# Documentation of the DIGRAPH3 software collection
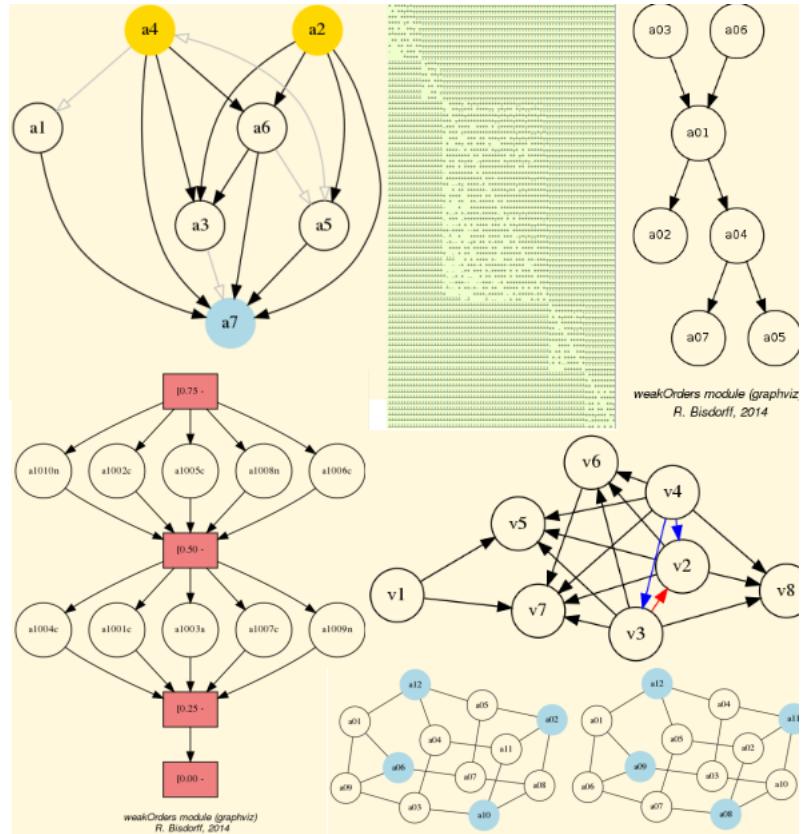
Tutorials and advanced topics

Raymond BISDORFF

Luxembourg, 2020

Last updated : December 24, 2020

This documentation is dedicated to our
late colleague and dear friend
Marc ROUBENS.

More documents are freely available here

Contents

Tutorials of the DIGRAPH3 Resources

HTML Version

The tutorials in this document describe the practical usage of our *Digraph3* Python3 software resources in the field of *Algorithmic Decision Theory* and more specifically in **outranking** based *Multiple Criteria Decision Aid* (MCDA). They mainly illustrate practical tools for a Master Course at the University of Luxembourg.

The document contains first a set of tutorials introducing the main objects available in the Digraph3 collection of Python3 modules, like *digraphs*, *outranking digraphs*, *performance tableaux* and *voting profiles*.

Some of the tutorials are decision problem oriented and show how to compute the potential *winner(s)* of an election, how to build a *best choice recommendation*, or how to *rate*

or *linearly rank* with multiple incommensurable performance criteria.

More graph theoretical tutorials follow. One on working with *undirected graphs*, followed by a tutorial on how to compute *non isomorphic maximal independent sets* (kernels) in the n-cycle graph.

Another tutorial is furthermore devoted on how to generally compute *kernels* in graphs, digraphs and, more specifically, *initial* and *terminal* kernels in outranking digraphs. Special tutorials are devoted to *perfect* graphs, like *split*, *interval* and *permutation* graphs, and to *tree-graphs* and *forests*.

1 Working with the *Digraph3* software resources

1.1 Purpose

The basic idea of the Digraph3 Python resources 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, maximal dominant or absorbent choices, rankings, outrankings, linear ordering, etc. Most of the available computing resources are meant to illustrate a Master Course on *Algorithmic Decision Theory* 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.

1.2 Downloading of the Digraph3 resources

Using the Digraph3 modules is easy. You only need to have installed on your system the Python (https://www.python.org/doc/) programming language of version 3.+ (readily available under Linux and Mac OS). Notice that, from Version 3.3 on, the Python standard decimal module implements very efficiently its decimal.Decimal class in C. Now, Decimal objects are mainly used in the Digraph3 characteristic r-valuation functions, which makes the recent python-3.7+ versions much faster (more than twice as fast) when extensive digraph operations are performed.

Several download options (easiest under Linux or Mac OS-X) are given.

1. Either, by using a git client either, from github

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

2. Or, from sourceforge.net

```
...$ git clone https://git.code.sf.net/p/digraph3/code Digraph3
```

3. Or, with a browser access, download and extract the latest distribution zip archive either, from the github link above (https://github.com/rbisdorff/Digraph3) or, from the sourceforge page (https://sourceforge.net/projects/digraph3/) .

1.3 Starting a Python3 session

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

```
1  $HOME/.../Digraph3$ python3
2  Python 3.9.0 (default, Nov  1 2020, 09:59:50)
3  [GCC 9.3.0] on linux
4  Type "help", "copyright", "credits" or
5        "license" for more information.
6  >>> ...
```

Listing 1.1: Generating a random digraph instance

```
1  >>> from randomDigraphs import RandomDigraph
2  >>> dg = RandomDigraph(order=5,arcProbability=0.5,seed=101)
3  >>> dg
4   *------- Digraph instance description ------*
5   Instance class    : RandomDigraph
6   Instance name     : randomDigraph
7   Digraph Order      : 5
8   Digraph Size       : 12
9   Valuation domain : [-1.00; 1.00]
```

(continues on next page)

5

```
10   Determinateness  : 100.000
11   Attributes       : ['actions', 'valuationdomain', 'relation',
12                      'order', 'name', 'gamma', 'notGamma']
13   >>> dg.save('tutorialDigraph')
14   *--- Saving digraph in file: <tutorialDigraph.py> ---*
```

From the `randomDigraphs` module we import the `randomDigraphs.RandomDigraph` class in order to generate, for instance, a crisp *digraph* object *dg* of order 5 - number of nodes or (decision) *actions* - and size 12 - number of directed *arcs* (see Listing 1.1 Lines 1-2).

We may directly inspect the content of python object *dg* (Line 3)

1.4 Digraph **object structure**

All `digraphs.Digraph` objects contain at least the following attributes (see Listing 1.1 Lines 11-12):

0. A **name** attribute, holding usually the actual name of the stored instance that was used to create the instance;

1. A collection of digraph nodes called **actions** (decision actions): an ordered dictionary of nodes with at least a 'name' attribute;

2. An **order** attribute containing the number of graph nodes (length of the actions dictionary) automatically added by the object constructor;

3. 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);

4. The digraph **relation** : a double dictionary indexed by an oriented pair of actions (nodes) and carrying a decimal characteristic value in the range of the previous valuation domain;

5. Its associated **gamma function** : a dictionary containing the direct successors, respectively predecessors of each action, automatically added by the object constructor;

6. 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.

6

1.5 Permanent storage

The `digraphs.Digraph.save()` method (see Listing 1.1 Line 13 above) stores the digraph object *dg* in a file named `tutorialDigraph.py` with the following content.

```
1  actions = {
2   'a1': {'shortName': 'a1', 'name': 'random decision action'},
3   'a2': {'shortName': 'a2', 'name': 'random decision action'},
4   'a3': {'shortName': 'a3', 'name': 'random decision action'},
5   'a4': {'shortName': 'a4', 'name': 'random decision action'},
6   'a5': {'shortName': 'a5', 'name': 'random decision action'},
7   }
8  valuationdomain = {'hasIntegerValuation': True,
9                     'min': -1.0,'med': 0.0,'max': 1.0}
10 relation = {
11  'a1': {'a1':-1.0, 'a2':-1.0, 'a3':1.0, 'a4':-1.0, 'a5':-1.0,},
12  'a2': {'a1':1.0, 'a2':-1.0, 'a3':-1.0, 'a4':1.0, 'a5':1.0,},
13  'a3': {'a1':1.0, 'a2':-1.0, 'a3':-1.0, 'a4':1.0, 'a5':-1.0,},
14  'a4': {'a1':1.0, 'a2':1.0, 'a3':1.0, 'a4':-1.0, 'a5':-1.0,},
15  'a5': {'a1':1.0, 'a2':1.0, 'a3':1.0, 'a4':-1.0, 'a5':-1.0,},
16  }
```

1.6 Inspecting a `Digraph` object

We may reload the previously saved digraph object from the file named `tutorialDigraph.py` with the `digraphs.Digraph` class constructor and the `digraphs.Digraph.showAll()` method output reveals us that *dg* is a *connected* and *irreflexive* digraph of *order* five, evaluated in an integer *valuation domain* [-1,0,+1} (see Listing 1.2).

Listing 1.2: Random crisp digraph example

```
1  >>> dg = Digraph('tutorialDigraph')
2  >>> dg.showAll()
3   *----- show detail -------------*
4   Digraph           : tutorialDigraph
5   *---- Actions ----*
6   ['a1', 'a2', 'a3', 'a4', 'a5']
7   *---- Characteristic valuation domain ----*
8   {'hasIntegerValuation': True,
9     'min': -1, 'med': 0, 'max': 1'}
10  * ---- Relation Table -----
11     S  |  'a1'  'a2'  'a3'  'a4'  'a5'
12   ------|----------------------------
13    'a1' |   -1    -1    1    -1    -1
14    'a2' |    1    -1    -1    1     1
15    'a3' |    1    -1    -1    1    -1
16    'a4' |    1     1    1    -1    -1
17    'a5' |    1     1    1    -1    -1
```

```
18    Valuation domain: [-1;+1]
19    *--- Connected Components ---*
20    1: ['a1', 'a2', 'a3', 'a4', 'a5']
21    Neighborhoods:
22      Gamma     :
23    'a1': in => {'a2', 'a4', 'a3', 'a5'}, out => {'a3'}
24    'a2': in => {'a5', 'a4'}, out => {'a1', 'a4', 'a5'}
25    'a3': in => {'a1', 'a4', 'a5'}, out => {'a1', 'a4'}
26    'a4': in => {'a2', 'a3'}, out => {'a1', 'a3', 'a2'}
27    'a5': in => {'a2'}, out => {'a1', 'a3', 'a2'}
28      Not Gamma :
29    'a1': in => set(), out => {'a2', 'a4', 'a5'}
30    'a2': in => {'a1', 'a3'}, out => {'a3'}
31    'a3': in => {'a2'}, out => {'a2', 'a5'}
32    'a4': in => {'a1', 'a5'}, out => {'a5'}
33    'a5': in => {'a1', 'a4', 'a3'}, out => {'a4'}
```

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 (see Fig. 1) , if the graphviz (https://graphviz.org/) tools are installed on your system[1].

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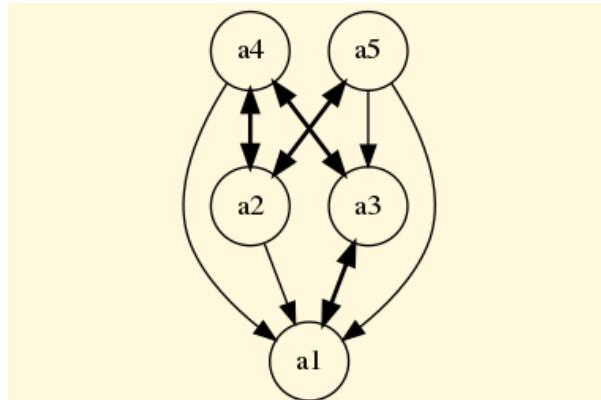

Rubis Python Server (graphviz), R. Bisdorff, 2008

Fig. 1: The tutorial crisp digraph

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

---

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

```
1  >>> dg.showStatistics()
2  *----- general statistics -------------*
3  for digraph             : <tutorialDigraph.py>
4  order                   : 5 nodes
5  size                    : 12 arcs
6  # undetermined          : 0 arcs
7  determinateness (%)     : 100.0
8  arc density             : 0.60
9  double arc density      : 0.40
10 single arc density      : 0.40
11 absence density         : 0.20
12 strict single arc density: 0.40
13 strict absence density  : 0.20
14 # components            : 1
15 # strong components     : 1
16 transitivity degree (%) : 53.0
17                         : [0, 1, 2, 3, 4, 5]
18 outdegrees distribution : [0, 1, 1, 3, 0, 0]
19 indegrees distribution  : [0, 1, 2, 1, 1, 0]
20 mean outdegree          : 2.40
21 mean indegree           : 2.40
22                         : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
23 symmetric degrees dist. : [0, 0, 0, 0, 1, 4, 0, 0, 0, 0, 0]
24 mean symmetric degree   : 4.80
25 outdegrees concentration index   : 0.1667
26 indegrees concentration index    : 0.2333
27 symdegrees concentration index   : 0.0333
28                         : [0, 1, 2, 3, 4, 'inf']
29 neighbourhood depths distribution: [0, 1, 4, 0, 0, 0]
30 mean neighbourhood depth       : 1.80
31 digraph diameter               : 2
32 agglomeration distribution     :
33 a1 : 58.33
34 a2 : 33.33
35 a3 : 33.33
36 a4 : 50.00
37 a5 : 50.00
38 agglomeration coefficient      : 45.00
```

1.7 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 (see Fig. 2).

```
1  >>> from digraphs import GridDigraph
2  >>> grid = GridDigraph(n=5,m=5,hasMedianSplitOrientation=True)
3  >>> grid.exportGraphViz('tutorialGrid')
```

```
4   *---- exporting a dot file for GraphViz tools ---------*
5   Exporting to tutorialGrid.dot
6   dot -Grankdir=BT -Tpng TutorialGrid.dot -o tutorialGrid.png
```


Fig. 2: The 5x5 grid graph median split oriented
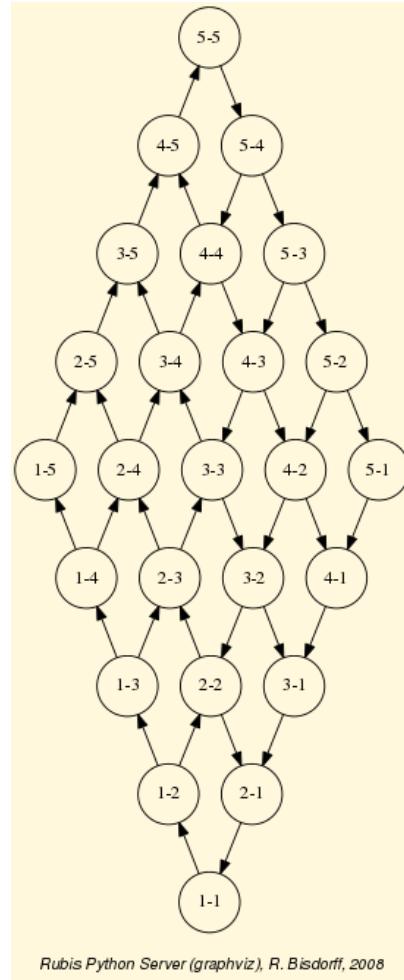
Back to *Content Table* (page 3)

2 Manipulating Digraph objects

2.1 Random digraphs

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 `randomDigraphs.RandomValuationDigraph` constructor.

Listing 2.1: Random bipolar-valued digraph instance

```
1  >>> from randomDigraphs import RandomValuationDigraph
2  >>> dg = RandomValuationDigraph(order=7,Normalized=True)
3  >>> dg.save('tutRandValDigraph')
4  >>> dg = Digraph('tutRandValDigraph')
5  >>> dg
6   *------- Digraph instance description ------*
7   Instance class      : Digraph
8   Instance name       : tutRandValDigraph
9   Digraph Order       : 7
10  Digraph Size        : 22
11  Valuation domain    : [-1.00;1.00]
12  Determinateness (%) : 75.24
13  Attributes          : ['name', 'actions', 'order',
14                         'valuationdomain', 'relation',
15                         'gamma', 'notGamma']
```

With the `save()` method (see Listing 2.1 Line 3) 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 genuine `digraphs.Digraph` class constructor may retore the *dg* object from the stored file (Line 4). We may easily inspect the content of *dg* (Lines 5-). The digraph size 22 indicates the number of positively valued arcs. The valuation domain is normalized in the interval *[-1.0; 1.0]* and the mean absolute arc valuation is 0.7524. All `digraphs.Digraph` objects contain at least the list of attributes shown here: a *name* (string), a dictionary of *actions* (digraph nodes), an *order* (integer) attribute containing the number of actions, a *valuationdomain* dictionary, a double dictionary *relation* representing the adjency table of the digraph relation, a gamma and a notGamma dictionary describing the direct neighbourhoods od each action.

11

The `Digraph` class now provides some generic methods for exploring a given `Digraph` object, like the `showShort()`, `showAll()`, `showRelationTable()` and the `showNeighborhoods()` methods.

Listing 2.2: Example of random valuation digraph

```
1  >>> dg.showAll()
2   *----- show detail -------------*
3    Digraph          : tutRandValDigraph
4   *---- Actions ----*
5    ['1', '2', '3', '4', '5', '6', '7']
6   *---- Characteristic valuation domain ----*
7    {'med': Decimal('0.0'), 'hasIntegerValuation': False,
8     'min': Decimal('-1.0'), 'max': Decimal('1.0')}
9   * ---- Relation Table -----
10  r(xSy) |  '1'    '2'    '3'   '4'    '5'    '6'   '7'
11  -------|-------------------------------------------
12  '1'    |   0.00 -0.48  0.70  0.86  0.30  0.38  0.44
13  '2'    |  -0.22  0.00 -0.38  0.50  0.80 -0.54  0.02
14  '3'    |  -0.42  0.08  0.00  0.70 -0.56  0.84 -1.00
15  '4'    |   0.44 -0.40 -0.62  0.00  0.04  0.66  0.76
16  '5'    |   0.32 -0.48 -0.46  0.64  0.00 -0.22 -0.52
17  '6'    |  -0.84  0.00 -0.40 -0.96 -0.18  0.00 -0.22
18  '7'    |   0.88  0.72  0.82  0.52 -0.84  0.04  0.00
19  *--- Connected Components ---*
20   1: ['1', '2', '3', '4', '5', '6', '7']
21  Neighborhoods:
22   Gamma:
23   '1': in => {'5', '7', '4'}, out => {'5', '7', '6', '3', '4'}
24   '2': in => {'7', '3'}, out => {'5', '7', '4'}
25   '3': in => {'7', '1'}, out => {'6', '2', '4'}
26   '4': in => {'5', '7', '1', '2', '3'}, out => {'5', '7', '1', '6'}
27   '5': in => {'1', '2', '4'}, out => {'1', '4'}
28   '6': in => {'7', '1', '3', '4'}, out => set()
29   '7': in => {'1', '2', '4'}, out => {'1', '2', '3', '4', '6'}
30   Not Gamma:
31   '1': in => {'6', '2', '3'}, out => {'2'}
32   '2': in => {'5', '1', '4'}, out => {'1', '6', '3'}
33   '3': in => {'5', '6', '2', '4'}, out => {'5', '7', '1'}
34   '4': in => {'6'}, out => {'2', '3'}
35   '5': in => {'7', '6', '3'}, out => {'7', '6', '2', '3'}
36   '6': in => {'5', '2'}, out => {'5', '7', '1', '3', '4'}
37   '7': in => {'5', '6', '3'}, out => {'5'}
```

Warning: Mind that most Digraph class methods will ignore the reflexive couples by considering that the reflexive relations are **indeterminate**, i.e. the characteristic value $r(x\,S\,x)$ for all action x is put to the *median*, i.e. *indeterminate* value 0 in this case (see Listing 2.2 Lines 12-18 and [BIS-2004]).

2.2 Graphviz drawings

We may have an even better insight into the `Digraph` object *dg* by looking at a graphviz (https://graphviz.org/) drawing[1] .

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


Rubis Python Server (graphviz), R. Bisdorff, 2008
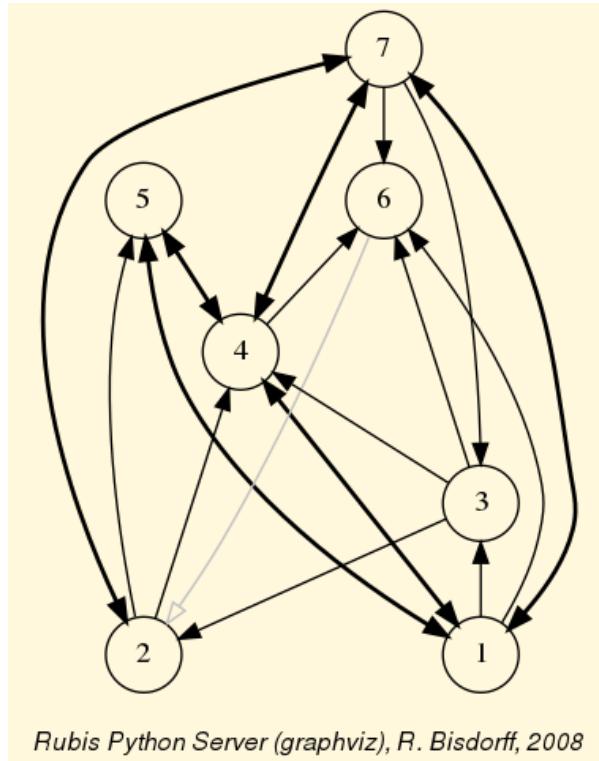
Fig. 1: The tutorial random valuation digraph

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 (see Fig. 1).

2.3 Asymmetric and symmetric parts

We may now extract both the *symmetric* as well as the *asymmetric* part of digraph *dg* with the help of two corresponding constructors (see Fig. 2).

```
1  >>> from digraphs import AsymmetricPartialDigraph,
2                          SymmetricPartialDigraph
3  >>> asymDg = AsymmetricPartialDigraph(dg)
4  >>> asymDg.exportGraphViz()
5  >>> symDG = SymmetricPartialDigraph(dg)
6  >>> symDg.exportGraphViz()
```
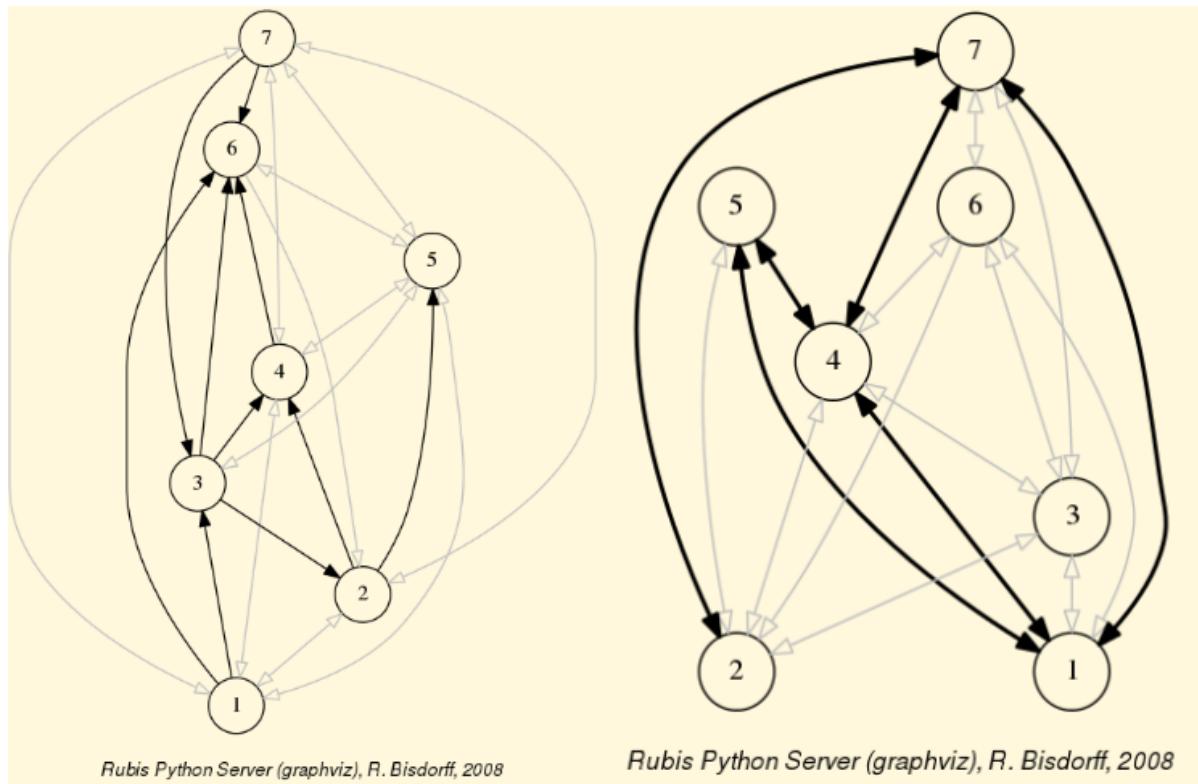

Fig. 2: Asymmetric and symmetric part of the tutorial random valuation digraph

Note: The constructor of the partial objects *asymDg* and *symDg* puts to the indeterminate characteristic value all *not-asymmetric*, respectively *not-symmetric* links between nodes (see Fig. 2).

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

```
1  def _constructRelation(self):
2      actions = self.actions
3      Min = self.valuationdomain['min']
4      Max = self.valuationdomain['max']
```

(continues on next page)

14

```python
 5        Med = self.valuationdomain['med']
 6        relationIn = self.relation
 7        relationOut = {}
 8        for a in actions:
 9            relationOut[a] = {}
10            for b in actions:
11                if a != b:
12                    if relationIn[a][b] >= Med and relationIn[b][a] <= Med:
13                        relationOut[a][b] = relationIn[a][b]
14                    elif relationIn[a][b] <= Med and relationIn[b][a] >= Med:
15                        relationOut[a][b] = relationIn[a][b]
16                    else:
17                        relationOut[a][b] = Med
18                else:
19                    relationOut[a][b] = Med
20        return relationOut
```

2.4 Border and inner parts

We may also extract the border -the part of a digraph induced by the union of its initial and terminal prekernels (see tutorial *On computing digraph kernels* (page 163))- as well as, the inner part -the *complement* of the border- with the help of two corresponding class constructors: `digraphs.GraphBorder` and `digraphs.GraphInner` (see Listing 2.3 Line 1).

Let us illustrate these parts on a linear ordering obtained from the tutorial random valuation digraph *dg* (see Listing 2.3 Line 2-3) with the *NetFlows ranking rule* (page 59)).

Listing 2.3: Border and inner part of a linear order

```
1  >>> from digraphs import GraphBorder, GraphInner
2  >>> from linearOrders import NetFlowsOrder
3  >>> nf = NetFlowsOrder(dg)
4  >>> nf.netFlowsOrder
5   ['6', '4', '5', '3', '2', '1', '7']
6  >>> bnf = GraphBorder(nf)
7  >>> bnf.exportGraphViz(worstChoice=['6'],bestChoice=['7'])
8  >>> inf = GraphInner(nf)
9  >>> inf.exportGraphViz(worstChoice=['6'],bestChoice=['7'])
```
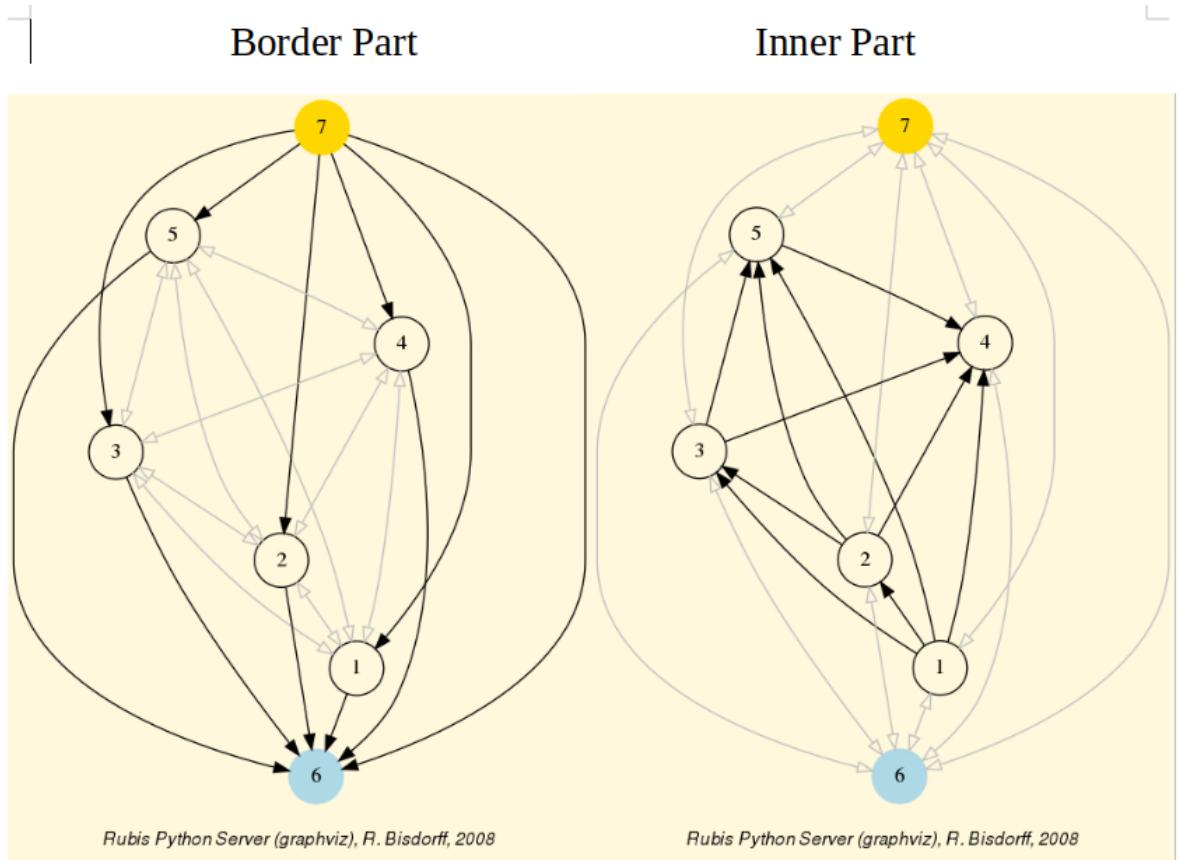

Fig. 3: *Border* and *inner* part of a linear order oriented by *terminal* and *initial* kernels

We may orient the graphviz drawings in Fig. 3 with the terminal node 6 (*worstChoice* parameter) and initial node 7 (*bestChoice* parameter), see Listing 2.3 Lines 7 and 9.

