the concerned subset of criteria,
- The *position* parameter (default = median performance) selects the constant performance in the respective scale of each performance criterion.

class perfTabs.**EmptyPerformanceTableau**
Bases: *perfTabs.PerformanceTableau*

Template for PerformanceTableau objects.

class perfTabs.**NormalizedPerformanceTableau**(*argPerfTab=None*, *lowValue=0*, *highValue=100*, *coalition=None*, *Debug=False*)

Bases: *perfTabs.PerformanceTableau*

specialsation of the PerformanceTableau class for constructing normalized, 0 - 100, valued PerformanceTableau instances from a given argPerfTab instance.

class perfTabs.**PartialPerformanceTableau**(*inPerfTab*, *actionsSubset=None*, *criteriaSubset=None*, *objectivesSubset=None*)

Bases: *perfTabs.PerformanceTableau*

Constructor for partial performance tableaux concerning a subset of actions and/or criteria and/or objectives

class perfTabs.**PerformanceTableau**(*filePerfTab=None*, *isEmpty=False*)

Bases: object

In this *Digraph3* module, the root *perfTabs.PerformanceTableau* class provides a generic **performance table model**. A given object of this class consists in:

1. a set of potential decision **actions** : an ordered dictionary describing the potential decision actions or alternatives with 'name' and 'comment' attributes,
2. an optional set of decision **objectives**: an ordered dictionary with name, comment, weight and list of concerned criteria per objective,
3. a coherent family of **criteria**: a ordered dictionary of criteria functions used for measuring the performance of each potential decision action with respect to the preference dimension captured by each criterion,
4. the **evaluations**: a dictionary of performance evaluations for each decision action or alternative on each criterion function.

Structure:

```
actions = OrderedDict([('a1', {'name': ..., 'comment': ...}),
           ('a2', {'name': ..., 'comment': ...}),
           ...])
objectives = OrderedDict([
            ('obj1', {'name': ..., 'comment': ..., 'weight': ..., 'criteria': ['g1
↪', ...]}),
            ('obj2', {'name': ..., 'comment', ..., 'weight': ..., 'criteria': ['g2
↪', ...]}),
            ...])
criteria = OrderedDict([
     ('g1', {'weight':Decimal("3.00"),
             'scale': (Decimal("0.00"),Decimal("100.00")),
             'thresholds' : {'pref': (Decimal('20.0'), Decimal('0.0')),
                             'ind': (Decimal('10.0'), Decimal('0.0')),
                             'veto': (Decimal('80.0'), Decimal('0.0'))},
             'objective': 'obj1',
             }),
     ('g2', {'weight':Decimal("5.00"),
             'scale': (Decimal("0.00"),Decimal("100.00")),
             'thresholds' : {'pref': (Decimal('20.0'), Decimal('0.0')),
                             'ind': (Decimal('10.0'), Decimal('0.0')),
                             'veto': (Decimal('80.0'), Decimal('0.0'))},
             'objective': 'obj2',
             }),
     ...])
evaluation = {'g1': {'a1':Decimal("57.28"),'a2':Decimal("99.85"), ...},
              'g2': {'a1':Decimal("88.12"),'a2':Decimal("33.25"), ...},
              ...}
```

With the help of the `perfTabs.RandomPerformanceTableau` class let us generate for illustration a random performance tableau concerning 7 decision actions or alternatives denoted *a01*, *a02*, ..., *a07*:

```
>>> from randomPerfTabs import RandomPerformanceTableau
>>> rt = RandomPerformanceTableau(seed=100)
>>> rt.showActions()
    *----- show decision action --------------*
    key:  a01
      short name: a01
      name:       random decision action
      comment:    RandomPerformanceTableau() generated.
    key:  a02
      short name: a02
      name:       random decision action
      comment:    RandomPerformanceTableau() generated.
    key:  a03
      short name: a03
      name:       random decision action
      comment:    RandomPerformanceTableau() generated.
    ...
    ...
    key:  a07
    name:       random decision action
    comment:    RandomPerformanceTableau() generated.
>>> ...
```

In this example we consider furthermore a family of seven equisignificant cardinal criteria functions *g01*, *g02*, ..., *g07*, measuring the performance of each alternative on a rational scale form 0.0 to 100.00. In order to capture the evaluation's uncertainty and imprecision, each criterion function *g1* to *g7* admits three performance discrimination thresholds of 10, 20 and 80 pts for warranting respectively any indifference, preference and veto situations:

```
>>> rt.showCriteria(IntegerWeights=True)
    *----  criteria -----*
    g1 'RandomPerformanceTableau() instance'
      Scale = (0.0, 100.0)
      Weight = 1
      Threshold ind : 10.00 + 0.00x ; percentile:  0.20
      Threshold veto : 80.00 + 0.00x ; percentile:  0.93
      Threshold pref : 20.00 + 0.00x ; percentile:  0.28
    g2 'RandomPerformanceTableau() instance'
      Scale = (0.0, 100.0)
      Weight = 1
      Threshold ind : 10.00 + 0.00x ; percentile:  0.18
      Threshold veto : 80.00 + 0.00x ; percentile:  1.0
      Threshold pref : 20.00 + 0.00x ; percentile:  0.37
    g3 'RandomPerformanceTableau() instance'
      Scale = (0.0, 100.0)
      Weight = 1
      Threshold ind : 10.00 + 0.00x ; percentile:  0.15
      Threshold veto : 80.00 + 0.00x ; percentile:  0.96
      Threshold pref : 20.00 + 0.00x ; percentile:  0.29
    ...
    ...
    g7 'RandomPerformanceTableau() instance'
      Scale = (0.0, 100.0)
      Weight = 1
      Threshold ind : 10.00 + 0.00x ; percentile:  0.17
```

```
      Threshold veto : 80.00 + 0.00x ; percentile:  0.97
      Threshold pref : 20.00 + 0.00x ; percentile:  0.37
>>> ...
```

The performance evaluations of each decision alternative on each criterion are gathered in a *performance tableau*:

```
>>> rt.showPerformanceTableau()
    *----  performance tableau -----*
    criteria | weights |   'a01'   'a02'   'a03'    ...   'a12'   'a13'
    ---------|-------------------------------------------------------------
       'g1'  |   1     |   14.57   45.49   77.08    ...   93.30   94.71
       'g2'  |   1     |   33.54   30.94   76.80    ...   55.54   90.12
       'g3'  |   1     |   81.80   16.04   64.85    ...   23.72   44.82
       'g4'  |   1     |   63.78   90.23   12.66    ...   52.82   34.33
       'g5'  |   1     |   85.42   36.30   48.36    ...   76.70   51.36
       'g6'  |   1     |   49.35   58.27   14.72    ...   21.91   30.99
       'g7'  |   1     |   62.12   65.08   74.87    ...   38.98   93.64
>>> ...
```

computeActionCriterionPerformanceDifferences(*refAction*, *refCriterion*, *comments=False*, *Debug=False*)
computes the performances differences observed between the reference action and the others on the given criterion

computeActionCriterionQuantile(*action*, *criterion*, *Debug=False*)
renders the quantile of the performance of action on criterion

computeActionQuantile(*action*, *Debug=True*)
renders the overall performance quantile of action

computeAllQuantiles(*Sorted=True*, *Comments=False*)
renders a html string showing the table of the quantiles matrix action x criterion

computeCriterionPerformanceDifferences(*c*, *Comments=False*, *Debug=False*)
Renders the ordered list of all observed performance differences on the given criterion.

computeDefaultDiscriminationThresholds(*criteriaList=None*, *quantile={'ind': 10, 'pref': 20, 'veto': 80, 'weakVeto': 60}*, *Debug=False*, *Comments=False*)
updates the discrimination thresholds with the percentiles from the performance differences. Parameters: quantile = {'ind': 10, 'pref': 20, 'weakVeto': 60, 'veto: 80}.

computeMinMaxEvaluations(*criteria=None*, *actions=None*)
renders minimum and maximum performances on each criterion in dictionary form: {'g': {'minimum': x, 'maximum': x}}

computeNormalizedDiffEvaluations(*lowValue=0.0*, *highValue=100.0*, *withOutput=False*, *Debug=False*)
renders and csv stores (withOutput=True) the list of normalized evaluation differences observed on the family of criteria Is only adequate if all criteria have the same evaluation scale. Therefore the performance tableau is normalized to 0.0-100.0 scales.

computePerformanceDifferences(*Comments=False*, *Debug=False*, *NotPermanentDiffs=True*, *WithMaxMin=False*)
Adds to the criteria dictionary the ordered list of all observed performance differences.

computeQuantileOrder(*q0=3*, *q1=0*, *Threading=False*, *nbrOfCPUs=1*, *Comments=False*)
Renders a linear ordering of the decision actions from a simulation of pre-ranked outranking digraphs.

The pre-ranking simulations range by default from quantiles=q0 to quantiles=min(100, max(10,len(self.actions)/10])).

The actions are ordered along a decreasing Borda score of their ranking results.

computeQuantilePreorder(*Comments=True*, *Debug=False*)
computes the preorder of the actions obtained from decreasing majority quantiles. The quantiles are re-computed with a call to the self.computeQuantileSort() method.

computeQuantileRanking(*q0=3*, *q1=0*, *Threading=False*, *nbrOfCPUs=1*, *Comments=False*)
Renders a linear ranking of the decision actions from a simulation of pre-ranked outranking digraphs.

The pre-ranking simulations range by default from quantiles=q0 to qantiles=min(100, max(10,len(self.actions)/10)).

The actions are ordered along an increasing Borda score of their ranking results.

computeQuantileSort()
shows a sorting of the actions from decreasing majority quantiles

computeQuantiles(*Debug=False*)
renders a quantiles matrix action x criterion with the performance quantile of action on criterion

computeThresholdPercentile(*criterion*, *threshold*, *Debug=False*)
computes for a given criterion the quantile of the performance differences of a given constant threshold.

computeVariableThresholdPercentile(*criterion*, *threshold*, *Debug=False*)
computes for a given criterion the quantile of the performance differences of a given threshold.

computeWeightPreorder()
renders the weight preorder following from the given criteria weights in a list of increasing equivalence lists of criteria.

computeWeightedAveragePerformances(*isNormalized=False*, *lowValue=0.0*, *highValue=100.0*, *isListRanked=False*)
Compute normalized weighted average scores Normalization transforms by default all the scores into a common 0-100 scale. A lowValue and highValue parameter can be provided for a specific normalisation.

convert2BigData()
Renders a cPerformanceTableau instance, by converting the action keys to integers and evaluations to floats, including the discrimination thresholds, the case given.

convertDiscriminationThresholds2Decimal()

convertDiscriminationThresholds2Float()

convertEvaluation2Decimal()
Convert evaluations from obsolete float format to decimal format

convertEvaluation2Float()
Convert evaluations from decimal format to float

convertInsite2BigData()
Convert in site a standard formated Performance tableau into a bigData formated instance.

convertInsite2Standard()
Convert in site a bigData formated Performance tableau back into a standard formated PerformanceTableau instance.

convertWeight2Decimal()
Convert significance weights from obsolete float format to decimal format.

convertWeight2Integer()
Convert significance weights from Decimal format to int format.

csvAllQuantiles (*fileName='quantiles'*)
save quantiles matrix criterionxaction in CSV format

hasOddWeightAlgebra (*Debug=False*)
Verify if the given criteria[self]['weight'] are odd or not. Return a Boolen value.

htmlPerformanceHeatmap (*argCriteriaList=None, argActionsList=None, SparseModel=False, minimalComponentSize=1, rankingRule='Copeland', quantiles=None, strategy='average', ndigits=2, ContentCentered=True, colorLevels=None, pageTitle='Performance Heatmap', Correlations=False, Threading=False, nbrOfCPUs=1, Debug=False*)
Renders the Brewer RdYlGn 5,7, or 9 levels colored heatmap of the performance table actions x criteria in html format.

See the corresponding perfTabs.showHTMLPerformanceHeatMap() method.

htmlPerformanceTable (*actions=None, isSorted=False, Transposed=False, ndigits=2, ContentCentered=True, title=None*)
Renders the performance table citerion x actions in html format.

mpComputePerformanceDifferences (*NotPermanentDiffs=True, nbrCores=None, Debug=False*)
Adds to the criteria dictionary the ordered list of all observed performance differences.

normalizeEvaluations (*lowValue=0.0, highValue=100.0, Debug=False*)
recode the evaluations between lowValue and highValue on all criteria

restoreOriginalEvaluations (*lowValue=0.0, highValue=100.0, Debug=False*)
recode the evaluations to their original values on all criteria

save (*fileName='tempperftab', isDecimal=True, valueDigits=2*)
Persistant storage of Performance Tableaux.

saveCSV (*fileName='tempPerfTab', Sorted=True, criteriaList=None, actionsList=None, ndigits=2, Debug=False*)
1 Store the performance Tableau self Actions x Criteria in CSV format.

saveXMCDA (*fileName='temp', category='New XMCDA Rubis format', user='digraphs Module (RB)', version='saved from Python session', variant='Rubis', valuationType='standard', servingD3=True*)
save performance tableau object self in XMCDA format.

saveXMCDA2 (*fileName='temp', category='XMCDA 2.0 Extended format', user='digraphs Module (RB)', version='saved from Python session', title='Performance Tableau in XMCDA-2.0 format.', variant='Rubis', valuationType='bipolar', servingD3=False, isStringIO=False, stringNA='NA', comment='produced by saveXMCDA2()', hasVeto=True*)
save performance tableau object self in XMCDA 2.0 format including decision objectives, the case given.

saveXMCDA2String (*fileName='temp', category='XMCDA 2.0 format', user='digraphs Module (RB)', version='saved from Python session', title='Performance Tableau in XMCDA-2.0 format.', variant='Rubis', valuationType='bipolar', servingD3=True, comment='produced by stringIO()', stringNA='NA'*)
save performance tableau object self in XMCDA 2.0 format. !!! obsolete: replaced by the isStringIO in the saveXMCDA2 method !!!

saveXML (*name='temp', category='standard', subcategory='standard', author='digraphs Module (RB)', reference='saved from Python'*)
save temporary performance tableau self in XML format.

saveXMLRubis (*name='temp', category='Rubis', subcategory='new D2 version', author='digraphs Module (RB)', reference='saved from Python'*)
save temporary performance tableau self in XML Rubis format.

showActions(*Alphabetic=False*)
presentation methods for decision actions or alternatives

showAll()
Show fonction for performance tableau

showAllQuantiles(*Sorted=True*)
prints the performance quantiles tableau in the session console.

showCriteria(*IntegerWeights=False*, *Alphabetic=False*, *ByObjectives=False*, *Debug=False*)
print Criteria with thresholds and weights.

showEvaluationStatistics()
renders the variance and standard deviation of the values observed in the performance Tableau.

showHTMLPerformanceHeatmap(*actionsList=None*, *criteriaList=None*, *colorLevels=7*, *pageTitle=None*, *ndigits=2*, *SparseModel=False*, *minimalComponentSize=1*, *rankingRule='Copeland'*, *quantiles=None*, *strategy='average'*, *Correlations=False*, *Threading=False*, *nbrOfCPUs=None*, *Debug=False*)
shows the html heatmap version of the performance tableau in a browser window (see perfTabs.htmlPerformanceHeatMap() method).

Parameters:

- *actionsList* and *criteriaList*, if provided, give the possibility to show the decision alternatives, resp. criteria, in a given ordering.
- *ndigits* = 0 may be used to show integer evaluation values.
- If no *actionsList* is provided, the decision actions are ordered from the best to the worst. This ranking is obtained by default with the Copeland rule applied on a standard *BipolarOutrankingDigraph*. When the *SparseModel* flag is put to *True*, a sparse *PreRankedOutrankingDigraph* construction is used instead.
- The *minimalComponentSize* allows to control the fill rate of the pre-ranked model. If *minimalComponentSize* = *n* (the number of decision actions) both the pre-ranked model will be in fact equivalent to the standard model.
- It may interesting in some cases to use *rankingRule* = 'NetFlows'.
- Quantiles used for the pre-ranked decomposition are put by default to *n* (the number of decision alternatives) for $n < 50$. For larger cardinalities up to 1000, quantiles = *n* /10. For bigger performance tableaux the *quantiles* parameter may be set to a much lower value not exceeding usually 1000.
- The pre-ranking may be obtained with three ordering strategies for the quantiles equivalence classes: 'average' (default), 'optimistic' or 'pessimistic'.
- With *Correlations* = *True* and *criteriaList* = *None*, the criteria will be presented from left to right in decreasing order of the correlations between the marginal criterion based ranking and the global ranking used for presenting the decision alternatives.
- For large performance Tableaux, *multiprocessing* techniques may be used by setting *Threading* = *True* in order to speed up the computations; especially when *Correlations* = *True*.
- By default, the number of cores available, will be detected. It may be efficient in a HPC context to indicate the exact number of singled threaded cores in fact allocated to the job.

```
>>> from randomPerfTabs import RandomPerformanceTableau
>>> rt = RandomPerformanceTableau(seed=100)
>>> rt.showHTMLPerformanceHeatmap(colorLevels=5,Correlations=True)
```

Heatmap of Performance Tableau 'randomperftab'

criteria	g6	g3	g4	g2	g1	g5	g7
weights	1	1	1	1	1	1	1
tau(*)	0.42	0.25	0.17	0.03	0.03	-0.06	-0.15
a07	75.39	77.35	71.73	34.70	80.00	65.15	11.40
a01	49.35	81.80	63.78	33.54	14.57	85.42	62.12
a11	59.54	91.38	59.04	95.61	4.79	71.47	20.91
a05	86.13	0.56	96.85	17.85	73.20	81.38	33.37
a03	14.72	64.85	12.66	76.80	77.08	48.36	74.87
a08	70.62	56.62	77.50	62.63	53.29	25.25	19.28
a04	67.60	12.41	55.40	20.39	70.55	76.15	56.82
a12	21.91	23.72	52.82	55.54	93.30	76.70	38.98
a13	30.99	44.82	34.33	90.12	94.71	51.36	93.64
a02	58.27	16.04	90.23	30.94	45.49	36.30	65.08
a09	12.10	19.26	50.71	96.33	8.02	84.74	52.53
a10	5.07	84.12	28.99	21.08	45.59	90.93	72.01
a06	16.47	39.55	60.91	18.86	43.35	89.05	1.25

Color legend:

quantile	0.20%	0.40%	0.60%	0.80%	1.00%

(*) tau: Ordinal (Kendall) correlation between marginal criterion and global ranking relation.

showHTMLPerformanceQuantiles(*Sorted=True*)
shows the performance quantiles tableau in a browser window.

showHTMLPerformanceTableau(*actionsSubset=None*, *isSorted=True*, *Transposed=False*, *ndigits=2*, *ContentCentered=True*, *title=None*)
shows the html version of the performance tableau in a browser window.

showObjectives()

showPairwiseComparison(*a*, *b*, *hasSymetricThresholds=True*, *Debug=False*, *isReturningHTML=False*, *hasSymmetricThresholds=True*)
renders the pairwise comprison parameters on all criteria in html format

showPerformanceTableau(*actionsSubset=None*, *Sorted=True*, *ndigits=2*)
Print the performance Tableau.

showQuantileSort(*Debug=False*)
Wrapper of computeQuantilePreorder() for the obsolete showQuantileSort() method.

showStatistics(*Debug=False*)
show statistics concerning the evaluation distributions on each criteria.

to_JSON()
Convert the performance table .__dict__ into a JSON string

class perfTabs.**XMCDA2PerformanceTableau**(*fileName='temp'*, *HasSeparatedWeights=False*, *HasSeparatedThresholds=False*, *stringInput=None*, *Debug=False*)
Bases: *perfTabs.PerformanceTableau*

Specialization of the general PerformanceTableau class for reading stored XMCDA 2.0 formatted instances with exact decimal numbers. Using the inbuilt module xml.etree (for Python 2.5+).

Parameters:

- fileName is given without the extension `.xml` or `.xmcda`,
- HasSeparatedWeights in XMCDA 2.0.0 encoding (default = False),
- HasSeparatedThresholds in XMCDA 2.0.0 encoding (default = False),
- stringInput: instantiates from an XMCDA 2.0 encoded string argument.

Back to the *Installation*

2.2.7 performanceQuantiles module

Digraph3 collection of python3 modules for Algorithmic Decision Theory applications

Module for incremental performance quantiles computation

class performanceQuantiles.**PerformanceQuantiles**(*perfTab=None*, *numberOfBins=4*, *LowerClosed=True*, *filePerfQuant=None*, *Debug=False*)

Bases: *perfTabs.PerformanceTableau*

Implements the incremental performance quantiles representation of a given performance tableau.

NumberOfBins may be either 'quartiles', 'deciles', ... or 'n', the integer number of bins.

Example python session:

```
>>> import performanceQuantiles
>>> from randomPerfTabs import RandomCBPerformanceTableau
>>> from randomPerfTabs import RandomCBPerformanceGenerator as␣
↪PerfTabGenerator
>>> nbrActions=1000
>>> nbrCrit = 7
>>> tp = RandomCBPerformanceTableau(numberOfActions=nbrActions,
...                                 numberOfCriteria=nbrCrit,seed=105)
>>> pq = performanceQuantiles.PerformanceQuantiles(tp,'quintiles',
...                                 LowerClosed=True,Debug=False)
>>> pq.showLimitingQuantiles(ByObjectives=True)
*----  performance quantiles -----*
Costs
criteria  | weights |  '0.0'   '0.25'  '0.5'   '0.75'  '1.0'
 ---------|-------------------------------------------------
    'c1'  |   6     | -97.12  -65.70  -46.08  -24.96   -1.85
Benefits
criteria  | weights |  '0.0'   '0.25'  '0.5'   '0.75'  '1.0'
 ---------|-------------------------------------------------
    'b1'  |   1     |   2.11   32.42   53.25   73.44   98.69
```

```
    'b2'  |   1     |   0.00    3.00    5.00    7.00   10.00
    'b3'  |   1     |   1.08   34.64   54.80   73.24   97.23
    'b4'  |   1     |   0.00    3.00    5.00    7.00   10.00
    'b5'  |   1     |   1.84   34.25   55.11   74.62   96.40
    'b6'  |   1     |   0.00    3.00    5.00    7.00   10.00
>>> tpg = PerfTabGenerator(tp,seed=105)
>>> newActions = tpg.randomActions(100)
>>> pq.updateQuantiles(newActions,historySize=None)
>>> pq.showHTMLLimitingQuantiles(Transposed=True)
```

Performance quantiles

Sampling sizes between 994 and 1004.

criterion	0.00	0.25	0.50	0.75	1.00
b1	2.11	24.39	49.80	69.75	98.69
b2	0.00	5.05	6.57	7.84	10.00
b3	1.08	53.61	59.43	80.16	97.23
b4	0.00	5.10	5.89	6.52	10.00
b5	1.84	32.00	39.16	59.81	96.40
b6	0.00	3.96	4.41	7.65	10.00
c1	-97.12	-73.58	-59.89	-42.08	-1.85

computeQuantileProfile (*p*, *qFreq=None*, *Debug=False*)
Renders the quantile *q(p)* on all the criteria.

save (*fileName='tempPerfQuant'*, *valueDigits=2*)
Persistant storage of a PerformanceQuantiles instance.

showActions ()

showCriteria (*IntegerWeights=False*, *Alphabetic=False*, *ByObjectives=True*, *Debug=False*)
print Criteria with thresholds and weights.

showCriterionStatistics (*g*, *Debug=False*)
show statistics concerning the evaluation distributions on each criteria.

showHTMLLimitingQuantiles (*Sorted=True*, *Transposed=False*, *ndigits=2*, *ContentCentered=True*, *title=None*)
shows the html version of the limiting quantiles in a browser window.

showLimitingQuantiles (*ByObjectives=False*, *Sorted=False*, *ndigits=2*)
Prints the performance quantile limits in table format: criteria x limits.

updateQuantiles (*newData*, *historySize=None*)
Update the PerformanceQuantiles with a set of new random decision actions. Parameter *historysize* allows to take more or less into account the historical situtaion. For instance, *historySize=0* does not take into account at all any past observations. Otherwise, if *historySize=None* (the default setting), the new observations become less and less influential compared to the historical data.

Back to the *Installation*

2.2.8 randomPerfTabs module

A tutorial with coding examples is available here: *Generating random performance tableaux*

class randomPerfTabs.**Random3ObjectivesPerformanceGenerator**(*argPerfTab*, *actionNamePrefix='a'*, *instanceCounter=0*, *seed=None*, *Debug=False*)

Bases: *randomPerfTabs.RandomPerformanceGenerator*

Generates and/or new decision actions with random evaluation for a given Random3ObjectivesPerformanceTableau instance.

class randomPerfTabs.**Random3ObjectivesPerformanceTableau**(*numberOfActions=20*, *numberOfCriteria=13*, *weightDistribution='equiobjectives'*, *weightScale=None*, *IntegerWeights=True*, *OrdinalScales=False*, *commonScale=None*, *commonThresholds=None*, *commonMode=None*, *valueDigits=2*, *vetoProbability=0.5*, *missingDataProbability=0.05*, *BigData=False*, *seed=None*, *Debug=False*)

Bases: *perfTabs.PerformanceTableau*

Specialization of the PerformanceTableau for 3 objectives: *Eco*, *Soc* and *Env*.

Each decision action is qualified at random as weak (-), fair (~) or good (+) on each of the three objectives.

Generator arguments:

- numberOf Actions := 20 (default)
- number of Criteria := 13 (default)
- weightDistribution := 'equiobjectives' (default)

  'equisignificant' (weights set all to 1)
  'random' (in the range 1 to numberOfCriteria)
- weightScale := [1,numerOfCriteria] (random default)
- IntegerWeights := True (default) / False
- OrdinalScales := True / False (default), if True commonScale is set to (0,10)
- commonScale := (Min, Max)

  when common Scale = False, (0.0,10.0) by default if OrdinalScales == True and CommonScale=None,
  and (0.0,100.0) by default otherwise
- commonThresholds := ((Ind,Ind_slope),(Pref,Pref_slope),(Veto,Veto_slope)) with

  Ind < Pref < Veto in [0.0,100.0] such that

(Ind/100.0*span + Ind_slope*x) < (Pref/100.0*span + Pref_slope*x) < (Pref/100.0*span + Pref_slope*x)

By default [(0.05*span,0.0),(0.10*span,0.0),(0.60*span,0.0)] if OrdinalScales=False

By default [(0.1*span,0.0),(0.2*span,0.0),(0.8*span,0.0)] otherwise

with span = commonScale[1] - commonScale[0].

- commonMode := ['triangular','variable',0.50] (default), A constant mode may be provided.

  ['uniform','variable',None], a constant range may be provided.

  ['beta','variable',None] (three alpha, beta combinations:

  (5.8661,2.62203),(5.05556,5.05556) and (2.62203, 5.8661)

  chosen by default for 'good', 'fair' and 'weak' evaluations.

  Constant parameters may be provided.

- valueDigits := 2 (default, for cardinal scales only)
- vetoProbability := x in]0.0-1.0[(0.5 default), probability that a cardinal criterion shows a veto preference discrimination threshold.
- Debug := True / False (default)

showActions(*Alphabetic=False*)

showObjectives()

class randomPerfTabs.**RandomCBPerformanceGenerator**(*argPerfTab*, *actionNamePrefix='a'*, *instanceCounter=None*, *seed=None*)

Bases: *randomPerfTabs.RandomPerformanceGenerator*

Instantiates a generator of new decision actions with associated random evaluations using the model parameters provided by a given RandomCBPerformanceTableau instance.

class randomPerfTabs.**RandomCBPerformanceTableau**(*numberOfActions=13*, *numberOfCriteria=7*, *name='randomCBperfTab'*, *weightDistribution='equiobjectives'*, *weightScale=None*, *IntegerWeights=True*, *NegativeWeights=False*, *commonScale=None*, *commonThresholds=None*, *commonPercentiles=None*, *samplingSize=100000*, *commonMode=None*, *valueDigits=2*, *missingDataProbability=0.01*, *BigData=False*, *seed=None*, *Threading=False*, *nbrCores=None*, *Debug=False*, *Comments=False*)

Bases: *perfTabs.PerformanceTableau*

Full automatic generation of random Cost versus Benefit oriented performance tableaux.

Parameters:

- If numberOfActions == None, a uniform random number between 10 and 31 of cheap, neutral or advantageous actions (equal 1/3 probability each type) actions is instantiated
- If numberOfCriteria == None, a uniform random number between 5 and 21 of cost or benefit criteria. Cost criteria have probability 1/3, whereas benefit criteria respectively 2/3 probability to be generated. However, at least one criterion of each kind is always instantiated.