Note: The constructor of the partial digraphs *bnf* and *inf* (see Listing 2.3 Lines 3 and 6) puts to the *indeterminate* characteristic value all links *not* in the *border*, respectively *not* in the *inner* part (see Fig. 4).

Being much *denser* than a linear order, the actual inner part of our tutorial random valuation digraph *dg* is reduced to a single arc between nodes 3 and 4 (see Fig. 4).
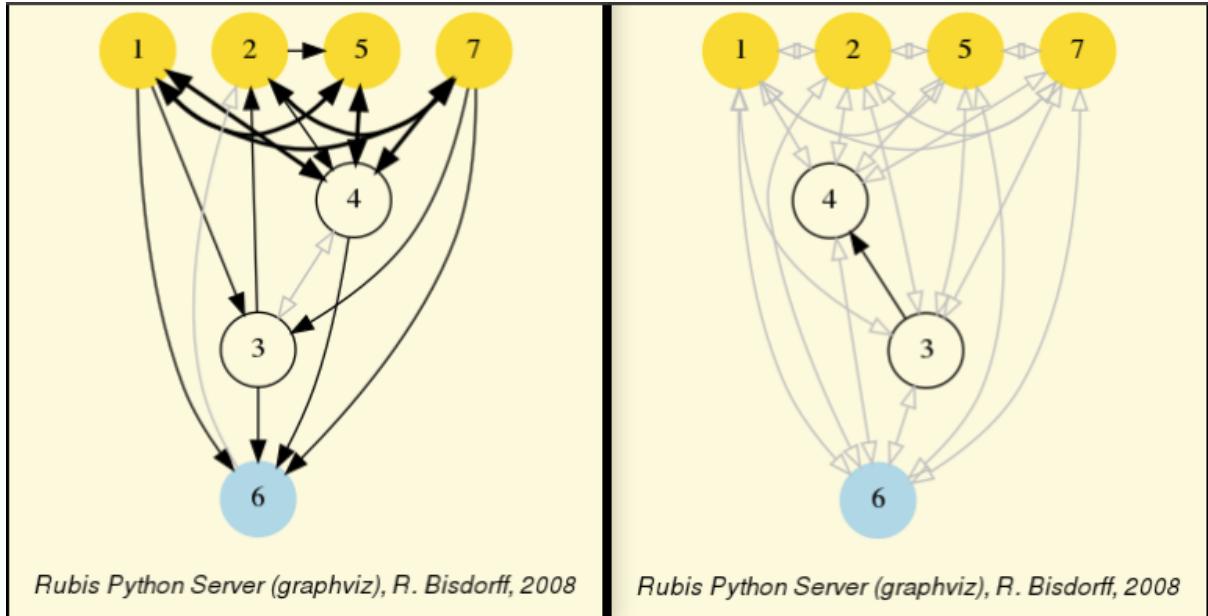
Fig. 4: Border and inner part of the tutorial random valuation digraph *dg*

Indeed, a *complete* digraph on the limit has no inner part (privacy!) at all, whereas *empty* and *indeterminate* digraphs admit both, an empty border and an empty inner part.

2.5 Fusion by epistemic disjunction

We may recover object *dg* from both partial objects *asymDg* and *symDg*, or as well from the border *bg* and the inner part *ig*, with a **bipolar fusion** constructor, also called **epistemic disjunction**, available via the `digraphs.FusionDigraph` class (see Listing 2.1 Lines 12- 21).

Listing 2.4: Epistemic fusion of partial diagraphs

```
>>> from digraphs import FusionDigraph
>>> fusDg = FusionDigraph(asymDg,symDg,operator='o-max')
>>> # fusDg = FusionDigraph(bg,ig,operator='o-max')
>>> 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
```

The epistemic disjunction operation **o-max** (see Listing 2.4 Line 2) works as follows.

Let r and r' characterise two bipolar-valued epistemic situations.

17

- o-max(r, r') = max(r, r') when both r and r' are *validated* (positive);

- o-max(r, r') = min(r, r') when both r and r' are *invalidated* (negative);

- o-max(r, r') = *indeterminate* otherwise.

2.6 Dual, converse and codual digraphs

We may as readily compute the **dual** (negated relation[14]), the **converse** (transposed relation) and the **codual** (transposed and negated relation) of the digraph instance *dg*.

```
1   >>> from digraphs import DualDigraph, ConverseDigraph, CoDualDigraph
2   >>> ddg = DualDigraph(dg)
3   >>> ddg.showRelationTable()
4    -r(xSy) |  '1'     '2'    '3'   '4'    '5'     '6'   '7'
5    --------|-------------------------------------------
6    '1 '    |   0.00   0.48 -0.70 -0.86 -0.30 -0.38 -0.44
7    '2'     |   0.22   0.00  0.38 -0.50  0.80  0.54 -0.02
8    '3'     |   0.42   0.08  0.00 -0.70  0.56 -0.84  1.00
9    '4'     |  -0.44   0.40  0.62  0.00 -0.04 -0.66 -0.76
10   '5'     |  -0.32   0.48  0.46 -0.64  0.00  0.22  0.52
11   '6'     |   0.84   0.00  0.40  0.96  0.18  0.00  0.22
12   '7'     |   0.88 -0.72 -0.82 -0.52  0.84 -0.04  0.00
13  >>> cdg = ConverseDigraph(dg)
14  >>> cdg.showRelationTable()
15   * ---- Relation Table -----
16    r(ySx) |  '1'     '2'    '3'   '4'    '5'    '6'    '7'
17    --------|-------------------------------------------
18   '1'      |   0.00 -0.22 -0.42  0.44  0.32 -0.84  0.88
19   '2'      |  -0.48  0.00  0.08 -0.40 -0.48  0.00  0.72
20   '3'      |   0.70 -0.38  0.00 -0.62 -0.46 -0.40  0.82
21   '4'      |   0.86  0.50  0.70  0.00  0.64 -0.96  0.52
22   '5'      |   0.30  0.80 -0.56  0.04  0.00 -0.18 -0.84
23   '6'      |   0.38 -0.54  0.84  0.66 -0.22  0.00  0.04
24   '7'      |   0.44  0.02 -1.00  0.76 -0.52 -0.22  0.00
25  >>> cddg = CoDualDigraph(dg)
26  >>> cddg.showRelationTable()
27   * ---- Relation Table -----
28    -r(ySx) |  '1'     '2'    '3'    '4'    '5'    '6'    '7'
29    --------|-------------------------------------------
30   '1'      |   0.00   0.22  0.42 -0.44 -0.32  0.84 -0.88
31   '2'      |   0.48   0.00 -0.08  0.40  0.48  0.00 -0.72
32   '3'      |  -0.70   0.38  0.00  0.62  0.46  0.40 -0.82
33   '4'      |  -0.86 -0.50 -0.70  0.00 -0.64  0.96 -0.52
34   '5'      |  -0.30 -0.80  0.56 -0.04  0.00  0.18  0.84
35   '6'      |  -0.38   0.54 -0.84 -0.66  0.22  0.00 -0.04
36   '7'      |  -0.44 -0.02  1.00 -0.76  0.52  0.22  0.00
```

---

[14] Not to be confused with the *dual graph* of a plane graph *g* that has a vertex for each face of *g*. Here we mean the *less than* (strict converse) relation corresponding to a *greater or equal* relation, or the *less than or equal* relation corresponding to a (strict) *better than* relation.

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.

Listing 2.5: Computing the *dual*, the *converse* and the *codual* of a digraph

```
1  >>> ddg = -dg    # dual of dg
2  >>> cdg = ~dg    # converse of dg
3  >>> cddg = ~(-dg) # = -(~(dg) codual of dg
4  >>> (-(~dg)).showRelationTable()
5   * ---- Relation Table -----
6   -r(ySx) |  '1'    '2'    '3'    '4'    '5'    '6'    '7'
7   --------|-------------------------------------------
8   '1'     |   0.00  0.22  0.42 -0.44 -0.32  0.84 -0.88
9   '2'     |   0.48  0.00 -0.08  0.40  0.48  0.00 -0.72
10  '3'     |  -0.70  0.38  0.00  0.62  0.46  0.40 -0.82
11  '4'     |  -0.86 -0.50 -0.70  0.00 -0.64  0.96 -0.52
12  '5'     |  -0.30 -0.80  0.56 -0.04  0.00  0.18  0.84
13  '6'     |  -0.38  0.54 -0.84 -0.66  0.22  0.00 -0.04
14  '7'     |  -0.44 -0.02  1.00 -0.76  0.52  0.22  0.00
```

2.7 Symmetric and transitive closures

Symmetric and transitive closure in-site constructors are also available (see Fig. 5). 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 `digraphs.Digraph.save()`, writes out in a named file the python content of the Digraph object in string representation.

Listing 2.6: Symmetric and transitive closures

```
1  >>> dg.save('tutRandValDigraph')
2  >>> dg.closeSymmetric()
3  >>> dg.closeTransitive()
4  >>> dg.exportGraphViz('strongComponents')
```


Rubis Python Server (graphviz), R. Bisdorff, 2008
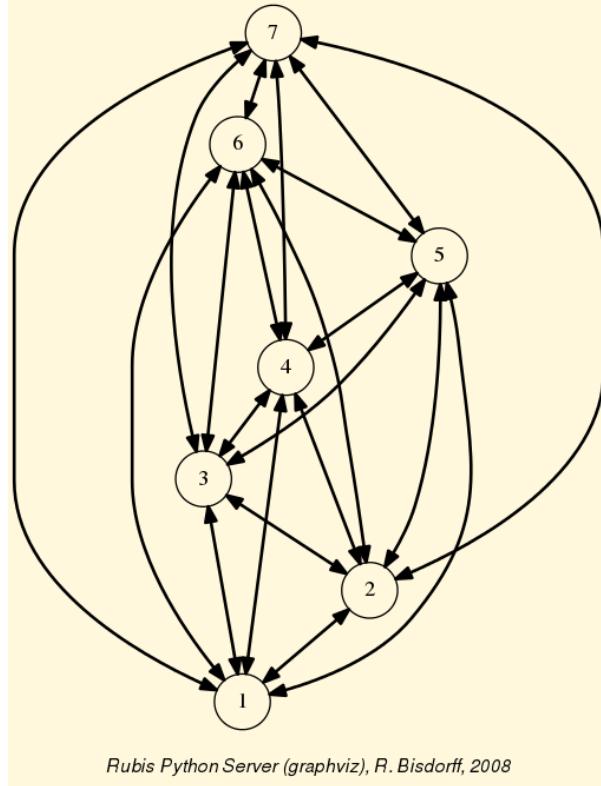
Fig. 5: Symmetric and transitive closure of the tutorial random valuation digraph

The `digraphs.Digraph.closeSymmetric()` method (see Listing 2.6 Line 2), of complexity $\mathcal{O}(n^2)$ where n denotes the digraph's order, changes, on the one hand, all single pairwise links it may detect into double links by operating a disjunction of the pairwise relations. On the other hand, the `digraphs.Digraph.closeTransitive()` method (see Listing 2.6 Line 3), implements the *Roy-Warshall* transitive closure algorithm of complexity $\mathcal{O}(n^3)$. ([17])

Note: The same `digraphs.Digraph.closeTransitive()` method with a *Reverse = True* flag may be readily used for eliminating all transitive arcs from a transitive digraph instance. We make usage of this feature when drawing *Hasse diagrams* of `transitiveDigraphs.TransitiveDigraph` objects.

---

[17] Roy, B. *Transitivité et connexité.* C. R. Acad. Sci. Paris 249, 216-218, 1959. Warshall, S. *A Theorem on Boolean Matrices.* J. ACM 9, 11-12, 1962.

2.8 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()
```

2.9 CSV storage

Sometimes it is required to exchange the graph valuation data in CSV format with a statistical package like R (https://www.r-project.org/). 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.

```
1  >>> dgcsv = CSVDigraph('tutRandValDigraph')
2  >>> dgcsv.showRelationTable(ReflexiveTerms=False)
3   * ---- Relation Table -----
4   r(xSy) |   '1'    '2'    '3'    '4'    '5'    '6'    '7'
5   -------|-------------------------------------------------------------
6   '1'    |   -    -0.48   0.70   0.86   0.30   0.38   0.44
7   '2'    | -0.22    -    -0.38   0.50   0.80  -0.54   0.02
8   '3'    | -0.42   0.08    -     0.70  -0.56   0.84  -1.00
9   '4'    |  0.44  -0.40  -0.62    -     0.04   0.66   0.76
10  '5'    |  0.32  -0.48  -0.46   0.64    -    -0.22  -0.52
11  '6'    | -0.84   0.00  -0.40  -0.96  -0.18    -    -0.22
12  '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 (see Fig. 6).

```
1  >>> 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

Fig. 6: The valued relation table shown in a browser window

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

2.10 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.

```
1  >>> from digraphs import CompleteDigraph,EmptyDigraph,
2                          IndeterminateDigraph
3  >>> help(CompleteDigraph)
4   Help on class CompleteDigraph in module digraphs:
5   class CompleteDigraph(Digraph)
6    |  Parameters:
7    |      order > 0; valuationdomain=(Min,Max).
8    |  Specialization of the general Digraph class for generating
9    |  temporary complete graphs of order 5 in {-1,0,1} by default.
10   |  Method resolution order:
11   |      CompleteDigraph
12   |      Digraph
13   |      builtins.object
14   ...
15  >>> e = EmptyDigraph(order=5)
16  >>> e.showRelationTable()
17   * ---- Relation Table -----
18     S   |    '1'     '2'     '3'     '4'     '5'
19    ---- -|----------------------------------
20    '1'   |   -1.00   -1.00   -1.00   -1.00   -1.00
21    '2'   |   -1.00   -1.00   -1.00   -1.00   -1.00
22    '3'   |   -1.00   -1.00   -1.00   -1.00   -1.00
23    '4'   |   -1.00   -1.00   -1.00   -1.00   -1.00
24    '5'   |   -1.00   -1.00   -1.00   -1.00   -1.00
25   >>> e.showNeighborhoods()
26   Neighborhoods:
27     Gamma     :
28  '1': in => set(), out => set()
29  '2': in => set(), out => set()
30  '5': in => set(), out => set()
31  '3': in => set(), out => set()
32  '4': in => set(), out => set()
33     Not Gamma :
34  '1': in => {'2', '4', '5', '3'}, out => {'2', '4', '5', '3'}
35  '2': in => {'1', '4', '5', '3'}, out => {'1', '4', '5', '3'}
36  '5': in => {'1', '2', '4', '3'}, out => {'1', '2', '4', '3'}
37  '3': in => {'1', '2', '4', '5'}, out => {'1', '2', '4', '5'}
38  '4': in => {'1', '2', '5', '3'}, out => {'1', '2', '5', '3'}
39  >>> i = IndeterminateDigraph()
40   * ---- Relation Table -----
41     S   |    '1'    '2'    '3'    '4'    '5'
42    ------|----------------------------
43    '1'   |   0.00   0.00   0.00   0.00   0.00
44    '2'   |   0.00   0.00   0.00   0.00   0.00
45    '3'   |   0.00   0.00   0.00   0.00   0.00
46    '4'   |   0.00   0.00   0.00   0.00   0.00
```

```
47   '5'  |  0.00  0.00  0.00  0.00  0.00
48  >>> i.showNeighborhoods()
49   Neighborhoods:
50     Gamma      :
51   '1': in => set(), out => set()
52   '2': in => set(), out => set()
53   '5': in => set(), out => set()
54   '3': in => set(), out => set()
55   '4': in => set(), out => set()
56     Not Gamma :
57   '1': in => set(), out => set()
58   '2': in => set(), out => set()
59   '5': in => set(), out => set()
60   '3': in => set(), out => set()
61   '4': in => set(), out => set()
```

Note: Mind 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* (see Listing 2.7 Lines 34-38) whereas, in the latter, *nothing is known* about the actual neighborhoods of the nodes (see Listing 2.7 Lines 57-61). These two cases illustrate why in the case of bipolar-valued digraphs, we may need both a *gamma* **and** a *notGamma* function.

Back to *Content Table* (page 3)

3 Working with the `outrankingDigraphs` module

See also the technical documentation of the outrankingDigraphs module.

3.1 Outranking digraph model

In this *Digraph3* module, the main `outrankingDigraphs.BipolarOutrankingDigraph`
class provides a generic **bipolar-valued 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-valued outranking digraph consisting of 7 decision actions denoted *a01*,
a02, . . . , *a07*.

```
1  >>> from outrankingDigraphs import RandomBipolarOutrankingDigraph
2  >>> odg = RandomBipolarOutrankingDigraph()
3  >>> odg.showActions()
4      *----- show digraphs actions --------------*
5      key:  a01
6      name:       random decision action
7      comment:    RandomPerformanceTableau() generated.
8      key:  a02
9      name:       random decision action
10     comment:    RandomPerformanceTableau() generated.
11     ...
12     ...
13     key:  a07
14     name:       random decision action
15     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.

```
1  >>> odg.showCriteria()
2   *----  criteria -----*
3   g01 'RandomPerformanceTableau() instance'
4      Scale = [0.0, 100.0]
5      Weight = 3.0
6      Threshold pref : 20.00 + 0.00x ; percentile:  0.28
7      Threshold ind : 10.00 + 0.00x ; percentile:  0.095
8      Threshold veto : 80.00 + 0.00x ; percentile:  1.0
9   g02 'RandomPerformanceTableau() instance'
10     Scale = [0.0, 100.0]
11     Weight = 3.0
12     Threshold pref : 20.00 + 0.00x ; percentile:  0.33
13     Threshold ind : 10.00 + 0.00x ; percentile:  0.19
14     Threshold veto : 80.00 + 0.00x ; percentile:  0.95
15  ...
16  ...
17  g07 'RandomPerformanceTableau() instance'
18     Scale = [0.0, 100.0]
19     Weight = 10.0
20     Threshold pref : 20.00 + 0.00x ; percentile:  0.476
21     Threshold ind : 10.00 + 0.00x ; percentile:  0.238
22     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*.

```
1  >>> odg.showPerformanceTableau()
2   *----  performance tableau -----*
3   criteria | 'a01'   'a02'   'a03'   'a04'   'a05'   'a06'   'a07'
4   ---------|-------------------------------------------------------
5     'g01'  |   9.6    48.8    21.7    37.3    81.9    48.7    87.7
6     'g02'  |  90.9    11.8    96.6    41.0    34.0    53.9    46.3
7     'g03'  |  97.8    46.4    83.3    30.9    61.5    85.4    82.5
8     'g04'  |  40.5    43.6    53.2    17.5    38.6    21.5    67.6
9     'g05'  |  33.0    40.7    96.4    55.1    46.2    58.1    52.6
10    'g06'  |  47.6    19.0    92.7    55.3    51.7    26.6    40.4
11    'g07'  |  41.2    64.0    87.7    71.6    57.8    59.3    34.7
```

3.2 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

Fig. 1: Visualizing a performance tableau in a browser window

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

Fig. 2: Ranked heatmap of the performance table

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.

3.3 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 bipolar-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')}
```

3.4 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-
 →veto
 --------------------------------------------------------------------------------
 →----
 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].

3.5 Recoding the digraph 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.

3.6 The strict outranking digraph

From the theory (see [BIS-2013], [ADT-L7]) we know that a bipolar-valued outranking digraph is **weakly complete**, i.e. if $r(x\,S\,y) < 0.0$ then $r(y\,S\,x) >= 0.0$. From this property follows that a bipolar-valued outranking relation verifies the **coduality** principle: the dual (strict negation -[14]) of the converse (inverse ˜) 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
```
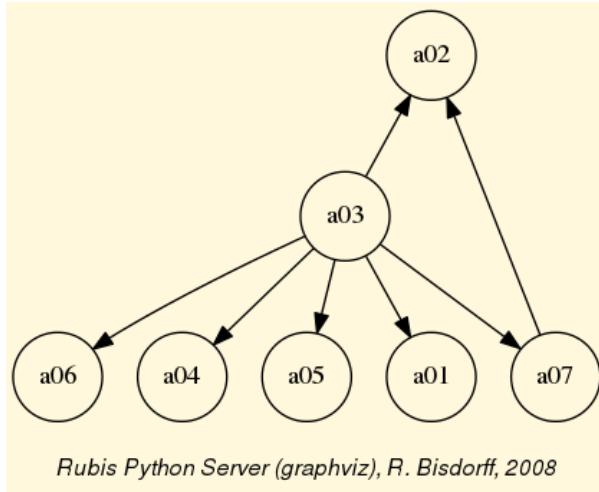
Fig. 3: Codual digraph

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.

3.7 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 (https://www.decision-deck.org/xmcda/) XML encoding scheme of MCDA data, as provided by the Decision Deck Project (see https://www.decision-deck.org/).

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

The resulting XML file may 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. Re-instantiating again a corresponding updated *odg* object goes like follow.

```
1  >>> pt = XMCDA2PerformanceTableau('tutorialPerfTab')
2  >>> odg = BipolarOutrankingDigraph(pt)
3  >>> odg.showRelationTable()
4   * ---- Relation Table -----
5     S  |  'a01'    'a02'   'a03'   'a04'   'a05'   'a06'   'a07'
6   ------|------------------------------------------------------------
7   'a01' |   +0.00   +29.73  -29.73  +13.51  +48.65  +40.54  +48.65
```

```
 8    'a02' |    +13.51   +0.00  -100.00   +37.84   +13.51   +43.24   -37.84
 9    'a03' |    +83.78  +100.00   +0.00   +91.89   +83.78   +83.78   +70.27
10    'a04' |    +24.32   +48.65  -56.76    +0.00   +24.32   +51.35   +24.32
11    'a05' |    +51.35  +100.00  -70.27   +72.97    +0.00   +51.35   +32.43
12    'a06' |    +16.22   +72.97  -51.35   +35.14   +32.43    +0.00   +37.84
13    'a07' |    +67.57   +45.95  -24.32   +27.03   +27.03   +45.95    +0.00
```

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

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

Back to *Content Table* (page 3)

4 Generating random performance tableaux

4.1 Introduction

The `randomPerfTabs` module provides several constructors for random performance tableaux models 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 most useful models.

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.

One of the most useful models, called **RandomCBPerformanceTableau**, proposes a performance tableau involving 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 conflicting, objectives. *Low costs*

will randomly be coupled with *low benefits*, whereas *high costs* will randomly be coupled with high benefits.

Many public policy decision problems involve three often conflicting decision objectives taking into account *economical, societal* as well as *environmental* aspects. For this type of performance tableau model, we provide a specific model, called **Random3ObjectivesPerformanceTableau**.

Deciding which students, based on the grades obtained in a number of examinations, validate or not their academic studies, is the genuine decision practice of universities and academies. To thouroughly study these kind of decision problems, we provide a performance tableau model, called **RandomAacademicPerformanceTableau**, which gathers grades obtained by a given number of students in a given number of weighted courses.

In order to study aggregation of election results (see the tutorial on *Computing the winner of an election* (page 47)) in the context of bipolar-valued outranking digraphs, we provide furthermore a specific performance tableau model called **RandomRankPerformanceTableau** which provides ranks (linearly ordered performances without ties) of a given number of election candidates (decision actions) for a given number of weighted voters (performance criteria).

4.2 Random standard performance tableaux

The `randomPerfTabs.RandomPerformanceTableau` class, the simplest of the kind, specializes the generic `prefTabs.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 performance 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 (default = ('beta',None,(2,2))):

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

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

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

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

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

- missingDataProbability := 0 <= float <= 1.0 ; probability of missing performance evaluation on a criterion for an alternative (default 0.025).

Code example.

Listing 4.1: Generating a random performance tableau

```
>>> from randomPerfTabs import RandomPerformanceTableau
>>> t = RandomPerformanceTableau(numberOfActions=21,numberOfCriteria=13,
↪seed=100)
>>> t.actions
    {'a01': {'comment': 'RandomPerformanceTableau() generated.',
            'name': 'random decision action'},
     'a02': { ... },
      ...
     }
>>> t.criteria
    {'g01': {'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'},
      'g02':  { ... },
       ...
      }
>>> t.evaluation
    {'g01': {'a01': Decimal('15.17'),
            'a02': Decimal('44.51'),
            'a03': Decimal('-999'),  # missing evaluation
             ...
            },
```

```
27          ...
28        }
29  >>> t.showHTMLPerformanceTableau()
```

Performance table randomperftab

criteria	g01	g02	g03	g04	g05	g06	g07	g08	g09	g10	g11	g12	g13
weight	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
a01	15.17	46.37	82.88	41.14	59.94	41.19	58.68	44.73	22.19	64.64	34.93	42.36	17.55
a02	44.51	16.22	41.66	53.58	31.39	65.22	71.96	57.84	78.08	77.37	8.30	63.41	61.55
a03	NA	21.53	12.82	56.93	26.80	48.03	54.35	62.42	94.27	73.57	71.11	21.81	56.90
a04	58.00	51.16	21.92	65.57	59.02	44.77	37.49	58.39	80.79	55.39	46.44	19.57	39.22
a05	24.22	77.01	75.74	83.87	40.85	8.55	85.44	67.34	57.40	39.08	64.83	29.37	96.39
a06	29.10	39.35	15.45	34.99	49.12	11.49	28.44	52.89	64.24	62.92	58.28	32.02	10.25
a07	96.58	32.06	6.05	49.56	NA	66.06	41.64	13.08	38.31	24.82	48.39	57.03	42.91
a08	82.29	47.67	9.96	79.43	29.45	84.17	31.99	90.88	39.58	50.78	61.88	44.40	48.26
a09	43.90	14.81	60.55	42.37	6.72	56.14	34.20	51.54	21.79	79.13	50.95	93.16	81.89
a10	38.75	79.70	27.88	42.39	71.88	66.09	58.33	58.88	17.10	44.25	48.73	30.63	52.73
a11	35.84	67.48	38.81	33.75	26.87	64.10	71.95	62.72	NA	85.80	58.37	49.33	NA
a12	29.12	13.97	67.45	38.60	48.30	11.87	NA	57.76	74.86	26.57	48.80	43.57	7.68
a13	34.79	90.72	38.93	57.38	64.14	97.86	91.16	43.80	33.68	38.98	28.87	63.36	60.03
a14	62.22	80.16	19.26	62.34	60.96	24.72	73.63	71.21	56.43	46.12	26.09	51.43	12.86
a15	44.23	69.62	94.95	34.95	63.46	52.97	98.84	78.74	36.64	65.12	22.46	55.52	68.79
a16	19.10	45.49	65.63	64.96	50.57	55.91	10.02	34.70	29.31	50.15	70.68	62.57	71.09
a17	27.73	22.03	48.00	79.38	23.35	74.03	58.74	59.42	50.95	82.27	49.20	43.27	38.61
a18	41.46	33.83	7.97	75.11	49.00	55.70	64.99	38.47	49.86	17.45	28.08	35.21	67.81
a19	22.41	NA	34.86	49.30	65.18	39.84	81.16	NA	55.99	66.55	55.38	43.08	29.72
a20	21.52	69.98	71.81	43.74	24.53	55.39	52.67	13.67	66.80	57.46	70.81	5.41	76.05
a21	56.90	48.80	31.66	15.31	40.57	58.14	70.19	67.23	61.10	31.04	60.72	22.39	70.38

Fig. 1: Browser view on random performance tableau instance

Note: Missing (NA) evaluation are registered in a performance tableau as *Decimal('-999')* value (see Listing 4.1 Line 24). Best and worst performance on each criterion are marked in *light green*, respectively in *light red*.

4.3 Random Cost-Benefit performance 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

Listing 4.2: Generating a random Cost-Benefit performance tableau

```
>>> from randomPerfTabs import RandomCBPerformanceTableau
>>> t = RandomCBPerformanceTableau(
        numberOfActions=7,\
        numberOfCriteria=5,\
        weightDistribution='equiobjectives',\
        commonPercentiles={'ind':5,'pref':10,'veto':95},\
```

```
7            seed=100)
8  >>> t.showActions()
9   *----- show decision action --------------*
10  key:  a1
11     short name: a1
12     name:       random cheap decision action
13  key:  a2
14     short name: a2
15     name:       random neutral decision action
16  ...
17  key:  a7
18     short name: a7
19     name:       random advantageous decision action
20 >>> t.showCriteria()
21  *----  criteria -----*
22  g1 'random ordinal benefit criterion'
23     Scale = (0, 10)
24     Weight = 0.167
25  g2 'random cardinal cost criterion'
26     Scale = (0.0, 100.0)
27     Weight = 0.250
28     Threshold ind  :  1.76 + 0.00x ; percentile:  0.095
29     Threshold pref :  2.16 + 0.00x ; percentile:  0.143
30     Threshold veto : 73.19 + 0.00x ; percentile:  0.952
31  ...
```

In the example above, we may notice the three types of decision actions (Listing 4.2 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.