- weightDistribution := {'equiobjectives'|'fixed'|'random'|'equisignificant'} By default, the sum of significance of the cost criteria is set equal to the sum of the significance of the benefit criteria.
- default weightScale for 'random' weightDistribution is 1 - numberOfCriteria.
- commonScale parameter is not used. The scale of cost criteria is cardinal or ordinal (0-10) with proabailities 1/4 respectively 3/4, whereas the scale of benefit criteria is ordinal or cardinal with probabilities 2/3, respectively 1/3.
- All cardinal criteria are evaluated with decimals between 0.0 and 100.0 wheras all ordinal criteria are evaluated with integers between 0 and 10.
- commonThresholds parameter is not used. Preference discrimination is specified as percentiles of concerned performance differences (see below).
- CommonPercentiles = {'ind':0.05, 'pref':0.10, 'veto':'95} are expressed in percentiles of the observed performance differences and only concern cardinal criteria.

Warning: Minimal number of decision actions required is 3 !

updateDiscriminationThresholds(*Comments=False*, *Debug=False*)
Recomputes performance discrimination thresholds from commonPercentiles.

Note: Overwrites all previous criterion discrimination thresholds !

class randomPerfTabs.**RandomPerformanceGenerator**(*argPerfTab*, *actionNamePrefix='a'*, *instanceCounter=None*, *seed=None*)
Bases: object

Wrapper for generating new decision actions with random evaluation for a given RandomPerformanceTableau instance.

randomActions(*nbrOfRandomActions=1*)
Generates nbrOfRandomActions.

randomPerformanceTableau(*nbrOfRandomActions=1*)
Generates nbrOfRandomActions.

class randomPerfTabs.**RandomPerformanceTableau**(*numberOfActions=13*, *actionNamePrefix='a'*, *numberOfCriteria=7*, *weightDistribution='equisignificant'*, *weightScale=None*, *IntegerWeights=True*, *commonScale=(0.0, 100.0)*, *commonThresholds=((2.5, 0.0), (5.0, 0.0), (80.0, 0.0))*, *commonMode=('beta', None, (2, 2))*, *valueDigits=2*, *missingDataProbability=0.025*, *BigData=False*, *seed=None*, *Debug=False*)
Bases: *perfTabs.PerformanceTableau*

Specialization of the generic perftabs.PerformanceTableau class for generating a temporary random performance tableau.

Parameters:

- numberOfActions := nbr of decision actions.
- numberOfCriteria := number performance criteria.

- weightDistribution := 'random' (default) | 'fixed' | 'equisignificant'.

  If 'random', weights are uniformly selected randomly
  form the given weight scale;
  If 'fixed', the weightScale must provided a corresponding weights
  distribution;
  If 'equisignificant', all criterion weights are put to unity.

- weightScale := [Min,Max] (default =[1,numberOfCriteria].
- IntegerWeights := True (default) | False (normalized to proportions of 1.0).
- commonScale := [Min;Max]; common performance measuring scales (default = [0;100])
- commonThresholds := [(q0,q1),(p0,p1),(v0,v1)]; common indifference(q), preference (p) and considerable performance difference discrimination thresholds. q0, p0 and v0 are expressed in percentige of the common scale amplitude: Max - Min.
- commonMode := common random distribution of random performance measurements:

  ('uniform',None,None), uniformly distributed between min and max values.
  ('normal',mu,sigma), truncated Gaussion distribution.
  ('triangular',mode,repartition), generalized triangular distribution
  ('beta',mod,(alpha,beta)), mode in]0,1[.

- valueDigits := <integer>, precision of performance measurements (2 decimal digits by default).

Code example:

```
>>> from randomPerfTabs import RandomPerformanceTableau
>>> t = RandomPerformanceTableau(numberOfActions=3,numberOfCriteria=1,
↪seed=100)
>>> t.actions
    {'a1': {'comment': 'RandomPerformanceTableau() generated.', 'name':
↪'random decision action'},
     'a2': {'comment': 'RandomPerformanceTableau() generated.', 'name':
↪'random decision action'},
     'a3': {'comment': 'RandomPerformanceTableau() generated.', 'name':
↪'random decision action'}}
>>> t.criteria
    {'g1': {'thresholds': {'ind' : (Decimal('10.0'), Decimal('0.0')),
                           'veto': (Decimal('80.0'), Decimal('0.0')),
                           'pref': (Decimal('20.0'), Decimal('0.0'))},
            'scale': [0.0, 100.0],
            'weight': Decimal('1'),
            'name': 'digraphs.RandomPerformanceTableau() instance',
            'comment': 'Arguments: ; weightDistribution=random;
                weightScale=(1, 1); commonMode=None'}}
>>> t.evaluation
    {'g01': {'a01': Decimal('45.95'),
             'a02': Decimal('95.17'),
             'a03': Decimal('17.47')
            }
    }
```

class randomPerfTabs.**RandomRankPerformanceTableau**(*numberOfActions=13*, *numberOfCriteria=7*, *weightDistribution='equisignificant'*, *weightScale=None*, *commonThresholds=None*, *IntegerWeights=True*, *BigData=False*, *seed=None*, *Debug=False*)

Bases: *perfTabs.PerformanceTableau*

Specialization of the PerformanceTableau class for generating a temporary random performance tableau.

Random generator for multiple criteria ranked (without ties) performances of a given number of decision actions. On each criterion, all decision actions are hence lineraly ordered. The RandomRankPerformanceTableau class is matching the RandomLinearVotingProfiles class (see the votingDigraphs module)

Parameters:

- number of actions,
- number of performance criteria,
- weightDistribution := equisignificant | random (default, see RandomPerformanceTableau)
- weightScale := (1, 1 | numberOfCriteria (default when random))
- IntegerWeights := Boolean (True = default)
- commonThresholds (default) := {

  'ind':(0,0),
  'pref':(1,0),
  'veto':(numberOfActions,0)
  } (default)

class randomPerfTabs.**RandomStdPerformanceGenerator**(*argPerfTab*, *actionNamePrefix='a'*, *instanceCounter=0*, *seed=None*)

Bases: *randomPerfTabs.RandomPerformanceGenerator*

Generates for a given standard RandomPerformanceTableau instance.

Back to the *Installation*

2.2.9 outrankingDigraphs module

A tutorial with coding examples is available here: *Working with the outrankingDigraphs module*

Python implementation of outranking digraphs.

class outrankingDigraphs.**BipolarIntegerOutrankingDigraph**(*argPerfTab=None*, *coalition=None*, *hasBipolarVeto=True*, *hasSymmetricThresholds=True*)

Bases: *outrankingDigraphs.BipolarOutrankingDigraph*, *perfTabs.PerformanceTableau*

Parameters:

performanceTableau (fileName of valid py code)

optional, coalition (sublist of criteria)

Specialization of the standard OutrankingDigraph class for generating bipolar integer-valued outranking digraphs.

savePy2Gprolog(*name='temp'*)
save digraph in gprolog version

showRelation()
prints the relation valuation in ##.## format.

class outrankingDigraphs.**BipolarOutrankingDigraph**(*argPerfTab=None*, *coalition=None*, *actionsSubset=None*, *hasNoVeto=False*, *hasBipolarVeto=True*, *Normalized=False*, *CopyPerfTab=True*, *BigData=False*, *Threading=False*, *tempDir=None*, *WithConcordanceRelation=True*, *WithVetoCounts=True*, *nbrCores=None*, *Debug=False*, *Comments=False*)

Bases: *outrankingDigraphs.OutrankingDigraph*

Specialization of the abstract OutrankingDigraph root class for generating bipolarly-valued outranking digraphs.

Parameters:

- argPerfTab: instance of PerformanceTableau class. If a file name string is given, the performance tableau will directly be loaded first.
- coalition: subset of criteria to be used for contructing the outranking digraph.
- hasNoVeto: veto desactivation flag (False by default).
- hasBipolarVeto: bipolar versus electre veto activation (true by default).
- Normalized: the valuation domain is set by default to [-100,+100] (bipolar percents). If True, the valuation domain is recoded to [-1.0,+1.0].
- WithConcordanceRelation: True by default when not threading. The self.concordanceRelation contains the significance majority margin of the "at least as good relation as" without the large performance difference polarization.
- WithVetoCounts: True by default when not threading. All vetos and countervetos are stored in self.vetos and self.negativeVetos slots, as well the counts of large performance differences in self.largePerformanceDifferencesCount slot.
- Threading: False by default. Allows to profit from SMP machines via the Python multiprocessing module.

- nbrCores: controls the maximal number of cores that will be used in the multiprocessing phases. If None is given, the os.cpu_count method is used in order to determine the number of availble cores on the SMP machine.

Warning: If Threading is True, WithConcordanceRelation and WithVetoCounts flags are automatically set both to False.

computeCriterionRelation(*c*, *a*, *b*, *hasSymmetricThresholds=True*)
Compute the outranking characteristic for actions x and y on criterion c.

computeSingleCriteriaNetflows()
renders the Promethee single criteria netflows matrix M

criterionCharacteristicFunction(*c*, *a*, *b*, *hasSymmetricThresholds=True*)
Renders the characteristic value of the comparison of a and b on criterion c.

saveSingleCriterionNetflows(*fileName='tempnetflows.prn'*, *delimiter=' '*, *Comments=True*)
Delimited save of single criteria netflows matrix

class outrankingDigraphs.**ConfidentBipolarOutrankingDigraph**(*argPerfTab=None*, *distribution='triangular'*, *betaParameter=2*, *confidence=90.0*, *coalition=None*, *hasNoVeto=False*, *hasBipolarVeto=True*, *Normalized=True*, *Threading=False*, *nbrOfCPUs=1*, *Debug=False*)

Bases: *outrankingDigraphs.BipolarOutrankingDigraph*

Confident bipolar outranking digraph based on multiple criteria of uncertain significance.

The digraph's bipolar valuation represents the bipolar outranking relation based on a sufficient likelihood of the at least as good as relation that is outranking without veto and counterveto.

By default, each criterion i' significance weight is supposed to be a triangular random variable of mode w_i in the range 0 to 2*w_i.

Parameters:

- argPerfTab: PerformanceTableau instance or the name (without extension) of a stored one. If None, a random instance is generated.
- distribution: {triangular|uniform|beta}, probability distribution used for generating random weights
- betaParameter: a = b (default = 2)
- confidence: required likelihood (in %) of the outranking relation
- other standard parameters from the BipolarOutrankingDigraph class (see documentation).

computeCLTLikelihoods(*distribution='triangular'*, *betaParameter=None*, *Threading=False*, *nbrOfCPUs=1*, *Debug=False*)
Renders the pairwise CLT likelihood of the at least as good as relation neglecting all considerable large performance differences polarisations.

showRelationTable(*IntegerValues=False*, *actionsSubset=None*, *Sorted=True*, *LikelihoodDenotation=True*, *hasLatexFormat=False*, *hasIntegerValuation=False*, *relation=None*, *Debug=False*)
prints the relation valuation in actions X actions table format.

class outrankingDigraphs.**DissimilarityOutrankingDigraph**(*argPerfTab=None*)
Bases: *outrankingDigraphs.OutrankingDigraph*, *perfTabs.PerformanceTableau*

Parameters: performanceTableau (fileName of valid py code)

Specialization of the OutrankingDigraph class for generating temporary dissimilarity random graphs

showAll()
specialize the general showAll method for the dissimilarity case

class outrankingDigraphs.**Electre3OutrankingDigraph**(*argPerfTab=None*, *coalition=None*, *hasNoVeto=False*)
Bases: *outrankingDigraphs.OutrankingDigraph*, *perfTabs.PerformanceTableau*

Specialization of the standard OutrankingDigraph class for generating classical Electre III outranking digraphs (with vetoes and no counter-vetoes).

Parameters:

performanceTableau (fileName of valid py code)

optional, coalition (sublist of criteria)

computeCriterionRelation(*c*, *a*, *b*, *hasSymmetricThresholds=False*)
compute the outranking characteristic for actions x and y on criterion c.

computeVetos(*cutLevel=None*, *realVetosOnly=False*)
prints all veto situations observed in the OutrankingDigraph instance.

showVetos(*cutLevel=None*, *realVetosOnly=False*, *Comments=True*)
prints all veto situations observed in the OutrankingDigraph instance.

class outrankingDigraphs.**EquiSignificanceMajorityOutrankingDigraph**(*argPerfTab=None*, *coalition=None*, *hasNoVeto=False*)
Bases: *outrankingDigraphs.BipolarOutrankingDigraph*, *perfTabs.PerformanceTableau*

Parameters: performanceTableau (fileName of valid py code)

Specialization of the general OutrankingDigraph class for temporary outranking digraphs with equisignificant criteria.

class outrankingDigraphs.**MultiCriteriaDissimilarityDigraph**(*perfTab=None*, *filePerfTab=None*)
Bases: *outrankingDigraphs.OutrankingDigraph*, *perfTabs.PerformanceTableau*

Parameters: performanceTableau (fileName of valid py code)

Specialization of the OutrankingDigraph class for generating temporary multiple criteria based dissimilarity graphs.

class outrankingDigraphs.**NewRobustOutrankingDigraph**(*filePerfTab=None*, *Debug=False*, *hasNoVeto=True*)
Bases: *outrankingDigraphs.BipolarOutrankingDigraph*, *perfTabs.PerformanceTableau*

Parameters: performanceTableau (fileName of valid py code)

Specialization of the general OutrankingDigraph class for new robustness anaylsis.

class outrankingDigraphs.**OrdinalOutrankingDigraph**(*argPerfTab=None*, *coalition=None*, *hasNoVeto=False*)

Bases: *outrankingDigraphs.OutrankingDigraph*, *perfTabs.PerformanceTableau*

Parameters: performanceTableau (fileName of valid py code)

Specialization of the general OutrankingDigraph class for temporary ordinal outranking digraphs

class outrankingDigraphs.**OutrankingDigraph**(*file=None*, *order=7*)

Bases: *digraphs.Digraph*, *perfTabs.PerformanceTableau*

Abstract root class for **outranking digraphs** inheritating methods both from the generic *digraphs.Digraph* and from the generic *perfTabs.PerformanceTableau* root classes. As such, our genuine outranking digraph model is a hybrid object appearing on the one side as digraph with a nodes set (the decision alternatives) and a binary relation (outranking situations) and, on the other side, as a performance tableau with a set of decision alternatives, performance criteria and a table of performance measurements.

Provides common methods to all specialized models of outranking digraphs, the standard outranking digraph model being provided by the *outrankingDigraphs.BipolarOutrankingDigraph* class.

A given object of this class consists at least in:

1. a potential set of decision **actions** : an (ordered) dictionary describing the potential decision actions or alternatives with 'name' and 'comment' attributes,
2. a coherent family of **criteria**: an (ordered) dictionary of criteria functions used for measuring the performance of each potential decision action with respect to the preference dimension captured by each criterion,
3. the **evaluations**: a dictionary of performance evaluations for each decision action or alternative on each criterion function.
4. the digraph **valuationdomain**, a dictionary with three entries: the *minimum* (-100, means certainly no link), the *median* (0, means missing information) and the *maximum* characteristic value (+100, means certainly a link),
5. the **outranking relation** : a double dictionary defined on the Cartesian product of the set of decision alternatives capturing the credibility of the pairwise *outranking situation* computed on the basis of the performance differences observed between couples of decision alternatives on the given family if criteria functions.

Warning: Cannot be called directly ! No __init__(self,...) method defined.

computeAMPLData(*OldValuation=False*)
renders the ampl data list

computeActionsCorrelations()
renders the comparison correlations between the actions

computeCriteriaCorrelationDigraph()
renders the ordinal criteria correlation digraph

computeCriteriaCorrelations()
renders the comparison correlations between the criteria

computeCriterionCorrelation(*criterion*, *Threading=False*, *nbrOfCPUs=None*, *Debug=False*, *Comments=False*)
Renders the ordinal correlation coefficient between the global outranking and the marginal criterion relation.

Uses the digraphs.computeOrdinalCorrelationMP().

Note: Renders a dictionary with the key 'correlation' containing the actual bipolar correlation index and the key 'determination' containing the minimal determination level D of the self outranking and the marginal criterion relation.

D = sum_{x != y} min(abs(self.relation(x,y)),abs(marginalCriterionRelation(x,y)) / n(n-1)

where n is the number of actions considered.

The correlation index with a completely indeterminate relation is by convention 0.0 at determination level 0.0 .

---

computeCriterionRelation(*c*, *a*, *b*)
compute the outranking characteristic for actions x and y on criterion c.

computeMarginalCorrelation(*args*, *Threading=False*, *nbrOfCPUs=None*, *Debug=False*, *Comments=False*)
Renders the ordinal correlation coefficient between the marginal criterion relation and a given normalized outranking relation.

args = (criterion,relation)

computeMarginalVersusGlobalOutrankingCorrelations(*Sorted=True*, *Threading=False*, *nbrCores=None*, *Comments=False*)
Method for computing correlations between each individual criterion relation with the corresponding global outranking relation.

Returns a list of tuples (correlation,criterionKey) sorted by default in decreasing order of the correlation.

If Threading is True, a multiprocessing Pool class is used with a parallel equivalent of the built-in map function.

If nbrCores is not set, the os.cpu_count() function is used to determine the number of available cores.

computeMarginalVersusGlobalRankingCorrelations(*ranking*, *Sorted=True*, *ValuedCorrelation=False*, *Threading=False*, *nbrCores=None*, *Comments=False*)
Method for computing correlations between each individual criterion relation with the corresponding global outranking relation.

Returns a list of tuples (correlation,criterionKey) sorted by default in decreasing order of the correlation.

If Threading is True, a multiprocessing Pool class is used with a parallel equivalent of the built-in map function.

If nbrCores is not set, the os.cpu_count() function is used to determine the number of available cores.

computePairwiseComparisons(*hasSymmetricThresholds=True*)
renders pairwise comparison parameters for all pairs of actions

computePairwiseCompleteComparison(*a*, *b*, *c*)
renders pairwise complete comparison parameters for actions a and b on criterion c.

computeQuantileSortRelation(*Debug=False*)
Renders the bipolar-valued relation obtained from the self quantile sorting result.

computeSingletonRanking(*Comments=False*, *Debug=False*)
Renders the sorted bipolar net determinatation of outrankingness minus outrankedness credibilities of all singleton choices.

res = ((netdet,singleton,dom,absorb)+)

computeVetoesStatistics(*level=None*)
renders the cut level vetos in dictionary format: vetos = {'all': n0, 'strong: n1, 'weak':n2}.

computeVetosShort()
renders the number of vetoes and real vetoes in an OutrankingDigraph.

computeWeightsConcentrationIndex()
Renders the Gini concentration index of the weight distribution

Based on the triangle summation formula.

defaultDiscriminationThresholds(*quantile={'ind': 10, 'pref': 20, 'veto': 80, 'weakVeto': 60}, Debug=False, comments=False*)
updates the discrimination thresholds with the percentiles from the performance differences.

Parameters: quantile = {'ind': 10, 'pref': 20, 'weakVeto': 60, 'veto: 80}.

export3DplotOfActionsCorrelation(*plotFileName='correlation', Type='pdf', Comments=False, bipolarFlag=False, dist=True, centeredFlag=False*)
use Calmat and R for producing a png plot of the principal components of the the actions ordinal correlation table.

export3DplotOfCriteriaCorrelation(*plotFileName='correlation', Type='pdf', Comments=False, bipolarFlag=False, dist=True, centeredFlag=False*)
use Calmat and R for producing a plot of the principal components of the criteria ordinal correlation table.

saveActionsCorrelationTable(*fileName='tempcorr.prn', delimiter=' ', Bipolar=True, Silent=False, Centered=False*)
Delimited save of correlation table

saveCriteriaCorrelationTable(*fileName='tempcorr.prn', delimiter=' ', Bipolar=True, Silent=False, Centered=False*)
Delimited save of correlation table

saveXMCDA2RubisChoiceRecommendation(*fileName='temp', category='Rubis', subcategory='Choice Recommendation', author='digraphs Module (RB)', reference='saved from Python', comment=True, servingD3=False, relationName='Stilde', graphValuationType='bipolar', variant='standard', instanceID='void', stringNA='NA', _OldCoca=True, Debug=False*)
save complete Rubis problem and result in XMCDA 2.0 format with unicode encoding.

saveXMCDAOutrankingDigraph(*fileName='temp', category='Rubis', subcategory='Choice Recommendation', author='digraphs Module (RB)', reference='saved from Python', comment=True, servingD3=False, relationName='Stilde', valuationType='bipolar', variant='standard', instanceID='void'*)
save complete Rubis problem and result in XMCDA format with unicode encoding.

saveXMLRubisOutrankingDigraph(*name='temp', category='Rubis outranking digraph', subcategory='Choice recommendation', author='digraphs Module (RB)', reference='saved from Python', noSilent=False, servingD3=True*)
save complete Rubis problem and result in XML format with unicode encoding.

showAll()
specialize the general showAll method with criteria and performance tableau output

showCriteriaCorrelationTable(*isReturningHTML=False*)
prints the criteriaCorrelationIndex in table format

showCriteriaHierarchy()
shows the Rubis clustering of the ordinal criteria correlation table

showCriterionRelationTable(*criterion*, *actionsSubset=None*)
prints the relation valuation in actions X actions table format.

showMarginalVersusGlobalOutrankingCorrelation(*Sorted=True*, *Threading=False*, *nbrOfCPUs=None*, *Comments=True*)
Show method for computeCriterionCorrelation results.

showPairwiseComparison(*a*, *b*, *Debug=False*, *isReturningHTML=False*, *hasSymmetricThresholds=True*)
renders the pairwise comprison parameters on all criteria in html format

showPairwiseComparisonsDistributions()
show the lt,leq, eq, geq, gt distributions for all pairs

showPairwiseOutrankings(*a*, *b*, *Debug=False*, *isReturningHTML=False*, *hasSymmetricThresholds=True*)

showPerformanceTableau(*actionsSubset=None*)
Print the performance Tableau.

showRelationTable(*IntegerValues=False*, *actionsSubset=None*, *Sorted=True*, *hasLPDDenotation=False*, *hasLatexFormat=False*, *hasIntegerValuation=False*, *relation=None*, *ReflexiveTerms=True*)
prints the relation valuation in actions X actions table format.

showShort()
specialize the general showShort method with the criteria.

showSingletonRanking(*Comments=True*, *Debug=False*)
Calls self.computeSingletonRanking(comments=True,Debug = False). Renders and prints the sorted bipolar net determinatation of outrankingness minus outrankedness credibilities of all singleton choices. res = ((netdet,sigleton,dom,absorb)+)

showVetos(*cutLevel=None*, *realVetosOnly=False*)
prints all veto situations observed in the OutrankingDigraph instance.

class outrankingDigraphs.**PolarisedOutrankingDigraph**(*digraph=None*, *level=None*, *KeepValues=True*, *AlphaCut=False*, *StrictCut=False*)
Bases: *digraphs.PolarisedDigraph*, *outrankingDigraphs.OutrankingDigraph*, *perfTabs.PerformanceTableau*

Specilised *digraphs.PolarisedDigraph* instance for Outranking Digraphs.

Warning: If called with argument *digraph=None*, a RandomBipolarOutrankingDigraph instance is generated first.

class outrankingDigraphs.**RandomBipolarOutrankingDigraph**(*numberOfActions=7, numberOfCriteria=7, weightDistribution='random', weightScale=[1, 10], commonScale=[0.0, 100.0], commonThresholds=[(10.0, 0.0), (20.0, 0.0), (80.0, 0.0), (80.0, 0.0)], commonMode=('uniform', None, None), hasBipolarVeto=True, Normalized=False*)

Bases: *outrankingDigraphs.BipolarOutrankingDigraph*, *perfTabs.PerformanceTableau*

Parameters:

n := nbr of actions, p := number criteria,
scale := [Min,Max], thresholds := [h,q,v]

Specialization of the OutrankingDigraph class for generating temporary Digraphs from random performance tableaux.

class outrankingDigraphs.**RandomElectre3OutrankingDigraph**(*numberOfActions=7, numberOfCriteria=7, weightDistribution='random', weightScale=[1, 10], commonScale=[0.0, 100.0], commonThresholds=[(10.0, 0.0), (20.0, 0.0), (80.0, 0.0)], commonMode=['uniform', None, None]*)

Bases: *outrankingDigraphs.Electre3OutrankingDigraph*, *perfTabs.PerformanceTableau*

Parameters:

n := nbr of actions, p := number criteria, scale := [Min,Max],
thresholds := [h,q,v]

Specialization of the OutrankingDigraph class for generating temporary Digraphs from random performance tableaux.

class outrankingDigraphs.**RandomOutrankingDigraph**(*numberOfActions=7, numberOfCriteria=7, weightDistribution='random', weightScale=[1, 10], commonScale=[0.0, 100.0], commonThresholds=[(10.0, 0.0), (20.0, 0.0), (80.0, 0.0), (80.0, 0.0)], commonMode=('uniform', None, None), hasBipolarVeto=True, Normalized=False*)

Bases: *outrankingDigraphs.RandomBipolarOutrankingDigraph*

Dummy for obsolete RandomOutrankingDigraph Class

class outrankingDigraphs.**RobustOutrankingDigraph**(*filePerfTab=None*, *Debug=False*, *hasNoVeto=True*)

Bases: *outrankingDigraphs.BipolarOutrankingDigraph*, *perfTabs.PerformanceTableau*

Parameters: performanceTableau (fileName of valid py code)

Specialization of the general OutrankingDigraph class for robustness anaylsis.

saveAMPLDataFile(*name='temp'*, *Unique=False*, *Comments=True*)
save the ampl reverse data for cplex

saveXMLRubisOutrankingDigraph(*name='temp'*, *category='Rubis outranking robustness digraph'*, *subcategory='Choice recommendation'*, *author='digraphs Module (RB)'*, *reference='saved from Python'*, *comment=True*, *servingD3=True*)
save complete robust Rubis problem and result in XML format with unicode encoding.

showRelationTable()
specialisation for integer values

class outrankingDigraphs.**RubisRestServer**(*host='http://leopold-loewenheim.uni.lu/cgi-bin/xmlrpc_cgi.py'*, *Debug=False*)

Bases: xmlrpc.client.ServerProxy

xmlrpc-cgi Proxy Server for accessing on-line a Rubis Rest Solver.

Example Python3 session:

```
>>> from outrankingDigraphs import RubisRestServer
>>> solver = RubisRestServer()
>>> solver.ping()
************************************************
* This is the Leopold-Loewenheim Apache Server  *
* of the University of Luxembourg.              *
* Welcome to the Rubis XMCDA 2.0 Web service    *
* R. Bisdorff (c) 2009-2013                     *
* November 2013, version REST/D4 1.1            *
************************************************
>>> from perfTabs import RandomCBPerformanceTableau
>>> t = RandomCBPerformanceTableau(numberOfActions=5,numberOfCriteria=7)
>>> solver.submitProblem(t)
The problem submission was successful !
Server ticket: l4qfAP0RfBBvyjsL
>>> solver.showHTMLSolution()
Created new window in existing browser session.
>>> solver.saveXMCDA2Solution()
The solution request was successful.
Saving XMCDA 2.0 encoded solution in file Solutionl4qfAP0RfBBvyjsL.xml
>>> ...
```

ping(*Debug=False*)

saveXMCDA2Solution(*fileName=None*, *Debug=False*)
Save the solution in XMCDA 2.0 encoding.

showHTMLSolution(*ticket=None*, *valuation='bipolar'*)
Show XMCDA 2.0 solution in a default browser window. The valuation parameter may set the correct style sheet.

Parameter:

- valuation: 'bipolar' or 'robust'. By default the valuation type is set automatically at problem submission.

submitProblem(*perfTab*, *valuation='bipolar'*, *hasVeto=True*, *argTitle='XMCDA 2.0 encoding'*, *Debug=False*)
Submit PerformanceTableau class instances.

Parameters:

- valuation: 'bipolar', 'robust', 'integer'
- hasVeto: Switch on or off vetoes
- argTitle: set specific application title

submitXMCDA2Problem(*fileName*, *valuation=None*, *Debug=False*)
Submit stored XMCDA 2.0 encoded performance tableau.

> **Warning:** An <_.xml> file extension is assumed !

class outrankingDigraphs.**StochasticBipolarOutrankingDigraph**(*argPerfTab=None*, *sampleSize=50*, *samplingSeed=None*, *distribution='triangular'*, *spread=1.0*, *likelihood=0.9*, *coalition=None*, *hasNoVeto=False*, *hasBipolarVeto=True*, *Normalized=False*, *Debug=False*, *SeeSampleCounter=False*)
Bases: *outrankingDigraphs.BipolarOutrankingDigraph*

Stochastic bipolar outranking digraph based on multiple criteria of uncertain significance.

The digraph's bipolar valuation represents the median of sampled outranking relations with a sufficient likelihood (default = 90%) to remain positive, repectively negative, over the possible criteria significance ranges.

Each criterion i' significance weight is supposed to be a triangular random variables of mode w_i in the range 0 to 2*w_i.

Parameters:

- argPerfTab: PerformanceTableau instance or the name of a stored one. If None, a random instance is generated.
- sampleSize: number of random weight vectors used for Monte Carlo simulation.
- distribution: {triangular|extTriangular|uniform|beta(2,2)|beta(4,4)}, probability distribution used for generating random weights
- spread: weight range = weight mode +- (weight mode * spread)
- likelihood: 1.0 - frequency of valuations of opposite sign compared to the median valuation.
- other standard parameters from the BipolarOutrankingDigraph class (see documentation).

computeCDF(*x*, *y*, *rValue*)
computes by interpolation the likelihood of a given rValue with respect to the sampled r(x,y) valuations.

Parameters:

- action key x
- action key y
- r(x,y)

computeCLTLikelihoods(*distribution='triangular'*, *Debug=False*)
Renders the pairwise CLT likelihood of the at least as good as relation neglecting all considerable large performance differences polarisations.

showRelationStatistics(*argument='likelihoods'*, *actionsSubset=None*, *hasLatexFormat=False*, *Bipolar=False*)
prints the relation statistics in actions X actions table format.

showRelationTable(*IntegerValues=False*, *actionsSubset=None*, *hasLPDDenotation=False*, *hasLatexFormat=False*, *hasIntegerValuation=False*, *relation=None*)
specialising BipolarOutrankingDigraph.showRelationTable() for stochstic instances.

class outrankingDigraphs.**UnanimousOutrankingDigraph**(*argPerfTab=None*, *coalition=None*, *hasNoVeto=False*)
Bases: *outrankingDigraphs.OutrankingDigraph*, *perfTabs.PerformanceTableau*

Parameters: performanceTableau (fileName of valid py code)

Specialization of the general OutrankingDigraph class for temporary unanimous outranking digraphs

Back to the *Installation*

2.2.10 xmcda module

xmcda.**saveRobustRubisChoiceXSL**(*fileName='xmcda2RubisRobustChoice.xsl'*)
Save the robust version of the Rubis Best-Choice XMCDA 2.0 XSL style sheet in the current working directory This style sheet allows to browse an XMCDA 2.0 encoded robust Rubis Best-Choice recommendation.

Note: When accessing an XMCDA encoded data file in a browser, for safety reasons, the corresponding XSL style sheet must be present in the same working directory.

---

xmcda.**saveRubisChoiceXSL**(*fileName='xmcda2RubisChoice.xsl'*)
Save the local Rubis Best-Choice XMCDA 2.0 XSL style sheet in the current working directory This style sheet allows to browse an XMCDA 2.0 encoded Rubis Best-Choice recommendation.

Note: When accessing an XMCDA encoded data file in a browser, for safety reasons, the corresponding XSL style sheet must be present in the same working directory.

---

xmcda.**saveRubisXSL**(*fileName='xmcda2Rubis.xsl'*, *Extended=True*)
Save the standard Rubis XMCDA 2.0 XSL style sheet in the current working directory. This style sheet allows to visualize an XMCDA 2.0 encoded performance tableau in a browser.

Note: When accessing an XMCDA encoded data file in a browser, for safety reasons, the corresponding XSL style sheet must be present in the same working directory.

---

xmcda.**saveXMCDARubisBestChoiceRecommendation**(*problemFileName=None*, *tempDir='.'*, *valuationType=None*)

Store an XMCDA2 encoded solution file of the Rubis best-choice recommendation.

The valuationType parameter allows to work:

- on the standard bipolar outranking digraph (valuationType = 'bipolar', default),
- on the normalized –[-1,1] valued– bipolar outranking digraph (valuationType = 'normalized'),
- on the robust –ordinal criteria weights– bipolar outranking digraph (valuationType = 'robust'),
- on the confident outranking digraph (valuationType = 'confident'),
- ignoring considerable performances differences (valuationType = 'noVeto').

Note: The method requires an Unix like OS like Ubuntu or Mac OSX and depends on:

- the R statistics package for Principal Component Analysis graphic,
- the C calmat matrix interpreter (On http://leopold-loewenheim.uni.lu/svn/repos/Calmat/ see README)
- the xpdf ressources (...$ apt-get install xpdf on Ubuntu) for converting pdf files to ppm format, and ppm files to png format.

---

xmcda.**showXMCDARubisBestChoiceRecommendation**(*problemFileName=None*, *valuationType=None*)

Launches a browser window with the XMCDA2 solution of the Rubis Solver computed from a stored XMCDA2 encoded performance tableau.

The valuationType parameter allows to work:

- on the standard bipolar outranking digraph (valuationType = 'bipolar', default),
- on the normalized –[-1,1] valued– bipolar outranking digraph (valuationType = 'normalized'),
- on the robust –ordinal criteria weights– bipolar outranking digraph (valuationType = 'robust'),
- on the confident outranking digraph (valuationType = 'confident'),
- ignoring considerable performances differences (valuationType = 'noVeto').

Example:

```
>>> import xmcda
>>> from randomPerfTabs import RandomCBPerformanceTableau
>>> t = RandomCBPerformanceTableau()
>>> t.saveXMCDA2('example')
>>> xmcda.showXMCDARubisBestChoiceRecommendation(problemFileName='example')
```

Back to the *Installation*

2.2.11 sparseOutrankingDigraphs module

Digraph3 collection of python3 modules for Algorithmic Decision Theory applications

Module for sparse pre-ranked outranking digraphs

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR ANY PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

class sparseOutrankingDigraphs.**PreRankedConfidentOutrankingDigraph**(*argPerfTab*, *quantiles=None*, *quantilesOrderingStrategy='average'*, *LowerClosed=False*, *componentRankingRule='Copeland'*, *minimalComponentSize=1*, *distribution='triangular'*, *betaParameter=2*, *confidence=90.0*, *Threading=False*, *tempDir=None*, *nbrOfCPUs=1*, *nbrOfThreads=1*, *save2File=None*, *CopyPerfTab=True*, *Comments=False*, *Debug=False*)

Bases: *sparseOutrankingDigraphs.PreRankedOutrankingDigraph*, *perfTabs.PerformanceTableau*

Main class for the multiprocessing implementation of pre-ranked sparse confident outranking digraphs.

The sparse outranking digraph instance is decomposed with a confident q-tiling sort into a partition of quantile equivalence classes which are linearly ordered by average quantile limits (default).

With each quantile equivalence class is associated a ConfidentBipolarOutrankingDigraph object which is restricted to the decision actions gathered in this quantile equivalence class.

By default, the number of quantiles is set to a tenth of the number of decision actions, i.e. quantiles = order//10. The effective number of quantiles can be much lower for large orders; for instance quantiles = 250 gives good

results for a digraph of order 25000.

For other parameters settings, see the corresponding classes: *sortingDigraphs.QuantilesSortingDigraph* and *outrankingDigraphs.ConfidentBipolarOutrankingDigraph*.

computeCLTLikelihoods(*distribution='triangular'*, *betaParameter=None*, *Debug=False*)
Renders the pairwise CLT likelihood of the at least as good as relation neglecting all considerable large performance differences polarisations.

showRelationTable(*IntegerValues=False*, *actionsSubset=None*, *Sorted=True*, *LikelihoodDenotation=True*, *hasLatexFormat=False*, *hasIntegerValuation=False*, *relation=None*, *Debug=False*)
prints the relation valuation in actions X actions table format.

class sparseOutrankingDigraphs.**PreRankedOutrankingDigraph**(*argPerfTab*, *quantiles=None*, *quantilesOrderingStrategy='average'*, *LowerClosed=False*, *componentRankingRule='Copeland'*, *minimalComponentSize=1*, *Threading=False*, *tempDir=None*, *nbrOfCPUs=1*, *nbrOfThreads=1*, *save2File=None*, *CopyPerfTab=True*, *Comments=False*, *Debug=False*)
Bases: *sparseOutrankingDigraphs.SparseOutrankingDigraph*, *perfTabs.PerformanceTableau*

Main class for the multiprocessing implementation of pre-ranked sparse outranking digraphs.

The sparse outranking digraph instance is decomposed with a q-tiling sort into a partition of quantile equivalence classes which are linearly ordered by average quantile limits (default).

With each quantile equivalence class is associated a BipolarOutrankingDigraph object which is restricted to the decision actions gathered in this quantile equivalence class.

See http://leopold-loewenheim.uni.lu/bisdorff/documents/DA2PL-RB-2016.pdf

By default, the number of quantiles is set to a tenth of the number of decision actions, i.e. quantiles = order//10. The effective number of quantiles can be much lower for large orders; for instance quantiles = 250 gives good results for a digraph of order 25000.

For other parameters settings, see the corresponding *sortingDigraphs.QuantilesSortingDigraph* class.

actionOrder(*action*, *ordering=None*)
Renders the order of a decision action in a given ordering

If ordering == None, the self.boostedOrder attribute is used.

actionRank(*action*, *ranking=None*)
Renders the rank of a decision action in a given ranking

If ranking == None, the self.boostedRanking attribute is used.

computeActionCategories(*action*, *Show=False*, *Debug=False*, *Comments=False*, *Threading=False*, *nbrOfCPUs=None*)
Renders the union of categories in which the given action is sorted positively or null into. Returns a tuple : action, lowest category key, highest category key, membership credibility !

computeBoostedOrdering(*orderingRule='Copeland'*)
Renders an ordred list of decision actions ranked in increasing preference direction following the orderingRule on each component.

computeBoostedRanking(*rankingRule='Copeland'*)
Renders an ordred list of decision actions ranked in decreasing preference direction following the rankingRule on each component.

computeCategoryContents(*Reverse=False*, *Comments=False*, *StoreSorting=True*, *Threading=False*, *nbrOfCPUs=None*)
Computes the sorting results per category.

computeCriterion2RankingCorrelation(*criterion*, *Threading=False*, *nbrOfCPUs=None*, *Debug=False*, *Comments=False*)
Renders the ordinal correlation coefficient between the global linar ranking and the marginal criterion relation.

computeMarginalVersusGlobalRankingCorrelations(*Sorted=True*, *ValuedCorrelation=False*, *Threading=False*, *nbrCores=None*, *Comments=False*)
Method for computing correlations between each individual criterion relation with the corresponding global ranking relation.

Returns a list of tuples (correlation,criterionKey) sorted by default in decreasing order of the correlation.

If Threading is True, a multiprocessing Pool class is used with a parallel equivalent of the built-in map function.

If nbrCores is not set, the os.cpu_count() function is used to determine the number of available cores.

computeNewActionCategories(*action*, *sorting*, *Debug=False*, *Comments=False*)
Renders the union of categories in which the given action is sorted positively or null into. Returns a tuple : action, lowest category key, highest category key, membership credibility !

computeNewSortingCharacteristics(*actions*, *relation*, *Comments=False*)
Renders a bipolar-valued bi-dictionary relation representing the degree of credibility of the assertion that "actions x in A belongs to category c in C", i.e. x outranks low category limit and does not outrank the high category limit (if LowerClosed).

showActionSortingResult(*action*)
shows the quantiles sorting result all (default) of a subset of the decision actions.

showActions()
Prints out the actions disctionary.

showComponents(*direction='increasing'*)

showCriteria(*IntegerWeights=False*, *Debug=False*)
print Criteria with thresholds and weights.

showCriteriaQuantiles()

showDecomposition(*direction='decreasing'*)

showMarginalVersusGlobalRankingCorrelation(*Sorted=True*, *Threading=False*, *nbrOfCPUs=None*, *Comments=True*)
Show method for computeCriterionCorrelation results.

showNewActionCategories(*action*, *sorting*)
Prints the union of categories in which the given action is sorted positively or null into.

showNewActionsSortingResult(*actions*, *sorting*, *Debug=False*)
shows the quantiles sorting result all (default) of a subset of the decision actions.

showRelationTable(*compKeys=None*)
Specialized for showing the quantiles decomposed relation table. Components are stored in an ordered dictionary.

showShort(*fileName=None*, *WithFileSize=True*)
Default (__repr__) presentation method for big outranking digraphs instances:

```
>>> from sparseOutrankingDigraphs import *
>>> t = RandomCBPerformanceTableau(numberOfActions=100,seed=1)
>>> g = PreRankedOutrankingDigraph(t,quantiles=10)
>>> print(g)
*----- show short --------------*
Instance name     : randomCBperftab_mp
# Actions         : 100
# Criteria        : 7
Sorting by        : 10-Tiling
Ordering strategy : average
Ranking rule      : Copeland
# Components      : 19
Minimal size      : 1
Maximal size      : 22
Median size       : 2
fill rate         : 0.116
----  Constructor run times (in sec.) ----
Total time        : 0.14958
QuantilesSorting  : 0.06847
Preordering       : 0.00071
Decomposing       : 0.07366
Ordering          : 0.00130
<class 'sparseOutrankingDigraphs.PreRankedOutrankingDigraph'> instance
```

showSorting(*Descending=True*, *isReturningHTML=False*, *Debug=False*)
Shows sorting results in decreasing or increasing (Reverse=False) order of the categories. If isReturningHTML is True (default = False) the method returns a htlm table with the sorting result.

class sparseOutrankingDigraphs.**SparseOutrankingDigraph**(*argPerfTab=None*, *coalition=None*, *actionsSubset=None*, *hasNoVeto=False*, *hasBipolarVeto=True*, *Normalized=False*, *CopyPerfTab=True*, *BigData=False*, *Threading=False*, *tempDir=None*, *WithConcordanceRelation=True*, *WithVetoCounts=True*, *nbrCores=None*, *Debug=False*, *Comments=False*)

Bases: *outrankingDigraphs.BipolarOutrankingDigraph*

Abstract root class for linearly decomposed sparse digraphs.

computeDecompositionSummaryStatistics()

Returns the summary of the distribution of the length of the components as follows:

```
summary = {'max': maxLength,
           'median':medianLength,
           'mean':meanLength,
           'stdev': stdLength,
           'fillrate': fillrate,
                       (see computeFillRate()}
```

computeDeterminateness()
Computes the Kendalll distance in % of self with the all median valued (indeterminate) digraph.

computeFillRate()
Renders the sum of the squares (without diagonal) of the orders of the component's subgraphs over the square (without diagonal) of the big digraph order.

computeOrderCorrelation(*order*, *Debug=False*)
Renders the ordinal correlation K of a sparse digraph instance when compared with a given linear order (from worst to best) of its actions

K = sum_{x != y} [min(max(-self.relation(x,y)),other.relation(x,y), max(self.relation(x,y),-other.relation(x,y))]

K /= sum_{x!=y} [min(abs(self.relation(x,y),abs(other.relation(x,y))]

Note: Renders a dictionary with the key 'correlation' containing the actual bipolar correlation index and the key 'determination' containing the minimal determination level D of self and the other relation.

D = sum_{x != y} min(abs(self.relation(x,y)),abs(other.relation(x,y)) / n(n-1)

where n is the number of actions considered.

The correlation index with a completely indeterminate relation is by convention 0.0 at determination level 0.0 .

Warning: self must be a normalized outranking digraph instance !

computeOrdinalCorrelation(*other*, *Debug=False*)
Renders the ordinal correlation K of a SpareOutrakingDigraph instance when compared with a given compatible (same actions set) other Digraph instance.

K = sum_{x != y} [min(max(-self.relation(x,y)),other.relation(x,y), max(self.relation(x,y),-other.relation(x,y))]

K /= sum_{x!=y} [min(abs(self.relation(x,y),abs(other.relation(x,y))]

Note: The global outranking relation of SparesOutrankingDigraph instances is contructed on the fly from the ordered dictionary of the components.

Renders a dictionary with a 'correlation' key containing the actual bipolar correlation index K and a 'determination' key containing the minimal determination level D of self and the other relation, where

D = sum_{x != y} min(abs(self.relation(x,y)),abs(other.relation(x,y)) / n(n-1)

and where n is the number of actions considered.

The correlation index K with a completely indeterminate relation is by convention 0.0 at determination level 0.0 .

---

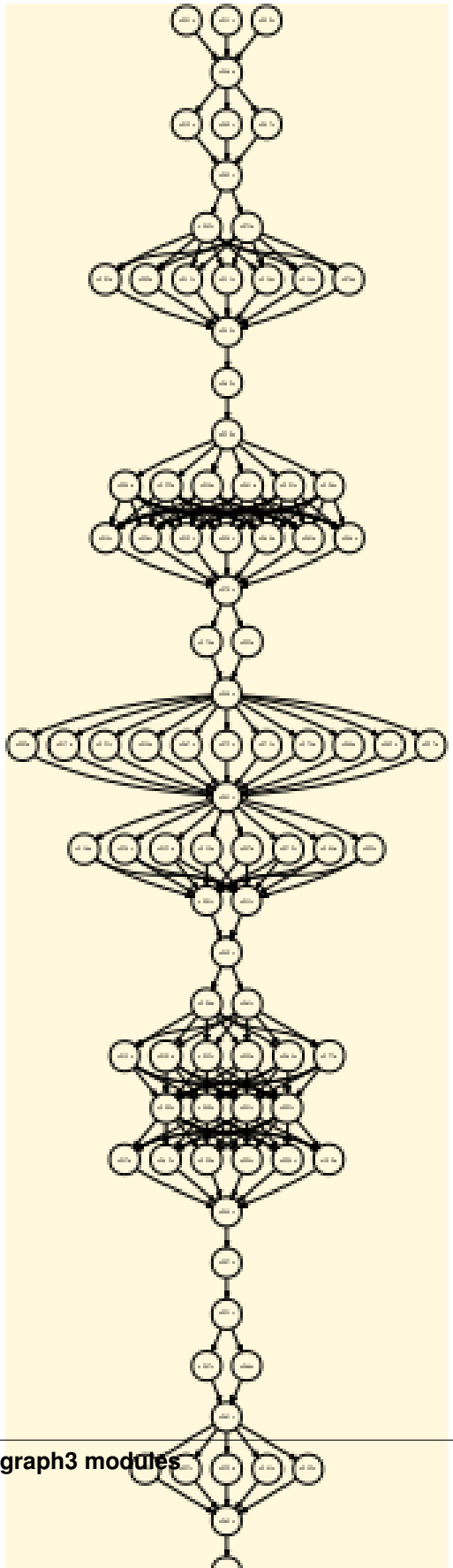
exportGraphViz (*fileName=None*, *actionsSubset=None*, *bestChoice=set()*, *worstChoice=set()*, *noSilent=True*, *graphType='png'*, *graphSize='7, 7'*, *relation=None*)
export GraphViz dot file for graph drawing filtering.

exportSortingGraphViz (*fileName=None*, *actionsSubset=None*, *direction='decreasing'*, *noSilent=True*, *graphType='pdf'*, *graphSize='7, 7'*, *fontSize=10*, *relation=None*, *Debug=False*)
export GraphViz dot file for weak order (Hasse diagram) drawing filtering from SortingDigraph instances.

Example:

```
>>> print('==>> Testing graph viz export of sorting Hasse diagram')
>>> MP  = True
>>> nbrActions=100
>>> tp = RandomCBPerformanceTableau(numberOfActions=nbrActions,
...                          Threading=MP,
...                          seed=100)
>>> bg = PreRankedOutrankingDigraph(tp,CopyPerfTab=True,quantiles=20,
...                              quantilesOrderingStrategy='average',
...                              componentRankingRule='Copeland',
...                              LowerClosed=False,
...                              minimalComponentSize=1,
...                             Threading=MP,nbrOfCPUs=8,
...                             #tempDir='.',
...                              nbrOfThreads=8,
...                              Comments=False,Debug=False)
>>> print(bg)
*----- show short --------------*
Instance name     : randomCBperftab_mp
# Actions         : 100
# Criteria        : 7
Sorting by        : 20-Tiling
Ordering strategy : average
Ranking rule      : Copeland
# Components      : 36
Minimal order     : 1
Maximal order     : 11
Average order     : 2.8
fill rate         : 4.121%
----  Constructor run times (in sec.) ----
Total time        : 0.15991
QuantilesSorting  : 0.11717
Preordering       : 0.00066
Decomposing       : 0.04009
Ordering          : 0.00000
<class 'sparseOutrankingDigraphs.PreRankedOutrankingDigraph'> instance
>>> bg.showComponents()
*--- Relation decomposition in increasing order---*
35: ['a010']
34: ['a024', 'a060']
33: ['a012']
32: ['a018']
31: ['a004', 'a054', 'a075', 'a082']
30: ['a099']
29: ['a065']
28: ['a025', 'a027', 'a029', 'a041', 'a059']
```

```
27: ['a063']
26: ['a047', 'a066']
25: ['a021']
24: ['a007']
23: ['a044']
22: ['a037', 'a062', 'a090', 'a094', 'a098', 'a100']
21: ['a005', 'a040', 'a051', 'a093']
20: ['a015', 'a030', 'a052', 'a055', 'a064', 'a077']
19: ['a006', 'a061']
18: ['a049']
17: ['a001', 'a033']
16: ['a016', 'a028', 'a032', 'a035', 'a057', 'a079', 'a084', 'a095']
15: ['a043']
14: ['a002', 'a017', 'a023', 'a034', 'a067', 'a072', 'a073', 'a074', 'a088',
↪'a089', 'a097']
13: ['a048']
12: ['a078', 'a092']
11: ['a070']
10: ['a014', 'a026', 'a039', 'a058', 'a068', 'a083', 'a086']
9: ['a008', 'a022', 'a038', 'a081', 'a091', 'a096']
8: ['a020']
7: ['a069']
6: ['a045']
5: ['a003', 'a009', 'a013', 'a031', 'a036', 'a056', 'a076']
4: ['a042', 'a071']
3: ['a085']
2: ['a019', 'a080', 'a087']
1: ['a046']
0: ['a011', 'a050', 'a053']
>>> bg.exportSortingGraphViz(actionsSubset=bg.boostedRanking[:100])
```



htmlRelationMap(*actionsSubset=None, tableTitle='Relation Map', relationName='r(x R y)', symbols=['+', '·', ' ', '-', '_'], Colored=True, ContentCentered=True*)
renders the relation map in actions X actions html table format.

ordering2Preorder(*ordering*)
Renders a preordering (a list of list) of a linar order (worst to best) of decision actions in increasing preference direction.

ranking2Preorder(*ranking*)
Renders a preordering (a list of list) of a ranking (best to worst) of decision actions in increasing preference direction.

recodeValuation(*newMin=-1*, *newMax=1*, *Debug=False*)
Specialization for recoding the valuation of all the partial digraphs and the component relation. By default the valuation domain is normalized to [-1;1]

relation(*x*, *y*, *Debug=False*)
Dynamic construction of the global outranking characteristic function *r(x S y)*.

showDecomposition(*direction='decreasing'*)
Prints on the console the decomposition structure of the sparse outranking digraph instance in *decreasing* (default) or *increasing* preference direction.

showHTMLMarginalQuantileLimits()
shows the marginal quantiles limits.

showHTMLRelationMap(*actionsSubset=None, Colored=True, tableTitle='Relation Map', relationName='r(x S y)', symbols=['+', '·', ' ', '–', '—']*)
Launches a browser window with the colored relation map of self.

showRelationMap(*fromIndex=None*, *toIndex=None*, *symbols=None*, *actionsList=None*)
Prints on the console, in text map format, the location of the diagonal outranking components of the sparse outranking digraph.

By default, symbols = {'max':'T','positive': '+', 'median': ' ', 'negative': '-', 'min': '_'}

Example:

```
>>> from sparseOutrankingDigraphs import *
>>> t = RandomCBPerformanceTableau(numberOfActions=50,seed=1)
>>> bg = PreRankedOutrankingDigraph(t,quantiles=10,minimalComponentSize=5)
>>> print(bg)
*----- show short --------------*
Instance name     : randomCBperftab_mp
# Actions         : 50
# Criteria        : 7
Sorting by        : 10-Tiling
Ordering strategy : average
Ranking Rule      : Copeland
# Components      : 7
Minimal size      : 5
Maximal size      : 13
Median size       : 6
fill rate         : 16.898%
----  Constructor run times (in sec.) ----
Total time        : 0.08494
QuantilesSorting  : 0.04339
Preordering       : 0.00034
Decomposing       : 0.03989
```

```
Ordering          : 0.00024
<class 'sparseOutrankingDigraphs.PreRankedOutrankingDigraph'> instance
>>> bg.showRelationMap()
 T+++TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
_ ++TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
 + ++TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
--- -TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
-_-+ TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
______ T-+T+TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
______   +TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
______+  +  TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
______-+- ++TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
_______  + TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
______   -  TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________ +++-+++++TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________+ +++++++++-+TTTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________+- +--++++++++TTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________--+ -+++++++-+TTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________++++ +-   ++ TTTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________--+-+ +++++++TTTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________-+-++- ++++--TTTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________-++-++- + -+-TTTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________---- ++- + ++TTTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________-+--++++- -++TTTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________--- --+++ ++TTTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________+-+-++-+-+ +TTTTTTTTTTTTTTTTTTTTTTTTTTTTT
____________-+- -+++-++ TTTTTTTTTTTTTTTTTTTTTTTTTTTTT
_________________________  -  + + TTTTTTTTTTTTTTTTTT
_________________________  -+ + ++T++TTTTTTTTTTTTTTT
_________________________++ +++++++++TTTTTTTTTTTTTT
_________________________ -- -+-++  TTTTTTTTTTTTTTT
_________________________++++ ++++++-TTTTTTTTTTTTTT
_________________________----- ++-T+TTTTTTTTTTTTTT
_________________________  +++- -++-+TTTTTTTTTTTTT
_________________________-----++ -++TTTTTTTTTTTTTT
_________________________ +-+-+-+ -++TTTTTTTTTTTTT
_________________________+   +++ T+TTTTTTTTTTTTT
_________________________-- --+++  -TTTTTTTTTTTTT
_________________________--_+ -_--+ TTTTTTTTTTTTT
_____________________________________ +++++++TTTTTT
_____________________________________+ +++-+TTTTTT
_____________________________________--  +++TTTTTT
_____________________________________--    ++TTTTT
_____________________________________+-+  +++TTTTT
_____________________________________ +- + --TTTTT
_____________________________________---+++ +TTTTT
_____________________________________- _-+++ TTTTT
_________________________________________  TTTT
_________________________________________  ++ T
_________________________________________ - -TT
_________________________________________ -+  T
_________________________________________  _  T
_________________________________________
Component ranking rule: Copeland
>>>
```

sortingRelation(*x*, *y*, *Debug=False*)

Dynamic construction of the quantiles sorting characteristic function *r(x QS y)*.

Back to the *Installation*

2.2.12 sortingDigraphs module

A tutorial with coding examples for solving multi-criteria rating problems is available here: *Rating with learned quantile norms*

Digraph3 collection of python3 modules for Algorithmic Decision Theory applications.

Module for sorting and rating applications.

class sortingDigraphs.**NormedQuantilesRatingDigraph**(*argPerfQuantiles=None*, *newData=None*, *quantiles=None*, *hasNoVeto=False*, *valuationScale=(-1, 1)*, *rankingRule='best'*, *WithSorting=False*, *Threading=False*, *tempDir=None*, *nbrOfCPUs=None*, *Comments=False*, *Debug=False*)

Bases: *sortingDigraphs.QuantilesSortingDigraph*, *performanceQuantiles.PerformanceQuantiles*

Specialisation of the sortingDigraph Class for absolute rating of a new set of decision actions with normed performance quantiles gathered from historical data.

Note: The constructor requires a valid *performanceQuantiles.PerformanceQuantiles* instance.