```
1  >>> t.showStatistics()
2   *-------- Performance tableau summary statistics -------*
3   Instance name      : randomCBperftab
4   #Actions           : 7
5   #Criteria          : 5
6   *Statistics per Criterion*
7   Criterion name      : g1
8     Criterion weight    : 2
9     criterion scale    : 0.00 - 10.00
10    mean evaluation    : 5.14
11    standard deviation : 2.64
12    maximal evaluation : 8.00
13    quantile Q3 (x_75) : 8.00
14    median evaluation  : 6.50
15    quantile Q1 (x_25) : 3.50
16    minimal evaluation : 1.00
```

```
17     mean absolute difference      : 2.94
18     standard difference deviation : 3.74
19  Criterion name          : g2
20    Criterion weight      : 3
21    criterion scale      : -100.00 - 0.00
22    mean evaluation      : -49.32
23    standard deviation : 27.59
24    maximal evaluation : 0.00
25    quantile Q3 (x_75) : -27.51
26    median evaluation  : -35.98
27    quantile Q1 (x_25) : -54.02
28    minimal evaluation : -91.87
29    mean absolute difference      : 28.72
30    standard difference deviation : 39.02
31  ...
```

A (potentially ranked) colored heatmap with 5 color levels is also provided.

```
>>> t.showHTMLPerformanceHeatmap(colorLevels=5,rankingRule=None)
```


Fig. 2: Unranked heatmap of a random Cost-Benefit performance tableau

Such a performance tableau may be stored and re-accessed as follows.

```
1  >>> t.save('temp')
2   *----- saving performance tableau in XMCDA 2.0 format  -------------*
3   File: temp.py saved !
4  >>> from perfTabs import PerformanceTableau
5  >>> t = PerformanceTableau('temp')
```

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

```
1  >>> t.saveCSV('temp')
2   * --- Storing performance tableau in CSV format in file temp.csv
```

```
1  ...$ less temp.csv
2   "actions","g1","g2","g3","g4","g5"
3   "a1",1.00,-17.92,-33.99,26.68,3.00
4   "a2",8.00,-30.71,-77.77,66.35,6.00
5   "a3",8.00,-41.65,-69.84,53.43,8.00
6   "a4",2.00,-39.49,-16.99,18.62,2.00
7   "a5",6.00,-91.87,-74.85,83.09,7.00
8   "a6",7.00,-32.47,-24.91,79.24,9.00
9   "a7",4.00,-91.11,-7.44,48.22,7.00
```

Back to

4.4 Random three objectives performance tableaux

We provide the `randomPerfTabs.Random3ObjectivesPerformanceTableau` class for generating random performance tableaux concerning potential public policies evaluated with respect to three preferential decision objectives taking respectively into account *economical*, *societal* as well as *environmental* aspects.

Each public policy 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 normalized 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,
- missingDataProbability = 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

Listing 4.3: Generating a random 3 Objectives performance tableau

```
1  >>> from randomPerfTabs import Random3ObjectivesPerformanceTableau
2  >>> t = Random3ObjectivesPerformanceTableau(
3              numberOfActions=31,
4              numberOfCriteria=13,
5              weightDistribution='equiobjectives',
6              seed=120)
7  >>> t.showObjectives()
8   *------ show objectives -------"
9   Eco: Economical aspect
10     g04 criterion of objective Eco 20
11     g05 criterion of objective Eco 20
12     g08 criterion of objective Eco 20
13     g11 criterion of objective Eco 20
14    Total weight: 80.00 (4 criteria)
15   Soc: Societal aspect
16     g06 criterion of objective Soc 16
17     g07 criterion of objective Soc 16
18     g09 criterion of objective Soc 16
19     g10 criterion of objective Soc 16
20     g13 criterion of objective Soc 16
21    Total weight: 80.00 (5 criteria)
22   Env: Environmental aspect
23     g01 criterion of objective Env 20
24     g02 criterion of objective Env 20
25     g03 criterion of objective Env 20
26     g12 criterion of objective Env 20
27    Total weight: 80.00 (4 criteria)
```

In Listing 4.3 above, we notice that 5 *equisignificant* criteria (g06, g07, g09, g10, g13) evaluate for instance the performance of the public policies from a **societal** point of view (Lines 16-21). 4 *equisignificant* criteria do the same from an **economical** (Lines 10-14), respectively an **environmental** point of view (Lines 21-27). The *equiobjectives* directive results hence in a balanced total weight (80.00) for each decision objective.

```
1  >>> t.showActions()
2   key:  p01
3     name:        random public policy Eco+ Soc- Env+
4     profile:     {'Eco': 'good', 'Soc': 'weak', 'Env': 'good'}
5   key:  p02
6   ...
7   key:  p26
8     name:        random public policy Eco+ Soc+ Env-
9     profile:     {'Eco': 'good', 'Soc': 'good', 'Env': 'weak'}
10  ...
11  key:  p30
12    name:        random public policy Eco- Soc- Env-
13    profile:     {'Eco': 'weak', 'Soc': 'weak', 'Env': 'weak'}
14  ...
```

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 public policy with respect to the three objectives. Policy *p01*, for instance, will probably show *good* performances wrt the *economical* and environmental aspects, and *weak* performances wrt the *societal* aspect.

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

```
1  >>> from perfTabs import PartialPerformanceTableau
2  >>> teco = PartialPerformanceTableau(t,criteriaSubset=\
3                              t.objectives['Eco']['criteria'])
4  >>> tsoc = PartialPerformanceTableau(t,criteriaSubset=\
5                              t.objectives['Soc']['criteria'])
6  >>> tenv = PartialPerformanceTableau(t,criteriaSubset=\
7                              t.objectives['Env']['criteria'])
```

One may thus compute a partial bipolar-valued outranking digraph for each individual objective.

```
1  >>> from outrankingDigraphs import BipolarOutrankingDigraph
2  >>> geco = BipolarOutrankingDigraph(teco)
3  >>> gsoc = BipolarOutrankingDigraph(tsoc)
4  >>> 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.

```
1  >>> from digraphs import FusionLDigraph
2  >>> gfus = FusionLDigraph([geco,gsoc,genv])
3  >>> gfus.strongComponents()
4   {frozenset({'p30'}),
5    frozenset({'p10', 'p03', 'p19', 'p08', 'p07', 'p04', 'p21', 'p20',
```

41

```
6                'p13', 'p23', 'p16', 'p12', 'p24', 'p02', 'p31', 'p29',
7                'p05', 'p09', 'p28', 'p25', 'p17', 'p14', 'p15', 'p06',
8                'p01', 'p27', 'p11', 'p18', 'p22'}),
9   frozenset({'p26'})}
10  >>> from digraphs import StrongComponentsCollapsedDigraph
11  >>> scc = StrongComponentsCollapsedDigraph(gfus)
12  >>> scc.showActions()
13   *----- show digraphs actions --------------*
14   key:  frozenset({'p30'})
15     short name: Scc_1
16     name:       _p30_
17     comment:    collapsed strong component
18   key:  frozenset({'p10', 'p03', 'p19', 'p08', 'p07', 'p04', 'p21', 'p20', 'p13',
19                    'p23', 'p16', 'p12', 'p24', 'p02', 'p31', 'p29', 'p05', 'p09',
    ↪ 'p28', 'p25',
20                    'p17', 'p14', 'p15', 'p06', 'p01', 'p27', 'p11', 'p18', 'p22'}
    ↪)
21     short name: Scc_2
22     name:       _p10_p03_p19_p08_p07_p04_p21_p20_p13_p23_p16_p12_p24_p02_p31_\
23                  p29_p05_p09_p28_p25_p17_p14_p15_p06_p01_p27_p11_p18_p22_
24     comment:    collapsed strong component
25   key:  frozenset({'p26'})
26     short name: Scc_3
27     name:       _p26_
28     comment:    collapsed strong component
```

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

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

42
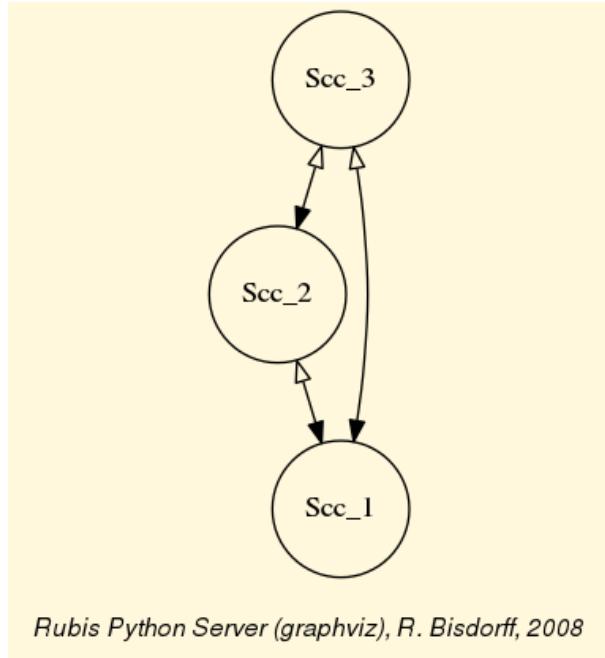
Fig. 3: Strong components digraph

Public policy *p26* (Eco+ Soc+ Env-) appears dominating the other policies, whereas policy *p30* (Eco- Soc- Env-) appears to be dominated by all the others.

4.5 Random academic performance tableaux

The `randomPerfTabs.RandomAcademicPerformanceTableau` class generates temporary performance tableaux with random grades for a given number of students in different courses (see Lecture 4: *Grading*, Algorithmic decision Theory Course http://hdl.handle. net/10993/37933)

Parameters:

- number of students,
- number of courses,
- weightDistribution := 'equisignificant' | 'random' (default)
- weightScale := (1, 1 | numberOfCourses (default when random))
- IntegerWeights := Boolean (True = default)
- commonScale := (0,20) (default)
- ndigits := 0
- WithTypes := Boolean (False = default)
- commonMode := ('triangular',xm=14,r=0.25) (default)
- commonThresholds := {'ind':(0,0), 'pref':(1,0)} (default)
- missingDataProbability := 0.0 (default)

43

When parameter *WithTypes* is set to *True*, the students are randomly allocated to one of the four categories: *weak* (1/6), *fair* (1/3), *good* (1/3), and *excellent* (1/3), in the bracketed proportions. In a default 0-20 grading range, the random range of a weak student is 0-10, of a fair student 4-16, of a good student 8-20, and of an excellent student 12-20. The random grading generator follows in this case a double triangular probablity law with *mode* (*xm*) equal to the middle of the random range and *median repartition* (*r* = 0.5) of probability each side of the mode.

Listing 4.4: Generating a random academic performance tableau

```
>>> from randomPerfTabs import RandomAcademicPerformanceTableau
>>> t = RandomAcademicPerformanceTableau(numberOfStudents=11,
              numberOfCourses=7, missingDataProbability=0.03,
              WithTypes=True, seed=100)
>>> t
 *------- PerformanceTableau instance description ------*
 Instance class    : RandomAcademicPerformanceTableau
 Seed              : 100
 Instance name     : randstudPerf
 # Actions         : 11
 # Criteria        : 7
 Attributes        : ['randomSeed', 'name', 'actions',
                      'criteria', 'evaluation', 'weightPreorder']
>>> t.showPerformanceTableau()
 *----  performance tableau -----*
  Courses |   'g1'   'g2'   'g3'   'g4'   'g5'   'g6'   'g7'
     ECTS |    2      1      3      4      1      1      5
  --------|--------------------------------------------
    's01f' |   12     13     15     08     16     06     15
    's02g' |   10     15     20     11     14     15     18
    's03g' |   14     12     19     11     15     13     11
    's04f' |   13     15     12     13     13     10     06
    's05e' |   12     14     13     16     15     12     16
    's06g' |   17     13     10     14     NA     15     13
    's07e' |   12     12     12     18     NA     13     17
    's08f' |   14     12     09     13     13     15     12
    's09g' |   19     14     15     13     09     13     16
    's10g' |   10     12     14     17     12     16     09
    's11w' |   10     10     NA     10     10     NA     08
>>> t.weightPreorder
 [['g2', 'g5', 'g6'], ['g1'], ['g3'], ['g4'], ['g7']]
```

The example tableau, generated for instance above with *missingDataProbability* = 0.03, *WithTypes* = True and *seed* = 100 (see Listing 4.4 Lines 2-4), results in a set of two excellent (*s05*, *s07*), five good (*s02*, *s03*, *s06*, *s09*, *s10*), three fair (*s01*, *s04*, *s08*) and one weak (*s11*) student performances. Notice that six students get a grade below the course validating threshold 10 and we observe four missing grades (NA), two in course *g5* and one in course *g3* and course *g6* (see Lines 19-29).

We may show a statistical summary of the students' grades obtained in the heighest

weighted course, namely *g7*, followed by a performance heatmap browser view showing a global ranking of the students' performances from best to weakest.

Listing 4.5: Student performance summary statistics per course

```
>>> t.showCourseStatistics('g7')
 *----- Summary performance statistics ------*
  Course name     : g7
  Course weight   : 5
  # Students      : 11
  grading scale   : 0.00 - 20.00
  # missing evaluations : 0
  mean evaluation       : 12.82
  standard deviation    : 3.79
  maximal evaluation    : 18.00
  quantile Q3 (x_75)    : 16.25
  median evaluation     : 14.00
  quantile Q1 (x_25)    : 10.50
  minimal evaluation    : 6.00
  mean absolute difference      : 4.30
  standard difference deviation : 5.35
>>> t.showHTMLPerformanceHeatmap(colorLevels=5,
                 pageTitle='Ranking the students')
```

Ranking the students

criteria	g7	g4	g3	g1	g2	g5	g6
weights	+5.00	+4.00	+3.00	+2.00	+1.00	+1.00	+1.00
s07e	17.00	18.00	12.00	12.00	12.00	NA	13.00
s02g	18.00	11.00	20.00	10.00	15.00	14.00	15.00
s09g	16.00	13.00	15.00	19.00	14.00	9.00	13.00
s05e	16.00	16.00	13.00	12.00	14.00	15.00	12.00
s06g	13.00	14.00	10.00	17.00	13.00	NA	15.00
s03g	11.00	11.00	19.00	14.00	12.00	15.00	13.00
s10g	9.00	17.00	14.00	10.00	12.00	12.00	16.00
s01f	15.00	8.00	15.00	12.00	13.00	16.00	6.00
s08f	12.00	13.00	9.00	14.00	12.00	13.00	15.00
s04f	6.00	13.00	12.00	13.00	15.00	13.00	10.00
s11w	8.00	10.00	NA	10.00	10.00	10.00	NA

Color legend:

quantile	20.00%	40.00%	60.00%	80.00%	100.00%

Fig. 4: Ranking the students with a performance heatmap view

The ranking shown here in Fig. 4 is produced with the default *NetFlows* rule (see tutorial *Ranking with multiple incommensurable criteria* (page 59)). With a mean marginal

45

correlation of $+0.361$ (see Listing 4.6 Lines 15-) associated with a low standard deviation (0.248), the result represents a rather *fair weighted consensus* made between the individual courses' marginal rankings.

Listing 4.6: Consensus quality of the students's ranking

```
1  >>> t.showRankingConsensusQuality(t.netFlowsRanking)
2   Consensus quality of ranking:
3    ['s07', 's02', 's09', 's05', 's06', 's03', 's10',
4     's01', 's08', 's04', 's11']
5   criterion (weight): correlation
6   -------------------------------
7    g7 (0.294): +0.727
8    g4 (0.235): +0.309
9    g2 (0.059): +0.291
10   g3 (0.176): +0.200
11   g1 (0.118): +0.109
12   g6 (0.059): +0.091
13   g5 (0.059): +0.073
14   Summary:
15   Weighted mean marginal correlation (a): +0.361
16   Standard deviation (b)                 : +0.248
17   Ranking fairness (a)-(b)               : +0.113
```

4.6 Random linearly ranked performance tableaux

Finally, we provide the `randomPerfTabs.RandomRankPerformanceTableau` class for generating multiple criteria ranked performance tableaux, i.e. on each criterion, all decision action's evaluations appear linearly ordered without ties.

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

Parameters:

- number of actions,
- number of performance criteria,
- weightDistribution := 'equisignificant' | 'random' (default, see 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 *Content Table* (page 3)

46

5 Computing the winner of an election

5.1 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.

```
1  candidates = OrderedDict([('a1',...), ('a2',...), ('a3', ...), ...}
2  voters = OrderedDict([('v1',{'weight':10}), ('v2',{'weight':3}), ...}
3  ## each voter specifies a linearly ranked list of candidates
4  ## from the best to the worst (without ties
5  linearBallot = {
6  'v1' : ['a2','a3','a1', ...],
7  'v2' : ['a1','a2','a3', ...],
8  ...
9  }
```

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.

Listing 5.1: Example of random linear voting profile

```
1  >>> from votingProfiles import RandomLinearVotingProfile
2  >>> v = RandomLinearVotingProfile(numberOfVoters=5,
3                                    numberOfCandidates=3,
4                                    RandomWeights=True)
5  >>> v.candidates
6  OrderedDict([ ('a1',{'name':'a1}), ('a2',{'name':'a2'}),
```

```
7                   ('a3',{'name':'a3'}) ])
8  >>> v.voters
9   OrderedDict([('v1',{'weight': 2}), ('v2':{'weight': 3}),
10    ('v3',{'weight': 1}), ('v4':{'weight': 5}),
11    ('v5',{'weight': 4})])
12 >>> v.linearBallot
13  {'v1': ['a1', 'a2', 'a3',],
14    'v2': ['a3', 'a2', 'a1',],
15    'v3': ['a1', 'a3', 'a2',],
16    'v4': ['a1', 'a3', 'a2',],
17    'v5': ['a2', 'a3', 'a1',]}
```

Notice that in this random example, the five voters are weighted (see Listing 5.1 Line 6-7). Their linear ballots can be viewed with the `votingProfiles.LinearVotingProfile.showLinearBallots()` method.

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

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

```
1  >>> v.save(fileName='tutorialLinearVotingProfile1')
2   *--- Saving linear profile in file: <tutorialLinearVotingProfile1.py> ---*
3  >>> v = LinearVotingProfile('tutorialLinearVotingProfile1')
```

5.2 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.

```
1  >>> v.computeUninominalVotes()
2   {'a2': 2, 'a1': 6, 'a3': 7}
3  >>> v.computeSimpleMajorityWinner()
4   ['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]).

```
1  >>> v.computeInstantRunoffWinner(Comments=True)
2   Half of the Votes =  7.50
3   ==> stage =  1
4       remaining candidates ['a1', 'a2', 'a3']
5       uninominal votes {'a1': 6, 'a2': 2, 'a3': 7}
6       minimal number of votes =  2
7       maximal number of votes =  7
8       candidate to remove =  a2
9       remaining candidates =  ['a1', 'a3']
10  ==> stage =  2
11      remaining candidates ['a1', 'a3']
12      uninominal votes {'a1': 8, 'a3': 7}
13      minimal number of votes =  7
14      maximal number of votes =  8
15      candidate a1 obtains an absolute majority
16  Instant run off winner: ['a1']
```

In stage 1, no candidate obtains an absolute majority of votes. Candidate *a2* obtains the minimal number of votes (2/15) and is, hence, eliminated. In stage 2, candidate *a1* obtains an absolute majority of the votes (8/15) and is eventually elected (see Listing 5.2).

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

Listing 5.3: Example of *Borda* rank scores

```
1  >>> v.computeRankAnalysis()
2   {'a2': [2, 5, 8], 'a1': [6, 9, 0], 'a3': [7, 1, 7]}
3  >>> v.computeBordaScores()
4   OrderedDict([
5     ('a1', {'BordaScore': 24, 'averageBordaScore': 1.6}),
6     ('a3', {'BordaScore': 30, 'averageBordaScore': 2.0}),
7     ('a2', {'BordaScore': 36, 'averageBordaScore': 2.4}) ])
8  >>> v.computeBordaWinners()
9   ['a1']
```

Candidate *a1* obtains the minimal *Borda* score, followed by candidate *a3* and finally candidate *a2* (see Listing 5.3). The corresponding *Borda* **rank analysis table** may be printed out with a corresponding `show` command.

Listing 5.4: Rank analysis example

```
1  >>> v.showRankAnalysisTable()
2   *----  Borda rank analysis tableau -----*
3   candi- | alternative-to-rank |     Borda
4   dates  |   1     2     3     | score  average
5   -------|----------------------------------
```

```
6     'a1'  |   6      9      0      | 24/15   1.60
7     'a3'  |   7      1      7      | 30/15   2.00
8     'a2'  |   2      5      8      | 36/15   2.40
```

In our randomly generated election results, we are lucky: The instant runoff winner and the *Borda* winner both are candidate *a1* (see Listing 5.2 and Listing 5.4). 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.

5.3 The Condorcet winner

For instance, candidate *a1* is ranked four times before and once behind candidate *a2*. Hence the corresponding **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 kind of digraphs.

Listing 5.5: Example of *Condorcet* digraph

```
1  >>> from votingProfiles import CondorcetDigraph
2  >>> cdg = CondorcetDigraph(v,hasIntegerValuation=True)
3  >>> cdg
4   *------- Digraph instance description ------*
5   Instance class      : CondorcetDigraph
6   Instance name       : rel_randomLinearVotingProfile1
7   Digraph Order       : 3
8   Digraph Size        : 3
9   Valuation domain    : [-15.00;15.00]
10  Determinateness (%) : 64.44
11  Attributes          : ['name', 'actions', 'voters',
12                         'ballot', 'valuationdomain',
13                         'relation', 'order',
14                         'gamma', 'notGamma']
15 >>> cdg.showAll()
16  *----- show detail -------------*
17  Digraph           : rel_randLinearVotingProfile1
18  *---- Actions ----*
19  ['a1', 'a2', 'a3']
20  *---- Characteristic valuation domain ----*
21  {'max': Decimal('15.0'), 'med': Decimal('0'),
22   'min': Decimal('-15.0'), 'hasIntegerValuation': True}
23  * ---- majority margins -----
24     M(x,y)   |  'a1'   'a2'   'a3'
25     ----------|-------------------
26      'a1'    |    0     11      1
27      'a2'    |  -11      0     -1
28      'a3'    |   -1      1      0
29  Valuation domain: [-15;+15]
```

50

Notice that in the case of linear voting profiles, majority margins always verify a zero sum property: $M(x,y) + M(y,x) = 0$ for all candidates x and y (see Listing 5.5 Lines 26-28). This is not true in general for arbitrary voting profiles. The *Condorcet* digraph of linear voting profiles defines in fact a *weak tournament* and belongs, hence, to the class of *self-codual* bipolar-valued digraphs ([13]).

Now, 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** (see Listing 5.5 Line 26).

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

By seeing the majority margins like a *bipolar-valued characteristic function* of a global preference relation defined on the set of candidates, we may use all operational resources of the generic `Digraph` class (see *Working with the Digraph3 software resources* (page 4)), and especially its `exportGraphViz` method[1], for visualizing an election result.

```
1   >>> cdg.exportGraphViz(fileName='tutorialLinearBallots')
2   *---- exporting a dot file for GraphViz tools ---------*
3   Exporting to tutorialLinearBallots.dot
4   dot -Grankdir=BT -Tpng tutorialLinearBallots.dot -o tutorialLinearBallots.png
```
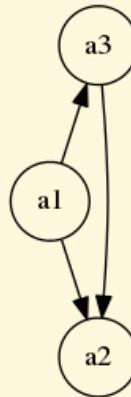

Fig. 1: Visualizing an election result

In Fig. 1 we notice that the *Condorcet* digraph from our example linear voting profile gives a linear order of the candidates: ['a1', 'a3', 'a2], the same actually as given by

---

[13] The class of *self-codual* bipolar-valued digraphs consists of all *weakly asymmetric* digraphs, i.e. digraphs containing only *asymmetric* and/or *indeterminate* links. Limit cases consists of, on the one side, *full tournaments* with *indeterminate reflexive links*, and, on the other side, *fully indeterminate* digraphs. In this class, the *converse* (inverse ~) operator is indeed identical to the *dual* (negation -) one.

the *Borda* scores (see Listing 5.3). This is by far not given in general. Usually, when aggregating linear ballots, there appear cyclic social preferences.

5.4 Cyclic social preferences

Let us consider for instance the following linear voting profile and construct the corresponding Condorcet digraph.

Listing 5.6: Example of cyclic social preferences

```
1  >>> v.showLinearBallots()
2   voters(weight)      candidates rankings
3   v1(1):        ['a1', 'a3', 'a5', 'a2', 'a4']
4   v2(1):        ['a1', 'a2', 'a4', 'a3', 'a5']
5   v3(1):        ['a5', 'a2', 'a4', 'a3', 'a1']
6   v4(1):        ['a3', 'a4', 'a1', 'a5', 'a2']
7   v5(1):        ['a4', 'a2', 'a3', 'a5', 'a1']
8   v6(1):        ['a2', 'a4', 'a5', 'a1', 'a3']
9   v7(1):        ['a5', 'a4', 'a3', 'a1', 'a2']
10  v8(1):        ['a2', 'a4', 'a5', 'a1', 'a3']
11  v9(1):        ['a5', 'a3', 'a4', 'a1', 'a2']
12  >>> cdg = CondorcetDigraph(v)
13  >>> cdg.showRelationTable()
14   * ---- Relation Table -----
15     S  |  'a1'   'a2'   'a3'   'a4'   'a5'
16    ------|-------------------------------------
17   'a1' |   -     0.11  -0.11  -0.56  -0.33
18   'a2' | -0.11    -     0.11   0.11  -0.11
19   'a3' |  0.11  -0.11    -    -0.33  -0.11
20   'a4' |  0.56  -0.11   0.33    -     0.11
21   'a5' |  0.33   0.11   0.11  -0.11    -
```

Now, we cannot find any completely positive row in the relation table (see Listing 5.6 Lines 17 -). 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 (see Fig. 2).

```
1  >>> cdg.exportGraphViz('cycles')
2   *---- exporting a dot file for GraphViz tools ---------*
3   Exporting to cycles.dot
4   dot -Grankdir=BT -Tpng cycles.dot -o cycles.png
```
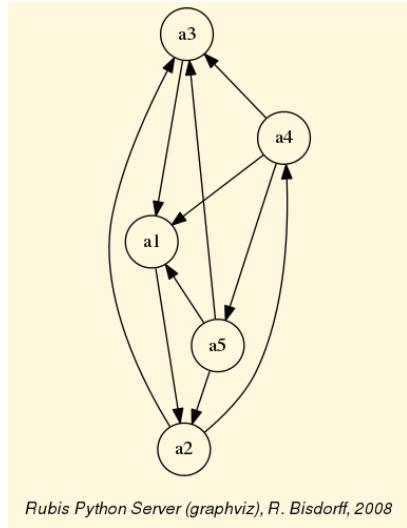
Fig. 2: Cyclic social preferences

But, there may be many cycles appearing in a *Condorcet* digraph, and, we may detect and enumerate all minimal chordless circuits in a Digraph instance with the `digraphs.Digraph.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 the tutorial on *ranking with multiple incommensurable criteria* (page 59) and [BIS-2008]).

Many more tools for exploiting voting results are available like the browser heat map view on voting profiles (see the technical documentation of the `votingProfiles` module).