Example Python session:

```
>>> from sortingDigraphs import *
>>> # historical data
>>> from randomPerfTabs import RandomCBPerformanceTableau
>>> nbrActions=1000
>>> nbrCrit = 13
>>> seed = 100
>>> tp = RandomCBPerformanceTableau(numberOfActions=nbrActions,
↪numberOfCriteria=nbrCrit,seed=seed)
>>> pq = PerformanceQuantiles(tp,numberOfBins='deciles',LowerClosed=True,
↪Debug=False)
>>> # new incoming decision actions of the same kind
>>> from randomPerfTabs import RandomCBPerformanceGenerator as␣
↪PerfTabGenerator
>>> tpg = PerfTabGenerator(tp,instanceCounter=0,seed=seed)
>>> newActions = tpg.randomActions(10)
```

```
>>> # rating the new set of decision actions after
>>> # updating the historical performance quantiles
>>> pq.updateQuantiles(newActions,historySize=None)
>>> nqr = NormedQuantilesRatingDigraph(pq,newActions,Debug=True)
>>> # inspecting the rating result
>>> nqr.showQuantilesRating()
 *-------- Normed quantiles rating result ---------
 [0.50 - 0.60[ ['a1', 'a7', 'a3', 'a10', 'a2']
 [0.40 - 0.50[ ['a6', 'a9', 'a8']
 [0.20 - 0.30[ ['a4', 'a5']
>>> nqr.showHTMLRatingHeatmap(pageTitle='Heatmap of Quantiles Rating')
```

Heatmap of Quantiles Rating

criteria	c02	b01	b05	b06	c03	b04	c04	c05	c06	b07	b03	b02	c01
weights	7	6	6	6	7	6	7	7	7	6	6	6	7
tau(*)	0.66	0.63	0.62	0.59	0.58	0.57	0.54	0.51	0.49	0.49	0.42	0.42	0.39
[0.90 - <[	-14.0	7.1	7.1	8.0	-18.2	7.1	-12.7	-2.4	-1.3	74.0	8.6	7.6	-2.0
[0.80 - 0.90[	-17.6	6.2	6.2	7.0	-21.9	5.4	-27.0	-2.9	-3.0	53.8	7.0	7.3	-2.2
[0.70 - 0.80[	-18.0	5.9	6.1	4.7	-27.3	4.3	-29.0	-3.5	-4.0	44.5	6.1	6.5	-2.3
[0.60 - 0.70[	-20.3	5.2	5.2	4.4	-34.3	3.5	-30.3	-4.4	-4.2	38.7	4.9	2.9	-2.5
a1c	-17.7	7.8	4.9	4.2	-25.1	1.4	-14.8	-2.2	-4.1	19.2	3.0	2.7	-2.9
a7c	-18.8	4.8	2.9	2.8	-37.5	7.3	-11.5	-5.0	-6.2	30.3	3.6	7.0	-2.6
a3n	-17.6	4.5	6.2	7.6	-38.7	3.5	-70.3	-4.7	-6.8	77.8	4.7	2.2	-9.7
a10c	-45.4	6.6	3.9	4.7	-18.6	2.5	-52.1	-7.5	-1.2	23.8	1.3	7.5	-2.4
a2n	-24.9	5.8	7.2	3.4	-44.5	5.3	-31.0	-7.9	-4.7	57.4	6.6	0.5	-2.8
[0.50 - 0.60[	-23.6	4.7	4.2	4.2	-38.1	3.4	-35.8	-5.0	-4.5	30.6	4.2	2.7	-2.7
a6c	-20.4	3.4	6.2	2.5	-12.0	4.1	-39.5	NA	-4.3	28.3	2.3	0.4	-9.1
a9c	-12.7	1.4	3.0	4.3	-66.5	2.5	-29.3	-7.9	-3.0	21.5	4.9	2.9	-2.0
a8n	-56.7	2.1	5.4	8.4	-53.5	2.6	-65.3	-2.9	-6.4	43.6	8.7	2.2	-2.3
[0.40 - 0.50[	-43.6	4.0	3.5	3.5	-43.4	2.7	-50.9	-7.1	-6.0	28.5	3.5	2.7	-2.9
[0.30 - 0.40[	-52.2	3.0	3.0	3.2	-50.8	2.5	-58.8	-7.5	-6.3	24.1	3.0	2.3	-7.0
a4a	-66.8	5.9	2.1	3.2	-68.3	3.4	-73.0	-7.8	-7.6	38.5	8.2	7.6	-7.8
a5c	-77.0	1.1	1.9	2.1	-32.9	2.5	-28.6	-3.6	-1.6	19.4	2.9	2.7	-1.5
[0.20 - 0.30[	-60.3	2.2	2.8	2.8	-65.5	2.5	-67.1	-7.8	-6.5	21.5	2.8	2.2	-7.8
[0.10 - 0.20[	-74.1	1.4	2.0	2.5	-67.1	2.0	-70.9	-7.9	-7.1	19.4	2.0	0.6	-9.1
[0.00 - 0.10[	-98.0	0.0	0.0	0.0	-96.6	0.0	-98.5	-10.0	-10.0	2.5	0.0	0.0	-10.0

Color legend:

quantile	14.29%	28.57%	42.86%	57.14%	71.43%	85.71%	100.00%

(*) tau: Ordinal (Kendall) correlation between marginal criterion and global ranking relation.
Ordinal (Kendall) correlation between global ranking and outranking relation: 0.97.

computeCategoryContents(*Debug=False*)
Computes the quantiles sorting results per quantile category.

computeQuantileOrdering(*strategy=None*, *Descending=True*, *HTML=False*, *Comments=False*, *Debug=False*)
Orders the quantiles sorting result of self.newActions.

Parameters:

- Descending: listing in *decreasing* (default) or *increasing* quantile order.
- strategy: ordering in an {'optimistic' (default) | 'pessimistic' | 'average'} in the uppest, the lowest

or the average potential quantile.

computeQuantilesRating (*Debug=True*)
Renders an ordered dictionary of non empty quantiles in ascending order.

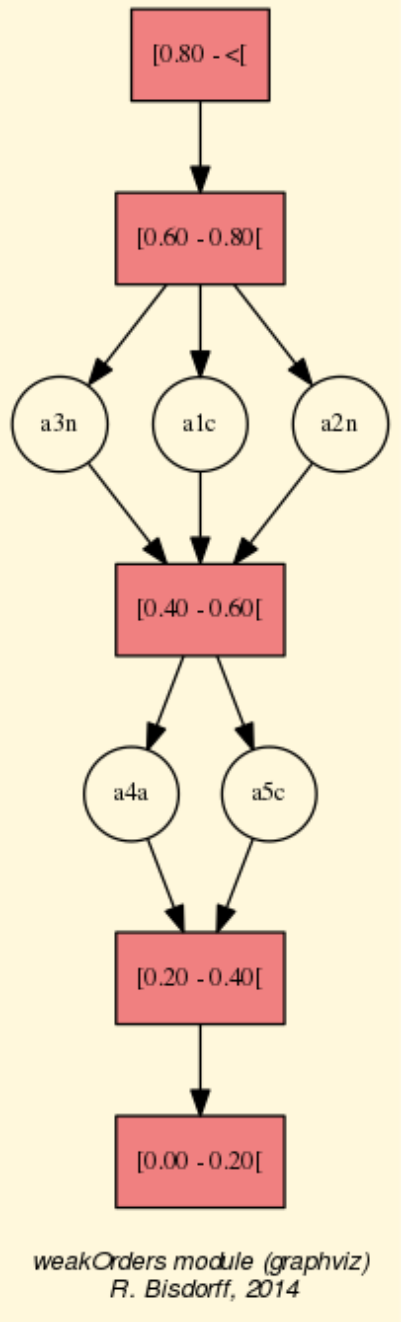
computeRatingRelation (*Debug=False*, *StoreRating=True*)
Computes a bipolar rating relation using a pre-ranking (list of lists) of the self-actions (self.newActions + self.profiles).

computeSortingCharacteristics (*action=None*, *Debug=False*)
Renders a bipolar-valued bi-dictionary relation (newActions x profiles) representing the degree of credibility of the assertion that "action x in A belongs to quantile category c profiles", If LowerClosed is True, x outranks the category low limit and x does not outrank the category high limit, or If LowerClosed is False, ie UPPERCLosed is True, the category low limit does not outrank x and the category high limit does outrank x.

exportRatingGraphViz (*fileName=None*, *relation=None*, *direction='best'*, *noSilent=True*, *graphType='png'*, *graphSize='7*, *7'*, *fontSize=10*)
The rating drawing is using the weakOrders.WeakOrder exportGraphViz() method for drawing oriented Hasse diagrams of weak orderings, ie the negation of the corresponding preorder relation.

Continuing the previous Python session:

```
>>> nqr.showQuantilesRating()
*-------- Quantile sorting result ---------
 [0.40 - 0.60[ ['a1', 'a2', 'a3']
 [0.20 - 0.40[ ['a4', 'a5']
>>> nqr.exportRatingGraphViz(noSilent=False)
*---- exporting a dot file for GraphViz tools ---------*
 Exporting to quantilesRatingDigraph.dot
 dot -Grankdir=TB -Tpng quantilesRatingDigraph.dot -o␣
↪quantilesRatingDigraph.png
```



Warning: For graphviz, nodes or action keys of the digraph must start with a letter and may not contain special characters like '-' or '_'.

htmlRatingHeatmap(*argCriteriaList=None*, *argActionsList=None*, *quantiles=None*, *ndigits=2*, *contentCentered=True*, *colorLevels=None*, *pageTitle='Rating Heatmap'*, *Correlations=False*, *Threading=False*, *nbrOfCPUs=1*, *Debug=False*)

Renders the Brewer RdYlGn 5,7, or 9 levels colored heatmap of the performance table actions x criteria in html format.

See the corresponding `perfTabs.showHTMLPerformanceHeatMap()` method.

showActionCategories(*action*, *Debug=False*, *Comments=True*)
Renders the union of categories in which the given action is sorted positively or null into. Returns a tuple : action, lowest category key, highest category key, membership credibility !

showActionsSortingResult(*actionSubset=None*, *Debug=False*)
Shows the quantiles sorting result of all (default) or a subset of the decision actions.

showHTMLQuantilesSorting(*Descending=True*, *strategy='average'*)
Shows the html version of the quantile sorting result in a browser window.

The ordring strategy is either:

- **optimistic**, following the upper quantile limits (default),
- **pessimistic**, following the lower quantile limits,
- **average**, following the averag of the upper and lower quantile limits.

showHTMLRatingHeatmap(*actionsList=None*, *criteriaList=None*, *colorLevels=7*, *pageTitle=None*, *ndigits=2*, *quantiles=None*, *Correlations=False*, *Threading=False*, *nbrOfCPUs=None*, *Debug=False*)
Specialisation of html heatmap version showing the performance tableau in a browser window; see `perfTabs.showHTMLPerformanceHeatMap()` method.

Parameters:

- *actionsList* and *criteriaList*, if provided, give the possibility to show the decision alternatives, resp. criteria, in a given ordering.
- *ndigits* = 0 may be used to show integer evaluation values.
- If no *actionsList* is provided, the decision actions are ordered from the best to the worst following the ranking of the NormedQuatilesRatingDigraph instance.
- It may interesting in some cases to use *RankingRule* = 'NetFlows'.
- With *Correlations* = *True* and *criteriaList* = *None*, the criteria will be presented from left to right in decreasing order of the correlations between the marginal criterion based ranking and the global ranking used for presenting the decision alternatives.
- Computing the marginal correlations may be boosted with Threading = True, if multiple parallel computing cores are available.

Suppose we observe the following rating result:

```
>>> nqr.showQuantilesRating()
 [0.50 - 0.75[ ['a1008', 'a1006', 'a1005', 'a1001', 'a1003', 'a1010']
 [0.25 - 0.50[ ['a1002']
 [0.00 - 0.25[ ['a1004', 'a1009', 'a1007']
>>> nqr.showHTMLRatingHeatmap(pageTitle='Heat map of the ratings',
...                           Correlations=True,
...                           colorLevels = 5)
```

Heat map of the ratings

Ranking rule: **Copeland**; *Ranking correlation*: **0.966**

criteria	c1	b4	b2	b5	b3	b6	b1
weights	6	1	1	1	1	1	1
tau(*)	0.80	0.45	0.25	0.24	0.20	0.16	-0.02
[0.75 - <[	-42.08	6.52	7.84	59.81	80.16	7.65	69.75
a1008c	-35.67	6.24	5.02	50.55	61.54	4.25	9.41
a1006n	-45.50	4.96	6.81	28.91	79.89	7.41	20.35
a1005c	-38.15	5.34	5.14	30.37	27.67	1.73	18.28
a1001a	-58.49	7.96	9.07	57.27	55.31	4.01	80.59
a1003n	-70.27	8.90	7.55	63.68	89.12	4.43	65.74
a1010n	-48.50	7.00	4.87	38.87	11.36	4.74	48.83
[0.50 - 0.75[	-59.89	5.89	6.57	39.16	59.43	4.41	49.80
a1002n	-70.86	6.17	7.47	30.26	59.79	3.94	49.84
[0.25 - 0.50[	-73.58	5.10	5.05	32.00	53.61	3.96	24.39
a1004a	-76.71	5.07	6.41	35.29	53.37	3.29	34.89
a1009n	-79.10	3.28	1.81	86.11	58.78	9.32	69.03
a1007a	-96.90	5.22	8.86	34.01	82.48	8.27	70.56
[0.00 - 0.25[	-97.12	0.00	0.00	1.84	1.08	0.00	2.11

Color legend:

quantile	20.00%	40.00%	60.00%	80.00%	100.00%

(*) tau: Ordinal (Kendall) correlation between marginal criterion and global ranking relation. Ordinal (Kendall) correlation between global ranking and outranking relation: 0.97.

showOrderedRelationTable(*relation=None*, *direction='decreasing'*)
Showing the relation table in decreasing (default) or increasing order.

showQuantilesRating(*Descending=True*, *Debug=False*)

showQuantilesSorting(*strategy='average'*)
Dummy show method for the commenting computeQuantileOrdering() method.

showRankingScores(*direction='descending'*)
Shows the ranking scores of the Copeland or the netflows ranking rule, the number of incoming arcs minus the number of outgoing arcs, resp. the sum of inflows minus the outflows.

class sortingDigraphs.**QuantilesSortingDigraph**(*argPerfTab=None*, *limitingQuantiles=None*, *LowerClosed=False*, *PrefThresholds=True*, *hasNoVeto=False*, *outrankingType='bipolar'*, *WithSortingRelation=True*, *CompleteOutranking=False*, *StoreSorting=False*, *CopyPerfTab=False*, *Threading=False*, *tempDir=None*, *nbrCores=None*, *nbrOfProcesses=None*, *Comments=False*, *Debug=False*)

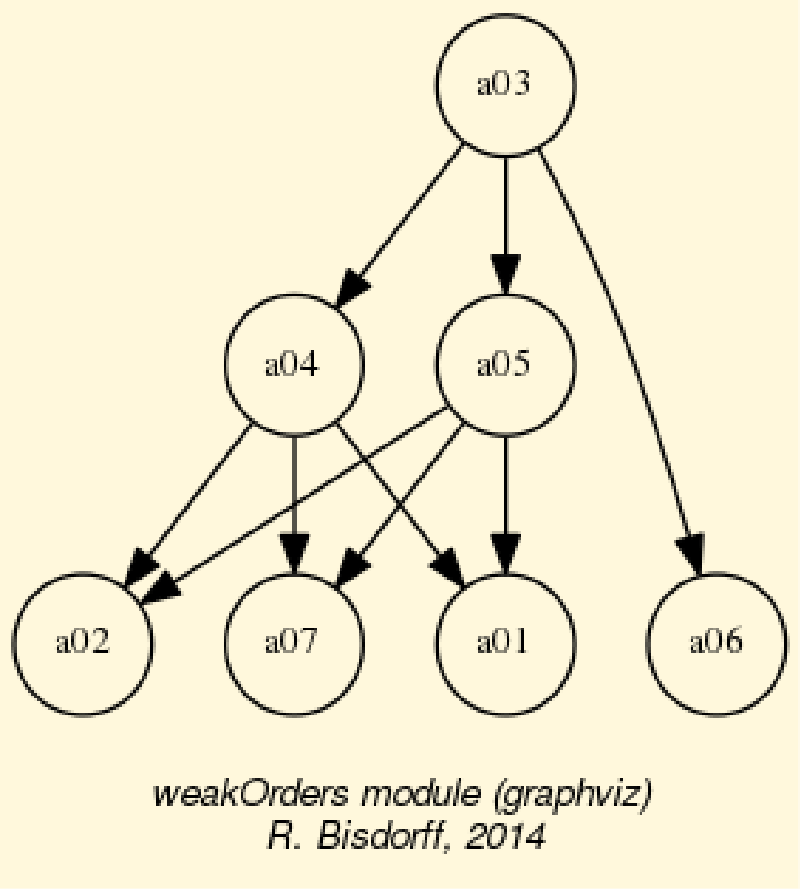
Bases: *sortingDigraphs.SortingDigraph*

Specialisation of the sortingDigraph Class for sorting of a large set of alternatives into quantiles delimited ordered classes.

Note: The constructor requires a valid PerformanceTableau instance. If no number of limiting quantiles is given, then a default profile with the limiting quartiles Q0,Q1,Q2, Q3 and Q4 is used on each criteria. By default upper closed limits of categories are supposed to be used in the sorting.

Example Python3 session:

```
>>> from sortingDigraphs import QuantilesSortingDigraph
>>> from randomPerfTabs import RandomCBPerformanceTableau
>>> t = RandomCBPerformanceTableau(numberOfActions=7,numberOfCriteria=5,
...                                weightDistribution='equiobjectives')
>>> qs = QuantilesSortingDigraph(t,limitingQuantiles=7)
>>> qs.showSorting()
*--- Sorting results in descending order ---*
]0.86 - 1.00]:       []
]0.71 - 0.86]:       ['a03']
]0.57 - 0.71]:       ['a04']
]0.43 - 0.57]:       ['a04', 'a05', 'a06']
]0.29 - 0.43]:       ['a01', 'a02', 'a06', 'a07']
]0.14 - 0.29]:       []
]< - 0.14]:          []
>>> qs.showQuantileOrdering()
]0.71-0.86] : ['a03']
]0.43-0.71] : ['a04']
]0.43-0.57] : ['a05']
]0.29-0.57] : ['a06']
]0.29-0.43] : ['a07', 'a02', 'a01']
>>> qs.exportGraphViz('quantilesSorting')
```



computeCategoryContents(*Reverse=False*, *Comments=False*, *StoreSorting=True*, *Threading=False*, *nbrOfCPUs=None*)
Computes the sorting results per category.

computeQuantileOrdering(*strategy=None*, *Descending=True*, *HTML=False*, *Comments=False*, *Debug=False*)

Parameters:

- Descending: listing in *decreasing* (default) or *increasing* quantile order.
- strategy: ordering in an {'optimistic' (default) | 'pessimistic' | 'average'} in the uppest, the lowest or the average potential quantile.

computeSortingCharacteristics(*action=None*, *Comments=False*, *StoreSorting=False*, *Debug=False*, *Threading=False*, *nbrOfCPUs=None*)
Renders a bipolar-valued bi-dictionary relation representing the degree of credibility of the assertion that "action x in A belongs to category c in C", ie x outranks low category limit and does not outrank the high category limit.

computeSortingRelation(*categoryContents=None*, *Debug=False*, *StoreSorting=True*, *Threading=False*, *nbrOfCPUs=None*, *Comments=False*)
constructs a bipolar sorting relation using the category contents.

computeWeakOrder(*Descending=True*, *Debug=False*)
Specialisation for QuantilesSortingDigraphs.

getActionsKeys(*action=None*, *withoutProfiles=True*)
extract normal actions keys()

showActionCategories(*action*, *Debug=False*, *Comments=True*, *Threading=False*, *nbrOfCPUs=None*)
Renders the union of categories in which the given action is sorted positively or null into. Returns a tuple : action, lowest category key, highest category key, membership credibility !

showActionsSortingResult(*actionSubset=None*, *Debug=False*)
shows the quantiles sorting result all (default) of a subset of the decision actions.

showHTMLQuantileOrdering(*Descending=True*, *strategy='optimistic'*)
Shows the html version of the quantile preordering in a browser window.

The ordring strategy is either:

- **optimistic**, following the upper quantile limits (default),
- **pessimistic**, following the lower quantile limits,
- **average**, following the averag of the upper and lower quantile limits.

showHTMLSorting(*Reverse=True*)
shows the html version of the sorting result in a browser window.

showOrderedRelationTable(*direction='decreasing'*)
Showing the relation table in decreasing (default) or increasing order.

showQuantileOrdering(*strategy=None*)
Dummy show method for the commenting computeQuantileOrdering() method.

showSorting(*Reverse=True*, *isReturningHTML=False*, *Debug=False*)
Shows sorting results in decreasing or increasing (Reverse=False) order of the categories. If isReturningHTML is True (default = False) the method returns a htlm table with the sorting result.

showSortingCharacteristics(*action=None*)
Renders a bipolar-valued bi-dictionary relation representing the degree of credibility of the assertion that "action x in A belongs to category c in C", ie x outranks low category limit and does not outrank the high category limit.

showWeakOrder(*Descending=True*)
Specialisation for QuantilesSortingDigraphs.

class sortingDigraphs.**SortingDigraph**(*argPerfTab=None*, *argProfile=None*, *scaleSteps=5*, *minValuation=-100.0*, *maxValuation=100.0*, *isRobust=False*, *hasNoVeto=False*, *LowerClosed=True*, *StoreSorting=True*, *Threading=False*, *tempDir=None*, *nbrCores=None*, *Debug=False*)
Bases: *outrankingDigraphs.BipolarOutrankingDigraph*, *perfTabs.PerformanceTableau*

Specialisation of the digraphs.BipolarOutrankingDigraph Class for Condorcet based multicriteria sorting of alternatives.

Besides a valid PerformanceTableau instance we require a sorting profile, i.e.:

- a dictionary <categories> of categories with 'name', 'order' and 'comment'
- a dictionary <criteriaCategoryLimits> with double entry:

  [criteriakey][categoryKey] containing a ['minimum'] and a ['maximum'] value in the scale of the criterion respecting the order of the categories.

Template of required data for a 4-sorting:

```
categories = {
               'c1': { 'name': 'week','order': 1,
                       'lowLimit': 0,'highLimit': 25,
                       'comment': 'lowest category',},
               'c2': { 'name': 'ok','order': 2,
                       'lowLimit': 25,'highLimit': 50,
                       'comment': 'medium category',},
               'c3': { 'name': 'good','order': 3,
                       'lowLimit': 50,'highLimit': 75,
                       'comment': 'highest category',},
               'c4': { 'name': 'excellent','order': 4,
                       'lowLimit': 75,'highLimit': 100,
                       'comment': 'highest category',},
 }
criteriaCategoryLimits['LowerClosed'] = True # default
criteriaCategoryLimits[g] = {
        'c1': {'minimum':0, 'maximum':25},
        'c2': {'minimum':25, 'maximum':50},
        'c3': {'minimum':50, 'maximum':75},
        'c4': {'minimum':75, 'maximum':200},
 }
```

A template named tempProfile.py is providied in the digraphs module distribution.

Note: We generally require a performanceTableau instance and a filename where categories and a profile my be read from. If no such filename is given, then a default profile with five, equally spaced, categories is used on each criteria. By default lower-closed limts of categories are supposed to be used in the sorting.

Example Python3 session:

```
>>> from sortingDigraphs import SortingDigraph
>>> from randomPerfTabs import RandomPerformanceTableau
>>> t = RandomPerformanceTableau(seed=1)
>>> [x for x in t.actions]
['a01', 'a02', 'a03', 'a04', 'a05', 'a06', 'a07', 'a08',
'a09', 'a10', 'a11', 'a12', 'a13']
>>> so = SortingDigraph(t,scaleSteps=5)
# so gives a sorting result into five lower closed ordered
# categories enumerated from 0 to 5.
>>> so.showSorting()
*--- Sorting results in descending order ---*
]> - 4]:     ['a02', 'a03', 'a11']
]4 - 3]:     ['a04', 'a07', 'a08', 'a09', 'a10', 'a11', 'a12', 'a13']
]3 - 2]:     ['a04', 'a05', 'a06', 'a09', 'a12']
]2 - 1]:     ['a01']
]1 - 0]:     []
# Notice that some alternatives, like a04, a09, a11 and a12 are sorted
# into more than one adjacent category. Weak ordering the sorting result
# into ordered adjacent categories gives following result:
>>> so.showWeakOrder(strategy='average',Descending=True)
Weak ordering by average normalized 5-sorting limits
]  >  -80.0] : ['a02', 'a03']
]100.0-60.0] : ['a11']
] 80.0-60.0] : ['a07', 'a08', 'a10', 'a13']
] 80.0-40.0] : ['a04', 'a09', 'a12']
] 60.0-40.0] : ['a05', 'a06']
```

```
] 40.0-20.0] : ['a01']
```

computeCategoryContents (*Reverse=False*, *StoreSorting=True*, *Comments=False*)
Computes the sorting results per category.

computeSortingCharacteristics (*action=None*, *StoreSorting=True*, *Comments=False*, *Debug=False*, *Threading=False*, *nbrOfCPUs=None*)
Renders a bipolar-valued bi-dictionary relation representing the degree of credibility of the assertion that "action x in A belongs to category c in C", ie x outranks low category limit and does not outrank the high category limit.

computeSortingRelation (*categoryContents=None*, *StoreSorting=True*, *Debug=False*)
constructs a bipolar sorting relation using the category contents.

computeWeakOrder (*Descending=False*, *strategy='average'*, *Comments=False*, *Debug=False*)
specialisation of the showWeakOrder method. The weak ordering strategy may be:

"optimistic" (ranked by highest category limits), "pessimistic" (ranked by lowest category limits) or "average" (ranked by average category limits)

exportDigraphGraphViz (*fileName=None*, *bestChoice=set()*, *worstChoice=set()*, *noSilent=True*, *graphType='png'*, *graphSize='7*, *7'*)
export GraphViz dot file for digraph drawing filtering.

exportGraphViz (*fileName=None*, *direction='decreasing'*, *noSilent=True*, *graphType='png'*, *graphSize='7*, *7'*, *fontSize=10*, *relation=None*, *Debug=False*)
export GraphViz dot file for weak order (Hasse diagram) drawing filtering from SortingDigraph instances.

getActionsKeys (*action=None*, *withoutProfiles=True*)
extract normal actions keys()

htmlCriteriaCategoryLimits (*tableTitle='Category limits'*)
Renders category minimum and maximum limits for each criterion as a html table.

orderedCategoryKeys (*Reverse=False*)
Renders the ordered list of category keys based on self.categories['order'] numeric values.

recodeValuation (*newMin=-1.0*, *newMax=1.0*, *Debug=False*)
Recodes the characteristic valuation domain according to the parameters given.

Note: Default values gives a normalized valuation domain

---

saveCategories (*fileName='tempCategories'*)

saveProfilesXMCDA2 (*fileName='temp'*, *category='XMCDA 2.0 format'*, *user='sortinDigraphs Module (RB)'*, *version='saved from Python session'*, *title='Sorting categories in XMCDA-2.0 format.'*, *variant='Rubis'*, *valuationType='bipolar'*, *isStringIO=False*, *stringNA='NA'*, *comment='produced by saveProfilesXMCDA2()'*)
Save profiles object self in XMCDA 2.0 format.

showActionCategories (*action*, *Debug=False*, *Comments=True*, *Threading=False*, *nbrOfCPUs=None*)
Renders the union of categories in which the given action is sorted positively or null into. Returns a tuple : action, lowest category key, highest category key, membership credibility !

showActionsSortingResult (*actionSubset=None*, *Debug=False*)
shows the quantiles sorting result all (default) of a subset of the decision actions.

showCriteriaCategoryLimits()
Shows category minimum and maximum limits for each criterion.

showOrderedRelationTable(*direction='decreasing'*)
Showing the relation table in decreasing (default) or increasing order.

showSorting(*Reverse=True*, *isReturningHTML=False*)
Shows sorting results in decreasing or increasing (Reverse=False) order of the categories. If isReturningHTML is True (default = False) the method returns a htlm table with the sorting result.

showSortingCharacteristics(*action=None*)
Renders a bipolar-valued bi-dictionary relation representing the degree of credibility of the assertion that "action x in A belongs to category c in C", ie x outranks low category limit and does not outrank the high category limit.

showWeakOrder(*Descending=False*, *strategy='average'*)
dummy for computeWeakOrder with Comments=True

Back to the *Installation*

2.2.13 votingProfiles module

A tutorial with coding examples is available here: *Computing the winner of an election*

Python 3 implementation of voting digraphs Refactored from revision 1.549 of the digraphs module Current revision $Revision: 2484 $ Copyright (C) 2011-2018 Raymond Bisdorff

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

class votingProfiles.**ApprovalVotingProfile**(*fileVotingProfile=None*, *seed=None*)
Bases: *votingProfiles.VotingProfile*

A specialised class for approval voting profiles

Structure:

```
candidates = OrderedDict([('a', {'name': ...}),
              ('b', {'name': ...}),
              ..., ...])
voters = OrderedDict([('v1',{'weight':1.0}),('v2':{'weight':1.0}), ...])
## each specifies the subset of candidates he approves on
approvalBallot = {
    'v1' : ['b'],
    'v2' : ['a','b'],
    ...
    }

## each specifies the subset -disjoint from the approvalBallot-  of candidates he␣
↪disapproves on
disApprovalBallot = {
    'v1' : ['a'],
```

```
    'v2' : [],
    ...
    }
```

computeBallot()
Computes a complete ballot from the approval Ballot.

Parameters: approvalEquivalence=False, disapprovalEquivalence=False.

save(*name='tempAVprofile'*)
Persistant storage of an approval voting profile.

Parameter: name of file (without <.py> extension!).

save2PerfTab(*fileName='votingPerfTab'*, *isDecimal=True*, *valueDigits=2*)
Persistant storage of an approval voting profile in the format of a standard performance tableau. For each voter *v*, the performance of candidate *x* corresponds to:

1, if approved; 0, if disapproved; -999, miising evalaution otherwise,

showApprovalResults()
Renders the approval obtained by each candidates.

showDisApprovalResults()
Renders the disapprovals obtained by each candidates.

showResults()

class votingProfiles.**CondorcetDigraph**(*argVotingProfile=None*, *approvalVoting=False*, *coalition=None*, *majorityMargins=True*, *hasIntegerValuation=True*)

Bases: *digraphs.Digraph*

Specialization of the general Digraph class for generating bipolar-valued marginal pairwise majority difference digraphs.