```
1  >>> v.showHTMLVotingHeatmap(rankingRule='NetFlows',
2                               Transposed=False)
```

Voting Heatmap

criteria	v5	v3	v8	v7	v6	v9	v4	v2	v1
weights	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00
tau(*)	+0.60	+0.60	+0.40	+0.40	+0.40	+0.20	+0.00	-0.40	-0.80
a4	1	3	2	2	2	3	2	3	5
a5	4	1	3	1	3	1	4	5	3
a2	2	2	1	5	1	5	5	2	4
a3	3	4	5	3	5	2	1	4	2
a1	5	5	4	4	4	4	3	1	1

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*
Ranking rule: **NetFlows**
Ordinal (Kendall) correlation between global ranking and global outranking relation: **+0.778**

Fig. 3: Visualizing a linear voting profile in a heatmap format

Notice that the importance weights of the voters are *negative*, which means that the preference direction of the criteria (in this case the individual voters) is *decreasing*, i.e. goes from lowest (best) rank to highest (worst) rank. Notice also, that the compromise *NetFlows* ranking *[a4,a5,a2,a1,a3]*, shown in this heatmap (see Fig. 3) results in an optimal *ordinal correlation* index of +0.778 with the pairwise majority voting margins (see the Adavanced topic on Ordinal Correlation equals Relational Equivalence and *Ranking with multiple incommensurable criteria* (page 59)). The number of voters is usually much larger than the number of candidates. In that case, it is better to generate a transposed *voters X candidates* view (see Listing 5.7 Line 2)

5.5 On generating random linear voting profiles

By default, the `votingProfiles.RandomLinearVotingProfile` class generates random linear voting profiles where every candidates has the same uniform probabilities to be ranked at a certain position by all the voters. For each voter's random linear ballot is indeed generated via a uniform shuffling of the list of candidates.

In reality, political election data appear quite different. There will usually be different favorite and marginal candidates for each political party. To simulate these aspects into our random generator, we are using two random exponentially distributed polls of the candidates and consider a bipartisan political landscape with a certain random balance (default theoretical party repartition = 0.50) between the two sets of potential party supporters (see `votingProfiles.LinearVotingProfile` class). A certain theoretical proportion (default = 0.1) will not support any party.

Let us generate such a linear voting profile for an election with 1000 voters and 15 candidates.

Listing 5.8: Generating a linear voting profile with random polls

```
1  >>> from votingProfiles import RandomLinearVotingProfile
2  >>> lvp = RandomLinearVotingProfile(numberOfCandidates=15,
3                                      numberOfVoters=1000,
4                                      WithPolls=True,
5                                      partyRepartition=0.5,
6                                      other=0.1,
7                                      seed=0.9189670954954139)
8  >>> lvp
9   *------- VotingProfile instance description ------*
10  Instance class    : RandomLinearVotingProfile
11  Instance name     : randLinearProfile
12  # Candidates      : 15
13  # Voters          : 1000
14  Attributes        : ['name', 'seed', 'candidates',
15                       'voters', 'RandomWeights',
16                       'sumWeights', 'poll1', 'poll2',
17                       'bipartisan', 'linearBallot', 'ballot']
18  >>> lvp.showRandomPolls()
19  Random repartition of voters
20   Party_1 supporters : 460 (46.0%)
21   Party_2 supporters : 436 (43.6%)
22   Other voters       : 104 (10.4%)
23  *---------------- random polls ----------------
24   Party_1(46.0%) | Party_2(43.6%)|  expected
25   ------------------------------------------------
26    a06 : 19.91%  | a11 : 22.94%  | a06 : 15.00%
27    a07 : 14.27%  | a08 : 15.65%  | a11 : 13.08%
28    a03 : 10.02%  | a04 : 15.07%  | a08 : 09.01%
29    a13 : 08.39%  | a06 : 13.40%  | a07 : 08.79%
30    a15 : 08.39%  | a03 : 06.49%  | a03 : 07.44%
31    a11 : 06.70%  | a09 : 05.63%  | a04 : 07.11%
32    a01 : 06.17%  | a07 : 05.10%  | a01 : 05.06%
33    a12 : 04.81%  | a01 : 05.09%  | a13 : 05.04%
34    a08 : 04.75%  | a12 : 03.43%  | a15 : 04.23%
35    a10 : 04.66%  | a13 : 02.71%  | a12 : 03.71%
36    a14 : 04.42%  | a14 : 02.70%  | a14 : 03.21%
37    a05 : 04.01%  | a15 : 00.86%  | a09 : 03.10%
38    a09 : 01.40%  | a10 : 00.44%  | a10 : 02.34%
39    a04 : 01.18%  | a05 : 00.29%  | a05 : 01.97%
40    a02 : 00.90%  | a02 : 00.21%  | a02 : 00.51%
```

In this example (see Listing 5.8 Lines 18-), we obtain 460 Party_1 supporters (46%), 436 Party_2 supporters (43.6%) and 104 other voters (10.4%). Favorite candidates of *Party_1* supporters, with more than 10%, appear to be *a06* (19.91%), *a07* (14.27%) and *a03* (10.02%). Whereas for *Party_2* supporters, favorite candidates appear to be *a11*

(22.94%), followed by *a08* (15.65%), *a04* (15.07%) and *a06* (13.4%). Being *first* choice for *Party_1* supporters and *fourth* choice for *Party_2* supporters, this candidate *a06* is a natural candidate for clearly winning this election game (see Listing 5.9).

Listing 5.9: The uninominal election winner

```
1  >>> lvp.computeSimpleMajorityWinner()
2   ['a06']
3  >>> lvp.computeInstantRunoffWinner()
4   ['a06']
5  >>> lvp.computeBordaWinners()
6   ['a06']
```

Is it also a *Condorcet* winner ? To verify, we start by creating the corresponding *Condorcet* digraph *cdg* with the help of the `votingProfiles.CondorcetDigraph` class. The created digraph instance contains 15 *actions* -the candidates- and 105 *oriented* arcs -the *positive* majority margins- (see Listing 5.10 Lines 6-7).

Listing 5.10: A Condorcet digraph constructed from a
linear voting profile

```
1  >>> from votingProfiles import CondorcetDigraph
2  >>> cdg = CondorcetDigraph(lvp)
3   *------- Digraph instance description ------*
4   Instance class      : CondorcetDigraph
5   Instance name       : rel_randLinearProfile
6   Digraph Order       : 15
7   Digraph Size        : 104
8   Valuation domain    : [-1000.00;1000.00]
9   Determinateness (%) : 67.08
10  Attributes          : ['name', 'actions', 'voters',
11                         'ballot', 'valuationdomain',
12                         'relation', 'order',
13                         'gamma', 'notGamma']
```

We may visualize the resulting pairwise majority margins by showing the HTML formated version of the *cdg* relation table in a browser view.

```
>>> cdg.showHTMLRelationTable(tableTitle='Pairwise majority margins',
                             relationName=M(x>y)')
```

Pairwise majority margins

M(x>y)	a01	a02	a03	a04	a05	a06	a07	a08	a09	a10	a11	a12	a13	a14	a15
a01	–	768	-138	108	478	-436	-198	-140	238	440	-268	148	50	202	218
a02	-768	–	-796	-484	-368	-858	-828	-772	-546	-496	-800	-722	-768	-696	-658
a03	138	796	–	160	590	-286	-80	-8	372	522	-158	280	210	360	338
a04	-108	484	-160	–	184	-370	-180	-288	160	136	-420	16	-62	56	30
a05	-478	368	-590	-184	–	-730	-640	-472	-234	-116	-550	-442	-522	-376	-386
a06	436	858	286	370	730	–	248	234	574	692	102	556	482	566	520
a07	198	828	80	180	640	-248	–	0	358	602	-94	304	266	384	420
a08	140	772	8	288	472	-234	0	–	436	396	-176	276	134	298	244
a09	-238	546	-372	-160	234	-574	-358	-436	–	116	-594	-126	-194	-90	-14
a10	-440	496	-522	-136	116	-692	-602	-396	-116	–	-510	-310	-442	-304	-266
a11	268	800	158	420	550	-102	94	176	594	510	–	388	268	474	292
a12	-148	722	-280	-16	442	-556	-304	-276	126	310	-388	–	-92	100	148
a13	-50	768	-210	62	522	-482	-266	-134	194	442	-268	92	–	158	186
a14	-202	696	-360	-56	376	-566	-384	-298	90	304	-474	-100	-158	–	68
a15	-218	658	-338	-30	386	-520	-420	-244	14	266	-292	-148	-186	-68	–

Valuation domain: [-1000; +1000]

Fig. 4: Browsing the majority margins of a *Condorcet* digraph

In Fig. 4, *light green* cells contain the positive majority margins, whereas *light red* cells contain the negative majority margins. A complete *light green* row reveals hence a *Condorcet* **winner**, whereas a complete *light green* column reveals a *Condorcet* **looser**. We recover again candidate *a06* as *Condorcet* winner ([15]), whereas the obvious *Condorcet* looser is here candidate *a02*, the candidate with the lowest support in both parties (see Listing 5.8 Line 40).

With a same *bipolar -first ranked* and *last ranked* candidate- selection procedure, we may *weakly rank* the candidates (with possible ties) by iterating these *first ranked* and *last ranked* choices among the remaining candidates ([BIS-1999]).

Listing 5.11: Ranking by iterating choosing the *first* and *last* remaining candidates

```
>>> cdg.showRankingByChoosing()
 Error: You must first run
  self.computeRankingByChoosing(CoDual=False(default)|True) !
>>> cdg.computeRankingByChoosing()
>>> cdg.showRankingByChoosing()
 Ranking by Choosing and Rejecting
  1st first ranked ['a06']
    2nd first ranked ['a11']
      3rd first ranked ['a07', 'a08']
        4th first ranked ['a03']
          5th first ranked ['a01']
            6th first ranked ['a13']
```

---

[15] The concept of *Condorcet* winner -a generalization of absolute majority winners- proposed by *Condorcet* in 1785, is an early historical example of *initial* digraph kernel (see the tutorial *On computing digraph kernels* (page 163)).

```
13                   7th first ranked ['a04']
14                   7th last ranked ['a12']
15                6th last ranked ['a14']
16             5th last ranked ['a15']
17          4th last ranked ['a09']
18        3rd last ranked ['a10']
19      2nd last ranked ['a05']
20    1st last ranked ['a02']
```

Before showing the *ranking-by-choosing* result, we have to compute the iterated bipolar selection procedure (see Listing 5.11 Line 2). The first selection concerns *a06* (first) and *a02* (last), followed by *a11* (first) opposed to *a05* (last), and so on, until there remains at iteration step 7 a last pair of candidates, namely *[a04, a12]* (see Lines 13-14).

Notice furthermore the first ranked candidates at iteration step 3 (see Listing 5.11 Line 9), namely the pair *[a07, a08]*. Both candidates represent indeed conjointly the *first ranked* choice. We obtain here hence a *weak ranking*, i.e. a ranking with a tie.

Let us mention that the *instant-run-off* procedure, we used before (see Listing 5.9 Line 3), when operated with a *Comments=True* parameter setting, will deliver a more or less similar *reversed* linear *ordering-by-rejecting* result, namely [*a02, a10, a14, a05, a09, a13, a12, a15, a04, a01, a08, a03, a07, a11, a06*], ordered from the *last* to the *first* choice.

Remarkable about both these *ranking-by-choosing* or *ordering-by-rejecting* results is the fact that the random voting behaviour, simulated here with the help of two discrete random variables ([16]), defined respectively by the two party polls, is rendering a ranking that is more or less in accordance with the simulated balance of the polls: -*Party_1* supporters : 460; *Party_2* supporters: 436 (see Listing 5.8 Lines 26-40 third column). Despite a random voting behaviour per voter, the given polls apparently show a *very strong incidence* on the eventual election result. In order to avoid any manipulation of the election outcome, public media are therefore in some countries not allowed to publish polls during the last weeks before a general election.

Note: Mind that the specific *ranking-by-choosing* procedure, we use here on the *Condorcet* digraph, operates the selection procedure by extracting at each step *initial* and *terminal* kernels, i.e. NP-hard operational problems (see tutorial *On computing digraph kernels* (page 163) and [BIS-1999]); A technique that does not allow in general to tackle voting profiles with much more than 30 candidates. The tutorial on *Ranking with multiple incommensurable criteria* (page 59) provides more adequate and efficient techniques for ranking from pairwise majority margins when a larger number of potential candidates is given.

Back to *Content Table* (page 3)

---

[16] Discrete random variables with a given empirical probability law (here the polls) are provided in the `randomNumbers` module by the `randomNumbers.DiscreteRandomVariable` class.

6 Ranking with multiple incommensurable criteria

- *The ranking problem* (page 59)
- *The* Copeland *ranking* (page 62)
- *The* NetFlows *ranking* (page 64)
- Kemeny *rankings* (page 66)
- Slater *rankings* (page 69)
- Kohler*'s ranking-by-choosing rule* (page 71)
- Tideman*'s ranked-pairs rule* (page 73)

6.1 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 bipolar-valued *strict outranking* characteristics, i.e. $r(x \succsim y)$ for all x, y in X (see *The strict outranking digraph* (page 30) and [BIS-2013]).

Let us consider a didactic outranking digraph g generated from a random *Cost-Benefit performance tableau* (page 35) concerning 9 decision alternatives evaluated on 13 performance criteria. We may compute the corresponding *strict outranking digraph* with a *codual transform* (page 18) as follows.

Listing 6.1: Random bipolar-valued strict outranking relation characteristics

```
>>> from outrankingDigraphs import *
>>> t = RandomCBPerformanceTableau(numberOfActions=9,
                                   numberOfCriteria=13,seed=200)
>>> g = BipolarOutrankingDigraph(t,Normalized=True)
>>> gcd = ~(-g) # codual digraph
>>> gcd.showRelationTable(ReflexiveTerms=False)
 * ---- Relation Table -----
 r(>) |  'a1'  'a2'  'a3'  'a4'  'a5'  'a6'  'a7'  'a8'  'a9'
 -----|-------------------------------------------------------
 'a1' |   -    0.00 +0.10 -1.00 -0.13 -0.57 -0.23 +0.10 +0.00
 'a2' | -1.00   -    0.00 +0.00 -0.37 -0.42 -0.28 -0.32 -0.12
 'a3' | -0.10  0.00   -   -0.17 -0.35 -0.30 -0.17 -0.17 +0.00
 'a4' |  0.00  0.00 -0.42   -   -0.40 -0.20 -0.60 -0.27 -0.30
 'a5' | +0.13 +0.22 +0.10 +0.40   -   +0.03 +0.40 -0.03 -0.07
 'a6' | -0.07 -0.22 +0.20 +0.20 -0.37   -   +0.10 -0.03 -0.07
 'a7' | -0.20 +0.28 -0.03 -0.07 -0.40 -0.10   -   +0.27 +1.00
 'a8' | -0.10 -0.02 -0.23 -0.13 -0.37 +0.03 -0.27   -   +0.03
 'a9' |  0.00 +0.12 -1.00 -0.13 -0.03 -0.03 -1.00 -0.03   -
```

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

Listing 6.2: Median cut polarised strict outranking relation characteristics

```
1  >>> ccd = PolarisedOutrankingDigraph(gcd,level=g.valuationdomain['med'],
2                                        KeepValues=False,StrictCut=True)
3  >>> ccd.showRelationTable(ReflexiveTerms=False,IntegerValues=True)
4   *---- Relation Table -----
5     r(>)_med | 'a1' 'a2' 'a3' 'a4' 'a5' 'a6' 'a7' 'a8' 'a9'
6     ---------|---------------------------------------------
7        'a1' |   -    0   +1   -1   -1   -1   -1   +1    0
8        'a2' |  -1    -   +0    0   -1   -1   -1   -1   -1
9        'a3' |  -1    0    -   -1   -1   -1   -1   -1    0
10       'a4' |   0    0   -1    -   -1   -1   -1   -1   -1
11       'a5' |  +1   +1   +1   +1    -   +1   +1   -1   -1
12       'a6' |  -1   -1   +1   +1   -1    -   +1   -1   -1
13       'a7' |  -1   +1   -1   -1   -1   -1    -   +1   +1
14       'a8' |  -1   -1   -1   -1   -1   +1   -1    -   +1
15       'a9' |   0   +1   -1   -1   -1   -1   -1   -1    -
```

Unfortunately, such crisp median-cut *Condorcet* digraphs, associated with a given strict outranking digraph, present only exceptionally a linear ordering. Usually, pairwise majority comparisons do not even render a *complete* or, at least, a *transitive* partial order. There may even frequently appear *cyclic* outranking situations (see the tutorial on *Computing the winner of an election* (page 47)).

To estimate how *difficult* this ranking problem here may be, we may have a look at the corresponding strict outranking digraph *graphviz* drawing ([1]).

```
1  >>> gcd.exportGraphViz('rankingTutorial')
2   *---- exporting a dot file for GraphViz tools ---------*
3   Exporting to rankingTutorial.dot
4   dot -Grankdir=BT -Tpng rankingTutorial.dot -o rankingTutorial.png
```

Rubis Python Server (graphviz), R. Bisdorff, 2008
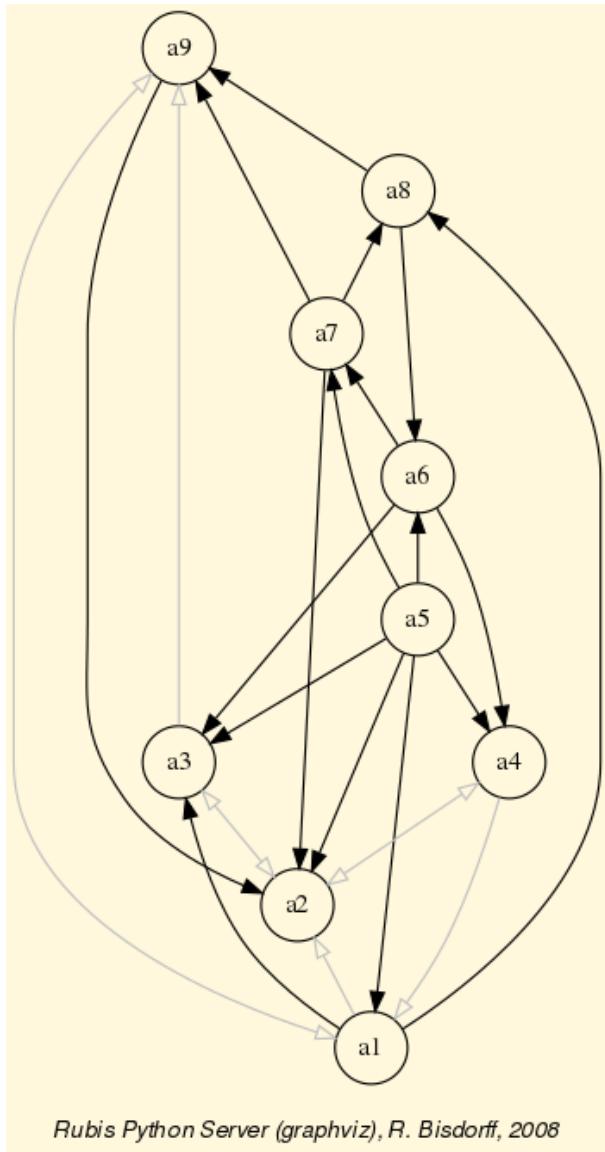
Fig. 1: The *strict outranking* digraph

The strict outranking relation $\succsim$ shown here is apparently *not transitive*: for instance, alternative *a8* outranks alternative *a6* and alternative *a6* outranks *a4*, however *a8* does not outrank *a4* (see Fig. 1). We may compute the transitivity degree of the outranking digraph, i.e. the ratio of the difference between the number of outranking arcs and the number of transitive arcs over the difference of the number of arcs of the transitive closure minus the transitive arcs of the digraph *gcd*.

```
>>> gcd.computeTransitivityDegree(Comments=True)
 Transitivity degree of graph <codual_rel_randomCBperftab>
  #triples x>y>z: 78, #closed: 38, #open: 40
  #closed/#triples =  0.487
```

With only 35% of the required transitive arcs, the strict outranking relation here 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.

```
1  >>> gcd.computeChordlessCircuits()
2  >>> gcd.showChordlessCircuits()
3   1 circuit(s).
4   *---- Chordless circuits ----*
5   1: ['a6', 'a7', 'a8'] , credibility : 0.033
```

There is one chordless circuit detected in the given strict outranking digraph *gcd*, namely *a6* outranks *a7*, the latter outranks *a8*, and *a8* outranks again *a6* (see Fig. 1). Any potential linear ordering of these three alternatives will, in fact, always contradict somehow the given outranking relation.

Now, 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*'s, *Arrow-Raynaud*'s or *Tideman*'s ranking rule.

6.2 The *Copeland* ranking

Copeland's rule, the most intuitive one as it works well for any strict outranking relation which models in fact a linear order, works on the *median cut* strict outranking digraph *ccd*. The rule computes for each alternative a score resulting from the sum of the differences between the crisp **strict outranking** characteristics $r(x \succsim y)_{>0}$ and the crisp **strict outranked** characteristics $r(y \succsim x)_{>0}$ for all pairs of alternatives where y is different from x. The alternatives are ranked in decreasing order of these *Copeland* scores; ties, the case given, being resolved by a lexicographical rule.

Listing 6.3: Computing a *Copeland* Ranking

```
1  >>> from linearOrders import CopelandRanking
2  >>> cop = CopelandRanking(gcd,Comments=True)
3   Copeland decreasing scores
4    a5 : 12
5    a1 : 2
6    a6 : 2
7    a7 : 2
8    a8 : 0
9    a4 : -3
10   a9 : -3
11   a3 : -5
12   a2 : -7
13  Copeland Ranking:
14  ['a5', 'a1', 'a6', 'a7', 'a8', 'a4', 'a9', 'a3', 'a2']
```

Alternative *a5* obtains here the best *Copeland* score (+12), followed by alternatives *a1*, *a6* and *a7* with same score (+2); following the lexicographic rule, *a1* is hence ranked before *a6* and *a6* before *a7*. Same situation is observed for *a4* and *a9* with a score of -3 (see Listing 6.3 Lines 4-12).

Copeland's ranking rule appears in fact **invariant** under the *codual transform* (page 18)

62

and renders a same linear order indifferently from digraphs *g* or *gcd* . The resulting ranking (see Listing 6.3 Line 14) is rather correlated (+0.463) with the given pairwise outranking relation in the ordinal *Kendall* sense (see Listing 6.4).

Listing 6.4: Checking the quality of the *Copeland* Ranking

```
>>> corr = g.computeRankingCorrelation(cop.copelandRanking)
>>> g.showCorrelation(corr)
 Correlation indexes:
  Crisp ordinal correlation : +0.463
  Valued equivalalence      : +0.107
  Epistemic determination   :  0.230
```

With an epistemic determination level of 0.230, the *extended Kendall tau* index (see [BIS-2012]) is in fact computed on 61.5% (100.0 x (1.0 + 0.23)/2) of the pairwise strict outranking comparisons. Furthermore, the bipolar-valued *relational equivalence* characteristics between the strict outranking relation and the *Copeland* ranking equals +0.107, i.e. a *majority* of 55.35% of the criteria significance supports the relational equivalence between the given strict outranking relation and the corresponding *Copeland* ranking.

The *Copeland* scores deliver actually only a unique *weak ranking*, i.e. a ranking with potential ties. This weak ranking may be constructed with the `transitiveDigraphs.WeakCopelandOrder` class.

Listing 6.5: Computing a weak *Copeland* ranking

```
>>> from transitiveDigraphs import WeakCopelandOrder
>>> wcop = WeakCopelandOrder(g)
>>> wcop.showRankingByChoosing()
 Ranking by Choosing and Rejecting
   1st ranked ['a5']
     2nd ranked ['a1', 'a6', 'a7']
       3rd ranked ['a8']
       3rd last ranked ['a4', 'a9']
     2nd last ranked ['a3']
   1st last ranked ['a2']
```

We recover in Listing 6.5 above, the ranking with ties delivered by the *Copeland* scores (see Listing 6.3). We may draw its corresponding *Hasse* diagram (see Listing 6.6).

Listing 6.6: Drawing a weak *Copeland* ranking

```
>>> wcop.exportGraphViz(fileName='weakCopelandRanking')
 *---- exporting a dot file for GraphViz tools ---------*
 Exporting to weakCopelandRanking.dot
 0 { rank = same; a5; }
 1 { rank = same; a1; a7; a6; }
 2 { rank = same; a8; }
 3 { rank = same; a4; a9}
 4 { rank = same; a3; }
 5 { rank = same; a2; }
```

63

```
10    dot -Grankdir=TB -Tpng weakCopelandRanking.dot\
11        -o weakCopelandRanking.png
```
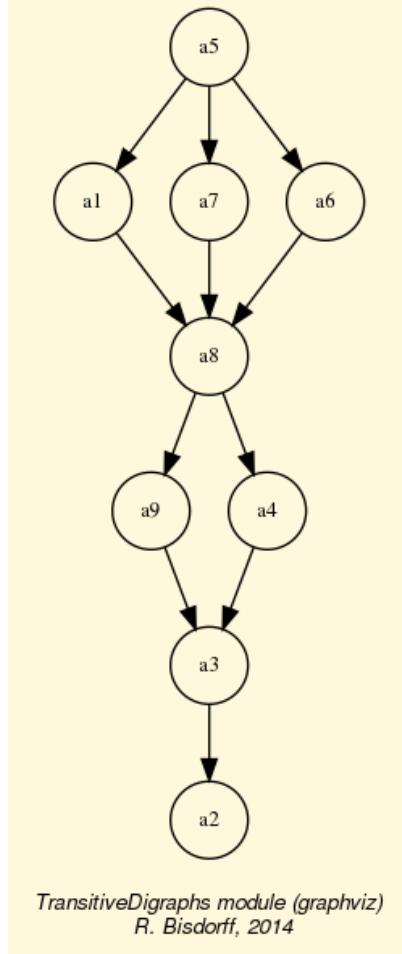

Fig. 2: A weak Copeland ranking

Let us now consider a similar ranking rule, but working directly on the *bipolar-valued* outranking digraph.

6.3 The *NetFlows* ranking

The valued version of the *Copeland* rule, called **NetFlows** rule, computes for each alternative x a *net flow* score, i.e. the sum of the differences between the **strict outranking** characteristics $r(x \succsim y)$ and the **strict outranked** characteristics $r(y \succsim x)$ for all pairs of alternatives where y is different from x.

Listing 6.7: Computing a *NetFlows* ranking

```
1  >>> from linearOrders import NetFlowsRanking
2  >>> nf = NetFlowsRanking(gcd,Comments=True)
3   Net Flows :
```

```
4   a5 : 3.600
5   a7 : 2.800
6   a6 : 1.300
7   a3 : 0.033
8   a1 : -0.400
9   a8 : -0.567
10  a4 : -1.283
11  a9 : -2.600
12  a2 : -2.883
13  NetFlows Ranking:
14  ['a5', 'a7', 'a6', 'a3', 'a1', 'a8', 'a4', 'a9', 'a2']
15  >>> cop.copelandRanking
16  ['a5', 'a1', 'a6', 'a7', 'a8', 'a4', 'a9', 'a3', 'a2']
```

It is worthwhile noticing again, that similar to the *Copeland* ranking rule seen before, the *NetFlows* ranking rule is also **invariant** under the *codual transform* (page 18) and delivers again the same ranking result indifferently from digraphs *g* or *gcd* (see Listing 6.7 Line 14).

In our example here, the *NetFlows* scores deliver a ranking *without ties* which is rather different from the one delivered by *Copeland*'s rule (see Listing 6.7 Line 16). It may happen, however, that we obtain, as with the *Copeland* scores above, only a ranking with ties, which may then be resolved again by following a lexicographic rule. In such cases, it is possible to construct again a *weak ranking* with the corresponding `transitiveDigraphs.WeakNetFlowsOrder` class.

The **NetFlows** ranking result appears to be slightly better correlated (+0.638) with the given outranking relation than its crisp cousin, the *Copeland* ranking (see Listing 6.4 Lines 4-6).

Listing 6.8: Checking the quality of the *NetFlows* Ranking

```
1   >>> corr = gcd.computeOrdinalCorrelation(nf)
2   >>> gcd.showCorrelation(corr)
3    Correlation indexes:
4     Extended Kendall tau      : +0.638
5     Epistemic determination   :  0.230
6     Bipolar-valued equivalence : +0.147
```

Indeed, the extended *Kendall* tau index of +0.638 leads to a bipolar-valued *relational equivalence* characteristics of +0.147, i.e. a *majority* of 57.35% of the criteria significance supports the relational equivalence between the given outranking digraphs *g* or *gcd* and the corresponding *NetFlows* ranking. This lesser ranking performance of the *Copeland* rule stems in this example essentially from the *weakness* of the actual ranking result and our subsequent *arbitrary* lexicographic resolution of the many ties given by the *Copeland* scores (see Fig. 2).

To appreciate now the more or less correlation of both the *Copeland* and the *NetFlows* rankings with the underlying pairwise outranking relation, it is useful to consider *Ke-*

meny's and *Slater*'s **best fitting** ranking rules.

6.4 *Kemeny* rankings

A **Kemeny** ranking is a linear ranking without ties which is *closest*, in the sense of the ordinal *Kendall* distance (see [BIS-2012]), to the given valued outranking digraphs g or *gcd*. This rule is also *invariant* under the *codual* transform.

Listing 6.9: Computing a *Kemeny* ranking

```
1  >>> from linearOrders import KemenyRanking
2  >>> ke = KemenyRanking(gcd,orderLimit=9) # default orderLimit is 7
3  >>> ke.showRanking()
4   ['a5', 'a6', 'a7', 'a3', 'a9', 'a4', 'a1', 'a8', 'a2']
5  >>> corr = gcd.computeOrdinalCorrelation(ke)
6  >>> gcd.showCorrelation(corr)
7   Correlation indexes:
8    Extended Kendall tau       : +0.779
9    Epistemic determination    :  0.230
10   Bipolar-valued equivalence : +0.179
```

So, **+0.779** represents the *highest possible* ordinal correlation (fitness) any potential linear ranking can achieve with the given pairwise outranking digraph (see Listing 6.9 Lines 7-10).

A *Kemeny* ranking may not be unique. In our example here, we obtain in fact two *Kemeny* rankings with a same **maximal** *Kemeny* index of 12.9.

Listing 6.10: Optimal *Kemeny* rankings

```
1  >>> ke.maximalRankings
2   [['a5', 'a6', 'a7', 'a3', 'a8', 'a9', 'a4', 'a1', 'a2'],
3    ['a5', 'a6', 'a7', 'a3', 'a9', 'a4', 'a1', 'a8', 'a2']]
4  >>> ke.maxKemenyIndex
5   Decimal('12.9166667')
```

We may visualize the partial order defined by the *epistemic disjunction* (page 17) of both optimal *Kemeny* rankings by using the `transitiveDigraphs.RankingsFusion` class as follows.

Listing 6.11: Computing the epistemic disjunction of all
optimal *Kemeny* rankings

```
1  >>> from transitiveDigraphs import RankingsFusion
2  >>> wke = RankingsFusion(ke,ke.maximalRankings)
3  >>> wke.exportGraphViz(fileName='tutorialKemeny')
4   *---- exporting a dot file for GraphViz tools ---------*
5   Exporting to tutorialKemeny.dot
6   0 { rank = same; a5; }
7   1 { rank = same; a6; }
8   2 { rank = same; a7; }
```

```
9   3 { rank = same; a3; }
10  4 { rank = same; a9; a8; }
11  5 { rank = same; a4; }
12  6 { rank = same; a1; }
13  7 { rank = same; a2; }
14  dot -Grankdir=TB -Tpng tutorialKemeny.dot -o tutorialKemeny.png
```
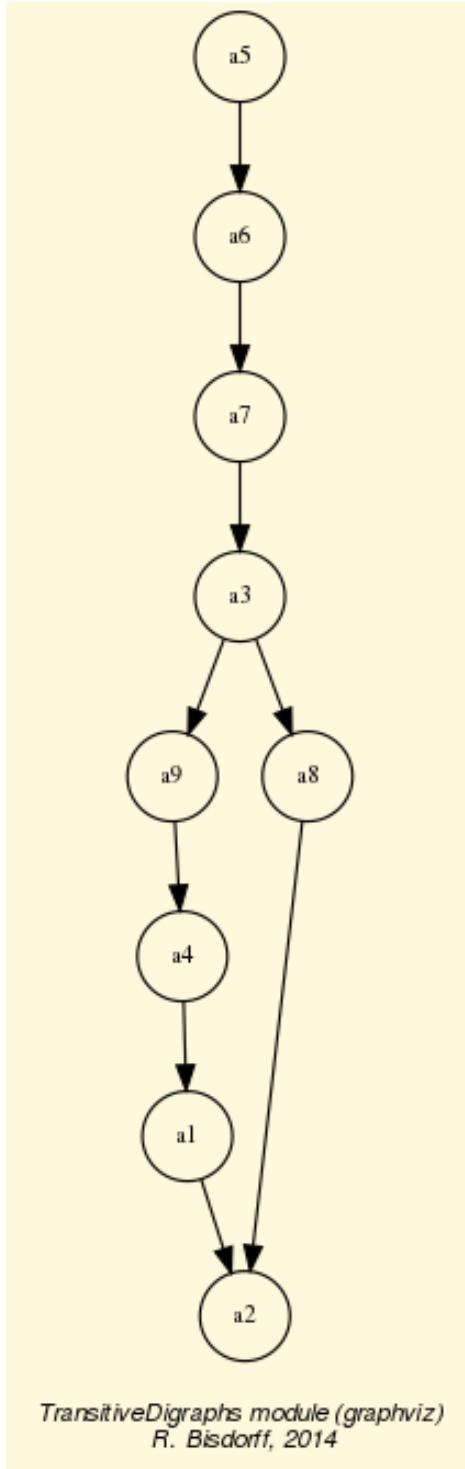

Fig. 3: Epistemic disjunction of optimal *Kemeny* rankings

It is interesting to notice in Fig. 3 and Listing 6.10, that both *Kemeny* rankings only differ in their respective positioning of alternative *a8*; either before or after alternatives *a9*, *a4* and *a1*.

To choose now a specific representative among all the potential rankings with maximal Kemeny index, we will choose, with the help of the `perfTabs.PerformanceTableau.` `showRankingConsensusQuality()` method, the *most consensual* one.

Listing 6.12: Computing Consensus Quality of Rankings

```
1   >>> g.showRankingConsensusQuality(ke.maximalRankings[0])
2    Consensus quality of ranking:
3     ['a5', 'a6', 'a7', 'a3', 'a8', 'a9', 'a4', 'a1', 'a2']
4    criterion (weight): correlation
5    -------------------------------
6     b09 (0.050): +0.361
7     b04 (0.050): +0.333
8     b08 (0.050): +0.292
9     b01 (0.050): +0.264
10    c01 (0.167): +0.250
11    b03 (0.050): +0.222
12    b07 (0.050): +0.194
13    b05 (0.050): +0.167
14    c02 (0.167): +0.000
15    b10 (0.050): +0.000
16    b02 (0.050): -0.042
17    b06 (0.050): -0.097
18    c03 (0.167): -0.167
19   Summary:
20    Weighted mean marginal correlation : +0.099
21    Standard deviation                 : +0.177
22   >>> g.showRankingConsensusQuality(ke.maximalRankings[1])
23    Consensus quality of ranking:
24     ['a5', 'a6', 'a7', 'a3', 'a9', 'a4', 'a1', 'a8', 'a2']
25    criterion (weight): correlation
26    -------------------------------
27     b09 (0.050): +0.306
28     b08 (0.050): +0.236
29     c01 (0.167): +0.194
30     b07 (0.050): +0.194
31     c02 (0.167): +0.167
32     b04 (0.050): +0.167
33     b03 (0.050): +0.167
34     b01 (0.050): +0.153
35     b05 (0.050): +0.056
36     b02 (0.050): +0.014
37     b06 (0.050): -0.042
38     c03 (0.167): -0.111
39     b10 (0.050): -0.111
40   Summary:
41    Weighted mean marginal correlation : +0.099
```

```
42      Standard deviation                   : 0.132
```

Both Kemeny rankings show the same *weighted mean marginal correlation* (+0.099, see Listing 6.12 Lines 19-21, 40-42) with all thirteen performance criteria. However, the second ranking shows a slightly lower *standard deviation* (+0.132 vs +0.177).

When several rankings with maximal Kemeny index are given, the `linearOrders.KemenyRanking` class constructor instantiates a *most consensual* one, i.e. a ranking with *highest* mean marginal correlation and, in case of ties, with *lowest* weighted standard deviation. Here we obtain ranking: ['a5', 'a6', 'a7', 'a3', 'a9', 'a4', 'a1', 'a8', 'a2'] (see Listing 6.9 Line 4).

6.5 *Slater* rankings

The **Slater** ranking rule is identical to *Kemeny*'s, but it is working, instead, on the *median cut polarised* digraph. *Slater*'s ranking rule is also *invariant* under the *codual* transform and delivers again indifferently on *g* or *gcd* the following results.

Listing 6.13: Computing a *Slater* ranking

```
1   >>> from linearOrders import SlaterRanking
2   >>> sl = SlaterRanking(gcd,orderLimit=9)
3   >>> sl.slaterRanking
4    ['a5', 'a6', 'a4', 'a1', 'a3', 'a7', 'a8', 'a9', 'a2']
5   >>> corr = gcd.computeOrderCorrelation(sl.slaterRanking)
6   >>> sl.showCorrelation(corr)
7    Correlation indexes:
8     Extended Kendall tau      : +0.676
9     Epistemic determination   :  0.230
10    Bipolar-valued equivalence : +0.156
11  >>> len(sl.maximalRankings)
12    7
```

We notice in Listing 6.13 Line 7 that the first *Slater* ranking is a rather good fit (+0.676), slightly better apparently than the *NetFlows* ranking result (+638). However, there are in fact 7 such potentially optimal *Slater* rankings (see Listing 6.13 Line 11). The corresponding *epistemic disjunction* (page 17) gives the following partial ordering.

Listing 6.14: Computing the epistemic disjunction of optimal *Slater* rankings

```
1   >>> slw = RankingsFusion(sl,sl.maximalRankings)
2   >>> slw.exportGraphViz(fileName='tutorialSlater')
3    *---- exporting a dot file for GraphViz tools ---------*
4    Exporting to tutorialSlater.dot
5    0 { rank = same; a5; }
6    1 { rank = same; a6; }
7    2 { rank = same; a7; a4; }
```

```
8    3 { rank = same; a1; }
9    4 { rank = same; a8; a3; }
10   5 { rank = same; a9; }
11   6 { rank = same; a2; }
12   dot -Grankdir=TB -Tpng tutorialSlater.dot -o tutorialSlater.png
```


TransitiveDigraphs module (graphviz)
R. Bisdorff, 2014
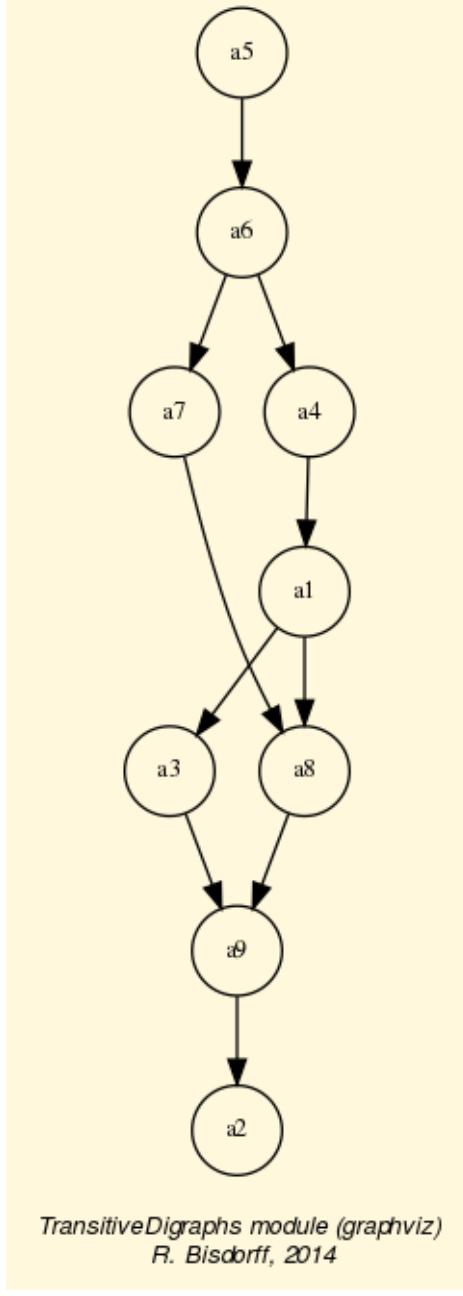
Fig. 4: Epistemic disjunction of optimal *Slater* rankings

What precise ranking result should we hence adopt ? *Kemeny*'s and *Slater*'s ranking rules are furthermore computationally *difficult* problems and effective ranking results are only computable for tiny outranking digraphs (< 20 objects).

More efficient ranking heuristics, like the *Copeland* and the *NetFlows* rules, are therefore

needed in practice. Let us finally, after these *ranking-by-scoring* strategies, also present two popular *ranking-by-choosing* strategies.

6.6 *Kohler*'s ranking-by-choosing rule

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

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

1. Compute for each row of the bipolar-valued *strict* outranking relation table (see Listing 6.1) 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 i;

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

Listing 6.15: Computing a *Kohler* ranking

```
1  >>> from linearOrders import KohlerRanking
2  >>> kocd = KohlerRanking(gcd)
3  >>> kocd.showRanking()
4   ['a5', 'a7', 'a6', 'a3', 'a9', 'a8', 'a4', 'a1', 'a2']
5  >>> corr = gcd.computeOrdinalCorrelation(kocd)
6  >>> gcd.showCorrelation(corr)
7   Correlation indexes:
8    Extended Kendall tau       : +0.747
9    Epistemic determination    :  0.230
10   Bipolar-valued equivalence : +0.172
```

With this *min-max* lexicographic *ranking-by-choosing* strategy, we find a correlation result (+0.747) that is until now clearly the nearest to an optimal *Kemeny* ranking (see Listing 6.10). Only two adjacent pairs: *[a6, a7]* and *[a8, a9]* are actually inverted here. Notice that *Kohler*'s ranking rule, contrary to the previously mentioned rules, is **not** *invariant* under the *codual* transform and requires to work on the *strict outranking* digraph *gcd* for a better correlation result.

```
1  >>> ko = KohlerRanking(g)
2  >>> corr = g.computeOrdinalCorrelation(ko)
3  >>> g.showCorrelation(corr)
4   Correlation indexes:
5    Crisp ordinal correlation  : +0.483
6    Epistemic determination    :  0.230
7    Bipolar-valued equivalence : +0.111
```

But *Kohler*'s ranking has a *dual* version, the prudent **Arrow-Raynaud** *ordering-by-choosing* rule, where a corresponding *max-min* strategy, when used on the *non-strict* outranking digraph *g*, for ordering the from *last* to *first* produces a similar ranking result (see [LAM-2009], [DIA-2010]).

Noticing that the *NetFlows* score of an alternative x represents in fact a bipolar-valued characteristic of the assertion '**alternative x is ranked first**', we may enhance *Kohler*'s or *Arrow-Raynaud*'s rules by replacing the *min-max*, respectively the *max-min*, strategy with an **iterated** maximal, respectively its *dual* minimal, *Netflows* score selection.

For a ranking (resp. an ordering) result, at step i (i goes from 1 to n) do the following:

1. Compute for each row of the bipolar-valued outranking relation table (see Listing 6.1) the corresponding *net flow score* (page 64) ;

2. Select the row where this score is maximal (resp. minimal); ties being resolved by lexicographic order;

3. Put the corresponding decision alternative at rank (resp. order) i;

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

A first *advantage* is that the so modified *Kohler*'s and *Arrow-Raynaud*'s rules become **invariant** under the *codual* transform. And we may get both the *ranking-by-choosing* as well as the *ordering-by-choosing* results with the `linearOrders`. `IteratedNetFlowsRanking` class constructor (see Listing 6.16 Lines 12-13).

Listing 6.16: Ordering-by-choosing with iterated minimal *NetFlows* scores

```
>>> from linearOrders import IteratedNetFlowsRanking
>>> inf = IteratedNetFlowsRanking(g)
>>> inf
*------- Digraph instance description ------*
Instance class       : IteratedNetFlowsRanking
Instance name        : rel_randomCBperftab_ranked
Digraph Order        : 9
Digraph Size         : 36
Valuation domain     : [-1.00;1.00]
Determinateness (%)  : 100.00
Attributes           : ['valuedRanks', 'valuedOrdering',
                         'iteratedNetFlowsRanking',
                         'iteratedNetFlowsOrdering',
                         'name', 'actions', 'order',
                         'valuationdomain', 'relation',
                         'gamma', 'notGamma']
>>> inf.iteratedNetFlowsOrdering
['a2', 'a9', 'a1', 'a4', 'a3', 'a8', 'a7', 'a6', 'a5']
>>> corr = g.computeOrderCorrelation(inf.iteratedNetFlowsOrdering)
>>> g.showCorrelation(corr)
 Correlation indexes:
  Crisp ordinal correlation  : +0.751
  Epistemic determination    :  0.230
  Bipolar-valued equivalence : +0.173
>>> inf.iteratedNetFlowsRanking
['a5', 'a7', 'a6', 'a3', 'a4', 'a1', 'a8', 'a9', 'a2']
>>> corr = g.computeRankingCorrelation(inf.iteratedNetFlowsRanking)
```

(continues on next page)

72

```
28   >>> g.showCorrelation(corr)
29    Correlation indexes:
30     Crisp ordinal correlation  : +0.743
31     Epistemic determination    :  0.230
32     Bipolar-valued equivalence : +0.171
```

The iterated *NetFlows* ranking and its *dual*, the iterated *NetFlows* ordering, do not usually deliver both the same result (Listing 6.16 Lines 18 and 26). With our example outranking digraph *g* for instance, it is the *ordering-by-choosing* result that obtains a slightly better correlation with the given outranking digraph *g* (+0.751), a result that is also slightly better than *Kohler*'s original result (+0.747, see Listing 6.15 Line 8).

With different *ranking-by-choosing* and *ordering-by-choosing* results, it may be useful to *fuse* now, similar to what we have done before with *Kemeny*'s and *Slaters*'s optimal rankings (see Listing 6.11 and Listing 6.14), both, the iterated *NetFlows* ranking and ordering into a partial ranking. But we are hence back to the practical problem of what linear ranking should we eventually retain ?

Let us finally mention another interesting *ranking-by-choosing* approach.

6.7 *Tideman*'s ranked-pairs rule

Tideman's *ranking-by-choosing* heuristic, the **RankedPairs** rule, working best this time on the non strict outranking digraph *g*, is based on a *prudent incremental* construction of linear orders that avoids on the fly any cycling outrankings (see [LAM-2009]). The ranking rule may be formulated as follows:

1. Rank the ordered pairs (x, y) of alternatives in decreasing order of $r(x \succsim y) + r(y \not\succsim x)$;

2. Consider the pairs in that order (ties are resolved by a lexicographic rule):

   - if the next pair does not create a *circuit* with the pairs already blocked, block this pair;

   - if the next pair creates a *circuit* with the already blocked pairs, skip it.

With our didactic outranking digraph *g*, we get the following result.

Listing 6.17: Computing a *RankedPairs* ranking

```
1  >>> from linearOrders import RankedPairsRanking
2  >>> rp = RankedPairsRanking(g)
3  >>> rp.showRanking()
4   ['a5', 'a6', 'a7', 'a3', 'a8', 'a9', 'a4', 'a1', 'a2']
```

The *RankedPairs* ranking rule renders in our example here luckily one of the two optimal *Kemeny* ranking, as we may verify below.

```
1   >>> ke.maximalRankings
2    [['a5', 'a6', 'a7', 'a3', 'a8', 'a9', 'a4', 'a1', 'a2'],
3     ['a5', 'a6', 'a7', 'a3', 'a9', 'a4', 'a1', 'a8', 'a2']]
4   >>> corr = g.computeOrdinalCorrelation(rp)
5   >>> g.computeCorrelation(corr)
6    Correlation indexes:
7     Extended Kendall tau       : +0.779
8     Epistemic determination    :  0.230
9     Bipolar-valued equivalence : +0.179
```

Similar to *Kohler*'s rule, the *RankedPairs* rule has also a prudent *dual* version, the **Dias-Lamboray** *ordering-by-choosing* rule, which produces, when working this time on the co-dual *strict outranking* digraph *gcd*, a similar ranking result (see [LAM-2009], [DIA-2010]).

Besides of not providing a unique linear ranking, the *ranking-by-choosing* rules, as well as their dual *ordering-by-choosing* rules, are unfortunately *not scalable* to outranking digraphs of larger orders (> 100). For such bigger outranking digraphs, with several hundred or thousands of alternatives, only the *Copeland*, the *NetFlows* ranking-by-scoring rules, with a polynomial complexity of $O(n^2)$, where n is the order of the outranking digraph, remain in fact computationally tractable.

Back to *Content Table* (page 3)

7 Computing a best choice recommendation

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

7.1 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** (see Table 1).

<div align="center">Table 1: The potential new office sites</div>

ID	Name	Address	Comment
A	Ave	Avenue de la liberté	High standing city center
B	Bon	Bonnevoie	Industrial environment
C	Ces	Cessange	Residential suburb location
D	Dom	Dommeldange	Industrial suburb environment
E	Bel	Esch-Belval	New and ambitious urbanization far from the city
F	Fen	Fentange	Out in the countryside
G	Gar	Avenue de la Gare	Main city 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 **coherent family of criteria**[26].

<div align="center">Table 2: The coherent family of performance criteria</div>

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**.

---

[26] A *coherent family* of performance criteria verifies: a) *Exhaustiveness*: No argument acceptable to all stakeholders can be put forward to justify a preference in favour of action x versus action y when x and y have the same performance level on each of the criteria of the family; b) *Cohesiveness*: Stakeholders unanimously recognize that action x must be preferred to action y whenever the performance level of x is significantly better than that of x on one of the criteria of positive weight, performance levels of x and y being the same on each of the other criteria; c) *Nonredundancy*: One of the above requirements is violated if one of the criteria is left out from the family. *Source*: European Working Group "*Multicriteria Aid for Decisions*" Series 3, no1, Spring, 2000.

Table 3: Performance evaluations of the potential office sites

Criterion	weight	A	B	C	D	E	F	G
Costs	45.0	35.0K€	17.8K€	6.7K€	14.1K€	34.8K€	18.6K€	12.0K€
Prox	32.0	100	20	80	70	40	0	60
Visi	26.0	60	80	70	50	60	0	100
Stan	23.0	100	10	0	30	90	70	20
Wksp	10.0	75	30	0	55	100	0	50
Wkcf	6.0	0	100	10	30	60	80	50
Park	3.0	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 in Table 3 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.

In Table 3 we may also see that the *Costs* criterion admits the highest significance (45.0), followed by the *Future turnover* criteria (32.0 + 26.0 + 23.0 = 81.0), The *Working conditions* criteria are the less significant (10.0 + 6.0, + 3.0 = 19.0). It follows that the CEO considers *maximizing the future turnover* the most important objective (81.0), followed by the *minizing yearly Costs* objective (45.0), and less important, the *maximizing working conditions* objective (19.0).

Concerning yearly costs, we suppose 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 Table 3, what is now the office site we may recommend to the CEO as **best choice** ?

7.2 Performance tableau

A Python encoded performance tableau is available for downloading here officeChoice.py.

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

```
1  >>> from perfTabs import *
2  >>> t = PerformanceTableau('officeChoice')
3  >>> help(t) # for discovering all the methods available
4  >>> t.showPerformanceTableau(Transposed=True)
5   *----  performance tableau -----*
6   criteria |   weights |     'A'        'B'        'C'        'D'        'E'        'F'␣
   ↪        'G'
7   ---------|-----------------------------------------------------------------------
   ↪------------
8   'C'      |     45    | -35000.00 -17800.00 -6700.00 -14100.00 -34800.00 -18600.
   ↪00 -12000.00
9   'Cf'     |      6    |     0.00    100.00    10.00     30.00     60.00     80.
   ↪00     50.00
10  'P'      |      3    |    90.00     30.00    100.00    90.00     70.00      0.
   ↪00     80.00
11  'Pr'     |     32    |   100.00     20.00    80.00     70.00     40.00      0.
   ↪00     60.00
12  'St'     |     23    |   100.00     10.00     0.00     30.00     90.00     70.
   ↪00     20.00
13  'V'      |     26    |    60.00     80.00    70.00     50.00     60.00      0.
   ↪00    100.00
14  'W'      |     10    |    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 `perfTabs.PerformanceTableau.showCriteria()` method.

```
1  >>> t.showCriteria(IntegerWeights=True)
2   *----  criteria -----*
3   C 'Costs'
4   Scale = (Decimal('0.00'), Decimal('50000.00'))
5   Weight = 45
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 = 6
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 on Fig. 1 by the **heatmap** 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,\
                                 rankingRule=None)
```

Heatmap of Performance Tableau 'officeChoice'

criteria	C	Pr	V	St	W	Cf	P
weights	+45.00	+32.00	+26.00	+23.00	+10.00	+6.00	+3.00
Ave	-35000.00	100.00	60.00	100.00	75.00	0.00	90.00
Bon	-17800.00	20.00	80.00	10.00	30.00	100.00	30.00
Ces	-6700.00	80.00	70.00	0.00	0.00	10.00	100.00
Dom	-14100.00	70.00	50.00	30.00	55.00	30.00	90.00
Bel	-34800.00	40.00	60.00	90.00	100.00	60.00	70.00
Fen	-18600.00	0.00	0.00	70.00	0.00	80.00	0.00
Gar	-12000.00	60.00	100.00	20.00	50.00	50.00	80.00

Color legend:

quantile	20.00%	40.00%	60.00%	80.00%	100.00%

Fig. 1: Unranked heatmap of the office choice performance tableau

Site *Ave* 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 *Ces*: 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 *Dom* and *Gar*. Finally, site *Fen* is clearly the less satisfactory alternative of all.

7.3 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 [-1.00, 1.00], 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 (see [BIS-2004]).

Valued Adjacency Matrix

r(x S y)	A	B	C	D	E	F	G
A	–	0.00	1.00	0.30	0.78	0.00	0.00
B	0.00	–	0.00	-0.56	0.00	1.00	-0.60
C	0.00	0.00	–	0.46	0.00	1.00	0.10
D	0.10	0.56	0.02	–	0.46	1.00	0.25
E	0.52	0.00	0.00	-0.10	–	1.00	-0.42
F	0.00	-1.00	-1.00	-1.00	-1.00	–	-1.00
G	0.00	0.92	-0.10	1.00	0.54	1.00	–

Valuation domain: [-1.00; +1.00]

Fig. 2: The office choice outranking digraph

For computing such a bipolar-valued outranking digraph from the given performance tableau t, we use the `BipolarOutrankingDigraph` constructor from the outrankingDigraphs module. The `Digraph.showHTMLRelationTable` method shows here the resulting bipolar-valued adjacency matrix in a system browser window (see Fig. 2).

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

In Fig. 2 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 (see Fig. 3).

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

Rubis Python Server (graphviz), R. Bisdorff, 2008
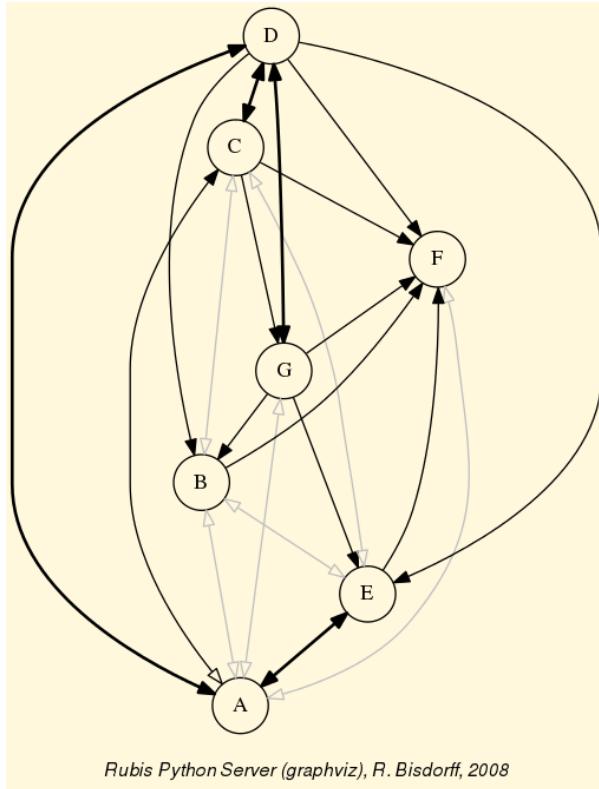
Fig. 3: The office choice outranking digraph

One may check that the outranking digraph g does not admit in fact any cyclic strict preference situation.

```
1  >>> g.computeChordlessCircuits()
2   []
3  >>> g.showChordlessCircuits()
4   No circuits observed in this digraph.
5   *---- Chordless circuits ----*
6   0 circuits.
```

7.4 *Rubis* best choice recommendations

Following the Rubis outranking method (see [BIS-2008]), potential best choice recommendations are determined by the outranking prekernels –*weakly independent* and *strictly outranking* choices– of the outranking digraph (see the tutorial on *On computing digraph kernels* (page 163)). The case given, we previously need to break open all chordless odd circuits at their weakest link.

```
1  >>> from digraphs import BrokenCocsDigraph
2  >>> bcg = BrokenCocsDigraph(g)
3  >>> bcg.brokenLinks
4   set()
```

80

As we observe indeed no such chordless circuits here, we may directly compute the *prek-ernels* of the outranking digraph *g*.

Listing 7.1: Computing outranking and outranked prek-ernels

```
1  >>> g.showPreKernels()
2   *--- Computing preKernels ---*
3   Dominant preKernels :
4   ['D']
5      independence :  1.0
6      dominance    :  0.02
7      absorbency   :  -1.0
8      covering     :  1.000
9   ['B', 'E', 'C']
10     independence :  0.00
11     dominance    :  0.10
12     absorbency   :  -1.0
13     covering     :  0.500
14  ['A', 'G']
15     independence :  0.00
16     dominance    :  0.78
17     absorbency   :  0.00
18     covering     :  0.700
19  Absorbent preKernels :
20  ['F', 'A']
21     independence :  0.00
22     dominance    :  0.00
23     absorbency   :  1.0
24     covering     :  0.700
25  *----- statistics -----
26  graph name:  rel_officeChoice.xml
27  number of solutions
28   dominant kernels :  3
29   absorbent kernels:  1
30  cardinality frequency distributions
31  cardinality     :  [0, 1, 2, 3, 4, 5, 6, 7]
32  dominant kernel :  [0, 1, 1, 1, 0, 0, 0, 0]
33  absorbent kernel:  [0, 0, 1, 0, 0, 0, 0, 0]
34  Execution time  : 0.00018 sec.
35  Results in sets: dompreKernels and abspreKernels.
```

We notice in Listing 7.1 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 best choice recommendation is now given by the **most determined** prekernel; the one supported by the most significant criteria coalition. This result is shown with the `showBestChoiceRecommendation` command. Notice that this method actually works by default on the broken chords digraph *bcg*.

Listing 7.2: Computing a best choice recommendation

```
1  >>> g.showBestChoiceRecommendation(CoDual=False)
2  ******************************************
3  Rubis best choice recommendation(s) (BCR)
4   (in decreasing order of determinateness)
5  Credibility domain: [-1.00,1.00]
6  === >> potential best choice(s)
7  * choice              : ['D']
8    independence        : 1.00
9    dominance           : 0.02
10   absorbency          : -1.00
11   covering (%)        : 100.00
12   determinateness (%) : 51.03
13   - most credible action(s) = { 'D': 0.02, }
14  === >> potential best choice(s)
15  * choice              : ['A', 'G']
16   independence        : 0.00
17   dominance           : 0.78
18   absorbency          : 0.00
19   covering (%)        : 70.00
20   determinateness (%) : 50.00
21   - most credible action(s) = { }
22  === >> potential best choice(s)
23  * choice              : ['B', 'C', 'E']
24   independence        : 0.00
25   dominance           : 0.10
26   absorbency          : -1.00
27   covering (%)        : 50.00
28   determinateness (%) : 50.00
29   - most credible action(s) = { }
30  === >> potential worst choice(s)
31  * choice              : ['A', 'F']
32   independence        : 0.00
33   dominance           : 0.00
34   absorbency          : 1.00
35   covered (%)         : 70.00
36   determinateness (%) : 50.00
37   - most credible action(s) = { }
38  Execution time: 0.014 seconds
```

We notice in Listing 7.2 (Line 7) above that the most significantly supported best choice recommendation is indeed the *Condorcet* winner D supported by a majority of 51.03% of the criteria significance (see Line 12). Both other potential best choice recommendations, as well as the potential worst choice recommendation, are not positively validated as best, resp. worst 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.