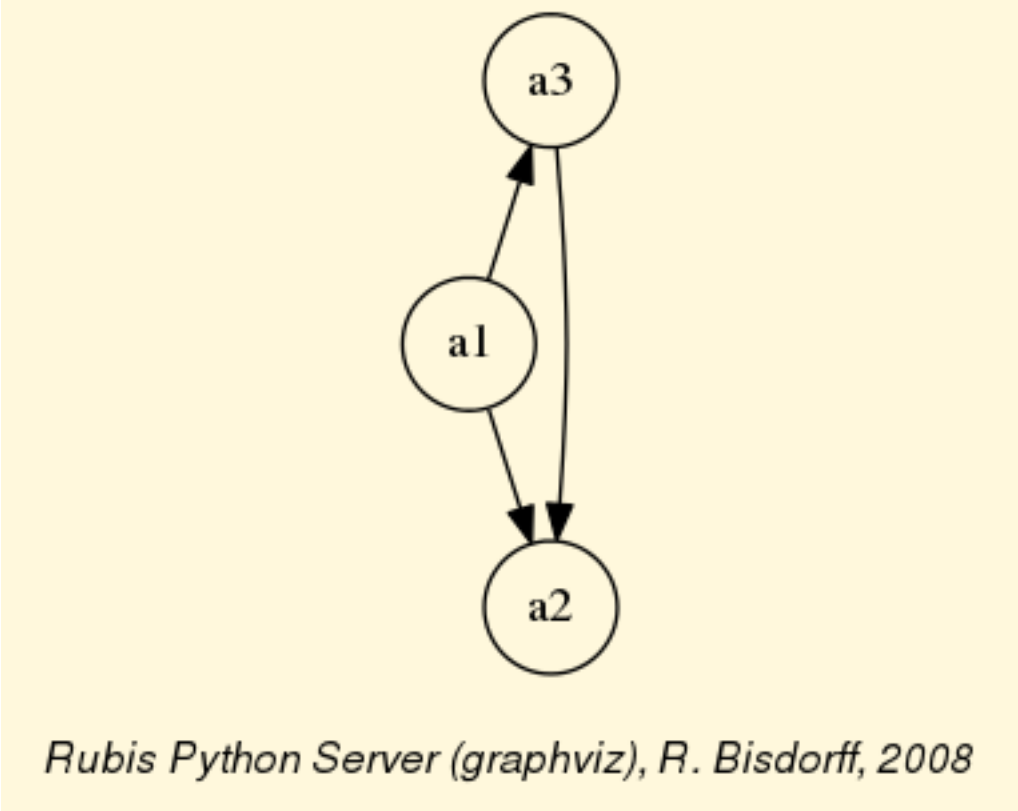
Parameters:

stored voting profile (fileName of valid py code) or voting profile object
optional, coalition (sublist of voters)

Example Python3 session

```
>>> from votingDigraphs import *
>>> v = RandomLinearVotingProfile(numberOfVoters=101,numberOfCandidates=5)
>>> v.showLinearBallots()
v101(1.0):   ['a5', 'a1', 'a2', 'a4', 'a3']
v100(1.0):   ['a4', 'a1', 'a5', 'a3', 'a2']
v89(1.0):    ['a4', 'a5', 'a1', 'a2', 'a3']
v88(1.0):    ['a3', 'a2', 'a5', 'a1', 'a4']
v87(1.0):    ['a5', 'a2', 'a4', 'a3', 'a1']
v86(1.0):    ['a5', 'a3', 'a1', 'a4', 'a2']
v85(1.0):    ['a5', 'a3', 'a2', 'a4', 'a1']
v84(1.0):    ['a3', 'a1', 'a2', 'a4', 'a5']
...
...
>>> g = CondorcetDigraph(v,hasIntegerValuation=True)
>>> g.showRelationTable()
* ---- Relation Table -----
 S    |  'a1'  'a2'   'a3'   'a4'  'a5'
------|-------------------------------------------------------
'a1' |   -     33     9      11    21
```

```
'a2' |  -33    -     -19     -1    -5
'a3' |  -9     19    -       5     -1
'a4' |  -11    1      -5     -     -9
'a5' |  -21    5       1     9      -
>>> g.computeCondorcetWinner()
['a1']
>>> g.exportGraphViz()
*---- exporting a dot file dor GraphViz tools ---------*
Exporting to rel_randLinearProfile.dot
dot -Grankdir=BT -Tpng rel_randLinearProfile.dot -o rel_randLinearProfile.png
```



computeArrowRaynaudRanking(*linearOrdered=True*, *Debug=False*)
Renders a ranking of the actions following Arrow&Raynaud's rule.

computeCondorcetWinner()
compute the Condorcet winner(s) renders always a, potentially empty, list

computeCopelandRanking(*Debug=False*)
Renders a ranking of the actions following Copeland's rule. Score(x_i) = Sum_j{ [[x_i > x_j]] - [[x_j > x_i]]}, where [[x > y]] = 1 if x>y is true, otherwise 0.

The alternatives are ranked in decreasing order of their Scores.

In case of a tie, we use a lexicographic rule applied to the identifiers.

computeKohlerRanking(*linearOrdered=True*, *Debug=False*)
Renders a ranking of the actions following Kohler's rule.

computeNetFlowsRanking(*Debug=False*)
Renders a ranking of the actions following the Net Flows rule. Score(x_i) = Sum_j{M(x_i,x_j)} for i,j = 1..n

The alternatives are ranked in decreasing order of their Scores.

In case of a tie, we use a lexicographic rule applied to the identifiers.

constructApprovalBallotRelation (*hasIntegerValuation=False*)
Renders the votes differences between candidates on the basis of an approval ballot.

constructBallotRelation (*hasIntegerValuation*)
Renders the marginal majority between candidates on the basis of a complete ballot.

constructMajorityMarginsRelation (*hasIntegerValuation=True*)
Renders the marginal majority between candidates on the basis of an approval ballot.

showMajorityMargins (***args*)
Wrapper for the Digraph.showRelationTable(Sorted=True, IntegerValues=False, actionsSubset=None, relation=None, ndigits=2, ReflexiveTerms=True)

See the *digraphs.Digraph.showRelationTable()* description.

class votingProfiles.**LinearVotingProfile** (*fileVotingProfile=None*, *numberOfCandidates=5*, *numberOfVoters=9*, *seed=None*)

Bases: *votingProfiles.VotingProfile*

A specialised class for linear voting profiles

Structure:

```
candidates = OrderedDict([('a', ...) ,('b', ...),('c', ...), ...])
voters = OrderedDict([('1',{'weight':1.0}), ('2',{'weight':1.0}), ...])
## each specifies a a ranked list of candidates
## from the best to the worst
linearBallot = {
    '1' : ['b','c','a', ...],
    '2' : ['a','b','c', ...],
    ...
    }
```

Sample Python3 session

```
>>> from votingDigraphs import *
>>> v = RandomLinearVotingProfile(numberOfVoters=5,numberOfCandidates=3)
>>> v.showLinearBallots()
voters(weight)       candidates rankings
v4(1.0):     ['a1', 'a2', 'a3']
v5(1.0):     ['a1', 'a2', 'a3']
v1(1.0):     ['a2', 'a1', 'a3']
v2(1.0):     ['a1', 'a2', 'a3']
v3(1.0):     ['a1', 'a3', 'a2']
>>> v.computeRankAnalysis()
{'a1': [4.0, 1.0, 0],
 'a2': [1.0, 3.0, 1.0],
 'a3': [0, 1.0, 4.0]}
>>> v.showRankAnalysisTable()
*----  Rank analysis tableau -----*
  ranks |  1    2    3    | Borda score
 -------|-----------------------------
   'a1' |  4    1    0    |   6
   'a2' |  1    3    1    |   10
   'a3' |  0    1    4    |   14
>>> v.computeUninominalVotes()
{'a1': 4.0, 'a3': 0, 'a2': 1.0}
```

```
>>> v.computeSimpleMajorityWinner()
['a1']
>>> v.computeBordaScores()
{'a1': 6.0, 'a3': 14.0, 'a2': 10.0}
>>> v.computeBordaWinners()
['a1']
>>> v.computeInstantRunoffWinner()
['a1']
```

computeBallot()
Computes a complete ballot from the linear Ballot.

computeBordaScores()
compute Borda scores from the rank analysis

computeBordaWinners()
compute the Borda winner from the Borda scores, ie the list of candidates with the minimal Borda score.

computeInstantRunoffWinner(*Comments=False*)
compute the instant runoff winner from a linear voting ballot

computeRankAnalysis()
compute the number of ranks each candidate obtains

computeSimpleMajorityWinner(*Comments=False*)
compute the winner in a uninominal Election from a linear ballot

computeUninominalVotes(*candidates=None*, *linearBallot=None*)
compute uninominal votes for each candidate in candidates sublist and restricted linear ballots

save(*name='templinearprofile'*)
Persistant storage of a linear voting profile.

Parameter: name of file (without <.py> extension!).

save2PerfTab(*fileName='votingPerfTab'*, *isDecimal=True*, *valueDigits=2*)
Persistant storage of a linear voting profile in the format of a rank performance Tableau. For each voter *v*, the rank performance of candidate *x* corresponds to:

number of candidates - linearProfile[v].index(x)

showBordaRanking()
show Borda ranking in increasing order of the Borda score

showHTMLVotingHeatmap(*criteriaList=None*, *actionsList=None*, *SparseModel=False*, *minimalComponentSize=1*, *rankingRule='Copeland'*, *quantiles=None*, *strategy='average'*, *ndigits=0*, *colorLevels=None*, *pageTitle='Voting Heatmap'*, *Correlations=True*, *Threading=False*, *nbrOfCPUs=1*, *Debug=False*)
Show the linear voting profile as a rank performance heatmap. The linear voting profile is previously saved to a stored Performance Tableau.

(see perfTabs.PerformanceTableau.showHTMLPerformanceHeatmap())

showLinearBallots(*IntegerWeights=True*)
show the linear ballots

showRankAnalysisTable(*Sorted=True*, *ndigits=0*, *Debug=False*)
Print the rank analysis tableau.

If Sorted (True by default), the candidates are ordered by increasing Borda Scores.

In case of decimal voters weights, ndigits allows to format the decimal precision of the numerical output.

class votingProfiles.**RandomApprovalVotingProfile**(*numberOfVoters=9*, *numberOfCandidates=5*, *minSizeOfBallot=1*, *maxSizeOfBallot=2*, *seed=None*)

Bases: *votingProfiles.ApprovalVotingProfile*

A specialized class for approval voting profiles.

generateRandomApprovalBallot(*minSizeOfBallot*, *maxSizeOfBallot*, *seed=None*)
Renders a randomly generated approval ballot.

generateRandomDisApprovalBallot(*minSizeOfBallot*, *maxSizeOfBallot*, *seed=None*)
Renders a randomly generated approval ballot.

class votingProfiles.**RandomLinearVotingProfile**(*numberOfVoters=9*, *numberOfCandidates=5*, *votersWeights=None*, *seed=None*)

Bases: *votingProfiles.LinearVotingProfile*

A specialized class for random linwear voting profiles. Random reation parameters:

numberOfVoters=5, numberOfCandidates=5, votersWeights = optional list of positive integers for instance [2,3,4,1,5].

generateRandomLinearBallot(*seed*)
Renders a randomly generated linear ballot.

class votingProfiles.**RandomVotingProfile**(*numberOfVoters=9*, *numberOfCandidates=5*, *hasRandomWeights=False*, *maxWeight=10*, *seed=None*, *Debug=False*)

Bases: *votingProfiles.VotingProfile*

A subclass for generating random voting profiles.

generateRandomBallot(*seed*, *Debug=False*)
Renders a randomly generated approval ballot from a shuffled list of candidates for each voter.

class votingProfiles.**VotingProfile**(*fileVotingProfile=None*, *seed=None*)

Bases: object

A generic class for storing voting profiles.

General structure:

```
candidates = OrderedDict([('a', ...),('b', ...),('c', ...), ( ... ) ])
voters = OrderedDict([
('1', {'weight':1.0}),
('2', {'weight':1.0}),
...,
])
ballot = {      # voters x candidates x candidates
    '1': {      # bipolar characteristic {-1,0,1} of each voter's
           'a': { 'a':0,'b':-1,'c':0, ...},    # pairwise preferences
           'b': { 'a':1,'b':0, 'c':1, ...},
           'c': { 'a':0,'b':-1,'c':0, ...},
           ...,
    },
    '2': { 'a': { 'a':0, 'b':0, 'c':1, ...},
           'b': { 'a':0, 'b':0, 'c':1, ...},
           'c': { 'a':-1,'b':-1,'c':0, ...},
           ...,
    },
    ...,
}
```

save(*name='tempVprofile'*)
Persistant storage of an approval voting profile.

showAll(*WithBallots=True*)
Show method for <VotingProfile> instances.

showVoterBallot(*voter*, *hasIntegerValuation=False*)
Show the actual voting of a voter.

Back to the *Installation*

2.2.14 linearOrders module

A tutorial with coding examples is available here: *Computing the winner of an election*

class linearOrders.**CopelandOrder**(*other*, *coDual=False*, *Debug=False*)
Bases: *linearOrders.LinearOrder*

instantiates the Copèeland Order from a given bipolar-valued Digraph instance

showScores(*direction='descending'*)

class linearOrders.**ExtendedPrudentDigraph**(*other*, *prudentBetaLevel=None*, *CoDual=False*, *Debug=False*)
Bases: *digraphs.Digraph*

Instantiates the associated extended prudent codual of the digraph enstance. Instantiates as other.__class__ ! Copies the case given the description, the criteria and the evaluation dictionary into self.

class linearOrders.**KemenyOrder**(*other*, *orderLimit=7*, *Debug=False*)
Bases: *linearOrders.LinearOrder*

instantiates the exact Kemeny Order from a given bipolar-valued Digraph instance of small order

class linearOrders.**KohlerOrder**(*other*, *coDual=False*, *Debug=False*)
Bases: *linearOrders.LinearOrder*

instantiates the Kohler Order from a given bipolar-valued Digraph instance

class linearOrders.**LinearOrder**(*file=None*, *order=7*)
Bases: *digraphs.Digraph*

abstract class for digraphs which represent linear orders.

computeKemenyIndex(*other*)
renders the Kemeny index of the self.relation (linear order) compared with a given bipolar-valued relation of a compatible other digraph (same nodes or actions).

computeOrder()
computes the linear ordering from lowest (worst) to highest (best) of an instance of the LinearOrcer class by sorting by indegrees (gamma[x][1]).

computeRanking()
computes the linear ordering from lowest (best, rankk = 1) to highest (worst rank=n) of an instance of the LinearOrcer class by sorting by outdegrees (gamma[x][0]).

exportDigraphGraphViz(*fileName=None*, *bestChoice=set()*, *worstChoice=set()*, *noSilent=True*, *graphType='png'*, *graphSize='7*, *7'*)
export GraphViz dot file for digraph drawing filtering.

exportGraphViz(*fileName=None*, *isValued=True*, *bestChoice=set()*, *worstChoice=set()*, *noSilent=True*, *graphType='png'*, *graphSize='7, 7'*)
export GraphViz dot file for linear order drawing filtering.

htmlOrder()
returns the html encoded presentation of a linear order

htmlRanking()
returns the html encoded presentation of a linear order

showOrdering()
shows the linearly ordered actions in list format.

showRanking()
shows the linearly ordered actions in list format.

class linearOrders.**NetFlowsOrder**(*other*, *coDual=False*, *Debug=False*)
Bases: *linearOrders.LinearOrder*

instantiates the net flows Order from a given bipolar-valued Digraph instance

showScores(*direction='descending'*)

class linearOrders.**OutFlowsOrder**(*other*, *coDual=False*, *Debug=False*)
Bases: *linearOrders.LinearOrder*

instantiates the out flows Order from a given bipolar-valued Digraph instance

showScores(*direction='descending'*)

class linearOrders.**PrincipalOrder**(*other*, *Colwise=True*, *imageType=None*, *plotFileName='principalOrdering'*, *tempDir=None*, *Debug=False*)
Bases: *linearOrders.LinearOrder*

instantiates the order from the scores obtained by the first princiapl axis of the eigen deomposition of the covariance of the outdegrees of the valued digraph 'other'.

class linearOrders.**RandomLinearOrder**(*numberOfActions=10*, *Debug=False*, *OutrankingModel=False*, *Valued=False*, *seed=None*)
Bases: *linearOrders.LinearOrder*

Instantiates random linear orders

class linearOrders.**RankedPairsOrder**(*other*, *coDual=False*, *Leximin=False*, *Cpp=False*, *isValued=False*, *isExtendedPrudent=False*, *Debug=False*)
Bases: *linearOrders.LinearOrder*

instantiates the Extended Prudent Ranked Pairs Order from a given bipolar-valued Digraph instance

Back to the *Installation*

2.2.15 weakOrders module

class weakOrders.**KemenyWeakOrder**(*other*, *orderLimit=7*, *Debug=False*)
Bases: *weakOrders.WeakOrder*

Specialization of the abstract WeakOrder class for weak orderings resulting from the epistemic disjunctive fusion (omax operator) of all potential Kemeny linear orderings.

class weakOrders.**KohlerArrowRaynaudFusionDigraph**(*outrankingDigraph*, *fusionOperator='o-max'*, *Threading=True*, *Debug=False*)
Bases: *weakOrders.WeakOrder*

Specialization of the abstract WeakOrder class for ranking-by-choosing orderings resulting from the epistemic disjunctive (o-max fusion) or conjunctive (o-min operator) fusion of a Kohler linear best ordering and an Arrow-Raynaud linear worst ordering.

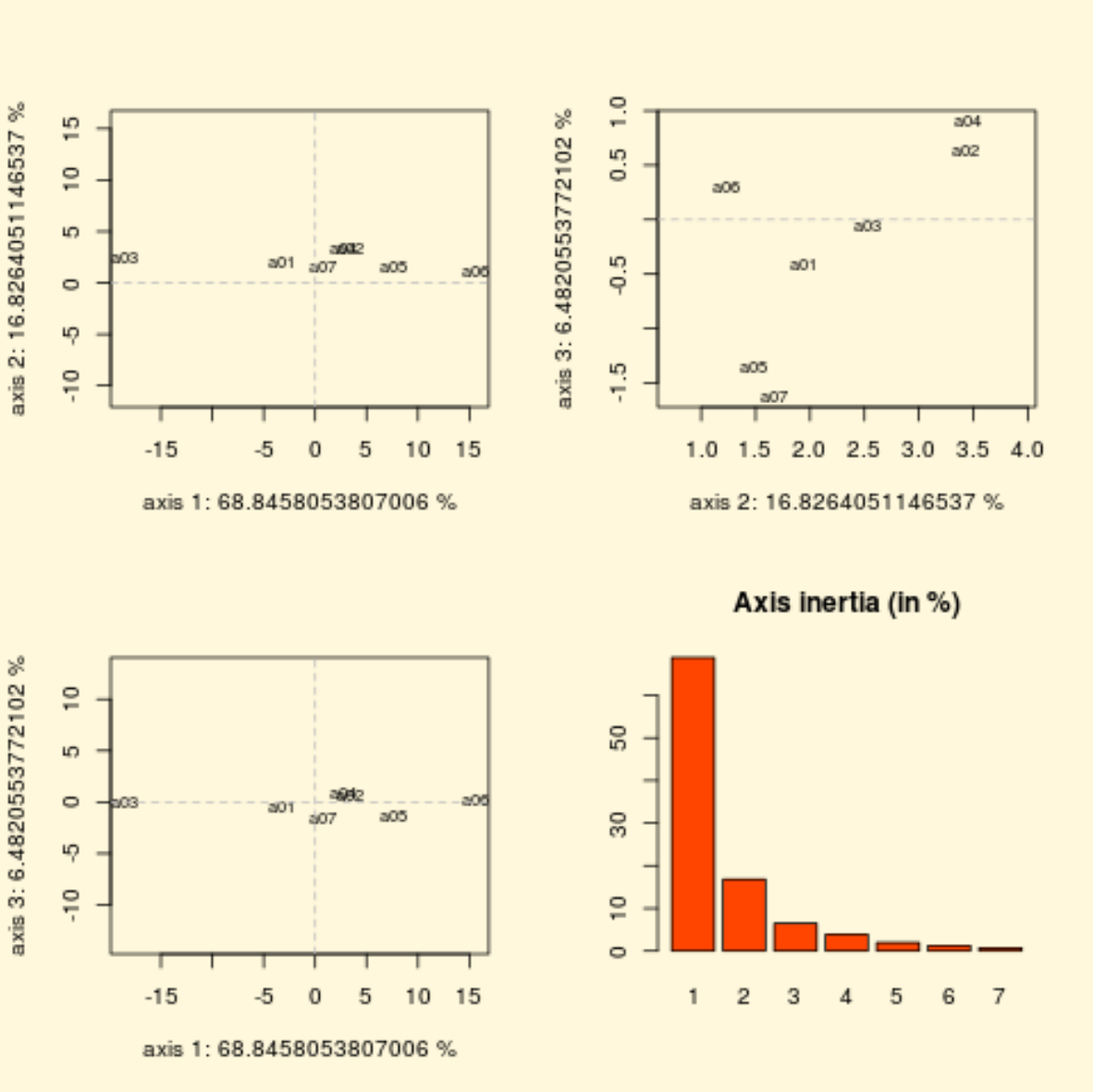
class weakOrders.**PrincipalInOutDegreesOrdering**(*other*, *fusionOperator='o-max'*, *imageType=None*, *plotFileName=None*, *Threading=True*, *Debug=False*)

Bases: *weakOrders.WeakOrder*

Specialization of abstract WeakOrder class for ranking by fusion of the principal orders of the variance-covariance of in- (Colwise) and outdegrees (Rowwise).

Example Python3 session with same outranking digraph g as shown in the RankingByChoosingDigraph example session (see below).

```
>>> from weakOrders import PrincipalInOutDegreesOrdering
>>> pro = PrincipalInOutDegreesOrdering(g,imageType="png",\
                plotFileName="proWeakOrdering")
>>> pro.showWeakOrder()
Ranking by Choosing and Rejecting
 1st ranked ['a06'] (1.00)
   2nd ranked ['a05'] (1.00)
     3rd ranked ['a02'] (1.00)
       4th ranked ['a04'] (1.00)
       4th last ranked ['a04'] (1.00)
     3rd last ranked ['a07'] (1.00)
   2nd last ranked ['a01'] (1.00)
 1st last ranked ['a03'] (1.00)
>>> pro.showPrincipalScores(ColwiseOrder=True)
List of principal scores
Column wise covariance ordered
action       colwise         rowwise
a06          15.52934        13.74739
a05          7.71195         4.95199
a02          3.40812         0.70554
a04          2.76502         0.15189
a07          0.66875         -1.77637
a01          -3.19392        -5.36733
a03          -18.51409       -21.09102
```



computeWeakOrder (*ColwiseOrder=False*)
Specialisation for PrincipalInOutDegreesOrderings.

exportGraphViz (*fileName=None*, *direction='ColwiseOrder'*, *Comments=True*, *graphType='png'*, *graphSize='7, 7'*, *fontSize=10*)
Specialisation for PincipalInOutDegrees class.

direction = "Colwise" (best to worst, default) | "Rowwise" (worst to best)

showPrincipalScores (*ColwiseOrder=False*, *RowwiseOrder=False*)
showing the principal in- (Colwise) and out-degrees (Rowwise) scores.

showWeakOrder (*ColwiseOrder=False*)
Specialisation for PrincipalInOutDegreesOrderings.

class weakOrders.**RankingByBestChoosingDigraph** (*digraph*, *Normalized=True*, *CoDual=False*, *Debug=False*)
Bases: *weakOrders.RankingByChoosingDigraph*

Specialization of abstract WeakOrder class for computing a ranking by best-choosing.

showWeakOrder()
Specialisation of showWeakOrder() for ranking-by-best-choosing results.

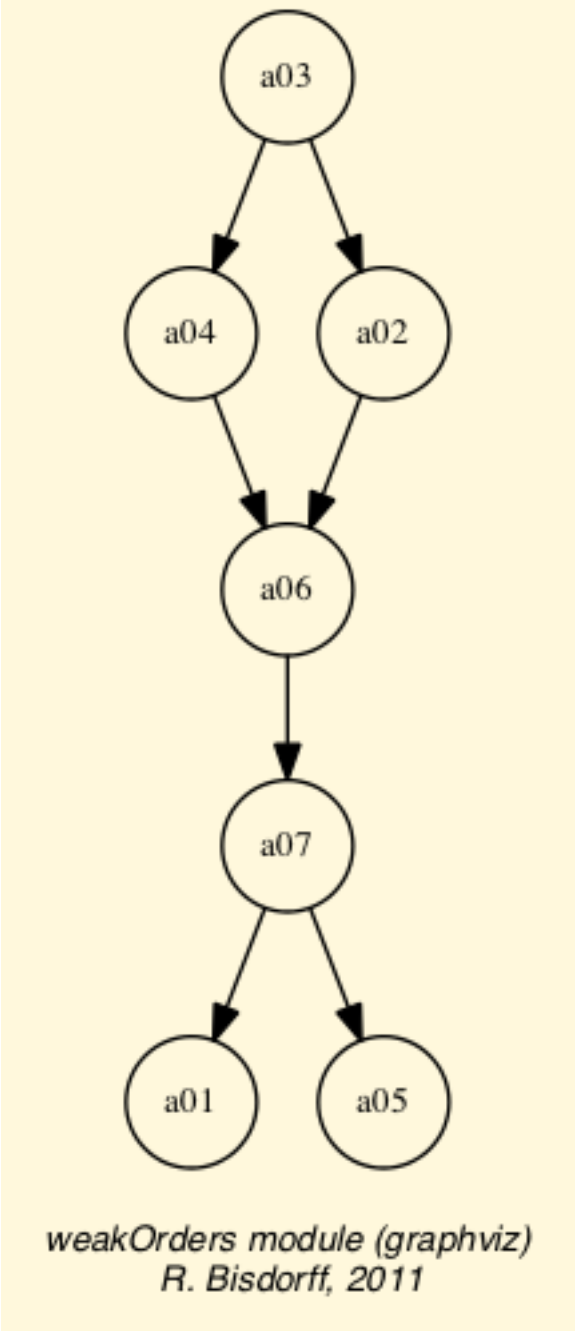
class weakOrders.**RankingByChoosingDigraph**(*other*, *fusionOperator='o-max'*, *CoDual=False*, *Debug=False*, *CppAgrum=False*, *Threading=True*)

Bases: *weakOrders.WeakOrder*

Specialization of the abstract WeakOrder class for ranking-by-Rubis-choosing orderings.

Example python3 session:

```
>>> from outrankingDigraphs import *
>>> t = RandomCBPerformanceTableau(numberOfActions=7,\
           numberOfCriteria=5,\
           weightDistribution='equiobjectives')
>>> g = BipolarOutrankingDigraph(t,Normalized=True)
>>> g.showRelationTable()
* ---- Relation Table -----
  S   |   'a01'  'a02'  'a03'  'a04'  'a05'  'a06'  'a07'
 -----|-------------------------------------------------------
'a01' |   +0.00  -1.00  -1.00  -0.33  +0.00  +0.00  +0.00
'a02' |   +1.00  +0.00  -0.17  +0.33  +1.00  +0.33  +0.67
'a03' |   +1.00  +0.67  +0.00  +0.33  +0.67  +0.67  +0.67
'a04' |   +0.33  +0.17  -0.33  +0.00  +1.00  +0.67  +0.67
'a05' |   +0.00  -0.67  -0.67  -1.00  +0.00  -0.17  +0.33
'a06' |   +0.33  +0.00  -0.33  -0.67  +0.50  +0.00  +1.00
'a07' |   +0.33  +0.00  -0.33  -0.67  +0.50  +0.17  +0.00
>>> from weakOrders import RankingByChoosingDigraph
>>> rbc = RankingByChoosingDigraph(g)
>>> rbc.showWeakOrder()
Ranking by Choosing and Rejecting
  1st ranked ['a03'] (0.47)
   2nd ranked ['a02', 'a04'] (0.58)
    3rd ranked ['a06'] (1.00)
    3rd last ranked ['a06'] (1.00)
   2nd last ranked ['a07'] (0.50)
  1st last ranked ['a01', 'a05'] (0.58)
>>> rbc.exportGraphViz('weakOrdering')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to converse-dual_rel_randomCBperftab.dot
dot -Grankdir=BT -Tpng converse-dual_rel_randomCBperftab.dot
   -o weakOrdering.png
```




```
>>> rbc.showOrderedRelationTable(direction="decreasing")
* ---- Relation Table -----
  S    |  'a03'  'a04'  'a02'  'a06'  'a07'  'a01'  'a05'
 ------|-------------------------------------------------
'a03' |   -      0.33   0.17   0.33   0.33   1.00   0.67
'a04' |  -0.33    -     0.00   0.67   0.67   0.33   1.00
'a02' |  -0.17   0.00    -     0.33   0.67   1.00   0.67
'a06' |  -0.33  -0.67  -0.33    -     0.17   0.33   0.17
'a07' |  -0.33  -0.67  -0.67  -0.17    -     0.33   0.33
'a01' |  -1.00  -0.33  -1.00  -0.33  -0.33    -     0.00
'a05' |  -0.67  -1.00  -0.67  -0.17  -0.33   0.00    -
```

computeRankingByBestChoosing(*Forced=False*)
Dummy for blocking recomputing without forcing.

computeRankingByLastChoosing(*Forced=False*)
Dummy for blocking recomputing without forcing.

showRankingByChoosing(*rankingByChoosing=None*)
Dummy for showWeakOrder method

showWeakOrder(*rankingByChoosing=None*)
Specialization of generic method. Without argument, a weak ordering is recomputed from the valued self relation.

class weakOrders.**RankingByLastChoosingDigraph**(*digraph*, *Normalized=True*, *CoDual=False*, *Debug=False*)
Bases: *weakOrders.RankingByChoosingDigraph*

Specialization of abstract WeakOrder class for computing a ranking by rejecting.

showWeakOrder()
Specialisation of showWeakOrder() for ranking-by-last-choosing results.

class weakOrders.**RankingByPrudentChoosingDigraph**(*digraph*, *CoDual=False*, *Normalized=True*, *Odd=True*, *Limited=0.2*, *Comments=False*, *Debug=False*, *SplitCorrelation=True*)
Bases: *weakOrders.RankingByChoosingDigraph*

Specialization for ranking-by-rejecting results with prudent single elimination of chordless circuits. By default, the cut level for circuits elimination is set to 20% of the valuation domain maximum (1.0).

class weakOrders.**WeakOrder**(*file=None*, *order=7*)
Bases: *digraphs.Digraph*

Abstract class for weak orderings' specialized methods.

exportDigraphGraphViz(*fileName=None*, *bestChoice=set()*, *worstChoice=set()*, *noSilent=True*, *graphType='png'*, *graphSize='7,7'*)
export GraphViz dot file for digraph drawing filtering.

exportGraphViz(*digraphClass=None*, *fileName=None*, *relation=None*, *direction='best'*, *noSilent=True*, *graphType='png'*, *graphSize='7,7'*, *fontSize=10*)
export GraphViz dot file for weak order (Hasse diagram) drawing filtering.

showOrderedRelationTable(*direction='decreasing'*, *originalRelation=False*)
Showing the relation table in decreasing (default) or increasing order.

showRankingByChoosing(*actionsList=None*, *rankingByChoosing=None*)
Dummy name for showWeakOrder() method

showWeakOrder(*rankingByChoosing=None*)
A show method for self.rankinByChoosing result.

class weakOrders.**WeakRankingOrder**(*other*, *rankings*, *Debug=False*)
Bases: *weakOrders.WeakOrder*

Specialization of the abstract WeakOrder class for weak orderings resulting from the epistemic disjunctive fusion (omax operator) of a list of rankings.

Example application:

```
>>> from weakOrders import WeakRankingOrder
>>> from sparseOutrankingDigraphs import PreRankedOutrankingDigraph
>>> t = RandomPerformanceTableau()
```

```
>>> pr = PreRankedOutrankingDigraph(t,10,quantilesOrderingStrategy='average')
>>> r1 = qr.boostedRanking
>>> pro = PreRankedOutrankingDigraph(t,10,quantilesOrderingStrategy='optimistic')
>>> r2 = pro.boostedRanking
>>> prp = QuantilesRankingDigraph(t,10,quantilesOrderingStrategy='pessimistic')
>>> r3 = prp.boostedRanking
>>> wqr = WeakQuantilesRankingOrder(pr,[r1,r2,r3])
>>> wqr.exportGraphViz('partialOrdering',graphType="pdf")
```

Back to the *Installation*

2.2.16 randomNumbers module

Python3+ implementation of random number generators Copyright (C) 2014-2018 Raymond Bisdorff

> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

class randomNumbers.**CauchyRandomVariable**(*position=0.0*, *scale=1.0*, *seed=None*, *Debug=False*)

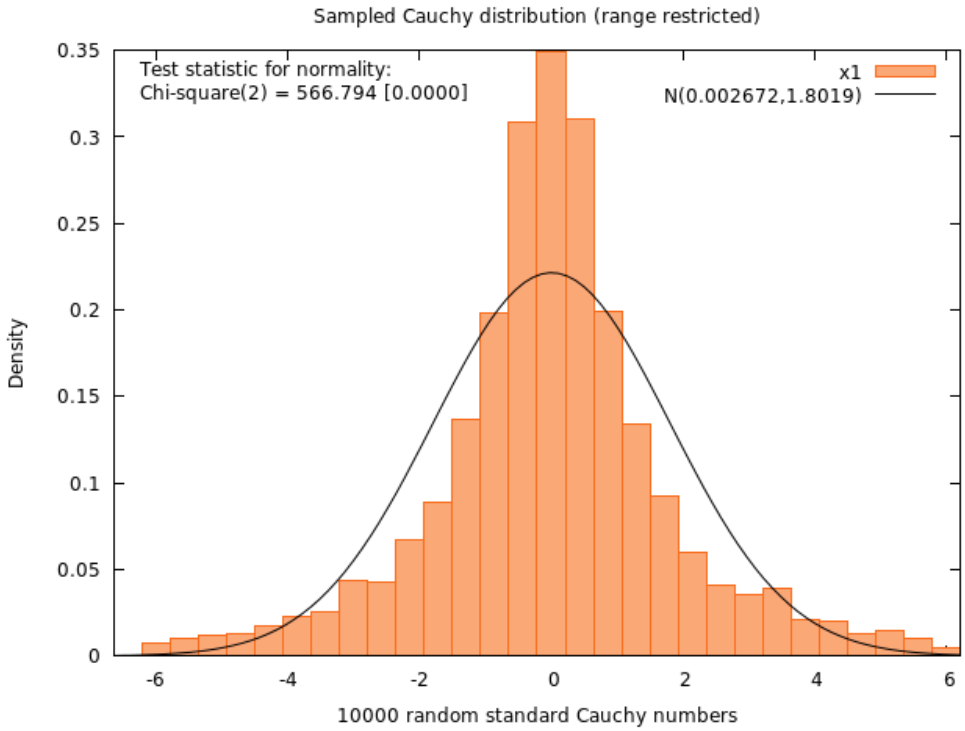
Bases: object

Cauchy random variable generator.

Parameters:

- position := median (default=0.0) of the Cauchy distribution
- scale := typical spread (default=1.0) with respect to median
- seed := integer (default=None) for fixing the sequence generation.

Cauchy quantile (inverse cdf) function: Q(x|position,scale) = position + scale*tan[pi(x-1/2)]



random()
generating a Cauchy random number.

class randomNumbers.**DiscreteRandomVariable**(*discreteLaw=None*, *seed=None*, *Debug=False*)

Bases: object

Discrete random variable generator

Parameters:

discreteLaw := dictionary with integer values
as keys and probabilities as float values,
seed := integer for fixing the sequence generation.

Example usage:

```
>>> from randomNumbers import DiscreteRandomVariable
>>> discreteLaw = {0:0.0478,
                   1:0.3349,
                   2:0.2392,
                   3:0.1435,
                   4:0.0957,
                   5:0.0670,
                   6:0.0478,
                   7:0.0096,
                   8:0.0096,
                   9:0.0048,}
## initialze the random generator
>>> rdv = DiscreteRandomVariable(discreteLaw,seed=1)
## sample discrete random variable and
## count frequencies of obtained values
>>> sampleSize = 1000
>>> frequencies = {}
```

```
>>> for i in range(sampleSize):
        x = rdv.random()
        try:
            frequencies[x] += 1
        except:
            frequencies[x] = 1
## print results
>>> results = [x for x in frequencies]
>>> results.sort()
>>> counts= 0.0
>>> for x in results:
        counts += frequencies[x]
        print  ('%s, %d, %.3f, %.3f' % (x, frequencies[x],
                  float(frequencies[x])/float(sampleSize),
                  discreteLaw[x]))
>>> print ('# of valid samples = %d' % counts)
```

random()
generating discrete random values from a discrete random variable.

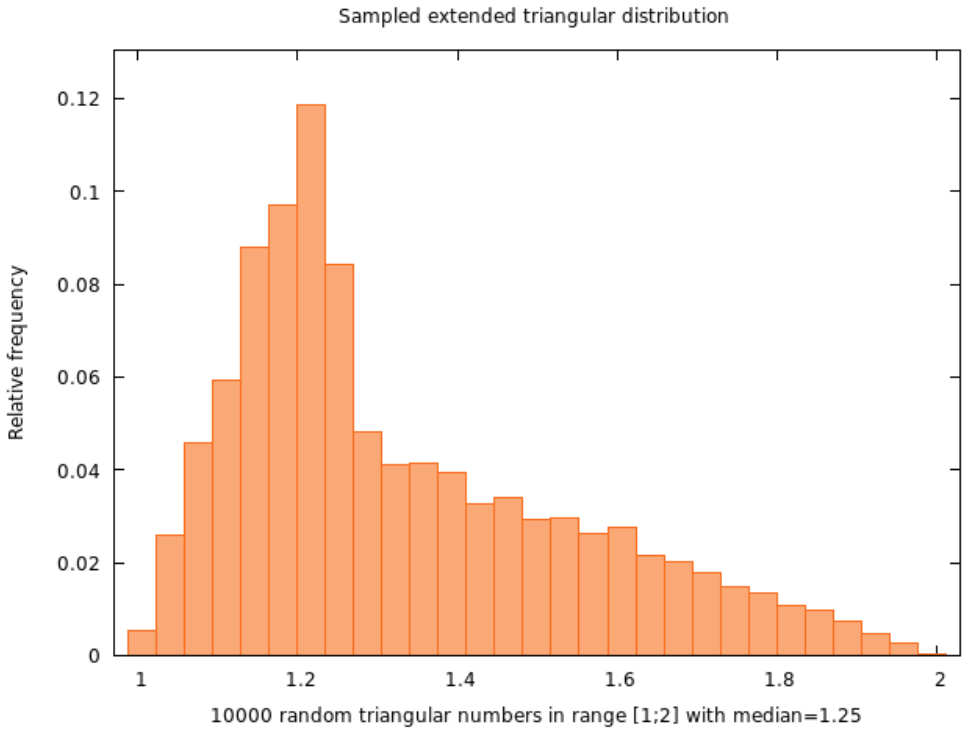
class randomNumbers.**ExtendedTriangularRandomVariable**(*lowLimit=0.0*, *highLimit=1.0*, *mode=None*, *probRepart=0.5*, *seed=None*, *Debug=False*)

Bases: object

Extended triangular random variable generator

Parameters:

- mode := most frequently observed value
- probRepart := probability mass distributed until the mode
- seed := integer for fixing the sequence generation.



random()
generating an extended triangular random number.

class randomNumbers.**QuasiRandomFareyPointSet**(*n=20*, *s=3*, *seed=None*, *Randomized=True*, *fileName='farey'*, *Debug=False*)

Bases: *randomNumbers.QuasiRandomPointSet*

Constructor for rendering a Farey point set of dimension *s* and max denominateor *n* which is *fully projection regular* in the s-dimensional real-valued [0,1]^s hypercube. The lattice constructor uses a randomly shuffled Farey series for the point construction. The resulting point set is stored in a self.pointSet attribute and saved by default in a CSV formatted file.

Parameters:

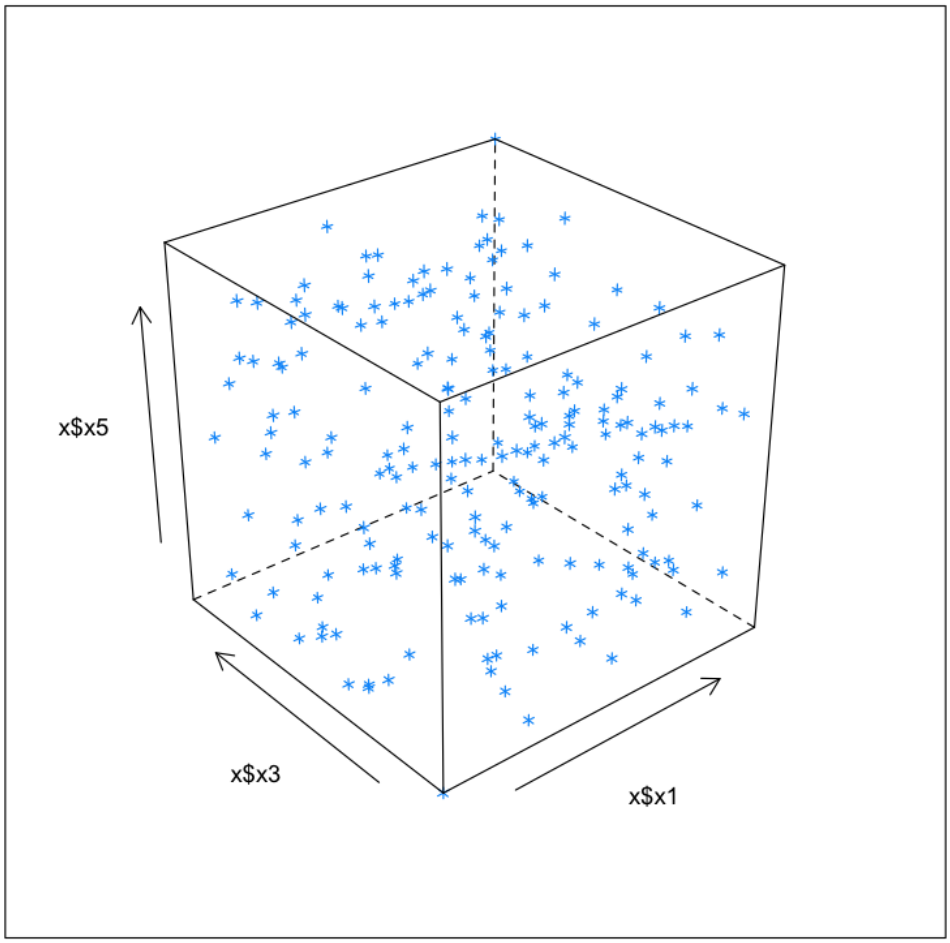
- *n* : (default=20) maximal denominator of the Farey series
- *s* : (default=3) dimension of the hypercube
- *seed* : for regenerating the same Farey point set
- *Randomized* : (default=True) On each dimension, the points are randomly shifted (mod 1) to avoid constant projections for equal dimension index distances.
- *fileName*: (default='farey') name -without the csv suffix- of the stored result file.

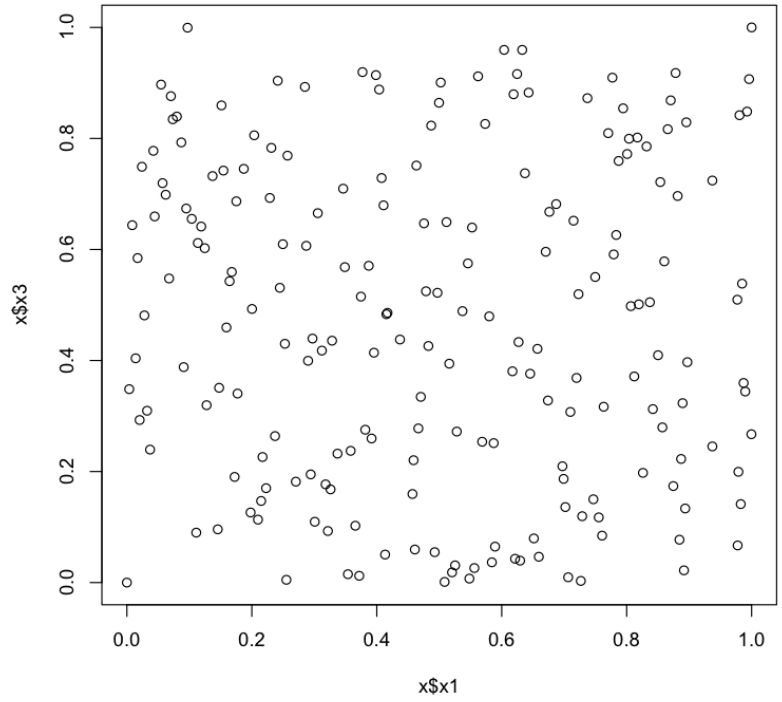
Sample Python session:

```
>>> from randomNumbers import QuasiRandomFareyPointSet
>>> qrfs = QuasiRandomFareyPointSet(n=20,s=5,Randomized=True,
                                    fileName='testFarey')
>>> print(qrfs.__dict__.keys())
dict_keys(['n', 's', 'Randomized', 'seed', 'fileName', 'Debug',
           'fareySeries', 'seriesLength', 'shuffledFareySeries',
           'pointSet', 'pointSetCardinality'])
>>> print(qrfs.fareySeries)
[0.0, 0.04, 0.04166, 0.0435, 0.04545, 0.0476, 0.05, 0.05263,
 0.0555, 0.058823529411764705, ...]
>>> print(qrfs.seriesLength)
201
>>> print(qrfs.pointSet[)
[(0.0, 0.0, 0.0, 0.0, 0.0), (0.5116, 0.4660, 0.6493, 0.41757, 0.3663),
 (0.9776, 0.1153, 0.0669, 0.7839, 0.5926), (0.6269, 0.5329, 0.4332, 0.0102, 0.
↪6126),
 (0.0445, 0.8992, 0.6595, 0.0302, 0.6704), ...]
>>> print(qrfs.pointSetCaridinality)
207
```

The resulting point set may be inspected in an R session:

```
 > x = read.csv('testFarey.csv')
 > x[1:5,]
   x1       x2       x3       x4       x5
 1   0.000000 0.000000 0.000000 0.000000 0.000000
 2   0.511597 0.466016 0.649321 0.417573 0.366316
 3   0.977613 0.115336 0.066893 0.783889 0.592632
 4   0.626933 0.532909 0.433209 0.010205 0.612632
 5   0.044506 0.899225 0.659525 0.030205 0.670410
> library('lattice')
 > cloud(x$x5 ~ x$x1 + x$x3)
 > plot(x$x1,x$x3)
```


x$x5
x$x3
x$x1



class randomNumbers.**QuasiRandomKorobovPointSet**(*n=997*, *s=3*, *a=383*, *Randomized=False*, *seed=None*, *fileName='korobov'*, *Debug=False*)

Bases: *randomNumbers.QuasiRandomPointSet*

Constructor for rendering a Korobov point set of dimension *n* which is *fully projection regular* in the s-dimensional real-valued [0,1)^s hypercube. The constructor uses a MLCG generator with potentially a full period. The point set is stored in a self.sequence attribute and saved in a CSV formatted file.

Source: Chr. Lemieux, Monte Carlo and quasi Monte Carlo Sampling Springer 2009 Fig. 5.12 p. 176.

Parameters:

- *n* : (default=997) number of Korobov points and modulus of the underlying MLCG
- *s* : (default=3) dimension of the hypercube
- *Randomized* : (default=False) the sequence is randomly shifted (mod 1) to avoid cycling when $s > n$
- *a* : (default=383) MLCG coefficient ($0 < a < n$), primitive with *n*. The choice of *a* and *n* is crucial for getting an MLCG with full period and hence a fully projection-regular sequence. A second good pair is given with $n = 1021$ (prime) and $a = 76$.
- *fileName*: (default='korobov') name -without the csv suffix- of the stored result.

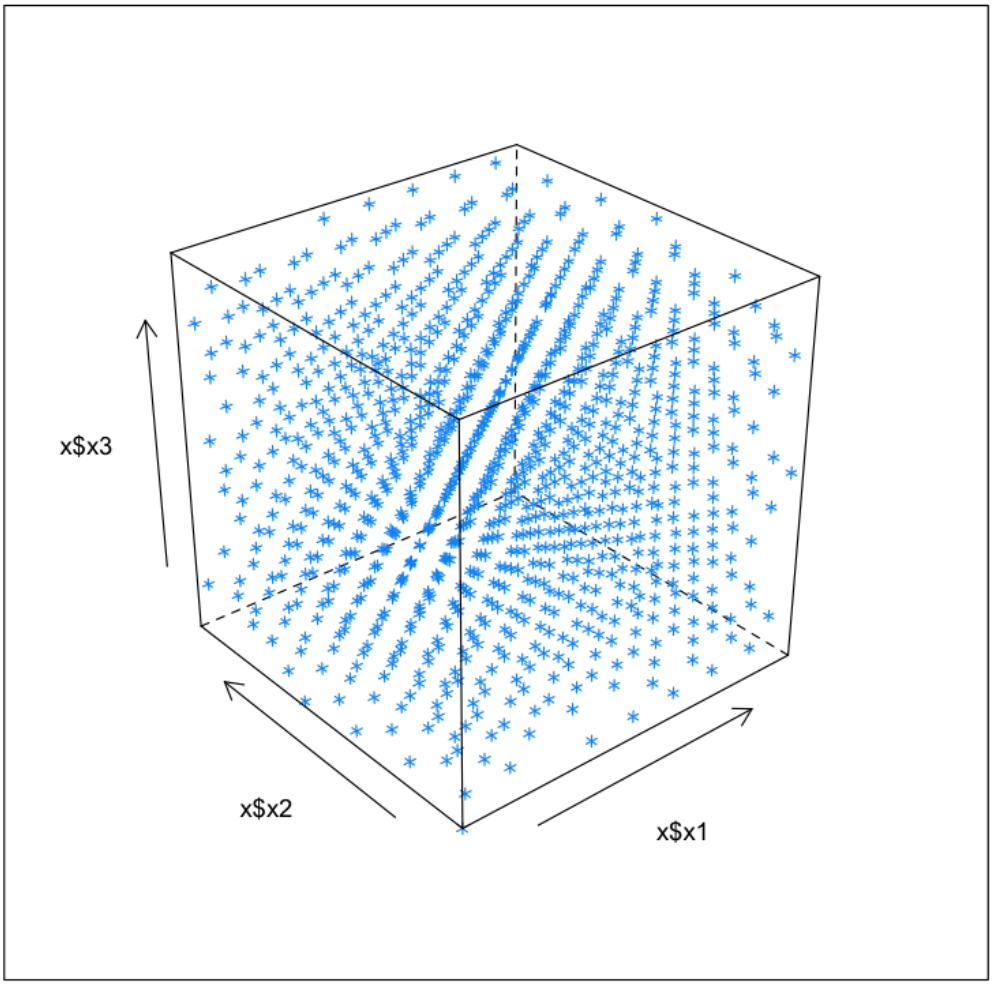
Sample Python session:

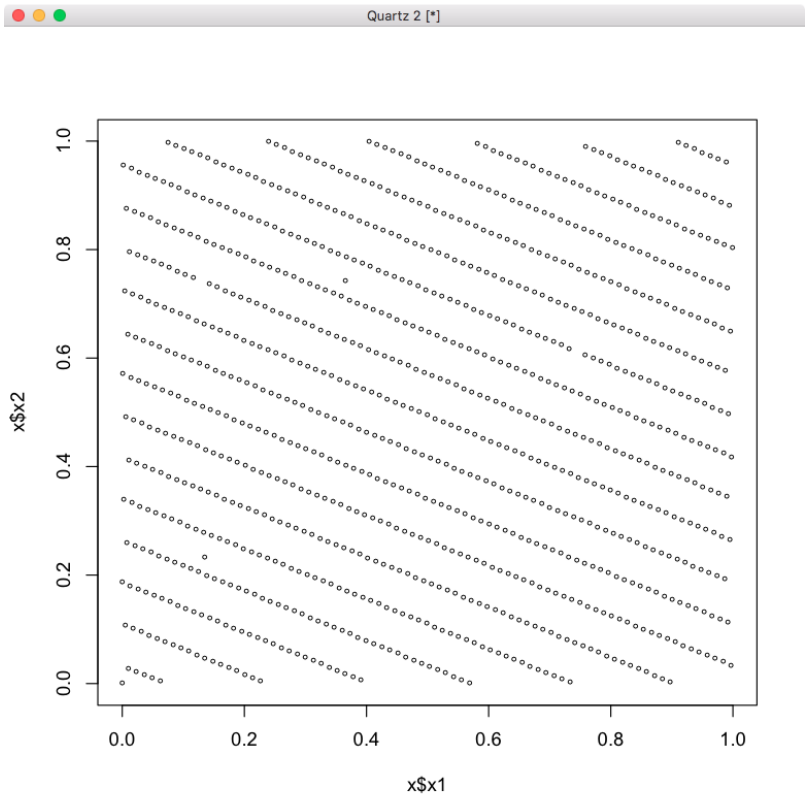
```
>>> from randomNumbers import QuasiRandomKorobovPointSet
>>> kor = QuasiRandomKorobovPointSet(Debug=True)
0 [0.0, 0.0, 0.0]
1 [0.13536725313948247, 0.23158619430934912, 0.8941657924971758]
2 [0.36595043842175035, 0.7415995294344084, 0.7035940773517395]
3 [0.8759637735468097, 0.5510278142889722, 0.714627176649633]
```

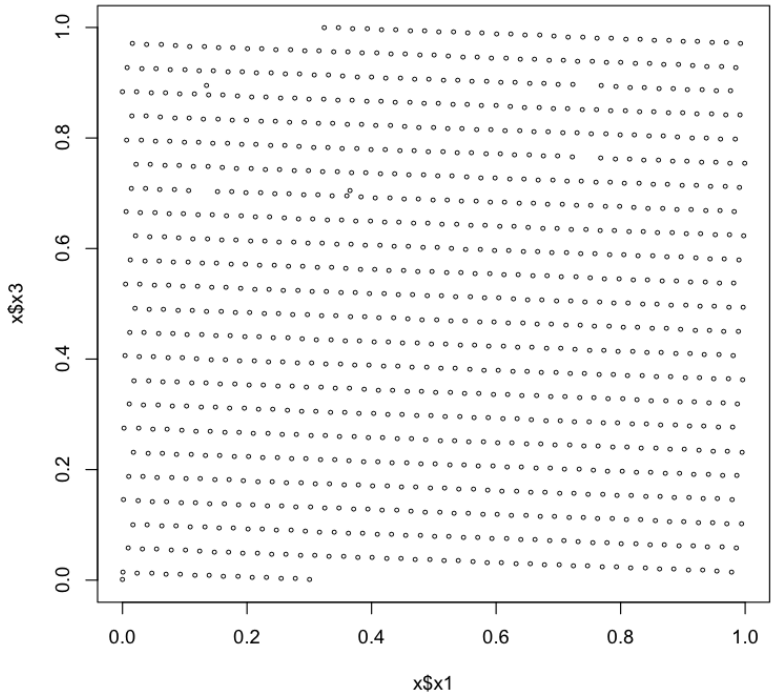
```
4 [0.6853920584013734, 0.5620609135868657, 0.9403042077429129]
5 [0.6964251576992669, 0.7877379446801456, 0.3746071164690914]
```

The resulting Korobov sequence may be inspected in an R session:

```
> x = read.csv('korobov.csv')
> x[1:5,]
>    x1       x2       x3
1 0.000000 0.000000 0.000000
2 0.135367 0.231586 0.894166
3 0.365950 0.741600 0.703594
4 0.875964 0.551028 0.714627
5 0.685392 0.562061 0.940304
> library('lattice')
> cloud(x$x3 ~ x$x1 + x$x2)
> plot(x$x1,x$x2,pch='°')
> plot(x$x1,x$x3,pch="°")
```







class randomNumbers.**QuasiRandomPointSet**

Bases: object

Abstract class for generic quasi random point set methods and tools.

countHits (*regionLimits*, *pointSet=None*)
Counting hits of a quasi random point set in given regionLimits.

testFct (*seq=None*, *buggyRegionLimits=(0.45*, *0.55)*)
Tiny buggy hypercube for testing a quasi random Korobov 3D sequence.

testUniformityDiscrepancy (*k=4*, *pointSet=None*, *fileName='testUniformity'*, *Debug=True*)
Count the number of point in each partial hypercube [(x-1)/k,x/k]^d where x integer ands 0 < x <= k.

class randomNumbers.**QuasiRandomUniformPointSet** (*n=20*, *s=3*, *seed=None*, *Randomized=True*, *fileName='uniform'*, *Debug=False*)

Bases: *randomNumbers.QuasiRandomPointSet*

Constructor for rendering a quai random point set of dimension *s* and max denominateor *n* which is *fully projection regular* in the s-dimensional real-valued [0,1]^s hypercube. The lattice constructor uses a randomly shuffled *uniform* series for the point construction. The resulting point set is stored in a self.pointSet attribute and saved by default in a CSV formatted file.

Parameters:

- *n* : (default=100) denominator of the uniform series x/n with 0 <= x <= n
- *s* : (default=3) dimension of the hypercube
- *seed* : for regenerating the same Farey point set
- *Randomized* : (default=True) On each dimension, the points are randomly shifted (mod 1) to avoid constant projections for equal dimension index distances.
- *fileName*: (default='uniform') name -without the csv suffix- of the stored result file.