7.5 Computing *strict best* choice recommendations

When comparing now the performances of alternatives D and G on a pairwise perspective (see below), 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) = $+145/145 = +1.0$.

```
1  >>> g.showPairwiseComparison('G','D')
2   *------------  pairwise comparison ----*
3   Comparing actions : (G, D)
4   crit. wght.  g(x)       g(y)    diff.  |   ind      pref     concord   |
5   ============================================================================
6   C    45.00 -12000.00 -14100.00 +2100.00 | 1000.00 2500.00    +45.00    |
7   Cf    6.00     50.00     30.00   +20.00 |   10.00   20.00     +6.00    |
8   P     3.00     80.00     90.00   -10.00 |   10.00   20.00     +3.00    |
9   Pr   32.00     60.00     70.00   -10.00 |   10.00   20.00    +32.00    |
10  St   23.00     20.00     30.00   -10.00 |   10.00   20.00    +23.00    |
11  V    26.00    100.00     50.00   +50.00 |   10.00   20.00    +26.00    |
12  W    10.00     50.00     55.00    -5.00 |   10.00   20.00    +10.00    |
13   ============================================================================
14   Valuation in range: -145.00 to +145.00; global concordance: +145.00
```

However, we must as well notice that the cheapest alternative C is in fact **strictly outranking** alternative G: r(C outranks G) = $+15/145 > 0.0$, and r(G outranks C) = $-15/145 < 0.0$.

```
1  >>> g.showPairwiseComparison('C','G')
2   *------------  pairwise comparison ----*
3   Comparing actions : (C, G)/(G, C)
4   crit. wght.  g(x)      g(y)       diff.  |   ind.   pref.     (C,G)/(G,C)  |
5   ============================================================================
6   C    45.00 -6700.00 -12000.00  +5300.00 | 1000.00 2500.00  +45.00/-45.00 |
7   Cf    6.00    10.00     50.00    -40.00 |   10.00   20.00   -6.00/ +6.00 |
8   P     3.00   100.00     80.00    +20.00 |   10.00   20.00   +3.00/ -3.00 |
9   Pr   32.00    80.00     60.00    +20.00 |   10.00   20.00  +32.00/-32.00 |
10  St   23.00     0.00     20.00    -20.00 |   10.00   20.00  -23.00/+23.00 |
11  V    26.00    70.00    100.00    -30.00 |   10.00   20.00  -26.00/+26.00 |
12  W    10.00     0.00     50.00    -50.00 |   10.00   20.00  -10.00/+10.00 |
13   ============================================================================
14   Valuation in range: -145.00 to +145.00; global concordance: +15.00/-15.00
```

To model these *strict outranking* situations, we may recompute the best choice recommendation on the **codual**, the converse (~) of the dual (-)[14], of the outranking digraph instance g (see [BIS-2013]), as follows.

Listing 7.3: Computing the strict best choice recommendation

```
1  >>> g.showBestChoiceRecommendation(CoDual=True,\
2                                     ChoiceVector=True)
```

(continues on next page)

```
3    * --- Best and worst choice recommendation(s) ---*
4     (in decreasing order of determinateness)
5    Credibility domain: [-1.00,1.00]
6    === >> potential best choice(s)
7    * choice              : ['A', 'C', 'D']
8      independence        : 0.00
9      dominance           : 0.10
10     absorbency          : 0.00
11     covering (%)        : 41.67
12     determinateness (%) : 50.59
13     - characteristic vector = { 'D': 0.02, 'G': 0.00, 'C': 0.00,
14                                 'A': 0.00, 'F': -0.02, 'E': -0.02,
15                                 'B': -0.02, }
16   === >> potential worst choice(s)
17   * choice              : ['A', 'F']
18     independence        : 0.00
19     dominance           : -0.52
20     absorbency          : 1.00
21     covered (%)         : 50.00
22     determinateness (%) : 50.00
23     - characteristic vector = { 'G': 0.00, 'F': 0.00, 'E': 0.00,
24                                 'D': 0.00, 'C': 0.00, 'B': 0.00,
25                                 'A': 0.00, }
```

It is interesting to notice in Listing 7.3 (Line 6) that the **strict best choice recommendation** consists in the set of weak Condorcet winners: 'A', 'C' and 'D'. 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-2006a], [BIS-2006b]), 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 an outranking prekernel, shows as well an indeterminate situation with respect to **being or not being** 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 as **potential worst choice recommendation** . This confirms again the global incomparability status of alternative A.

7.6 Weakly ordering the outranking digraph

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

```
1  >>> from transitiveDigraphs import RankingByChoosingDigraph
2  >>> gcd = ~(-g)
3  >>> rbc = RankingByChoosingDigraph(gcd)
4   Threading ...  ## multiprocessing if 2 cores are available
5   Exiting computing threads
6  >>> rbc.showRankingByChoosing()
7   Ranking by Choosing and Rejecting
8   1st ranked ['D']
9      2nd ranked ['C', 'G']
10     2nd last ranked ['B', 'C', 'E']
11  1st last ranked ['A', 'F']
12 >>> rbc.exportGraphViz('officeChoiceRanking')
13  *---- exporting a dot file for GraphViz tools ---------*
14  Exporting to officeChoiceRanking.dot
15  0 { rank = same; A; C; D; }
16  1 { rank = same; G; }
17  2 { rank = same; E; B; }
18  3 { rank = same; F; }
19  dot -Grankdir=TB -Tpng officeChoiceRanking.dot -o officeChoiceRanking.png
```


weakOrders module (graphviz)
R. Bisdorff, 2014
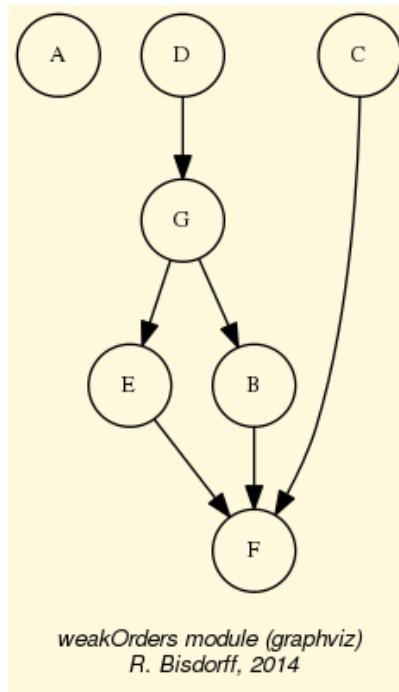
Fig. 4: Ranking-by-choosing from the office choice outranking digraph

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 decision objectives he is considering. In the setting here, where he considers that *maximizing the future turnover* is the most important objective followed by *minimizing the Costs* and, less important, *maximizing the working conditions*, site D represents actually the best compromise. However, if *Costs* do not play much a 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 these criteria significance weights in the `officeChoice.py` data file in such a way that

- all criteria under an objective appear *equi-significant*, and

- all three decision objectives are considered *equally important*.

What will become the best choice recommendation under this working hypothesis?

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

8 Alice's best choice: A case study[19]

Alice D. , 19 years old German student finishing her secondary studies in Köln (Germany), desires to undertake foreign languages studies. She will probably receive her "Abitur" with satisfactory and/or good marks and wants to start her further studies thereafter.

She would not mind staying in Köln, yet is ready to move elsewhere if necessary. The length of the higher studies do concern her, as she wants to earn her life as soon as

---

[19] This case study is inspired by a *Multiple Criteria Decision Analysis* case study published in Eisenführ Fr., Langer Th., and Weber M., *Fallstudien zu rationalem Entscheiden*, Springer 2001, pp. 1-17.

possible. Her parents however agree to financially support her study fees, as well as, her living costs during her studies.

8.1 The decision problem

Alice has already identified 10 **potential study programs**.

Table 1: Alice's potential study programs

ID	Diploma	Institution	City
T-UD	Qualified translator (T)	University (UD)	Düsseldorf
T-FHK	Qualified translator (T)	Higher Technical School (FHK)	Köln
T-FHM	Qualified translator (T)	Higher Technical School (FHM)	München
I-FHK	Graduate interpreter (I)	Higher Technical School (FHK)	Köln
T-USB	Qualified translator (T)	University (USB)	Saarbrücken
I-USB	Graduate interpreter (I)	University (USB)	Saarbrücken
T-UHB	Qualified translator (T)	University (UHB)	Heidelberg
I-UHB	Graduate interpreter (I)	University (UHB)	Heidelberg
S-HKK	Specialized secretary (S)	Chamber of Commerce (HKK)	Köln
C-HKK	Foreign correspondent (C)	Chamber of Commerce (HKK)	Köln

In Table 1 we notice that Alice considers three *Graduate Interpreter* studies (8 or 9 Semesters), respectively in Köln, in Saarbrücken or in Heidelberg; and five *Qualified translator* studies (8 or 9 Semesters), respectively in Köln, in Düsseldorf, in Saarbrücken, in Heidelberg or in Munich. She also considers two short (4 Semesters) study programs at the Chamber of Commerce in Köln.

Four **decision objectives** of more or less equal importance are guiding Alice's choice:

1. *maximize* the attractiveness of the study place (GEO),

2. *maximize* the attractiveness of her further studies (LEA),

3. *minimize* her financial dependency on her parents (FIN),

4. *maximize* her professional perspectives (PRA).

The decision consequences Alice wishes to take into account for evaluating the potential study programs with respect to each of the four objectives are modelled by the following **coherent family of criteria**[26].

Table 2: Alice's family of performance criteria

ID	Name	Comment	Objective	Weight
DH	Proximity	Distance in km to her home (min)	GEO	3
BC	Big City	Number of inhabitants (max)	GEO	3
AS	Studies	Attractiveness of the studies (max)	LEA	6
SF	Fees	Annual study fees (min)	FIN	2
LC	Living	Monthly living costs (min)	FIN	2
SL	Length	Length of the studies (min)	FIN	2
AP	Profession	Attractiveness of the profession (max)	PRA	2
AI	Income	Annual income after studying (max)	PRA	2
PR	Prestige	Occupational prestige (max)	PRA	2

Within each decision objective, the performance criteria are considered to be equisignificant. Hence, the four decision objectives show a same importance weight of 6 (see Table 2).

8.2 The performance tableau

The actual evaluations of Alice's potential study programs are stored in a file named AliceChoice.py of `perfTabs.PerformanceTableau` format[21].

Listing 8.1: Alice's performance tableau

```
1  >>> from perfTabs import PerformanceTableau
2  >>> t = PerformanceTableau('AliceChoice')
3  >>> t.showObjectives()
4    *------ decision objectives -------"
5    GEO: Geographical aspect
6      DH Distance to parent's home 3
7      BC Number of inhabitants     3
8      Total weight: 6 (2 criteria)
9    LEA: Learning aspect
10     AS Attractiveness of the study program 6
11     Total weight: 6.00 (1 criteria)
12   FIN: Financial aspect
13     SF Annual registration fees 2
14     LC Monthly living costs     2
15     SL Study time               2
16     Total weight: 6.00 (3 criteria)
17   PRA: Professional aspect
```

---

[21] Alice's performance tableau `AliceChoice.py` is available in the `examples` directory of the Digraph3 software collection.

```
18     AP Attractiveness of the profession        2
19     AI Annual professional income after studying 2
20     OP Occupational Prestige                   2
21     Total weight: 6.00 (3 criteria)
```

Details of the performance criteria may be consulted in a browser view (see Fig. 1 below).

```
>>> t.showHTMLCriteria()
```

AliceChoice: Family of Criteria

#	Identifyer	Name	Comment	Weight	Scale			Thresholds (ax + b)		
					direction	min	max	indifference	preference	veto
1	AI	Annual professional income after studying	Professional aspect measured in x / 1000 Euros	2.00	max	0.00	50.00	0.00x + 0.00	0.00x + 1.00	
2	AP	Attractiveness of the profession	Professional aspect subjectively measured on a three-level scale: 0 (weak), 1 (fair), 2 (good)	2.00	max	0.00	2.00	0.00x + 0.00	0.00x + 1.00	
3	AS	Attractiveness of the study program	Learning aspect subjectively measured from 0 (weak) to 10 (excellent)	6.00	max	0.00	10.00	0.00x + 0.00	0.00x + 1.00	0.00x + 7.00
4	BC	Number of inhabitants	Geographical aspect: measured in x / 1000	3.00	max	0.00	2000.00	0.01x + 0.00	0.05x + 0.00	
5	DH	Distance to parent's home	Geographical aspect measured in km	3.00	min	0.00	1000.00	0.00x + 0.00	0.00x + 10.00	
6	LC	Monthly living costs	Financial aspect measured in Euros	2.00	min	0.00	1000.00	0.00x + 0.00	0.00x + 100.00	
7	OP	Occupational Prestige	Professional aspect measured in SIOPS points	2.00	max	0.00	100.00	0.00x + 0.00	0.00x + 10.00	
8	SF	Annual registration fees	Financial aspect measured in Euros	2.00	min	400.00	4000.00	0.00x + 0.00	0.00x + 100.00	
9	SL	study time	Financial aspect measured in number of semesters	2.00	min	0.00	10.00	0.00x + 0.00	0.00x + 0.50	

Fig. 1: Alice's performance criteria

It is worthwhile noticing in Fig. 1 above that, on her subjective attractiveness scale of the study programs (criterion AS), Alice considers a performance differences of 7 points to be *considerable* and triggering, the case given, a *veto situation*. Notice also the proportional *indifference* (1%) and *preference* (5%) discrimination thresholds shown on criterion BC-number of inhabitants.

In the following *heatmap view*, we may now consult Alice's performance evaluations.

```
>>> t.showHTMLPerformanceHeatmap(
        colorLevels=5,Correlations=True,ndigits=0)
```

Heatmap of Performance Tableau 'AliceChoice'

criteria	AS	AP	SF	OP	AI	DH	LC	BC	SL
weights	+6.00	+2.00	+2.00	+2.00	+2.00	+3.00	+2.00	+3.00	+2.00
tau$^{(*)}$	+0.71	+0.64	+0.36	+0.36	+0.24	+0.03	-0.04	-0.07	-0.24
I-FHK	8	2	-400	62	35	0	0	1015	-8
I-USB	8	2	-400	62	45	-269	-1000	196	-9
T-FHK	5	1	-400	62	35	0	0	1015	-8
I-UHB	8	2	-400	62	45	-275	-1000	140	-9
T-UD	5	1	-400	62	45	-41	-1000	567	-9
T-USB	5	1	-400	62	45	-260	-1000	196	-9
T-FHM	4	1	-400	62	35	-631	-1000	1241	-8
T-UHB	5	1	-400	62	45	-275	-1000	140	-9
C-HKK	2	0	-4000	44	30	0	0	1015	-4
S-HKK	1	0	-4000	44	30	0	0	1015	-4

Color legend:

quantile	20.00%	40.00%	60.00%	80.00%	100.00%

(*) tau: *Ordinal (Kendall) correlation between marginal criterion and global ranking relation*
Ranking rule: **NetFlows**
Ordinal (Kendall) correlation between global ranking and global outranking relation: **+0.692**

Fig. 2: Heatmap of Alice's performance tableau

Alice is subjectively evaluating the *Attractiveness* of the studies (criterion *AS*) on an ordinal scale from 0 (weak) to 10 (excellent). Similarly, she is subjectively evaluating the *Attractiveness* of the respective professions (criterion *AP*) on a three level ordinal scale from 0 (*weak*), 1 (*fair*) to 2 (*good*). Considering the *Occupational Prestige* (criterion *OP*), she looked up the SIOPS[20]. All the other evaluation data she found on the internet (see Fig. 2).

Notice by the way that evaluations on performance criteria to be *minimized*, like *Distance to Home* (criterion *DH*) or *Study time* (criterion *SL*), are registered as *negative* values, so that smaller measures are, in this case, preferred to larger ones.

Her ten potential study programs are ordered with the *NetFlows* ranking rule applied to the corresponding bipolar-valued outranking digraph[23]. *Graduate interpreter* studies in Köln (*I-FHK*) or Saarbrücken (*I-USB*), followed by *Qualified Translator* studies in Köln (*T-FHK*) appear to be Alice's most preferred alternatives. The least attractive study programs for her appear to be studies at the Chamber of Commerce of Köln (*C-HKK*, *S-HKK*).

It is finally interesting to observe in Fig. 2 (third row) that the *most significant* performance criteria, appear to be for Alice, on the one side, the *Attractiveness* of the study program (criterion *AS*, tau = +0.72) followed by the *Attractiveness* of the future profession (criterion *AP*, tau = +0.62). On the other side, *Study times* (criterion *SL*, tau =

---

[20] Ganzeboom H.B.G, Treiman D.J. "Internationally Comparable Measures of Occupational Status for the 1988 International Standard Classification of Occupations", *Social Science Research* 25, 201–239 (1996).

[23] See the tutorial on *ranking with multiple incommensurable criteria* (page 59).

-0.24), *Big city* (criterion *BC*, tau = -0.07) as well as *Monthly living costs* (criterion *LC*, tau = -0.04) appear to be for her *not so* significant[27].

8.3 Building a best choice recommendation

Let us now have a look at the resulting pairwise outranking situations.

Listing 8.2: Alice's outranking digraph

```
1  >>> from outrankingDigraphs import BipolarOutrankingDigraph
2  >>> dg = BipolarOutrankingDigraph(t)
3  >>> dg
4   *------- Object instance description ------*
5   Instance class      : BipolarOutrankingDigraph
6   Instance name       : rel_AliceChoice
7   # Actions           : 10
8   # Criteria          : 9
9   Size                : 67
10  Determinateness (%) : 73.91
11  Valuation domain    : [-1.00;1.00]
12 >>> dg.computeSymmetryDegree(Comments=True)
13  Symmetry degree of graph <rel_AliceChoice> : 0.49
```

From Alice's performance tableau we obtain 67 positively validated pairwise outranking situations in the digraph *dg*, supported by a 74% majority of criteria significance (see Listing 8.2 Line 9-10).

Due to the poorly discriminating performance evaluations, nearly half of these outranking situations (see Line 12) are *symmetric* and reveal actually *more or less indifference* situations between the potential study programs. This is well illustrated in the **relation map** of the outranking digraph (see Fig. 3).

```
>>> dg.showHTMLRelationMap(tableTitle='Outranking relation map',\
                           rankingRule='Copeland')
```

---

[27] See also the corresponding Advanced Topic in the Digraph3 documentation.

Outranking relation map

Ranking rule: Copeland

r(x S y)	I-FHK	I-USB	I-UHB	T-FHK	T-UD	T-USB	T-UHB	T-FHM	C-HKK	S-HKK
I-FHK		·	·	+	·	·	·	·	·	+
I-USB	·		+	·	·	·	+	·	·	+
I-UHB	·	·		·	·	·	+	·	·	+
T-FHK	·	·	·		·	·	·	·	·	·
T-UD	-	·	·	·		+	+	·	·	·
T-USB	-	·	·	·	·		+	·	·	·
T-UHB	-	-	·	·	·	·		·	·	·
T-FHM	-	-	-	·	·	·	·		·	·
C-HKK	-	-	-	-	-	-	-	-		+
S-HKK	—	—	—	-	-	-	-	-	·	

Semantics	
+	certainly valid
·	valid
	indeterminate
-	invalid
—	certainly invalid

Fig. 3: 'Copeland'-ranked outranking relation map

We have mentioned that Alice considers a performance difference of 7 points on the *Attractiveness of studies* criterion AS to be considerable which triggers, the case given, a potential polarisation of the outranking characteristics. In Fig. 3 above, these polarisations appear in the last column and last row. We may inspect the occurrence of such polarisations as follows.

Listing 8.3: Veto and counter-veto situations

```
>>> dg.showVetos()
 *----  Veto situations ---
 number of veto situations : 3
 1: r(S-HKK >= I-FHK) = -0.17
  criterion: AS
  Considerable performance difference : -7.00
  Veto discrimination threshold       : -7.00
  Polarisation: r(S-HKK >= I-FHK) = -0.17 ==> -1.00
 2: r(S-HKK >= I-USB) = -0.17
  criterion: AS
  Considerable performance difference : -7.00
  Veto discrimination threshold       : -7.00
  Polarisation: r(S-HKK >= I-USB) = -0.17 ==> -1.00
 3: r(S-HKK >= I-UHB) = -0.17
  criterion: AS
  Considerable performance difference : -7.00
```

```
17    Veto discrimination threshold       : -7.00
18    Polarisation: r(S-HKK >= I-UHB) = -0.17 ==> -1.00
19   *----  Counter-veto situations ---
20   number of counter-veto situations : 3
21   1: r(I-FHK >= S-HKK) = 0.83
22    criterion: AS
23    Considerable performance difference : 7.00
24    Counter-veto threshold              : 7.00
25    Polarisation: r(I-FHK >= S-HKK) = 0.83 ==> +1.00
26   2: r(I-USB >= S-HKK) = 0.17
27    criterion: AS
28    Considerable performance difference : 7.00
29    Counter-veto threshold              : 7.00
30    Polarisation: r(I-USB >= S-HKK) = 0.17 ==> +1.00
31   3: r(I-UHB >= S-HKK) = 0.17
32    criterion: AS
33    Considerable performance difference : 7.00
34    Counter-veto threshold              : 7.00
35    Polarisation: r(I-UHB >= S-HKK) = 0.17 ==> +1.00
```

In Listing 8.3, we see that *considerable performance differences* concerning the *Attractiveness of the studies* (*AS* criterion) are indeed observed between the *Specialised Secretary* study programm offered in Köln and the *Graduate Interpreter* study programs offered in Köln, Saarbrücken and Heidelberg. They polarise, hence, three *more or less invalid* outranking situations to *certainly invalid* (Lines 9, 14, 19) and corresponding three *more or less valid* converse outranking situations to *certainly valid* ones (Lines 25, 30, 35).

We may finally notice in the relation map, shown in Fig. 3, that the four best-ranked study programs, *I-FHK*, *I-USB*, *I-UHB* and *T-FHK*, are in fact *Condorcet* winners (see Listing 8.4 Line 2), i.e. they are all four *indifferent* one of the other **and** positively *outrank* all other alternatives, a result confirmed below by our best choice recommendation (Line 8).

Listing 8.4: Alice's best choice recommendation

```
1   >>> dg.computeCondorcetWinners()
2    ['I-FHK', 'I-UHB', 'I-USB', 'T-FHK']
3   >>> dg.showBestChoiceRecommendation()
4    Best choice recommendation(s) (BCR)
5    (in decreasing order of determinateness)
6    Credibility domain: [-1.00,1.00]
7    === >> potential best choice(s)
8    choice            : ['I-FHK','I-UHB','I-USB','T-FHK']
9     independence     : 0.17
10    dominance        : 0.08
11    absorbency       : -0.83
12    covering (%)     : 62.50
13    determinateness (%) : 68.75
14    most credible action(s) = {'I-FHK': 0.75,'T-FHK': 0.17,
```

```
15                          'I-USB': 0.17,'I-UHB': 0.17}
16   === >> potential worst choice(s)
17   choice              : ['C-HKK', 'S-HKK']
18    independence       : 0.50
19    dominance          : -0.83
20    absorbency         : 0.17
21    covered (%)        : 100.00
22    determinateness (%) : 58.33
23    most credible action(s) = {'S-HKK': 0.17,'C-HKK': 0.17}
```

Most credible best choice among the four best-ranked study programs eventually becomes the *Graduate Interpreter* study program at the *Technical High School* in *Köln* (see Listing 8.4 Line 14) supported by a $(0.75 + 1)/2.0 = 87.5\%$ (18/24) majority of global criteria significance[24].

In the relation map, shown in Fig. 3, we see in the left lower corner that the *asymmetric part* of the outranking relation, i.e. the corresponding *strict* outranking relation, is actually *transitive* (see Listing 8.5 Line 2). Hence, a graphviz drawing of its *skeleton*, oriented by the previous *best*, respectively *worst* choice, may well illustrate our *best choice recommendation*.

Listing 8.5: Drawing the best choice recommendation

```
1   >>> dgcd = ~(-dg)
2   >>> dgcd.isTransitive()
3    True
4   >>> dgcd.closeTransitive(Reverse=True)
5   >>> dgcd.exportGraphViz('aliceBestChoice',
6           bestChoice=['I-FHK'],
7           worstChoice=['S-HKK','C-HKK'])
8    *---- exporting a dot file for GraphViz tools ---------*
9     Exporting to aliceBestChoice.dot
10    dot -Grankdir=BT -Tpng aliceBestChoice.dot -o aliceBestChoice.png
```

---

[24] See also the Advanced Topic about computing best choice membership characteristics in the Digraph3 documentation.
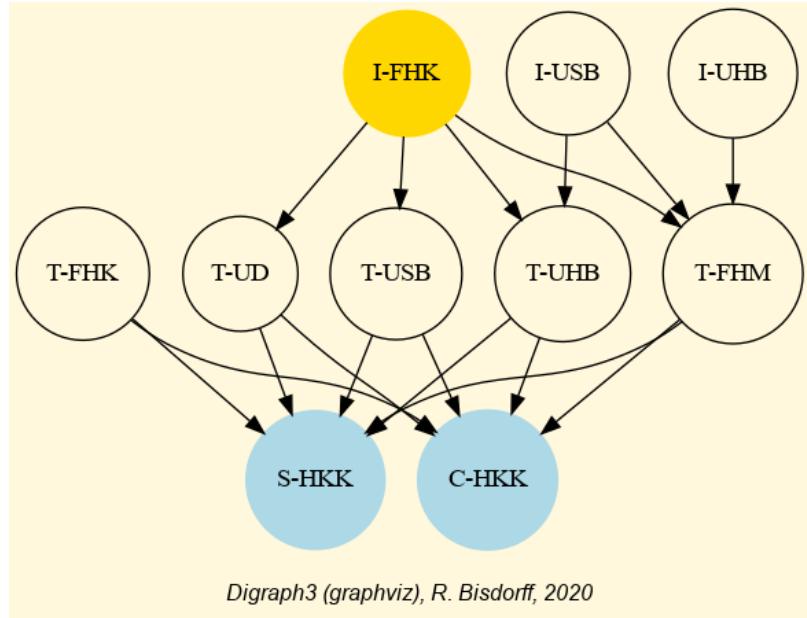
Fig. 4: Alice's best choice recommendation

In Fig. 4 we notice that the *Graduate Interpreter* studies come first, followed by the *Qualified Translator* studies. Last come the *Chamber of Commerce*'s specialised studies. This confirms again the high significance that Alice attaches to the *attractiveness* of her further studies and of her future profession (see criteria *AS* and *AP* in Fig. 2).

Let us now, for instance, check the pairwise outranking situations observed between the first and second-ranked alternative, i.e. *Garduate Interpreter* studies in *Köln* versus *Graduate Interpreter* studies in *Saabrücken* (see *I-FHK* and *I-USB* in Fig. 2).

```
>>> dg.showHTMLPairwiseOutrankings('I-FHK','I-USB')
```

Pairwise Comparison

Comparing actions : (I-FHK,I-USB)

crit.	wght.	g(x)	g(y)	diff	ind	pref	concord	v	polarisation
AI	2.00	35.00	+45.00	-10	0.00	1.00	-2.00		
AP	2.00	2.00	+2.00	0	0.00	1.00	+2.00		
AS	6.00	8.00	+8.00	0	0.00	1.00	+6.00		
BC	3.00	1015.00	+196.00	819	10.15	50.75	+3.00		
DH	3.00	0.00	-269.00	269	0.00	10.00	+3.00		
LC	2.00	0.00	-1000.00	1000	0.00	100.00	+2.00		
OP	2.00	62.00	+62.00	0	0.00	10.00	+2.00		
SF	2.00	-400.00	-400.00	0	0.00	100.00	+2.00		
SL	2.00	-8.00	-9.00	1	0.00	0.50	+2.00		

Valuation in range: -24.00 to +24.00; global concordance: +20.00

Pairwise Comparison

Comparing actions : (I-USB,I-FHK)

crit.	wght.	g(x)	g(y)	diff	ind	pref	concord	v	polarisation
AI	2.00	45.00	+35.00	10	0.00	1.00	+2.00		
AP	2.00	2.00	+2.00	0	0.00	1.00	+2.00		
AS	6.00	8.00	+8.00	0	0.00	1.00	+6.00		
BC	3.00	196.00	+1015.00	-819	10.15	50.75	-3.00		
DH	3.00	-269.00	+0.00	-269	0.00	10.00	-3.00		
LC	2.00	-1000.00	+0.00	-1000	0.00	100.00	-2.00		
OP	2.00	62.00	+62.00	0	0.00	10.00	+2.00		
SF	2.00	-400.00	-400.00	0	0.00	100.00	+2.00		
SL	2.00	-9.00	-8.00	-1	0.00	0.50	-2.00		

Valuation in range: -24.00 to +24.00; global concordance: +4.00

Fig. 5: Comparing the first and second best-ranked study programs

The *Köln* alternative is performing **at least as well as** the *Saarbrücken* alternative on all the performance criteria, except the *Annual income* (of significance 2/24). Conversely, the *Saarbrücken* alternative is clearly **outperformed** from the *geographical* (0/6) as well as from the *financial* perspective (2/6).