Back to the *Installation*

2.2.17 digraphsTools module

Python3+ implementation of Digraph3 tools

digraphsTools.**all_perms** (*str*)

digraphsTools.**flatten** (*iterable*, *ltypes=<class 'collections.abc.Iterable'>*)
Flattens a list of lists into a flat list.

Main usage:

```
>>> listOfLists = [[1,2],[3],[4]]
>>> [x for x in flatten(listOfLists)]
[1,2,3,4]
```

digraphsTools.**generateBipolarGrayCode**(*n*)
Bipolar version of generateGrayCode. X is a partially determined -1 vector.

digraphsTools.**generateGrayCode**(*n*)
Knuth ACP (4) 7.2.1.1. p.6 Algorithm G

digraphsTools.**generateLooplessGrayCode**(*n*)
Knuth ACP (4) 7.2.1.1. p.7 Algorithm L

digraphsTools.**grayCode**(*n*)

digraphsTools.**omax**(*Med*, *L*, *Debug=False*)
epistemic disjunction for bipolar outranking characteristics computation: Med is the valuation domain median and L is a list of r-valued statement characteristics.

digraphsTools.**omin**(*Med*, *L*, *Debug=False*)
epistemic conjunction of a list L of bipolar outranking characteristics. Med is the given valuation domain median.

digraphsTools.**powerset**(*S*)
Power set generator iterator.

Parameter S may be any object that is accepted as input by the set class constructor.

digraphsTools.**ranking2preorder**(*R*)

digraphsTools.**timefn**(*fn*)
A decorator for automate run time measurements from "High Performance Python" by M Gorelick & I Ozswald O'Reilly 2014 p.27

digraphsTools.**total_size**(*o*, *handlers={}*, *verbose=False*)
Returns the approximate memory footprint of an object and all of its contents.

Automatically finds the contents of the following containers and their subclasses: tuple, list, deque, dict, set, frozenset, Digraph and BigDigraph. To search other containers, add handlers to iterate over their contents:

handlers = {SomeContainerClass: iter, OtherContainerClass: OtherContainerClass.get_elements}

See http://code.activestate.com/recipes/577504/

Back to the *Installation*

2.2.18 arithmetics module

class arithmetics.**QuadraticResiduesDigraph**(*integers=[3, 5, 7, 11, 13, 17, 19]*)
Bases: *digraphs.Digraph*

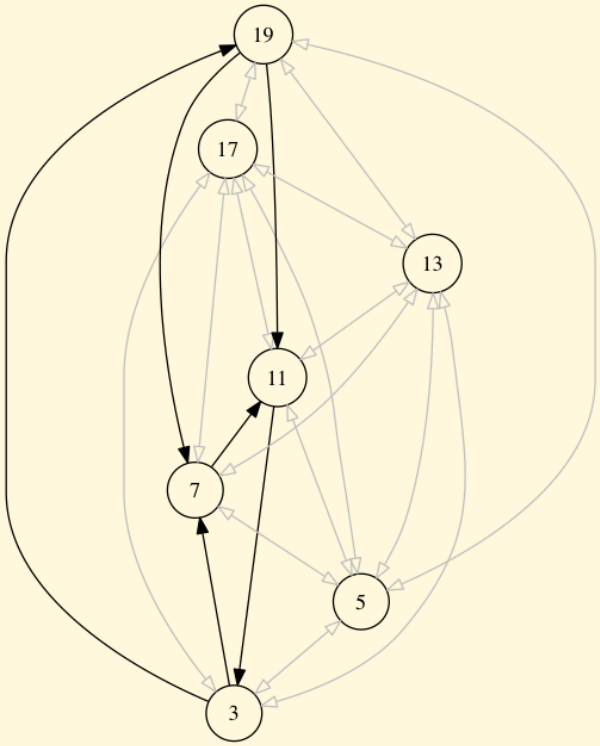
The **Legendre** symbol *(a/p)* of any pair of non null integers *a* and *p* is:

- **0** if *a* = 0 (mod p);
- **1** if *a* is a quadratic residue in *Zp*, ie *a* in *Qp*;
- **-1** if *a* is a non quadratic residue unit in *Zp*, ie *a* in *Up - Qp*.

The Legendre symbol hence defines a bipolar valuation on pairs of non null integers. The **reciprocity theorem** of the Legendre symbol states that, for *p* being an odd prime, *(a/p) = (p/a)*, apart from those pairs *(a/p)*, where *a = p* = 3 (mod 4). In this case, *(a/p) = -(p/a)*.

We may graphically illustrate the reciprocity theorem as follows:

```
>>> from arithmetics import *
>>> leg = QuadraticResiduesDigraph(primesBelow(20,Odd=True))
>>> from digraphs import AsymmetricPartialDigraph
>>> aleg = AsymmetricPartialDigraph(leg)
>>> aleg.exportGraphViz('legendreAsym')
```



arithmetics.**bezout**(*a*, *b*)

Renders d = gcd(a,b) and the Bezout coefficient x, y such that d = ax + by.

arithmetics.**computeFareySeries**(*n=7*, *AsFloats=False*, *Debug=False*)

Renders the Farey series, ie the ordered list of positive rational fractions with positive denominator lower or equal to n. For *n* = 1, we obtain: [[0,1],[1,1]].

Parametrs:

n: strictly positive integer (default = 7). *AsFloats*: If True (defaut False), renders the list of approximate floats corresponding to the rational fractions.

Source: Graham, Knuth, Patashnik, Sec. 4.5 in Concrete Mathematics 2nd Ed., Addison-Wesley 1994, pp 115-123.

arithmetics.**computePiDecimals**(*decimalWordLength=4*, *nbrOfWords=600*, *Comments=False*)

Renders at least decimalWordLenght * nbrOfWords (default: 4x600=2400) decimals of π. The Python transcription here recodes an original C code of unknown author (see[1]).

Uses the following infinite Euler series:

$$\pi = 2 * \sum_{n=0}^{\infty}[(n!)/(1 * 3 * 5... * (2n + 1))].$$

The series gives a new π decimal after adding in average 3.32 terms.

[1] *Source:* J.-P. Delahaye "*Le fascinant nombre* π", Pour la science Belin 1997 p.95.

```
>>> from arithmetics import computePiDecimals
>>> from time import time
>>> t0=time();piDecimals = computePiDecimals(decimalWordLength=3,nbrOfWords=100);
↪t1= time()
>>> print('pi = '+piDecimals[0]+'.'+piDecimals[1:])
pi = 3.14159265358979323846264338327950288419716939937510582097494459
23078164062862089986280348253421170679821480865132823066470938446 09
55058223172535940812848111745028410270193852110555964462294895493 03
81964428810975665933446128475648233786783165271201909145648566923 46
03486104543266482133936072602491412737245870066063 0
>>> print('precision = '+str(len(piDecimals[1:]))+'decimals')
precision = 314 decimals
>>> print('%.4f' % (t1-t0)+' sec.')
0.0338 sec.
```

arithmetics.**divisors**(*n*, *Sorted=True*)
Renders the list of divisors of integer n.

arithmetics.**divisorsFunction**(*k*, *n*)
generic divisor function:

- the number of divisors of *n* is divisorsFunction(0,n)
- the sum of the divisors of *n* is divisorsFunction(1,n)

arithmetics.**factorization**(*n*)

arithmetics.**gcd**(*a*, *b*)
Renders the greatest common divisor of a and b.

arithmetics.**invSternBrocot**(*sb=['L', 'R', 'R', 'L'], Debug=False*)
Computing the rational which corresponds to the Stern-Brocot string sb.

Source: Graham, Knuth, Patashnik, Sec. 4.5 in Concrete Mathematics 2nd Ed., Addison-Wesley 1994, pp 115-123.

arithmetics.**isprime**(*n*, *precision=7*)
http://en.wikipedia.org/wiki/Miller-Rabin_primality_test#Algorithm_and_running_time

arithmetics.**lcm**(*a*, *b*)
Renders the least common multiple of a and b.

arithmetics.**moebius_mu**(*n*)
Implements the Moebius mu function on N based on n's prime factorization: n = p1^e1 * ... * pk^ek with each ei >= 1 for i = 1, ..., k.

mu = 0 if ei > 1 for some i = 1, ... k else mu = (-1)^k.

arithmetics.**pollard_brent**(*n*)
https://comeoncodeon.wordpress.com/2010/09/18/pollard-rho-brent-integer-factorization/

arithmetics.**primeFactors**(*n*, *sort=True*)

arithmetics.**primesBelow**(*N*, *Odd=False*)
http://stackoverflow.com/questions/2068372/fastest-way-to-list-all-primes-below-n-in-python/3035188#3035188 Input N>=6, Returns a list of primes, 2 <= p < N

arithmetics.**solPartEqnDioph**(*a*, *b*, *c*)
renders a particular integer solution of the Diophantian equation ax + by = c.

arithmetics.**sternBrocot**(*m=5*, *n=7*, *Debug=False*)
Renders the Stern-Brocot representation of the rational m/n (m and n are positive integers). For instance, stern-Brocot(5,7) = ['L','R','R','L'].

Source: Graham, Knuth, Patashnik, Sec. 4.5 in Concrete Mathematics 2nd Ed., Addison-Wesley 1994, pp 115-123.

arithmetics.**totient**(*n*)
Implements the totient function rendering Euler's number of coprime elements a in Zn.

arithmetics.**zn_squareroots**(*n*, *Comments=False*)
Renders the quadratic residues of Zn as a dictionary.

arithmetics.**zn_units**(*n*, *Comments=False*)
Renders the set of units of Zn.

Back to the *Installation*

2.2.19 Cythonized modules for big digraphs

The following modules are compiled C-extensions using the Cython pre-compiler. No Python code source is provided for inspection. To distinguish them from the corresponding pure Python modules, a c- prefix is used.

cRandPerfTabs module

c-Extension for the Digraph3 collection. Module cRandPerfTabs.py is a c-compiled version of the *randomPerfTabs* module for generating random performance tableaux of Big Data type, ie with integer action keys and float performance evaluations.

Conversions methods are provided to switch from the standard to the BigData format and back.

class cRandPerfTabs.**NormalizedPerformanceTableau**
Bases: *cRandPerfTabs.cPerformanceTableau*

specialsation of the cPerformanceTableau class for constructing normalized, 0 - 100, valued PerformanceTableau instances from a given argPerfTab instance.

Parameters:

- argPerfTab=None,
- lowValue=0.0,
- highValue=100.0,
- coalition=None,
- Debug=False

normalizeEvaluations
recode the evaluations between lowValue and highValue on all criteria.

Parameters:

- lowValue=0.0,
- highValue=100.0,
- Debug=False

class cRandPerfTabs.**Random3ObjectivesPerformanceTableau**
Bases: *cRandPerfTabs.cPerformanceTableau*

Specialization of the cPerformanceTableau for 3 objectives: *Eco*, *Soc* and *Env*.

Each decision action is qualified at random as weak (-), fair (~) or good (+) on each of the three objectives.

Parameters:

- numberOf Actions := 20 (default),
- number of Criteria := 13 (default),
- weightDistribution := 'equiobjectives' (default)

  'equisignificant' (weights set all to 1)
  'random' (in the range 1 to numberOfCriteria),
- weightScale := [1,numerOfCriteria] (random default),
- commonScale := (0.0, 100.0) (default)

  (0.0,10.0) if OrdinalScales == True,
- commonThresholds := ((Ind,Ind_slope),(Pref,Pref_slope),(Veto,Veto_slope)) with

  Ind < Pref < Veto in [0.0,100.0] such that
  (Ind/100.0*span + Ind_slope*x) < (Pref/100.0*span + Pref_slope*x) < (Pref/100.0*span + Pref_slope*x)
  By default [(0.05*span,0.0),(0.10*span,0.0),(0.60*span,0.0)] if OrdinalScales=False
  By default [(0.1*span,0.0),(0.2*span,0.0),(0.8*span,0.0)] otherwise
  with span = commonScale[1] - commonScale[0].
- commonMode := ['triangular','variable',0.50] (default), A constant mode may be provided.

  ['uniform','variable',None], a constant range may be provided.
  ['beta','variable',None] (three alpha, beta combinations:
  (5.8661,2.62203),(5.05556,5.05556) and (2.62203, 5.8661)
  chosen by default for 'good', 'fair' and 'weak' evaluations.
  Constant parameters may be provided.
- valueDigits := 2 (default, for cardinal scales only),
- vetoProbability := x in]0.0-1.0[(0.5 default), probability that a cardinal criterion shows a veto preference discrimination threshold.
- missingDataProbability := x in]0.0-1.0[(0.05 default), probability that an action x criterion evaluation is missing.
- Debug := True / False (default).

showActions

Parameter:

- Alphabetic=False

showObjectives

class cRandPerfTabs.**RandomCBPerformanceTableau**
Bases: *cRandPerfTabs.cPerformanceTableau*

Full automatic generation of random Cost versus Benefit oriented performance tableaux.

Parameters:

- If numberOfActions == None, a uniform random number between 10 and 31 of cheap, neutral or advantageous actions (equal 1/3 probability each type) actions is instantiated
- If numberOfCriteria == None, a uniform random number between 5 and 21 of cost or benefit criteria (1/3 respectively 2/3 probability) is instantiated
- weightDistribution := {'equiobjectives'|'fixed'|'random'|'equisignificant' (default = 'equisignificant')}
- default weightScale for 'random' weightDistribution is 1 - numberOfCriteria
- commonScale parameter is obsolete. The scale of cost criteria is cardinal or ordinal (0-10) with proabailities 1/4 respectively 3/4, whereas the scale of benefit criteria is ordinal or cardinal with probabilities 2/3, respectively 1/3.
- All cardinal criteria are evaluated with decimals between 0.0 and 100.0 wheras all ordinal criteria are evaluated with integers between 0 and 10.
- commonThresholds is obsolete. Preference discrimination is specified as percentiles of concerned performance differences (see below).
- CommonPercentiles = {'ind':0.05, 'pref':0.10, ['weakveto':0.90,] 'veto':'95} are expressed in centiless (reversed for vetoes) and only concern cardinal criteria.
- missingDataProbability := x in]0.0-1.0[(0.05 default), probability that an action x criterion evaluation is missing.

Warning: Minimal number of decision actions required is 3 !

class cRandPerfTabs.**RandomCoalitionsPerformanceTableau**

Bases: *cRandPerfTabs.cPerformanceTableau*

Full automatic generation of performance tableaux with random coalitions of criteria

Parameters:

- numberOf Actions := 20 (default)
- number of Criteria := 13 (default)
- weightDistribution := 'equisignificant' (default with all weights = 1.0), 'random', 'fixed' (default w_1 = numberOfCriteria-1, w_{i!=1} = 1
- weightScale := [1,numerOfCriteria] (random default), [w_1, w_{i!=1] (fixed)
- commonScale := (0.0, 100.0) (default)
- commonThresholds := [(1.0,0.0),(2.001,0.0),(8.001,0.0)] if OrdinalSacles, [(0.10001*span,0),(0.20001*span,0.0),(0.80001*span,0.0)] with span = commonScale[1] - commonScale[0].
- commonMode := ['triangular',50.0,0.50] (default), ['uniform',None,None], ['beta', None,None] (three alpha, beta combinations (5.8661,2.62203) chosen by default for high('+'), medium ('~') and low ('-') evaluations.
- valueDigits := 2 (default, for cardinal scales only)
- Coalitions := True (default)/False, three coalitions if True
- VariableGenerators := True (default) / False, variable high('+'), medium ('~') or low ('-') law generated evaluations.
- OrdinalScales := True / False (default)
- Debug := True / False (default)

- RandomCoalitions = True / False (default) zero or more than three coalitions if Coalitions == False.
- vetoProbability := x in]0.0-1.0[/ None (default), probability that a cardinal criterion shows a veto preference discrimination threshold.

class cRandPerfTabs.**RandomPerformanceTableau**

Bases: *cRandPerfTabs.cPerformanceTableau*

Specialization of the cPerformanceTableau class for generating a temporary random performance tableau.

Parameters:

- numberOfActions := nbr of decision actions.
- numberOfCriteria := number performance criteria.
- weightDistribution := 'random' (default) | 'fixed' | 'equisignificant'.

  If 'random', weights are uniformly selected randomly
  form the given weight scale;
  If 'fixed', the weightScale must provided a corresponding weights
  distribution;
  If 'equisignificant', all criterion weights are put to unity.
- weightScale := [Min,Max] (default =[1,numberOfCriteria].
- IntegerWeights := True (in the BigData format)
- commonScale := [Min;Max]; common performance measuring scales (default = [0;100])
- commonThresholds := [(q0,q1),(p0,p1),(v0,v1)]; common indifference(q), preference (p) and considerable performance difference discrimination thresholds.
- commonMode := common random distribution of random performance measurements:

  ('uniform',Min,Max), uniformly distributed between min and max values.
  ('normal',mu,sigma), truncated Gaussion distribution.
  ('triangular',mode,repartition), generalized triangular distribution
  ('beta',mode,(alpha,beta)), by default Mode=None, alpha=beta=2.
- valueDigits := <integer>, precision of performance measurements (2 decimal digits by default).

Code example::

```
>>> from cRandPerfTabs import RandomPerformanceTableau
>>> t = RandomPerformanceTableau(numberOfActions=3,numberOfCriteria=1,
↪seed=100)
>>> t.actions
OrderedDict([
(0: {'name': 'a1'}),
(1: {'name': 'a2'}),
(3: {'name': 'a3'})
])
>>> t.criteria
OrderedDict([
('g1': {'thresholds': {'ind' : (10.0, 0.0) ),
                       'veto': (80.0, 0.0) ),
                       'pref': (20.0, 0.0) },
          'scale': (0.0, 100.0),
          'weight': 1,
          'name': 'cRandPerfTabs.RandomPerformanceTableau() instance',
          'comment': 'Arguments: weightDistribution=random;
```

```
                weightScale=(1, 1); commonMode=None'}).
])
>>> t.evaluation
{'g01': {0: 45.95, 1: 95.17, 2: 17.47.}}
```

class cRandPerfTabs.**RandomRankPerformanceTableau**
Bases: *cRandPerfTabs.cPerformanceTableau*

Specialization of the cPerformanceTableau class for generating a temporary random performance tableau.

Random generator for multiple criteria ranked (without ties) performances of a given number of decision actions. On each criterion, all decision actions are hence lineraly ordered. The RandomRankPerformanceTableau class is matching the RandomLinearVotingProfiles class (see the votingDigraphs module)

Parameters:

- number of actions,
- number of performance criteria,
- weightDistribution := equisignificant | random (default, see RandomPerformanceTableau)
- weightScale := (1, 1 | numberOfCriteria (default when random))
- integerWeights := Boolean (True = default)
- commonThresholds (default) := {

  'ind':(0,0),
  'pref':(1,0),
  'veto':(numberOfActions,0)
  } (default)

class cRandPerfTabs.**cPerformanceTableau**(*filePerfTab=None*, *isEmpty=False*)
Bases: *perfTabs.PerformanceTableau*

Abstract root class for cythenized performace tableau methods.

convert2Standard
Renders a standard *perfTabs.PerformanceTableau* class instance from a deepcopy of a BigData instance.

convertDiscriminationThresholds2Decimal
Converts perrformance discrimination thresholds from float to Decimal format.

convertDiscriminationThresholds2Float
Converts perrformance discrimination thresholds from Decimal to float format.

convertEvaluation2Decimal
Converts evaluations from float to Decimal format.

convertEvaluation2Float
Converts evaluations from Decimal to float format.

convertInsite2BigData
Converts in site weights, evaluations and discrimination thresholds to bigData float format.

convertInsite2Standard
Converts in site weights, evaluations and discrimination thresholds to standard Decimal format.

convertWeight2Decimal
Converts significance weights from int to Decimal format.

convertWeight2Integer
Converts significance weights from Decimal to int format.

normalizeEvaluations
Recodes the evaluations between lowValue and highValue on all criteria.

Parameters:

- lowValue=0.0,
- highValue=100.0,
- Debug=False

showCriteria
Prints self.criteria with thresholds and weights.

Parameters:

- IntegerWeights=True,
- Alphabetic=False,
- ByObjectives=True,
- Debug=False

showHTMLPerformanceHeatmap
shows the html heatmap version of the performance tableau in a browser window (see perfTabs.htmlPerformanceHeatMap() method).

Parameters:

- *actionsList* and *criteriaList*, if provided, give the possibility to show the decision alternatives, resp. criteria, in a given ordering.
- *ndigits* = 0 may be used to show integer evaluation values.
- If no *actionsList* is provided, the decision actions are ordered from the best to the worst. This ranking is obtained by default with the Copeland rule applied on a standard *BipolarOutrankingDigraph*. When the *SparseModel* flag is put to *True*, a sparse *PreRankedOutrankingDigraph* construction is used instead.
- The *minimalComponentSize* allows to control the fill rate of the pre-ranked model. If *minimalComponentSize* = *n* (the number of decision actions) both the pre-ranked model will be in fact equivalent to the standard model.
- It may interesting in some cases to use *rankingRule* = 'NetFlows'.
- Quantiles used for the pre-ranked decomposition are put by default to *n* (the number of decision alternatives) for $n < 50$. For larger cardinalities up to 1000, quantiles = *n* /10. For bigger performance tableaux the *quantiles* parameter may be set to a much lower value not exceeding usually 1000.
- The pre-ranking may be obtained with three ordering strategies for the quantiles equivalence classes: 'average' (default), 'optimistic' or 'pessimistic'.
- With *Correlations* = *True* and *criteriaList* = *None*, the criteria will be presented from left to right in decreasing order of the correlations between the marginal criterion based ranking and the global ranking used for presenting the decision alternatives.
- For large performance Tableaux, *multiprocessing* techniques may be used by setting.
- *Threading* = *True* in order to speed up the computations; especially when *Correlations* = *True*.
- By default, the number of cores available, will be detected. It may be necessary in a HPC context to indicate the exact number of singled threaded cores that are actually allocated to the running job.

```
>>> from cRandomPerfTabs import RandomPerformanceTableau
>>> rt = RandomPerformanceTableau(seed=100)
>>> rt.showHTMLPerformanceHeatmap(colorLevels=5,Correlations=True)
```

Heatmap of Performance Tableau 'randomperftab'

criteria	g6	g3	g4	g2	g1	g5	g7
weights	1	1	1	1	1	1	1
tau(*)	0.42	0.25	0.17	0.03	0.03	-0.06	-0.15
a07	75.39	77.35	71.73	34.70	80.00	65.15	11.40
a01	49.35	81.80	63.78	33.54	14.57	85.42	62.12
a11	59.54	91.38	59.04	95.61	4.79	71.47	20.91
a05	86.13	0.56	96.85	17.85	73.20	81.38	33.37
a03	14.72	64.85	12.66	76.80	77.08	48.36	74.87
a08	70.62	56.62	77.50	62.63	53.29	25.25	19.28
a04	67.60	12.41	55.40	20.39	70.55	76.15	56.82
a12	21.91	23.72	52.82	55.54	93.30	76.70	38.98
a13	30.99	44.82	34.33	90.12	94.71	51.36	93.64
a02	58.27	16.04	90.23	30.94	45.49	36.30	65.08
a09	12.10	19.26	50.71	96.33	8.02	84.74	52.53
a10	5.07	84.12	28.99	21.08	45.59	90.93	72.01
a06	16.47	39.55	60.91	18.86	43.35	89.05	1.25

Color legend:

quantile	0.20%	0.40%	0.60%	0.80%	1.00%

(*) tau: Ordinal (Kendall) correlation between marginal criterion and global ranking relation.

Back to the *Installation*

cIntegerOutrankingDigraphs module

c-Extension for the Digraph3 collection. Module cIntegerOutrankingDigraphs.py is a c-compiled part of the *outrankingDigraphs* module for handling random performance tableaux of Big Data type, ie with integer action keys and float performance evaluations.

class cIntegerOutrankingDigraphs.**IntegerBipolarOutrankingDigraph**

Bases: *outrankingDigraphs.BipolarOutrankingDigraph*, *perfTabs.PerformanceTableau*

Specialization of the abstract OutrankingDigraph root class for generating integer-valued bipolar outranking digraphs.

Parameters:

- argPerfTab: instance of PerformanceTableau class. If a file name string is given, the performance tableau will directly be loaded first.
- coalition: subset of criteria to be used for contruction the outranking digraph.
- hasNoVeto: veto desactivation flag (False by default).

- hasBipolarVeto: bipolar versus electre veto activation (true by default).
- Threading: False by default. Allows to profit from SMP machines via the Python multiprocessing module.
- nbrCores: controls the maximal number of cores that will be used in the multiprocessing phases. If None is given, the os.cpu_count method is used in order to determine the number of availble cores on the SMP machine.

Example Python session:

```
>>> from cRandPerfTabs import *
>>> t = RandomPerformanceTableau(numberOfActions=9,seed=100)
>>> t
*------- PerformanceTableau instance description ------*
Instance class   : RandomPerformanceTableau
Instance name    : cRandomperftab
# Actions        : 9
# Criteria       : 7
Attributes       : ['name', 'actions', 'criteria', 'evaluation',
↪'weightPreorder']
>>> from cIntegerOutrankingDigraphs import *
>>> ig = IntegerBipolarOutrankingDigraph(t)
>>> ig
*------- Object instance description ------*
Instance class   : IntegerBipolarOutrankingDigraph
Instance name    : rel_cRandomperftab
# Actions        : 9
# Criteria       : 7
Size             : 57
Determinateness  : 37.302
Valuation domain : {'min': -7, 'med': 0, 'max': 7,
                    'hasIntegerValuation': True}
----  Constructor run times (in sec.) ----
Total time       : 0.00243
Data input       : 0.00034
Compute relation : 0.00202
Gamma sets       : 0.00006
#Threads         : 1
>>> ig.showRelationTable()
* ---- Relation Table -----
 R  |   '0' '1' '2' '3' '4' '5' '6' '7' '8'
----|------------------------------------
'0' |   +0  +0  -1  -1  +2  +1  -3  +0  +1
'1' |   +3  +0  -7  -7  +1  +2  -1  +1  +1
'2' |   +3  +7  +0  +4  +3  +3  +4  +1  +3
'3' |   +2  +7  +4  +0  +1  +3  +5  +2  +0
'4' |   +5  +2  +2  +1  +0  +3  +1  +1  +3
'5' |   +1  +2  -1  -1  +1  +0  -1  +0  +3
'6' |   +3  +5  +5  +4  +3  +2  +0  +1  +3
'7' |   +5  +5  +3  +4  +3  +7  +1  +0  +5
'8' |   +1  +3  +2  +7  +0  +2  +2  +0  +0
>>> ig.showRubisBestChoiceRecommendation()
**********************
Rubis best choice recommendation(s) (BCR)
 (in decreasing order of determinateness)
Credibility domain: [-7.00,7.00]
 === >> potential best choice(s)
* choice              : [2, 3, 4, 6, 7, 8]
```

```
  +-irredundancy      : 0.00
  independence        : 0.00
  dominance           : 1.00
  absorbency          : -2.00
  covering (%)        : 50.00
  determinateness (%) : 55.56
  - most credible action(s) = { '7': 1.00, '6': 1.00, '4': 1.00, '2': 1.00, }
 === >> potential worst choice(s)
* choice              : [0, 1, 4, 5, 7, 8]
  +-irredundancy      : 0.00
  independence        : 0.00
  dominance           : -1.00
  absorbency          : 3.00
  covering (%)        : 0.00
  determinateness (%) : 54.76
  - most credible action(s) = { '8': 1.00, '5': 1.00, '0': 1.00, }
Execution time: 0.005 seconds
****************************
>>> ig.computeCopelandRanking()
[2, 6, 7, 3, 4, 8, 1, 5, 0]
```

computeCriterionRelation

Parameters:

- c,
- a,
- b,
- hasSymmetricThresholds=True.

Compute the outranking characteristic for actions x and y on criterion c.

computeDeterminateness

Computes the Kendalll distance in % of self with the all median valued (indeterminate) digraph.

computeOrderCorrelation

Parameters:

- order (ordered sequence from worst to best of action keys),
- bint Debug=False.

wrapper for the self.computeRankingCorrelation method The given argOrder is previously reversed.

computeOrdinalCorrelation

Parameters:

- other,
- Debug=False.

Renders the ordinal correlation K of an integer Digraph instance when compared with a given compatible (same actions set) other integer Digraph or Digraph instance.

Formulas:

K = sum_{x != y} [min(max(-self.relation(x,y)),other.relation(x,y), max(self.relation(x,y),-other.relation(x,y))]

K /= sum_{x!=y} [min(abs(self.relation(x,y),abs(other.relation(x,y))]

Note: The global outranking relation of BigDigraph instances is contructed on the fly from the ordered dictionary of the components.

Renders a tuple with at position 0 the actual bipolar correlation index and in position 1 the minimal determination level D of self and the other relation.