In a similar way, we may finally compute a *weak ranking* of all the potential study programs with the help of the `transitiveDigraphs.RankingByChoosingDigraph` constructor (see Listing 8.6 below), who computes a bipolar ranking by conjointly *best-choosing* and *last-rejecting* [BIS-1999].

```
1  >>> from transitiveDigraphs import\
2                RankingByChoosingDigraph
3  >>> rbc = RankingByChoosingDigraph(dg)
4  >>> rbc.showRankingByChoosing()
5   Ranking by Choosing and Rejecting
6    1st ranked ['I-FHK']
7      2nd ranked ['I-USB']
8        3rd ranked ['I-UHB']
9          4th ranked ['T-FHK']
10            5th ranked ['T-UD']
11            5th last ranked ['T-UD']
12          4th last ranked ['T-UHB', 'T-USB']
13        3rd last ranked ['T-FHM']
14      2nd last ranked ['C-HKK']
15    1st last ranked ['S-HKK']
```

In Listing 8.6, we find confirmed that the *Interpreter* studies appear all preferrred to the *Translator* studies. Furthermore, the *Interpreter* studies in *Saarbrücken* appear preferred to the same studies in *Heidelberg*. The *Köln* alternative is apparently the preferred one of all the *Translater* studies. And, the *Foreign Correspondent* and the *Specialised Secretary* studies appear second-last and last ranked.

Yet, how *robust* are our findings with respect to potential settings of the decision objectives' importance and the performance criteria significance ?

8.4 Robustness analysis

Alice considers her four decision objectives as being *more or less* equally important. Here we have, however, allocated *strictly equal* importance weights with *strictly* equi-significant criteria per objective. How robust is our previous best choice recommendation when, now, we would consider the importance of the objectives and, hence, the significance of the respective performance criteria to be *more or less uncertain* ?

To answer this question, we will consider the respective criteria significance weights wj to be **triangular random variables** in the range 0 to *2wj* with *mode = wj*. We may compute a corresponding **90%-confident outranking digraph** with the help of the outrankingDigraphs.ConfidentBipolarOutrankingDigraph constructor[22].

Listing 8.7: The 90% confident outranking digraph

```
1  >>> from outrankingDigraphs import\
2          ConfidentBipolarOutrankingDigraph
3  >>> cdg = ConfidentBipolarOutrankingDigraph(t,\
4                distribution='triangular',confidence=90.0)
```

---

[22] See also the corresponding Advanced Topic in the Digraph3 documentation.

```
5   >>> cdg
6    *------- Object instance description ------*
7    Instance class       : ConfidentBipolarOutrankingDigraph
8    Instance name        : rel_AliceChoice_CLT
9    # Actions            : 10
10   # Criteria           : 9
11   Size                 : 44
12   Valuation domain     : [-1.00;1.00]
13   Uncertainty model    : triangular(a=0,b=2w)
14   Likelihood domain    : [-1.0;+1.0]
15   Confidence level     : 90.0%
16   Confident majority   : 14/24 (58.3%)
17   Determinateness (%)  : 68.19
```

Of the original 67 valid outranking situations, we retain 44 outranking situations as being 90%-*confident* (see Listing 8.7 Line 10). The corresponding 90%-*confident* **qualified majority** of criteria significance amounts to $14/24 = 58.3\%$ (Line 15).

Concerning now a 90%-*confident* best choice recommendation, we are lucky (see Listing 8.8 below).

Listing 8.8: The 90% confident best choice recommendation

```
1   >>> cdg.computeCondorcetWinners()
2    ['I-FHK']
3   >>> cdg.showBestChoiceRecommendation()
4    ***********************
5    Best choice recommendation(s) (BCR)
6     (in decreasing order of determinateness)
7     Credibility domain: [-1.00,1.00]
8     === >> potential best choice(s)
9     choice               : ['I-FHK','I-UHB','I-USB',
10                             'T-FHK','T-FHM']
11     independence        : 0.00
12     dominance           : 0.42
13     absorbency          : 0.00
14     covering (%)        : 20.00
15     determinateness (%) : 61.25
16     - most credible action(s) = { 'I-FHK': 0.75, }
```

The *Graduate Interpreter* studies in Köln remain indeed a 90%-confident *Condorcet* winner (Line 2). Hence, the same study program also remains our 90%-confident most credible best choice supported by a continual 18/24 (87.5%) majority of the global criteria significance (see Lines 9 and 15).

When previously comparing the two best-ranked study programs (see Fig. 5), we have observed that *I-FHK* actually positively outranks *I-USB* on all four decision objectives. When admitting equi-significant criteria significances per objective, this outranking situation is hence valid independently of the importance weights Alice may allocate to each

of her decision objectives.

We may compute these **unopposed** outranking situations[25] with help of the `outrankingDigraphs.UnOpposedBipolarOutrankingDigraph` constructor.

Listing 8.9: Computing the unopposed outranking situations

```
1  >>> from outrankingDigraphs import UnOpposedBipolarOutrankingDigraph
2  >>> uop = UnOpposedBipolarOutrankingDigraph(t)
3  >>> uop
4   *------- Object instance description ------*
5    Instance class       : UnOpposedBipolarOutrankingDigraph
6    Instance name        : AliceChoice_unopposed_outrankings
7    # Actions            : 10
8    # Criteria           : 9
9    Size                 : 28
10   Oppositeness (%)     : 58.21
11   Determinateness (%)  : 62.94
12   Valuation domain     : [-1.00;1.00]
13  >>> uop.isTransitivity()
14   True
```

We keep 28 out the 67 standard outranking situations, which leads to an **oppositeness degree** of (1.0 - 28/67) = 58.21% (Listing 8.9 Line 10). Remarkable furthermore is that this unopposed outranking digraph *uop* is actually *transitive*, i.e. modelling a *partial ranking* of the study programs (Line 14).

We may hence make use of the `exportGraphViz` method of the `transitiveDigraphs.TransitiveDigraph` class for drawing the corresponding partial ranking.

```
>>> from transitiveDigraphs import TransitiveDigraph
>>> TransitiveDigraph.exportGraphViz(uop,\
                            'AliceChoice_unopposed')
 *---- exporting a dot file for GraphViz tools ---------*
  Exporting to AliceChoice_unopposed.dot
 dot -Grankdir=TB -Tpng AliceChoice_unopposed.dot\
                -o AliceChoice_unopposed.png
```

---

[25] See also the corresponding Advanced Topic in the Digraph3 documentation.
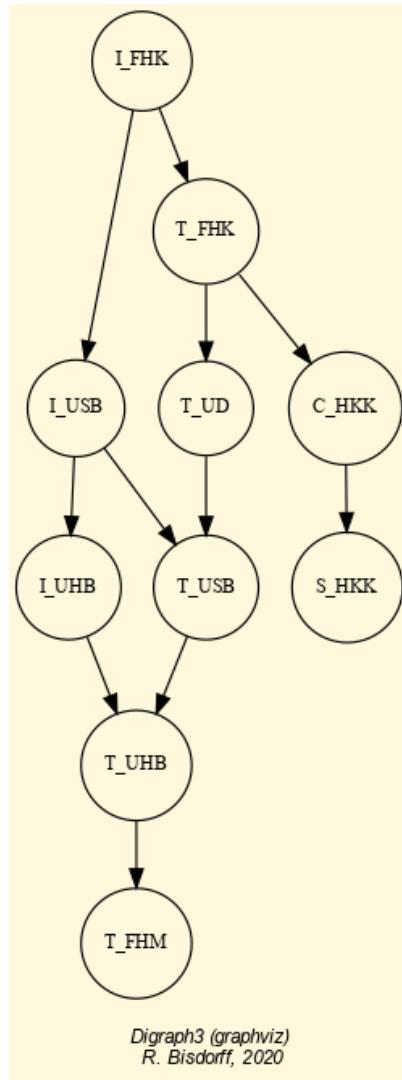
Fig. 6: Unopposed partial ranking of the potential study programs

Again, when *equi-signficant* performance criteria are assumed per decision objective, we observe in Fig. 6 that *I-FHK* remains the stable best choice, *independently* of the actual importance weights that Alice may wish to allocate to her four decision objectives.

In view of her performance tableau in Fig. 2, *Graduate Interpreter* studies at the *Technical High School Köln*, thus, represent definitely **Alice's very best choice**.

For further reading about the *Rubis* Best Choice methodology, one may consult in [BIS-2015] the study of a *real decision aid case* about choosing a best poster in a scientific conference.

Back to *Content Table* (page 3)

_____

9 Rating with multiple incommensurable criteria

- *K-sorting on a single criterion* (page 101)
- *Quantiles sorting with multiple criteria* (page 102)
- *Ranking big performance tableaux* (page 106)

We apply order statistics for sorting a set X of n potential decision actions, evaluated on m incommensurable performance criteria, into q quantile equivalence classes, based on pairwise outranking characteristics involving the quantile class limits observed on each criterion. Thus we may implement a weak ordering algorithm of complexity $O(nmq)$.

9.1 K-sorting on a single criterion

A single criterion sorting category K is a (usually) lower-closed interval $[m_k; M_k[$ on a real-valued performance measurement scale, with $m_k \leq M_k$. If x is a measured performance on this scale, we may distinguish three sorting situations.

1. $x < m_k$ and ($x < M_k$): The performance x is lower than category K.

2. $x \geqslant m_k$ and $x < M_k$: The performance x belongs to category K.

3. $x > m_k$ and $x \geqslant M_k$: The performance x is higher than category K.

As the relation $<$ is the dual of $\geqslant$ ($\not\geqslant$), it will be sufficient to check that $x \geqslant m_k$ as well as $x \not\geqslant M_k$ are true for x to be considered a member of category K.

Upper-closed categories (in a more mathematical integration style) may as well be considered. In this case it is sufficient to check that $m_k \not\geqslant x$ as well as $M_k \geq x$ are true for x to be considered a member of category K. It is worthwhile noticing that a category K such that $m_k = M_k$ is hence always empty by definition. In order to be able to properly sort over the complete range of values to be sorted, we will need to use a special, two-sided closed last, respectively first, category.

Let $K = K_1, ..., K_q$ be a non trivial partition of the criterion's performance measurement scale into $q \geq 2$ ordered categories K_k – i.e. lower-closed intervals $[m_k; M_k[$ – such that $m_k < M_k$, $M_k = m_{k+1}$ for $k = 0, \ldots, q$ - 1 and $M_q = \infty$. And, let $A = \{a_1, a_2, a_3, ...\}$ be a finite set of not all equal performance measures observed on the scale in question.

Property: For all performance measure $x \in A$ there exists now a unique k such that $x \in K_k$. If we assimilate, like in descriptive statistics, all the measures gathered in a category K_k to the central value of the category – i.e. $(m_k + M_k)/2$ – the sorting result will hence define a weak order (complete preorder) on A.

Let $Q = \{Q_0, Q_1, ..., Q_q\}$ denote the set of $q + 1$ increasing order-statistical quantiles –like quartiles or deciles– we may compute from the ordered set A of performance measures observed on a performance scale. If $Q_0 = \min(X)$, we may, with the following intervals: $[Q_0; Q_1[$, $[Q_1; Q_2[$, $\ldots$, $[Q_{q-1}; \infty[$, hence define a set of q lower-closed sorting categories. And, in the case of upper-closed categories, if $Q_q = \max(X)$, we would obtain the intervals

$]-\infty; Q_1],]Q_1; Q_2], \ldots,]Q_{q-1}; Q_q]$. The corresponding sorting of A will result, in both cases, in a repartition of all measures x into the q quantile categories K_k for $k = 1, \ldots, q$.

Example: Let $A = \{ a_7 = 7.03, a_{15} = 9.45, a_{11} = 20.35, a_{16} = 25.94, a_{10} = 31.44, a_9 = 34.48, a_{12} = 34.50, a_{13} = 35.61, a_{14} = 36.54, a_{19} = 42.83, a_5 = 50.04, a_2 = 59.85, a_{17} = 61.35, a_{18} = 61.61, a_3 = 76.91, a_6 = 91.39, a_1 = 91.79, a_4 = 96.52, a_8 = 96.56, a_{20} = 98.42 \}$ be a set of 20 increasing performance measures observed on a given criterion. The lower-closed category limits we obtain with quartiles ($q = 4$) are: $Q_0 = 7.03 = a_7$, $Q_1 = 34.485$, $Q_2 = 54.945$ (median performance), and $Q_3 = 91.69$. And the sorting into these four categories defines on A a complete preorder with the following four equivalence classes: $K_1 = \{a_7, a_{10}, a_{11}, a_{10}, a_{15}, a_{16}\}$, $K_2 = \{a_5, a_9, a_{13}, a_{14}, a_{19}\}$, $K_3 = \{a_2, a_3, a_6, a_{17}, a_{18}\}$, and $K_4 = \{a_1, a_4, a_8, a_{20}\}$.

9.2 Quantiles sorting with multiple criteria

Let us now suppose that we are given a performance tableau with a set X of n decision alternatives evaluated on a coherent family of m performance criteria associated with the corresponding outranking relation $\succsim$ defined on X. We denote x_j the performance of alternative x observed on criterion j.

Suppose furthermore that we want to sort the decision alternatives into q upper-closed quantile equivalence classes. We therefore consider a series : $k = k/q$ for $k = 0, \ldots, q$ of $q+1$ equally spaced quantiles, like quartiles: 0, 0.25, 0.5, 0.75, 1; quintiles: 0, 0.2, 0.4, 0.6, 0.8, 1: or deciles: 0, 0.1, 0.2, \ldots, 0.9, 1, for instance.

The upper-closed $\mathbf{q}^k$ class corresponds to the m quantile intervals $]q_j(p_{k-1}); q_j(p_k)]$ observed on each criterion j, where $k = 2, \ldots, q$, $q_j(p_q) = \max_X(x_j)$, and the first class gathers all performances below or equal to $Q_j(p_1)$.

The lower-closed $\mathbf{q}_k$ class corresponds to the m quantile intervals $[q_j(p_{k-1}); q_j(p_k)[$ observed on each criterion j, where $k = 1, \ldots, q\text{-}1$, $q_j(p_0) = \min_X(x_j)$, and the last class gathers all performances above or equal to $Q_j(p_{q-1})$.

We call **q-tiles** a complete series of $k = 1, \ldots, q$ upper-closed $\mathbf{q}^k$, respectively lower-closed $\mathbf{q}_k$, multiple criteria quantile classes.

Property: With the help of the bipolar-valued characteristic of the outranking relation $r(\succsim)$ we may compute the bipolar-valued characteristic of the assertion: x belongs to upper-closed q-tiles class $\mathbf{q}^k$ class, resp. lower-closed class $\mathbf{q}_k$, as follows.

$$r(x \in \mathbf{q}^k) \;=\; \min\left[\, - r\big(\mathbf{q}(p_{q-1}) \succsim x\big),\, r\big(\mathbf{q}(p_q) \succsim x\big)\right]$$

$$r(x \in \mathbf{q}_k) \;=\; \min\left[r\big(x \succsim \mathbf{q}(p_{q-1})\big),\, -r\big(x \succsim \mathbf{q}(p_q)\big)\right]$$

The outranking relation $\succsim$ verifying the coduality principle, $-r\big(\mathbf{q}(p_{q-1}) \succsim x\big) = r\big(\mathbf{q}(p_{q-1}) \prec x\big)$, resp. $-r\big(x \succsim \mathbf{q}(p_q)\big) = r\big(x \prec \mathbf{q}(p_q)\big)$.

We may compute, for instance, a five-tiling of a given random performance tableau with the help of the `sortingDigraphs.QuantilesSortingDigraph` class.

Listing 9.1: Computing a quintiles sorting result

```
1  >>> from randomPerfTabs import *
2  >>> t = RandomPerformanceTableau(numberOfActions=50,seed=5)
3  >>> from sortingDigraphs import QuantilesSortingDigraph
4  >>> qs = QuantilesSortingDigraph(t,limitingQuantiles=5)
5  >>> qs
6   *-----  Object instance description -----------*
7    Instance class   : QuantilesSortingDigraph
8    Instance name    : sorting_with_5-tile_limits
9    # Actions        : 50
10   # Criteria       : 7
11   # Categories      : 5
12   Lowerclosed       : False
13   Size              : 841
14   Valuation domain  : [-100.00;100.00]
15   Determinateness (%) : 81.39
16   Attributes          : ['actions', 'actionsOrig',
17        'criteria', 'evaluation', 'runTimes', 'name',
18        'limitingQuantiles', 'LowerClosed',
19        'categories', 'criteriaCategoryLimits',
20        'profiles', 'profileLimits', 'hasNoVeto',
21        'valuationdomain', 'nbrThreads', 'relation',
22        'categoryContent', 'order', 'gamma', 'notGamma']
23   *------  Constructor run times (in sec.) ------*
24   # Threads         : 1
25   Total time        : 0.03120
26   Data input        : 0.00300
27   Compute profiles  : 0.00075
28   Compute relation  : 0.02581
29   Weak Ordering     : 0.00052
30  >>> qs.showCriteriaCategoryLimits()
31   Quantile Class Limits (q = 5)
32   Upper-closed classes
33   crit.       0.20    0.40    0.60    0.80    1.00
34   *-------------------------------------------
35    g1         31.35   41.09   58.53   71.91   98.08
36    g2         27.81   39.19   49.87   61.66   96.18
37    g3         25.10   34.78   49.45   63.97   92.59
38    g4         24.61   37.91   53.91   71.02   89.84
39    g5         26.94   36.43   52.16   72.52   96.25
40    g6         23.94   44.06   54.92   67.34   95.97
41    g7         30.94   47.40   55.46   69.04   97.10
42  >>> qs.showSorting()
43  *--- Sorting results in descending order ---*
44   ]0.80 - 1.00]: ['a22']
45   ]0.60 - 0.80]: ['a03', 'a07', 'a08', 'a11', 'a14', 'a17',
46                   'a19', 'a20', 'a29', 'a32', 'a33', 'a37',
47                   'a39', 'a41', 'a42', 'a49']
48   ]0.40 - 0.60]: ['a01', 'a02', 'a04', 'a05', 'a06', 'a08',
```

```
49                    'a09', 'a16', 'a17', 'a18', 'a19', 'a21',
50                    'a24', 'a27', 'a28', 'a30', 'a31', 'a35',
51                    'a36', 'a40', 'a43', 'a46', 'a47', 'a48',
52                    'a49', 'a50']
53    ]0.20 - 0.40]: ['a04', 'a10', 'a12', 'a13', 'a15', 'a23',
54                    'a25', 'a26', 'a34', 'a38', 'a43', 'a44',
55                    'a45', 'a49']
56    ]  < - 0.20]: ['a44']
```

Most of the decision actions (26) are gathered in the median quintile $]0.40 - 0.60]$ class, whereas the highest quintile $]0.80 - 1.00]$ and the lowest quintile $] < -0.20]$ classes gather each one a unique decision alternative (*a22*, resp. *a44*) (see Listing 9.1 Lines 43-).

We may inspect as follows the details of the corresponding sorting characteristics.

Listing 9.2: Bipolar-valued sorting characteristics (extract)

```
1  >>> qs.valuationdomain
2   {'min': Decimal('-100.0'), 'med': Decimal('0'),
3    'max': Decimal('100.0')}
4  >>> qs.showSortingCharacteristics()
5   x  in  q^k          r(q^k-1 < x)   r(q^k >= x)   r(x in q^k)
6   a22 in ]< - 0.20]       100.00        -85.71        -85.71
7   a22 in ]0.20 - 0.40]     85.71        -71.43        -71.43
8   a22 in ]0.40 - 0.60]     71.43        -71.43        -71.43
9   a22 in ]0.60 - 0.80]     71.43        -14.29        -14.29
10  a22 in ]0.80 - 1.00]     14.29        100.00         14.29
11  ...
12  ...
13  a44 in ]< - 0.20]       100.00          0.00          0.00
14  a44 in ]0.20 - 0.40]      0.00         57.14          0.00
15  a44 in ]0.40 - 0.60]    -57.14         85.71        -57.14
16  a44 in ]0.60 - 0.80]    -85.71         85.71        -85.71
17  a44 in ]0.80 - 1.00]    -85.71         85.71        -85.71
18  ...
19  ...
20  a49 in ]< - 0.20]       100.00        -42.86        -42.86
21  a49 in ]0.20 - 0.40]     42.86          0.00          0.00
22  a49 in ]0.40 - 0.60]      0.00          0.00          0.00
23  a49 in ]0.60 - 0.80]      0.00         57.14          0.00
24  a49 in ]0.80 - 1.00]    -57.14         85.71        -57.14
```

Alternative *a22* verifies indeed positively both sorting conditions only for the highest quintile $[0.80 - 1.00]$ class (see Listing 9.2 Lines 10). Whereas alternatives *a44* and *a49*, for instance, weakly verify both sorting conditions each one for two, resp. three, adjacent quintile classes (see Lines 13-14 and 21-23).

Quantiles sorting results indeed always verify the following **Properties**.

1. **Coherence**: Each object is sorted into a non-empty subset of *adjacent* q-tiles

classes. An alternative that would *miss* evaluations on all the criteria will be sorted conjointly in all q-tiled classes.

2. **Uniqueness**: If $r(x \in \mathbf{q}^k) \neq 0$ for $k = 1, \ldots, q$, then performance x is sorted into *exactly one single* q-tiled class.

3. **Separability**: Computing the sorting result for performance x is independent from the computing of the other performances' sorting results. This property gives access to efficient parallel processing of class membership characteristics.

The *q-tiles* sorting result leaves us hence with more or less *overlapping* ordered quantile equivalence classes. For constructing now a linearly ranked q-tiles partition of X, we may apply three strategies:

1. **Average** (default): In decreasing lexicographic order of the average of the lower and upper quantile limits and the upper quantile class limit;

2. **Optimistic**: In decreasing lexicographic order of the upper and lower quantile class limits;

3. **Pessimistic**: In decreasing lexicographic order of the lower and upper quantile class limits;

Listing 9.3: Weakly ranking the quintiles sorting result

```
1  >>> qs.showQuantileOrdering(strategy='average')
2   ]0.80-1.00] : ['a22']
3   ]0.60-0.80] : ['a03', 'a07', 'a11', 'a14', 'a20', 'a29',
4                   'a32', 'a33', 'a37', 'a39', 'a41', 'a42']
5   ]0.40-0.80] : ['a08', 'a17', 'a19']
6   ]0.20-0.80] : ['a49']
7   ]0.40-0.60] : ['a01', 'a02', 'a05', 'a06', 'a09', 'a16',
8                   'a18', 'a21', 'a24', 'a27', 'a28', 'a30',
9                   'a31', 'a35', 'a36', 'a40', 'a46', 'a47',
10                  'a48', 'a50']
11  ]0.20-0.60] : ['a04', 'a43']
12  ]0.20-0.40] : ['a10', 'a12', 'a13', 'a15', 'a23', 'a25',
13                  'a26', 'a34', 'a38', 'a45']
14  ]  < -0.40] : ['a44']
```

following, for instance, the *average* ranking strategy, we find confirmed in the weak ranking shown in Listing 9.3, that alternative *a49* is indeed sorted into three adjacent quintiles classes, namely $]0.20 - 0.80]$ (see Line 6) and precedes the $]0.40 - 0.60]$ class, of same average of lower and upper limits.

Noticing the computational efficiency of the quantiles sorting construction (see Listing 9.1 Lines 23-29), coupled with the separability property of the quantile class membership characteristics computation, we will make usage of the `sortingDigraphs`. `QuantilesSortingDigraph` class for ranking big performance tableaux.

9.3 Ranking big performance tableaux

Indeed, none of the usual ranking heuristics (see previous tutorial), using essentially only the information given by the pairwise outranking characteristics, 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** using the quantiles sorting technique seen in the previous Section. This decomposition will lead to a *pre-ranked sparse* outranking digraph model.

In the coding example in Listing 9.4, we generate for instance, first (Lines 2-4), a cost benefit performance tableau of 100 decision alternatives and, secondly (Lines 5-6), we construct a `sparseOutrankingDigraphs.PreRankedOutrankingDigraph` instance called *bg*. Notice by the way the *BigData* flag (Line 4) used here for generating a parsimoniously commented performance tableau.

Listing 9.4: Computing a *pre-ranked* outranking digraph

```
>>> from sparseOutrankingDigraphs import\
                PreRankedOutrankingDigraph
>>> tp = RandomCBPerformanceTableau(numberOfActions=100,\
                BigData=True,seed=100)
>>> bg = PreRankedOutrankingDigraph(tp,quantiles=10,\
                LowerClosed=False,\
                componentRankingRule='NetFlows')
>>> bg
*----- Object instance description ------*
Instance class    : PreRankedOutrankingDigraph
Instance name     : randomCBperftab_pr
# Actions         : 100
# Criteria        : 7
Sorting by        : 10-Tiling
Ordering strategy : average
Ranking rule      : NetFlows
# Components      : 20
Minimal order     : 1
Maximal order     : 20
Average order     : 5.0
fill rate         : 11.475%
----  Constructor run times (in sec.) ----
#Threads          : 1
Total time        : 0.14232
Data imput        : 0.00230
QuantilesSorting  : 0.06648
Preordering       : 0.00032
Decomposing       : 0.06890
Ordering          : 0.00603
```

106

The total run time of the `sparseOutrankingDigraphs.PreRankedOutrankingDigraph` constructor is about 0.14 sec (see Listing 9.4 Line 24). The deciles sorting, preordering and decomposing leads to 20 linearly ordered quantiles equivalence classes. The corresponding pre-ranked decomposition may be visualized as follows.

Listing 9.5: The quantiles decomposition of a pre-ranked outranking digraph

```
1  >>> bg.showDecomposition()
2   *--- quantiles decomposition in decreasing order---*
3   c01. ]0.70-0.90] : [46, 67, 100]
4   c02. ]0.70-0.80] : [16, 19, 42, 56, 66, 79, 86, 87]
5   c03. ]0.50-0.80] : [39]
6   c04. ]0.60-0.70] : [13, 36, 41, 68, 69, 80, 85, 94]
7   c05. ]0.40-0.80] : [49]
8   c06. ]0.50-0.70] : [14, 20, 45]
9   c07. ]0.40-0.70] : [43]
10  c08. ]0.50-0.60] : [3, 9, 21, 23, 27, 35, 37, 38, 50, 54, 55,
11                      60, 72, 73, 74, 78, 82, 88, 92, 97]
12  c09. ]0.30-0.70] : [70]
13  c10. ]0.40-0.60] : [24]
14  c11. ]0.20-0.70] : [6]
15  c12. ]0.30-0.60] : [33]
16  c13. ]0.40-0.50] : [7, 8, 15, 25, 30, 32, 44, 48, 52, 57,
17                      58, 61, 64, 71, 77, 81, 84, 89, 91, 98]
18  c14. ]0.20-0.60] : [1]
19  c15. ]0.30-0.50] : [2, 17, 62, 93]
20  c16. ]0.30-0.40] : [5, 18, 22, 26, 28, 29, 31, 34, 47, 51,
21                      76, 83, 90, 95]
22  c17. ]0.20-0.40] : [63, 96]
23  c18. ]0.20-0.30] : [11, 12, 40, 53, 59, 65, 75, 99]
24  c19. ]0.10-0.30] : [10]
25  c20. ]0.10-0.20] : [4]
```

The best deciles class (]70%-90%]) gathers decision alternatives *46*, *67*, and *100*. Worst decile (]10%-20%]) gathers alternative *4* (see Listing 9.5 Lines 3 and 25).

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 stored in a *bg.boostedOrder* attribute. A reversed linear ranking (from the best to the worst) is stored in a *bg.boostedRanking* attribute.

Listing 9.6: Showing the componentwise *NetFlows* ranking

```
1  >>> bg.boostedRanking
2   [100, 67, 46, 16, 79, 87, 86, 56, 42, 66, 19, 39, 13, 94,
3     85, 69, 80, 36, 68, 41, 49, 14, 45, 20, 43, 55, 50, 92,
4     23, 97, 54, 21, 74, 78, 35, 9, 38, 88, 82, 3, 27, 37,
5     60, 73, 72, 70, 24, 6, 33, 58, 25, 15, 48, 30, 89, 77,
```

(continues on next page)

107

```
6        52, 7, 32, 98, 61, 57, 71, 81, 91, 64, 84, 44, 8, 1, 2,
7        93, 62, 17, 83, 26, 28, 90, 47, 31, 18, 29, 22, 76, 95,
8        51, 34, 5, 63, 96, 59, 40, 65, 75, 99, 11, 53, 12, 10, 4]
```

Alternative *100* appears *first ranked*, whereas alternative *4* is *last ranked* (see Listing 9.6 Line 2 and 8). The quality of this ranking result may be assessed by first, computing its ordinal correlation with the corresponding standard outranking relation; And, secondly, by showing the fairness of the ranking consensus.