D = sum_{x != y} min(abs(self.relation(x,y)),abs(other.relation(x,y)) / n(n-1)

where n is the number of actions considered.

The correlation index with a completely indeterminate relation is by convention 0.0 at determination level 0.0 .

computeOrdinalCorrelationMP

Parameters:

- other (digraph instance),
- Threading=True,
- nbrOfCPUs=True,
- Comments=False,
- Debug=False.

Multi processing version of the digraphs.computeOrdinalCorrelation() method.

Note: The relation filtering and the MedinaCut option are not implemented in the MP version.

computeRankingCorrelation

Parameters:

- ranking (ordered sequence from best to worst of action keys),
- Debug=False.

Renders the ordinal correlation K of an integer digraph instance when compared with a given linear ranking of its actions

K = sum_{x != y} [min(max(-self.relation(x,y)),other.relation(x,y), max(self.relation(x,y),-other.relation(x,y))]

K /= sum_{x!=y} [min(abs(self.relation(x,y),abs(other.relation(x,y))]

Note: The global outranking relation of BigDigraph instances is contructed on the fly from the ordered dictionary of the components.

Renders a tuple with at position 0 the actual bipolar correlation index and in position 1 the minimal determination level D of self and the other relation.

D = sum_{x != y} min(abs(self.relation(x,y)),abs(other.relation(x,y)) / n(n-1)

where n is the number of actions considered.

The correlation index with a completely indeterminate relation is by convention 0.0 at determination level 0.0 .

computeSize
Renders the number of validated non reflexive arcs

convertValuation2Decimal

criterionCharacteristicFunction

Parameters:

- c,
- a,
- b,
- hasSymmetricThresholds=True.

Renders the characteristic value of the comparison of a and b on criterion c.

showActions

Parameter:

- Alphabetic=False.

Presentation methods for decision actions or alternatives.

Back to the *Installation*

cIntegerSortingDigraphs module

c-Extension for the Digraph3 collection. Module cIntegerOutrankingDigraphs.py is a c-compiled part of the *outrankingDigraphs* module for handling random performance tableaux of Big Data type, ie with integer action keys and float performance evaluations.

class cIntegerSortingDigraphs.**IntegerQuantilesSortingDigraph**
Bases: *cIntegerOutrankingDigraphs.IntegerBipolarOutrankingDigraph*

Parameters:

- argPerfTab=None,
- limitingQuantiles=4,
- LowerClosed=False,
- PrefThresholds=True,
- hasNoVeto=False,
- outrankingType = “bipolar”,
- CompleteOutranking = False,
- StoreSorting=False,
- CopyPerfTab=False,
- Threading=False,
- tempDir=None,
- nbrCores=None,
- nbrOfProcesses=None,

- Comments=False,
- Debug=False.

Cythonized c-Extension of the *sortingDigraphs.QuantilesSortingDigraph* class for the sorting of very large sets of alternatives into quantiles delimited ordered classes.

A *cIntegerOutrankingDigraphs.IntegerBipolarOutrankingDigraph* class specialisation.

Note: We generally require an PerformanceTableau instance or a valid filename. If no limitingQuantiles parameter quantity is given, a default profile with the limiting quartiles Q0,Q1,Q2, Q3 and Q4 is used on each criteria.

By default, upper closed limits of categories are used in the sorting algorithm.

Example Python session:

```
>>> from cRandPerfTabs import *
>>> t = RandomPerformanceTableau(numberOfActions=25)
>>> from cIntegerSortingDigraphs import *
>>> so = IntegerQuantilesSortingDigraph(t,limitingQuantiles='quintiles')
>>> so.showSorting()
*--- Sorting results in descending order ---*
]0.80 - 1.00]:   [10, 17, 23, 24]
]0.60 - 0.80]:   [0, 2, 4, 5, 6, 10, 11, 12, 17, 19, 23]
]0.40 - 0.60]:   [1, 2, 3, 8, 9, 12, 13, 14, 16, 18, 19, 20, 21, 22]
]0.20 - 0.40]:   [7, 13, 15, 20]
]< - 0.20]:      []
>>> so.showSorting(Reverse=False)
*--- Sorting results in ascending order ---*
]< - 0.20]:      []
]0.20 - 0.40]:   [7, 13, 15, 20]
]0.40 - 0.60]:   [1, 2, 3, 8, 9, 12, 13, 14, 16, 18, 19, 20, 21, 22]
]0.60 - 0.80]:   [0, 2, 4, 5, 6, 10, 11, 12, 17, 19, 23]
]0.80 - 1.00]:   [10, 17, 23, 24]
>>> so.showQuantileOrdering(strategy='average')
]0.80-1.00] : [24]
]0.60-1.00] : [10, 17, 23]
]0.60-0.80] : [0, 4, 5, 6, 11]
]0.40-0.80] : [2, 12, 19]
]0.40-0.60] : [1, 3, 8, 9, 14, 16, 18, 21, 22]
]0.20-0.60] : [13, 20]
]0.20-0.40] : [7, 15]
```

computeCategoryContents

Parameters:

- Reverse=False,
- Comments=False,
- StoreSorting=True,
- Threading=False,
- nbrOfCPUs=None.

Computes the sorting results per category.

computeQuantileOrdering

Parameters:

- Descending: listing in *decreasing* (default) or *increasing* quantile order.
- strategy: ordering in an { 'optimistic' | 'pessimistic' | 'average'(default) } in the uppest, the lowest or the average potential quantile.
- HTML=False (for generating a HTML version of the result)
- Comments=False,
- Debug=False

computeSortingCharacteristics

Parameters:

- action=None,
- Comments=False,
- StoreSorting=False,
- Debug=False,
- Threading=False,
- nbrOfCPUs=None.

Renders a bipolar-valued bi-dictionary relation representing the degree of credibility of the assertion that "action x in A belongs to category c in C", ie x outranks low category limit and does not outrank the high category limit.

computeSortingRelation

Parameters:

- categoryContents=None,
- Debug=False,
- StoreSorting=True,
- Threading=False,
- nbrOfCPUs=None,
- Comments=False.

constructs a bipolar sorting relation using the category contents.

computeWeakOrder

Parameters:

- Descending=True,
- Debug=False.

Specialisation for QuantilesSortingDigraphs.

getActionsKeys

Parameters:

- action=None,
- withoutProfiles=True.

Extract normal actions keys()

orderedCategoryKeys

Parameter:

- Reverse=False.

Renders the ordered list of category keys based on self.categories['order'] numeric values.

showActionCategories

Parameters:

- action,
- Debug=False,
- Comments=True,
- Threading=False,
- nbrOfCPUs=None.

Renders the union of categories in which the given action is sorted positively or null into. Returns a tuple : action, lowest category key, highest category key, membership credibility !

showActionsSortingResult

Parameters:

- actionSubset=None,
- Debug=False.

shows the quantiles sorting result all (default) of a subset of the decision actions.

showHTMLQuantileOrdering

Parameters:

- Descending=True,
- strategy='average'.

Shows the html version of the quantile preordering in a browser window.

The ordring strategy is either:

- **optimistic**, following the upper quantile limits (default),
- **pessimistic**, following the lower quantile limits,
- **average**, following the average of the upper and lower quantile limits.

showHTMLSorting

"Parameter*:

- Reverse=True.

Shows the html version of the sorting result in a browser window.

showOrderedRelationTable

Parameter:

- direction="decreasing".

Showing the relation table in decreasing (default) or increasing order.

showQuantileOrdering

Parameter:

- strategy=None ('average' by default').

Dummy show method for the commenting computeQuantileOrdering() method.

showSorting

Parameters:

- Reverse=True,
- isReturningHTML=False,
- Debug=False.

Shows sorting results in decreasing or increasing (Reverse=False) order of the categories. If isReturningHTML is True (default = False) the method returns a htlm table with the sorting result.

showSortingCharacteristics

Parameter:

- action=None.

Renders a bipolar-valued bi-dictionary relation representing the degree of credibility of the assertion that "action x in A belongs to category c in C", ie x outranks low category limit and does not outrank the high category limit.

showWeakOrder

Parameter:

- Descending=True.

Specialisation for QuantilesSortingDigraphs.

Back to the *Installation*

cSparseIntegerOutrankingDigraphs module

c-Extension for the Digraph3 collection. Module cBigIntegerOutrankingDigraphs.py is a c-compiled partial version of the *sparseOutrankingDigraphs* module for handling outranking digraphs of very large order.

class cSparseIntegerOutrankingDigraphs.**SparseIntegerDigraph**
Bases: object

Abstract root class for linearly decomposed big digraphs (order > 1000) using multiprocessing ressources.

computeDecompositionSummaryStatistics
Returns the summary of the distribution of the length of the components as follows:

```
summary = {'max': maxLength,
           'median':medianLength,
           'mean':meanLength,
           'stdev': stdLength,
           'fillrate': fillrate,
                       (see computeFillRate()}
```

computeFillRate

Parameters:

- Debug=False.

Renders the sum of the squares (without diagonal) of the orders of the component's subgraphs over the square (without diagonal) of the big digraph order.

computeOrdinalCorrelation

Parameters:

- other (digraph instance),
- Debug=False.

Renders the ordinal correlation K of a SparseDigraph instance when compared with a given compatible (same actions set) other Digraph or SparseDigraph instance.

K = sum_{x != y} [min(max(-self.relation(x,y)),other.relation(x,y), max(self.relation(x,y),-other.relation(x,y))]

K /= sum_{x!=y} [min(abs(self.relation(x,y),abs(other.relation(x,y))]

Note: The global outranking relation of SparseDigraph instances is contructed on the fly from the ordered dictionary of the components.

Renders a tuple with at position 0 the actual bipolar correlation index and in position 1 the minimal determination level D of self and the other relation.

D = sum_{x != y} min(abs(self.relation(x,y)),abs(other.relation(x,y)) / n(n-1)

where n is the number of actions considered.

The correlation index with a completely indeterminate relation is by convention 0.0 at determination level 0.0 .

---

computeRankingCorrelation

Parameters:

- ranking (ordered list from best to worst),
- Debug=False.

Renders the ordinal correlation K of a SparseDigraph instance when compared with a given linear ranking of its actions

K = sum_{x != y} [min(max(-self.relation(x,y)),other.relation(x,y), max(self.relation(x,y),-other.relation(x,y))]

K /= sum_{x!=y} [min(abs(self.relation(x,y),abs(other.relation(x,y))]

Note: The global outranking relation of SparseDigraph instances is contructed on the fly from the ordered dictionary of the components.

Renders a tuple with at position 0 the actual bipolar correlation index and in position 1 the minimal determination level D of self and the other relation.

D = sum_{x != y} min(abs(self.relation(x,y)),abs(other.relation(x,y)) / n(n-1)

where n is the number of actions considered.

The correlation index with a completely indeterminate relation is by convention 0.0 at determination level 0.0 .

---

ordering2Preorder

Parameter:

- ordering (list from worst to best).

Renders a preordering (a list of list) of a linar order (worst to best) of decision actions in increasing preference direction.

ranking2Preorder

Parameter:

- ranking (list from best to worst).

Renders a preordering (a list of list) of a ranking (best to worst) of decision actions in increasing preference direction.

relation

Parameters:

- x (int action key),
- y (int action key).

Dynamic construction of the global outranking characteristic function *r(x S y)*.

showBestChoiceRecommendation

Parameters:

- Comments=False,
- ChoiceVector=False,
- Debug=False.

Update of rubisBestChoice Recommendation for big digraphs. To do: limit to best choice; worst choice should be a separate method()

showDecomposition

Parameter:

- direction='decreasing'.

Prints on the console the decomposition structure of the sparse outranking digraph instance in *decreasing* (default) or *increasing* preference direction.

showHTMLRelationMap

Parameters:

- fromIndex=0,
- toIndex=0,
- Colored=True,
- tableTitle='Sparse Relation Map',
- relationName='r(x S y)',
- symbols=['+','·',' ','–','&#151'].

Launches a browser window with the colored relation map of self. See corresponding *digraphs.Digraph.showRelationMap()* method.

showRelationMap

Parameters:

- fromIndex=0,
- toIndex=0,
- symbols=None.

Prints on the console, in text map format, the location of the diagonal outranking components of the big outranking digraph.

By default, symbols := {'max':'T','positive': '+', 'median': ' ', 'negative': '-', 'min': '_'}

The default ordering of the output is following the quantiles sorted boosted net flows ranking rule from best to worst actions. Further available ranking rules are Kohler's (rankingRule="kohler") and Tideman's ranked pairs rule (rankingRule="rankedPairs").

showRubisBestChoiceRecommendation

Parameters:

- g0=None (first component of self by default),
- Comments=False,
- ChoiceVector=True,
- Debug=False,
- _OldCoca=False,
- Cpp=False.

Renders the Rubis Best choice recommendation of the first component.

showRubisWorstChoiceRecommendation

Parameters:

- g0=None (last component of self by default),
- Comments=False,
- ChoiceVector=True,
- Debug=False,
- _OldCoca=False,
- Cpp=False.

Renders the Rubis Worst choice recommendation of the first component.

class cSparseIntegerOutrankingDigraphs.**SparseIntegerOutrankingDigraph**

Bases: *cSparseIntegerOutrankingDigraphs.SparseIntegerDigraph*, *perfTabs.PerformanceTableau*

Parameters:

- argPerfTab,
- quantiles=4,
- quantilesOrderingStrategy="average",

- LowerClosed=False,
- componentRankingRule="Copeland",
- minimalComponentSize=1,
- Threading=False,
- tempDir=None,
- componentThreadingThreshold=1000, * nbrOfCPUs=1,
- save2File=None,
- CopyPerfTab=False,
- Comments=False,
- Debug=False.

Main class for the multiprocessing implementation of big outranking digraphs.

The big outranking digraph instance is decomposed with a q-tiling sort into a partition of quantile equivalence classes which are linearly ordered by average quantile limits (default).

With each quantile equivalence class is associated a BipolarOutrankingDigraph object which is restricted to the decision actions gathered in this quantile equivalence class.

By default, the number of quantiles q is set to quartiles. However, the ranking quality and the best choice results get better with a finer grained quantiles decomposition.

For other parameters settings, see the corresponding *sortingDigraphs.QuantilesSortingDigraph* class.

Example python3.6 session:

```
>>> from cRandPerfTabs import *
>>> tp = RandomCBPerformanceTableau(numberOfActions=1000,
                                    Threading=True,seed=100)
>>> tp
*------- PerformanceTableau instance description ------*
Instance class   : RandomCBPerformanceTableau
Instance name    : randomCBperftab
# Actions        : 1000
# Objectives     : 2
# Criteria       : 7
Attributes       : ['name', 'actions', 'objectives',
                    'criteriaWeightMode', 'criteria',
                    'evaluation', 'weightPreorder']
>>> from cSparseIntegerOutrankingDigraphs import *
>>> bg = SparseIntegerOutrankingDigraph(tp,quantiles=35,
...                         quantilesOrderingStrategy='average',
...                         LowerClosed=False,
...                         minimalComponentSize=10,
...                         Threading=True,Debug=False)
>>> bg
  *----- Object instance description --------------*
  Instance class    : SparseIntegerOutrankingDigraph
  Instance name     : randomCBperftab_mp
  # Actions         : 1000
  # Criteria        : 7
  Sorting by        : 35-Tiling
  Ordering strategy : average
  Ranking rule      : Copeland
```

```
  # Components      : 75
  Minimal order     : 10
  Maximal order     : 36
  Average order     : 13.3
  fill rate         : 1.489%
  ----  Constructor run times (in sec.) ----
  Nbr of threads    : 1
  Nbr of threads    : 8
  Total time        : 0.54866
  QuantilesSorting  : 0.39175
  Preordering       : 0.00509
  Decomposing       : 0.15179
  Ordering          : 0.00000
>>> bg.showBestChoiceRecommendation()
  ***********************
  * --- Best choice recommendation(s) ---*
  (in decreasing order of determinateness)
  Credibility domain:  {'min': -24, 'med': 0, 'max': 24,
                        'hasIntegerValuation': True}
  * choice              : [131, 151, 388]
    +-irredundancy      : 0.00
    independence        : 0.00
    dominance           : 2.00
    absorbency          : -10.00
    covering (%)        : 61.90
    determinateness (%) : 50.00
    - most credible action(s) = { }
  ***********************
  * --- Worst choice recommendation(s) ---*
  (in decreasing order of determinateness)
  Credibility domain:  {'min': -24, 'med': 0, 'max': 24,
                        'hasIntegerValuation': True}
  * choice              : [312]
  +-irredundancy      : 24.00
  independence        : 24.00
  dominance           : -10.00
  absorbency          : 4.00
  covering (%)        : 0.00
  determinateness (%) : 58.33
  - most credible action(s) = { '312': 4.00, }
>>> print(bg.boostedRanking[:10],' ... ',bg.boostedRanking[-10:] )
  [388, 131, 151, 275, 679, 406, 741, 623, 579, 894]  ...
  [278, 886, 202, 473, 841, 878, 713, 62, 17, 312]
>>>
```

computeActionCategories

Parameters:

- action (int key),
- Show=False,
- Debug=False,
- Comments=False,
- Threading=False,
- nbrOfCPUs=1.

Renders the union of categories in which the given action is sorted positively or null into. Returns a tuple : action, lowest category key, highest category key, membership credibility !

computeBoostedOrdering

Parameter:

- orderingRule='Copeland'.

Renders an ordred list of decision actions ranked in increasing preference direction following by default the Copeland rule on each component.

computeBoostedRanking

Parameter:

- rankingRule='Copeland'.

Renders an ordred list of decision actions ranked in decreasing preference direction following the net flows rule on each component.

computeCriterion2RankingCorrelation

Parameters:

- criterion,
- Threading=False,
- nbrOfCPUs=1,
- Debug=False,
- Comments=False.

Renders the ordinal correlation coefficient between the global outranking and the marginal criterion relation.

computeDeterminateness

Parameter:

- InPercent=True.

Computes the Kendalll distance in % of self with the all median valued (indeterminate) digraph.

deter = (sum_{x,y in X} abs[r(xSy) - Med])/(oder*order-1)

computeMarginalVersusGlobalOutrankingCorrelations

Parameters:

- Sorted=True,
- ValuedCorrelation=False,
- Threading=False,
- nbrCores=None,
- Comments=False.

Method for computing correlations between each individual criterion relation with the corresponding global outranking relation.

Returns a list of tuples (correlation,criterionKey) sorted by default in decreasing order of the correlation.

If Threading is True, a multiprocessing Pool class is used with a parallel equivalent of the built-in map function.

If nbrCores is not set, the os.cpu_count() function is used to determine the number of available cores.

showActions
Prints out the actions disctionary.

showActionsSortingResult

Parameter:

- actionsSubset=None.

Shows the quantiles sorting result all (default) of a subset of the decision actions.

showComponents

Parameter:

- direction='increasing'.

showCriteria

Parameters:

- IntegerWeights=False,
- Debug=False.

print Criteria with thresholds and weights.

showDecomposition

Parameter:

- direction='increasing'.

showMarginalVersusGlobalOutrankingCorrelation

Parameters:

- Sorted=True,
- Threading=False,
- nbrOfCPUs=1,
- Comments=True.

Show method for computeCriterionCorrelation results.

showRelationTable

Parameters:

- IntegerValues=True,
- compKeys=None.

Specialized for showing the quantiles decomposed relation table. Components are stored in an ordered dictionary.

showShort

Parameter:

- WithFileSize=False.

Default (__repr__) presentation method for big outranking digraphs instances:

class cSparseIntegerOutrankingDigraphs.**cQuantilesRankingDigraph**
Bases: *cSparseIntegerOutrankingDigraphs.SparseIntegerOutrankingDigraph*

Parameters:

- argPerfTab,
- quantiles=4,
- quantilesOrderingStrategy="average",
- LowerClosed=False,
- componentRankingRule="Copeland",
- minimalComponentSize=1,
- Threading=False,
- tempDir=None,
- nbrOfCPUs=1,
- save2File=None,
- CopyPerfTab=False,
- Comments=False,
- Debug=False.

Cythonized class for the multiprocessing implementation of multiple criteria quantiles ranking of very big performance tableaux - > 100000.

By default, the number of quantiles q is set to quartiles. However, the ranking quality gets better with a finer grained quantiles decomposition.

For other parameters settings, see the corresponding *sortingDigraphs.QuantilesSortingDigraph* class.

Example python3.6 session:

```
>>> from cRandPerfTabs import *
>>> tp = RandomCBPerformanceTableau(numberOfActions=1000,
                                    Threading=True,seed=100)
>>> tp
*------- PerformanceTableau instance description ------*
Instance class    : RandomCBPerformanceTableau
Instance name     : randomCBperftab
# Actions         : 1000
# Objectives      : 2
# Criteria        : 7
Attributes        : ['name', 'actions', 'objectives',
                     'criteriaWeightMode', 'criteria',
                     'evaluation', 'weightPreorder']
>>> from cSparseIntegerOutrankingDigraphs import *
>>> bg = cQuantilesRankingDigraph(tp,quantiles=35,
...                           quantilesOrderingStrategy='average',
...                           LowerClosed=False,
...                           minimalComponentSize=10,
...                           Threading=True,nbrOfCPUs=8,Debug=False)
>>> bg
  *----- Object instance description --------------*
  Instance class     : cQuantilesRankingDigraph
  Instance name      : randomCBperftab_mp
  # Actions          : 1000
  # Criteria         : 7
  Sorting by         : 35-Tiling
```

```
  Ordering strategy : average
  Ranking rule      : Copeland
  # Components      : 75
  Minimal order     : 10
  Maximal order     : 36
  Average order     : 13.3
  fill rate         : 1.489%
  ----  Constructor run times (in sec.) ----
  Nbr of threads    : 8
  Total time        : 0.54866
  QuantilesSorting  : 0.39175
  Preordering       : 0.00509
  Decomposing       : 0.15179
  Ordering          : 0.00000
>>> bg.showBestChoiceRecommendation()
  ***********************
  * --- Best choice recommendation(s) ---*
  (in decreasing order of determinateness)
  Credibility domain:  {'min': -24, 'med': 0, 'max': 24,
                        'hasIntegerValuation': True}
  * choice              : [131, 151, 388]
    +-irredundancy      : 0.00
    independence        : 0.00
    dominance           : 2.00
    absorbency          : -10.00
    covering (%)        : 61.90
    determinateness (%) : 50.00
    - most credible action(s) = { }
  ***********************
  * --- Worst choice recommendation(s) ---*
  (in decreasing order of determinateness)
  Credibility domain:  {'min': -24, 'med': 0, 'max': 24,
                        'hasIntegerValuation': True}
  * choice              : [312]
  +-irredundancy      : 24.00
  independence        : 24.00
  dominance           : -10.00
  absorbency          : 4.00
  covering (%)        : 0.00
  determinateness (%) : 58.33
  - most credible action(s) = { '312': 4.00, }
>>> print(bg.boostedRanking[:10],' ... ',bg.boostedRanking[-10:] )
  [388, 131, 151, 275, 679, 406, 741, 623, 579, 894]  ...
  [278, 886, 202, 473, 841, 878, 713, 62, 17, 312]
>>>
```

computeActionCategories

Parameters:

- action (int key),
- Show=False,
- Debug=False,
- Comments=False,
- Threading=False,
- nbrOfCPUs=1.

Renders the union of categories in which the given action is sorted positively or null into. Returns a tuple : action, lowest category key, highest category key, membership credibility !

computeCriterion2RankingCorrelation

Parameters:

- criterion,
- Threading=False,
- nbrOfCPUs=1,
- Debug=False,
- Comments=False.

Renders the ordinal correlation coefficient between the global outranking and the marginal criterion relation.

computeDeterminateness

Parameter:

- InPercent=True.

Computes the Kendalll distance in % of self with the all median valued (indeterminate) digraph.

deter = (sum_{x,y in X} abs[r(xSy) - Med])/(oder*order-1)

computeMarginalVersusGlobalOutrankingCorrelations

Parameters:

- Sorted=True,
- ValuedCorrelation=False,
- Threading=False,
- nbrCores=None,
- Comments=False.

Method for computing correlations between each individual criterion relation with the corresponding global outranking relation.

Returns a list of tuples (correlation,criterionKey) sorted by default in decreasing order of the correlation.

If Threading is True, a multiprocessing Pool class is used with a parallel equivalent of the built-in map function.

If nbrCores is not set, the os.cpu_count() function is used to determine the number of available cores.

relation

Parameters:

- x (int action key),
- y (int action key).

Dynamic construction of the global outranking characteristic function *r(x S y)*.

showActions

Prints out the actions disctionary.

showActionsSortingResult

Parameter:

- actionsSubset=None.

Shows the quantiles sorting result all (default) of a subset of the decision actions.

showComponents

Parameter:

- direction='increasing'.

showCriteria

Parameters:

- IntegerWeights=False,
- Debug=False.

print Criteria with thresholds and weights.

showDecomposition

Parameter:

- direction='increasing'.

showMarginalVersusGlobalOutrankingCorrelation

Parameters:

- Sorted=True,
- Threading=False,
- nbrOfCPUs=1,
- Comments=True.

Show method for computeCriterionCorrelation results.

showRelationTable

Parameters:

- IntegerValues=True,
- compKeys=None.

Specialized for showing the quantiles decomposed relation table. Components are stored in an ordered dictionary.

showShort

Parameter:

- WithFileSize=False.

Default (__repr__) presentation method for big outranking digraphs instances:

Back to the *Installation*

2.2.20 Indices and tables

- genindex
- modindex
- search

2.2.21 Tutorials

- Tutorial

2.3 References

For further scientific documentation of the Digraph3 resources, see .

BIBLIOGRAPHY

[CPSTAT-L5] 18. Bisdorff (2017) *Simulating from abitrary empirical random distributions*. MICS Computational Statistics course, Lecture 5. FSTC/ILIAS University of Luxembourg, Winterr Semester 2017 (PDF 2x2 reduced presentation slides 211kB downloadable here)

[BIS-2016] 18. Bisdorff (2016). *On linear ranking from trillions of pairwise outranking situations*. Research Note 16-1, FSTC/ILIAS Decision Systems Group, University of Luxembourg pp. 1-6 (dowloadable PDF file 625.3 kB)

[ADT-L2] 18. Bisdorff (2014) *Who wins the election*? MICS Algorithmic Decision Theory course, Lecture 2. FSTC/ILIAS University of Luxembourg, Summer Semester 2014 (PDF 2x2 reduced presentation slides 195 kB downloadable here)

[ADT-L7] 18. Bisdorff (2014) *Best multiple criteria choice: the Rubis outranking method*. MICS Algorithmic Decision Theory course, Lecture 7. FSTC/ILIAS University of Luxembourg, Summer Semester 2014 (PDF 2x2 reduced slides 310kB downloadable here)

[BIS-2013] 18. Bisdorff (2013) "On Polarizing Outranking Relations with Large Performance Differences" *Journal of Multi-Criteria Decision Analysis* (Wiley) **20**:3-12 (downloadable preprint PDF file 403.5 Kb)

[BIS-2012] 18. Bisdorff (2012). "On measuring and testing the ordinal correlation between bipolar outranking relations". In Proceedings of DA2PL'2012 *From Multiple Criteria Decision Aid to Preference Learning*, University of Mons 91-100. (downloadable preliminary version PDF file 408.5 kB)

[BIS-2008] 18. Bisdorff, P. Meyer and M. Roubens (2008) "RUBIS: a bipolar-valued outranking method for the choice problem". 4OR, *A Quarterly Journal of Operations Research* Springer-Verlag, Volume 6, Number 2 pp. 143-165. (Online) Electronic version: DOI: 10.1007/s10288-007-0045-5 (downloadable preliminary version PDF file 271.5Kb)

[NR3-2007] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery (2007) "Single-Pass Estimation of Arbitrary Quantiles" Section 5.8.2 in *Numerical Recipes: The Art of Scientific Computing 3rd Ed.*, Cambridge University Press, pp 435-438.

[CHAM-2006] J.M. Chambers, D.A. James, D. Lambert and S. Vander Wiel (2006) "Monitoring Networked Applications with Incremental Quantile Estimation". *Statistical Science*, Vol. 21, No.4, pp.463-475. DOI: 10 12140/088342306000000583.

[BIS-2006] 18. Bisdorff, M. Pirlot and M. Roubens (2006). "Choices and kernels from bipolar valued digraphs". *European Journal of Operational Research*, 175 (2006) 155-170. (Online) Electronic version: DOI:10.1016/j.ejor.2005.05.004 (downloadable preliminary version PDF file 257.3Kb)

[FMCAA] 15. Häggström (2002) *Finite Markov Chains and Algorithmic Applications*. Cambridge University Press.

[ISOMIS-08] 18. Bisdorff and J.L. Marichal (2008). Counting non-isomorphic maximal independent sets of the n-cycle graph. *Journal of Integer Sequences*, Vol. 11 Article 08.5.7 (openly accessible here)

PYTHON MODULE INDEX