Listing 9.7: Quality of the componentwise *NetFlows*'s ranking result

```
1   >>> g = BipolarOutrankingDigraph(tp,Normalized=True)
2   >>> corr = g.computeRankingCorrelation(bg.boostedRanking)
3   >>> g.showCorrelation(corr)
4    Correlation indexes:
5     Extended Kendall tau       : +0.685
6     Epistemic determination    :  0.344
7     Bipolar-valued equivalence : +0.235
8   >>> g.showRankingConsensusQuality(bg.boostedRanking)
9    criterion (weight): correlation
10   ------------------------------
11    c1 (0.167): +0.238
12    c3 (0.167): +0.227
13    b2 (0.125): +0.221
14    b4 (0.125): +0.181
15    c2 (0.167): +0.160
16    b1 (0.125): +0.145
17    b3 (0.125): +0.139
18   Summary:
19    Weighted mean marginal correlation (a): +0.190
20    Standard deviation (b)                 : +0.039
21    Ranking fairness (a)-(b)               : +0.152
```

The *NetFlows* as well as the *Copeland* ranking heuristics are readily scalable with ad hoc HPC tuning to several millions of decision alternatives (see [BIS-2016]).

Back to *Content Table* (page 3)

10 Rating with learned quantile norms

10.1 Introduction

In this tutorial we address the problem of **rating multiple criteria performances** of a set of potential decision alternatives with respect to empirical order statistics, i.e. performance quantiles learned from historical performance data gathered from similar decision alternatives 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 alternatives, named *a1001* and *a1010*, marked on 7 **incommensurable** preference criteria: 2 **costs** criteria *c1* and *c2* (to **minimize**) and 6 **benefits** criteria *b1* to *b5* (to **maximize**).

Criterion	b1	b2	b3	b4	b5	c1	c2
weight	2	2	2	2	2	5	5
a1001	37.0	2	2	61.0	31.0	-4	-40.0
a1010	32.0	9	6	55.0	51.0	-4	-35.0

The performances on *benefits* criteria *b1*, *b4* and *b5* are measured on a cardinal scale from 0.0 (worst) to 100.0 (best) whereas, the performances on the *benefits* criteria *b2* and *b3* and on the *cost* criterion *c1* are measured on an ordinal scale from 0 (worst) to 10 (best), respectively -10 (worst) to 0 (best). The performances on the *cost* criterion *c2* are again measured on a cardinal negative scale from -100.00 (worst) to 0.0 (best).

The importance (sum of weights) of the *costs* criteria is **equal** to the importance (sum of weights) of the *benefits* criteria taken all together.

The non trivial decision problem we now face here, is to decide, how the multiple criteria performances of *a1001*, respectively *a1010*, 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 already encountered in the past.

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

10.2 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 distribution 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:

- Ordered **objectives** and a **criteria** dictionaries 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.

Below, an example Python session concerning 900 decision alternatives randomly generated from a *Cost-Benefit* Performance tableau model from which are also drawn the performances of alternatives *a1001* and *a1010* above.

Listing 10.1: Computing performance quantiles from a given performance tableau

```
1  >>> from performanceQuantiles import PerformanceQuantiles
2  >>> from randomPerfTabs import RandomCBPerformanceTableau
3  >>> nbrActions=900
4  >>> nbrCrit = 7
5  >>> seed = 100
6  >>> tp = RandomCBPerformanceTableau(numberOfActions=nbrActions,\
7                    numberOfCriteria=nbrCrit,seed=seed)
8  >>> pq = PerformanceQuantiles(tp,\
9                    numberOfBins = 'quartiles',\
10                   LowerClosed=True)
11 >>> pq
12  *------- PerformanceQuantiles instance description ------*
13  Instance class   : PerformanceQuantiles
14  Instance name    : 4-tiled_performances
15  # Objectives     : 2
16  # Criteria       : 7
```

```
17  # Quantiles       : 4
18  # History sizes   : {'c1': 887, 'b1': 888, 'b2': 891, 'b3': 895,
19                       'b4': 892, 'c2': 893, 'b5': 887}
20  Attributes        : ['perfTabType', 'valueDigits', 'actionsTypeStatistics',
21                       'objectives', 'BigData', 'missingDataProbability',
22                       'criteria', 'LowerClosed', 'name',
23                       'quantilesFrequencies', 'historySizes',
24                       'limitingQuantiles', 'cdf']
```

The `performanceQuantiles.PerformanceQuantiles` class parameter *numberOfBins* (see Listing 10.1 Line 9 above), choosing the wished number of quantile frequencies, may be either **quartiles** (4 bins), **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**.

Listing 10.2: Printing out the estimated quartile limits

```
1  >>> pq.showLimitingQuantiles(ByObjectives=True)
2   ----  Historical performance quantiles -----*
3  Costs
4  criteria | weights |  '0.00'    '0.25'    '0.50'    '0.75'    '1.00'
5  ---------|-------------------------------------------------------
6    'c1'   |    5    |   -10       -7        -5        -3         0
7    'c2'   |    5    | -96.37    -70.65    -50.10    -30.00     -1.43
8  Benefits
9  criteria | weights |  '0.00'    '0.25'    '0.50'    '0.75'    '1.00'
10  ---------|-------------------------------------------------------
11    'b1'   |    2    |  1.99     29.82     49,44     70.73     99.83
12    'b2'   |    2    |   0         3         5         7        10
13    'b3'   |    2    |   0         3         5         7        10
14    'b4'   |    2    |  3.27     30.10     50.82     70.89     98.05
15    'b5'   |    2    |  0.85     29.08     48.55     69.98     97.56
```

Both objectives are **equi-important**; the sum of weights (10) of the *costs* criteria balance the sum of weights (10) of the *benefits* criteria (see Listing 10.2 column 2). The preference direction of the *costs* criteria *c1* and *c2* is **negative**; the lesser the costs the better it is, whereas all the *benefits* criteria *b1* to *b5* show **positive** preference directions, i.e. the higher the benefits the better it is. The columns entitled '0.00', resp. '1.00' show the *quartile Q0*, resp. *Q4*, i.e. the **worst**, resp. **best** performance observed so far on each criterion. Column '0.50' shows the **median** (*Q2*) performance observed on the criteria.

New decision alternatives 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 new Performance Tableau with 100 new decision alternatives, the so far estimated historical quantile limits may be updated as follows:

Listing 10.3: Generating 100 new random decision alternatives of the same model

```
1  >>> from randomPerfTabs import RandomPerformanceGenerator
2  >>> rpg = RandomPerformanceGenerator(tp,seed=seed)
3  >>> newTab = rpg.randomPerformanceTableau(100)
4  >>> # Updating the quantile norms shown above
5  >>> pq.updateQuantiles(newTab,historySize=None)
```

Parameter *historySize* (see Listing 10.3 Line 5) 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 900/1000 (90%, weight of historical data) against 100/1000 (10%, 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 incoming data. The updated quantile limits may be shown in a browser view (see Fig. 1).

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

Performance quantiles

Sampling sizes between 986 and 995.

criterion	0.00	0.25	0.50	0.75	1.00
b1	1.99	28.77	49.63	75.27	99.83
b2	0.00	2.94	4.92	6.72	10.00
b3	0.00	2.90	4.86	8.01	10.00
b4	3.27	35.91	58.58	72.00	98.05
b5	0.85	32.84	48.09	69.75	99.00
c1	-10.00	-7.35	-5.39	-3.38	0.00
c2	-96.37	-72.22	-52.27	-33.99	-1.43

Fig. 1: Showing the updated quantiles limits

10.3 Rating new performances with quantile norms

For *absolute rating* of a newly given set of decision alternatives with the help of empirical performance quantiles estimated from historical data, we provide the `sortingDigraphs.NormedQuantilesRatingDigraph` class, a specialisation of the `sortingDigraphs.SortingDigraph` 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 alternatives, but also the historical quantile profiles obtained from a given `performanceQuantiles.PerformanceQuantiles` class instance, i.e. 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 list of 10 new decision alternatives generated with the same random performance tableau model and including the two decision alternatives *a1001* and *a1010* mentioned at the beginning.

Listing 10.4: Computing a normed rating of 10 new decision alternatives

```
>>> 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 profiles  : 4
 # New actions        : 10
 Size                 : 96
 Determinateness (%)      : 53.00
 Attributes: ['runTimes','objectives','criteria',
  'LowerClosed','quantilesFrequencies','limitingQuantiles',
  'historySizes','cdf','name','newActions','evaluation',
  'categories','criteriaCategoryLimits','profiles','profileLimits',
  'hasNoVeto','actions','completeRelation','relation',
  'concordanceRelation','valuationdomain','order','gamma',
  'notGamma','rankingRule','rankingCorrelation','rankingScores',
  'actionsRanking','ratingCategories','ratingRelation','relationOrig']
 *---- Constructor run times (in sec.)
```

```
22    #Threads         : 1
23     Total time       : 0.02218
24     Data input       : 0.00134
25     Quantile classes : 0.00008
26     Compute profiles : 0.00021
27     Compute relation : 0.01869
28     Compute rating   : 0.00186
29     Compute sorting  : 0.00000
```

Data input to the `sortingDigraphs.NormedQuantilesRatingDigraph` class constructor (see Listing 10.4 Line 3) are a valid PerformanceQuantiles object *pq* and a compatible list *newActions* of new decision alternatives generated from the same random origin.

Let us have a look at the digraph's nodes, here called **newActions**.

Listing 10.5: Performance tableau of the new incoming
decision alternatives

```
1  >>> nqr.showPerformanceTableau(actionsSubset=nqr.newActions)
2   *----   performance tableau -----*
3   criteria | a1001 a1002 a1003 a1004 a1005 a1006 a1007 a1008 a1009 a1010
4   ---------|-------------------------------------------------------------
5      'b1'  |  37.0  27.0  24.0  16.0  42.0  33.0  39.0  64.0  42.0  32.0
6      'b2'  |   2.0   5.0   8.0   3.0   3.0   3.0   6.0   5.0   4.0   9.0
7      'b3'  |   2.0   4.0   2.0   1.0   6.0   3.0   2.0   6.0   6.0   6.0
8      'b4'  |  61.0  54.0  74.0  25.0  28.0  20.0  20.0  49.0  44.0  55.0
9      'b5'  |  31.0  63.0  61.0  48.0  30.0  39.0  16.0  96.0  57.0  51.0
10     'c1'  |  -4.0  -6.0  -8.0  -5.0  -1.0  -5.0  -1.0  -6.0  -6.0  -4.0
11     'c2'  | -40.0 -23.0 -37.0 -37.0 -24.0 -27.0 -73.0 -43.0 -94.0 -35.0
```

Among the 10 new incoming decision alternatives (see Listing 10.5), we recognize alternatives *a1001* (see column 2) and *a1010* (see last column) we have mentioned in our introduction.

The `sortingDigraphs.NormedQuantilesRatingDigraph` class instance's *actions* dictionary includes as well the closed lower limits of the four quartile classes: *m1* = [0.0- [, *m2* = [0.25- [, *m3* = [0.5- [, *m4* = [0.75 - [. We find these limits in a *profiles* attribute (see Listing 10.6 below).

Listing 10.6: Showing the limiting profiles of the rating
quantiles

```
1  >>> nqr.showPerformanceTableau(actionsSubset=nqr.profiles)
2   *----   Quartiles limit profiles -----*
3   criteria |  'm1'    'm2'    'm3'    'm4'
4   ---------|----------------------------
5      'b1'  |   2.0    28.8    49.6    75.3
6      'b2'  |   0.0     2.9     4.9     6.7
7      'b3'  |   0.0     2.9     4.9     8.0
8      'b4'  |   3.3    35.9    58.6    72.0
```

```
9      'b5'  |   0.8    32.8   48.1    69.7
10     'c1'  | -10.0   -7.4   -5.4    -3.4
11     'c2'  | -96.4  -72.2  -52.3   -34.0
```

The main run time (see Listing 10.4 Lines 23-29) is spent by the class constructor in computing a bipolar-valued outranking relation on the extended actions set including both the new alternatives as well as the quartile class limits. In case of large volumes, i.e. many new decision alternatives 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 alternatives as well as 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 Listing 10.6 Line 2) 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.

Listing 10.7: Copeland ranking of new alternatives and
historical quartile limits

```
1  >>> nqr.rankingRule
2   'Copeland'
3  >>> nqr.actionsRanking
4   ['m4', 'a1005', 'a1010', 'a1002', 'a1008', 'a1006', 'a1001',
5    'a1003', 'm3', 'a1007', 'a1004', 'a1009', 'm2', 'm1']
6  >>> nqr.showCorrelation(nqr.rankingCorrelation)
7   Correlation indexes:
8    Crisp ordinal correlation  : +0.945
9    Epistemic determination    :  0.522
10   Bipolar-valued equivalence : +0.493
```

We achieve here (see Listing 10.7) a linear ranking without ties (from best to worst) of the digraph's actions set, i.e. including the new decision alternatives as well as the quartile limits *m1* to *m4*, which is very close in an ordinal sense ($\tau = 0.945$) to the underlying strict outranking relation.

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

```
1  >>> nqr.ratingCategories
2   OrderedDict([
3   ('m2', ['a1007','a1004','a1009']),
4   ('m3', ['a1005','a1010','a1002','a1008','a1006','a1001','a1003'])
5   ])
```

We notice above that no new decision alternatives are actually rated in the lowest [0.0-

0.25[, respectively highest [0.75- [quartile classes. Indeed, the rating result is shown, in descending order, as follows:

Listing 10.8: Showing a quantiles rating result

```
1  >>> nqr.showQuantilesRating()
2   *-------- Quartiles rating result ---------
3   [0.50 - 0.75[ ['a1005', 'a1010', 'a1002', 'a1008',
4                 'a1006', 'a1001', 'a1003']
5   [0.25 - 0.50[ ['a1007', 'a1004', 'a1009']
```

The same result may more conveniently be consulted in a browser view via a specialised rating heatmap format (see `perfTabs:PerformanceTableau.showHTMLPerformanceHeatmap()` method (see Fig. 2).

```
1  >>> nqr.showHTMLRatingHeatmap(pageTitle='Heatmap of Quartiles Rating',
2                 Correlations=True,colorLevels=5)
```

Heatmap of Quartiles Rating

Ranking rule: **Copeland**; *Ranking correlation:* **0.938**

criteria	c2	b3	c1	b4	b1	b2	b5
weights	5	2	5	2	2	2	2
tau$^{(*)}$	+0.64	+0.54	+0.43	+0.37	+0.37	+0.35	+0.34
[0.75 -	-30.00	7.00	-3.00	70.89	70.73	7.00	69.98
a1005c	-24.00	6.00	-1.00	28.00	42.00	3.00	30.00
a1010n	-35.00	6.00	-4.00	55.00	32.00	9.00	51.00
a1002c	-23.00	4.00	-6.00	54.00	27.00	5.00	63.00
a1008n	-43.00	6.00	-6.00	49.00	64.00	5.00	96.00
a1006c	-27.00	3.00	-5.00	20.00	33.00	3.00	39.00
a1001c	-40.00	2.00	-4.00	61.00	37.00	2.00	31.00
a1003a	-37.00	2.00	-8.00	74.00	24.00	8.00	61.00
[0.50 -	-50.10	5.00	-5.00	50.82	49.44	5.00	48.55
a1007c	-73.00	2.00	-1.00	20.00	39.00	6.00	16.00
a1004c	-37.00	1.00	-5.00	25.00	16.00	3.00	48.00
a1009n	-94.00	6.00	-6.00	44.00	42.00	4.00	57.00
[0.25 -	-70.65	3.00	-7.00	30.10	29.82	3.00	29.08
[0.00 -	-96.37	0.00	-10.00	3.27	1.99	0.00	0.85

Color legend:

quantile	20.00%	40.00%	60.00%	80.00%	100.00%

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

Fig. 2: Heatmap of normed quartiles ranking

Using furthermore a specialised version of the `transitiveDigraphs.TransitiveDigraph.exportGraphViz()` method allows drawing the same rating

result in a Hasse diagram format (see Fig. 3).

```
1  >>> nqr.exportRatingGraphViz('normedRatingDigraph')
2   *---- exporting a dot file for GraphViz tools ---------*
3    Exporting to normedRatingDigraph.dot
4    dot -Grankdir=TB -Tpng normedRatingDigraph.dot -o normedRatingDigraph.png
```


Fig. 3: Normed quartiles rating digraph
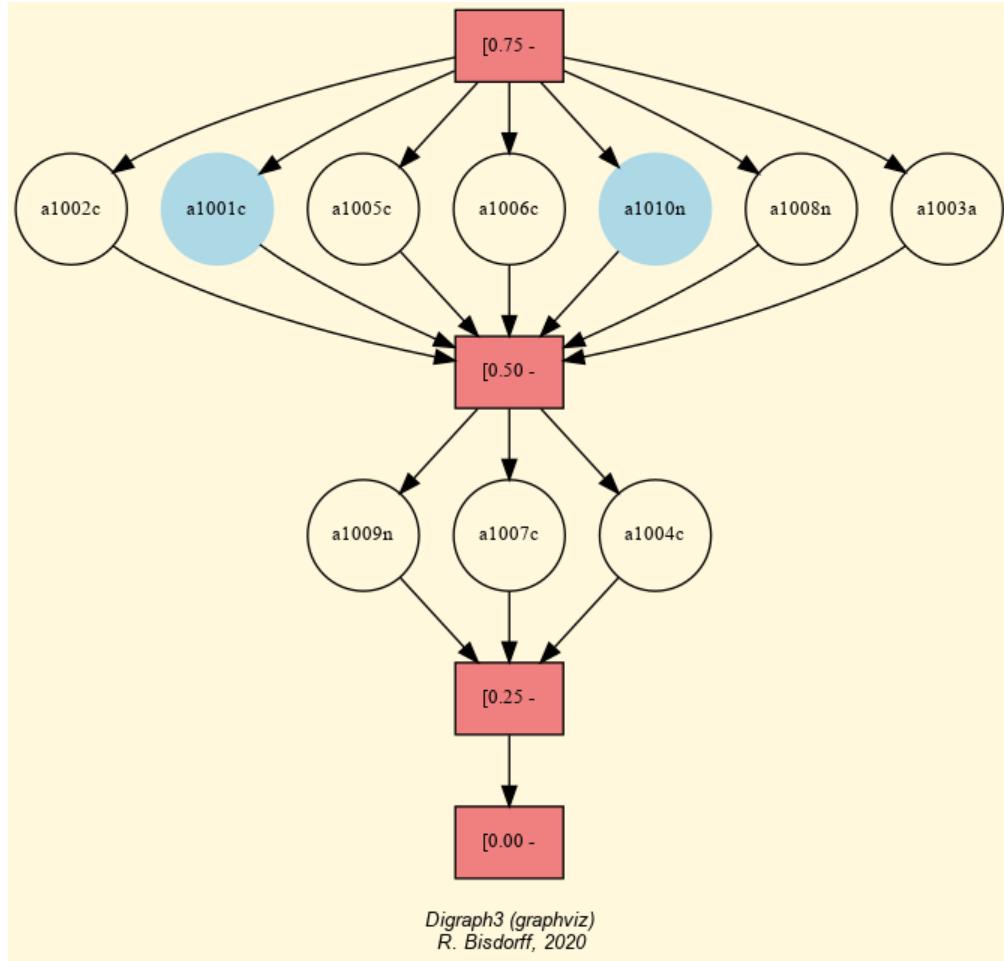
We may now answer the **normed rating decision problem** stated at the beginning. Decision alternative *a1001* and alternative *a1010* (see below) are both rated into the same quartile **Q3** class (see Fig. 3), even if the *Copeland* ranking, obtained from the underlying strict outranking digraph (see Fig. 2), suggests that alternative *a1010* is effectively *better performing than* alternative *a1001*.

Criterion	b1	b2	b3	b4	b5	c1	c2
weight	2	2	2	2	2	5	5
a1001	37.0	2	2	61.0	31.0	-4	-40.0
a1010	32.0	9	6	55.0	51.0	-4	-35.0

A preciser rating result may indeed be achieved when using **deciles** instead of *quartiles* for estimating the historical marginal cumulative distribution functions.

Listing 10.9: Normed deciles rating result

```
1  >>> pq1 = PerformanceQuantiles(tp, numberOfBins = 'deciles',\
2                  LowerClosed=True)
3  >>> pq1.updateQuantiles(newTab,historySize=None)
4  >>> nqr1 = NormedQuantilesRatingDigraph(pq1,newActions,rankingRule='best')
5  >>> nqr1.showQuantilesRating()
6   *-------- Deciles rating result ---------
7   [0.60 - 0.70[ ['a1005', 'a1010', 'a1008', 'a1002']
8   [0.50 - 0.60[ ['a1006', 'a1001', 'a1003']
9   [0.40 - 0.50[ ['a1007', 'a1004']
10  [0.30 - 0.40[ ['a1009']
```

Compared with the quartiles rating result, we notice in Listing 10.9 that the seven alternatives ($a1001$, $a1002$, $a1003$, $a1005$, $a1006$, $a1008$ and $a1010$), rated before into the third quartile class [0.50-0.75[, are now divided up: alternatives $a1002$, $a1005$, $a1008$ and $a1010$ attain now the 7th decile class [0.60-0.70[, whereas alternatives $a1001$, $a1003$ and $a1006$ attain only the 6th decile class [0.50-0.60[. Of the three $Q2$ [0.25-0.50[rated alternatives ($a1004$, $a1007$ and $a1009$), alternatives $a1004$ and $a1007$ are now rated into the 5th decile class [0.40-0.50[and $a1009$ is lowest rated into the 4th decile class [0.30-0.40[.

A browser view may again more conveniently illustrate this refined rating result (see Fig. 4).

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

Heatmap of Deciles rating

Ranking rule: **NetFlows**; *Ranking correlation:* **0.960**

criteria	c2	b3	c1	b1	b5	b2	b4
weights	5	2	5	2	2	2	2
tau(*)	0.67	0.65	0.58	0.57	0.53	0.53	0.48
[0.90 -	-20.32	7.73	-2.53	86.83	82.16	7.66	82.04
[0.80 -	-29.70	7.26	-3.35	79.30	75.15	6.64	74.66
[0.70 -	-37.97	6.67	-4.14	70.95	60.20	5.88	69.76
a1005c	-24.00	6.00	-1.00	42.00	30.00	3.00	28.00
a1010n	-35.00	6.00	-4.00	32.00	51.00	9.00	55.00
a1008n	-43.00	6.00	-6.00	64.00	96.00	5.00	49.00
a1002c	-23.00	4.00	-6.00	27.00	63.00	5.00	54.00
[0.60 -	-44.23	5.92	-5.04	60.56	56.01	5.37	62.23
a1006c	-27.00	3.00	-5.00	33.00	39.00	3.00	20.00
a1001c	-40.00	2.00	-4.00	37.00	31.00	2.00	61.00
a1003a	-37.00	2.00	-8.00	24.00	61.00	8.00	74.00
[0.50 -	-52.22	4.64	-6.02	49.56	48.07	4.83	58.45
a1007c	-73.00	2.00	-1.00	39.00	16.00	6.00	20.00
a1004c	-37.00	1.00	-5.00	16.00	48.00	3.00	25.00
[0.40 -	-60.50	3.84	-6.69	39.61	40.16	4.25	49.82
a1009n	-94.00	6.00	-6.00	42.00	57.00	4.00	44.00
[0.30 -	-67.14	3.12	-7.32	30.85	34.33	3.30	40.89
[0.20 -	-77.07	2.55	-7.94	23.84	29.57	2.27	30.45
[0.10 -	-83.04	1.99	-8.48	16.64	16.91	1.58	24.78
[0.00 -	-96.37	0.00	-10.00	1.99	0.85	0.00	3.27

Color legend:

quantile	20.00%	40.00%	60.00%	80.00%	100.00%

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

Fig. 4: Heatmap of normed deciles rating

In this *deciles* rating, decision alternatives *a1001* and *a1010* are now, as expected, rated in the *6th* decile (D6), respectively in the *7th* decile (D7).

To avoid having to recompute performance deciles from historical data when wishing to refine a rating result, it is useful, depending on the actual size of the historical data, to initially compute performance quantiles with a relatively high number of bins, for instance *dodeciles* or *centiles*. It is then possible to correctly interpolate *quartiles* or *deciles* for instance, when constructing the rating digraph.

Listing 10.10: From deciles interpolated quartiles rating result

```
1  >>> nqr2 = NormedQuantilesRatingDigraph(pq1,newActions,
2                  quantiles='quartiles')
3  >>> nqr2.showQuantilesRating()
4   *-------- Deciles rating result ---------
5   [0.50 - 0.75[ ['a1005', 'a1010', 'a1002', 'a1008',
6                  'a1006', 'a1001', 'a1003']
7   [0.25 - 0.50[ ['a1004', 'a1007', 'a1009']
```

With the *quantiles* parameter (see Listing 10.10 Line 2), we may recover by interpolation the same quartiles rating as obtained directly with historical performance quartiles (see Listing 10.8). Mind that a correct interpolation of quantiles from a given cumulative distribution function requires more or less uniform distributions of observations in each bin.

More generally, in the case of industrial production monitoring problems, for instance, where large volumes of historical performance data may be available, it may be of interest to estimate even more precisely the marginal cumulative distribution functions, especially when **tail** rating results, i.e. distinguishing **very best**, or **very worst** multiple criteria performances, become a critical issue. Similarly, the *historySize* parameter may be used for monitoring on the fly **unstable** random multiple criteria performance data.

Back to *Content Table* (page 3)

---

11 Where do study the best students ? : A case study

In 2004, the German magazine *Der Spiegel* conducted an extensive online survey assessing the apparent quality of German University students[28]. About 70,000 university students from 15 academic disciplines, like German studies, Economics, Physics, Mathematics, Medecine, were questioned on their 'Abitur' and university examens' marks, time of studies and age, grants, awards and publications, IT proficiency, linguistic skills, practical work experience, foreign mobility and civil engagement. Each student received in return a global quality score by a specific weighing of the collected data depending on the disicpline of the student.[29].

---

[28] Ref: Der Spiegel 48/2004 p.181 Url: https://www.spiegel.de/thema/studentenspiegel/ .

[29] The methlogy guiding the *Spiegel* survey may be consulted in German here .

The published results of the survey concern eventually about 50,000 students majorly enroled in one of the fifteen academic disciplines at the 41 German Universities or Higher Technical Institutes offering at least 8 of the fifteen disciplines.

We would like now to rate these 41 German higher education institutions with respect to their apparent enrolment quality.

11.1 The performance tableau

The published data of the 2004 *Spiegel* student survey are stored in a file named studentenSpiegel04.py of `perfTabs.PerformanceTableau` format.

Listing 11.1: The 2004 Spiegel students survey global results

```
1  >>> from perfTabs import PerformanceTableau
2  >>> t = PerformanceTableau('studentenSpiegel04')
3  >>> disciplines = [t for t in t.criteria]
4  >>> t.showHTMLPerformanceHeatmap(\
5                   criteriaList=disciplines,ndigits=1)
```

criteria	germ	pol	psy	soc	law	eco	mgt	phys	chem	math	info	elec	mec	bio	med
weights	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00
frei	57.8	60.5	64.1	57.5	50.7	53.3	NA	61.6	57.0	60.6	58.1	NA	NA	55.4	54.2
tum	NA	NA	NA	NA	NA	NA	68.0	62.8	58.8	62.6	58.2	58.2	56.9	53.6	60.1
mu	57.2	60.1	60.9	54.0	47.3	55.8	57.5	62.0	58.1	59.6	57.1	NA	NA	50.4	52.6
hei	61.4	59.5	59.8	52.2	51.1	54.4	NA	60.9	56.7	61.3	NA	NA	NA	55.2	55.5
leip	57.4	60.4	62.4	59.5	46.3	NA	54.6	62.2	57.0	NA	3.0	NA	NA	56.9	51.5
kons	56.9	65.9	59.1	54.7	48.3	59.0	NA	59.9	58.0	NA	59.7	NA	NA	55.7	NA
berh	57.3	58.5	59.8	59.2	48.8	59.5	55.5	61.9	53.2	57.9	55.8	NA	NA	55.0	52.3
stu	52.5	58.4	NA	NA	NA	NA	56.6	61.5	55.2	60.6	59.8	60.2	57.8	57.1	NA
tueb	57.9	57.7	58.4	1.0	46.8	60.8	54.4	61.6	57.5	NA	55.2	NA	NA	53.7	52.1
aug	57.9	54.3	NA	54.8	45.6	NA	54.3	62.3	NA	61.2	58.1	NA	NA	NA	NA
mnst	55.4	56.7	62.2	56.3	46.4	54.1	56.3	55.2	55.0	56.9	57.7	NA	NA	51.2	52.8
mnh	52.2	57.2	61.1	55.0	45.0	57.0	59.7	NA	NA	NA	58.6	NA	NA	NA	NA
jena	56.5	55.8	58.5	52.8	45.3	NA	56.2	61.6	57.8	NA	57.2	NA	NA	52.7	51.1
reg	54.8	55.4	62.1	NA	46.1	52.3	55.5	60.5	55.8	59.2	NA	NA	NA	54.6	50.9
wrzb	56.9	56.0	59.8	NA	46.4	53.3	52.8	60.2	56.6	NA	55.9	NA	NA	53.0	52.2
dres	55.2	55.9	60.6	56.2	44.0	56.7	54.8	59.9	55.8	57.8	56.2	56.1	54.8	55.3	49.2
marb	53.6	54.7	57.6	59.8	40.3	NA	55.5	62.8	57.8	NA	55.6	NA	NA	53.2	51.1
saar	57.9	NA	56.5	NA	48.1	NA	52.2	61.2	56.2	NA	57.6	NA	NA	NA	49.6
ksl	NA	NA	NA	NA	NA	NA	55.9	59.7	54.6	62.2	56.2	57.5	56.3	50.8	NA
berf	54.7	61.4	59.8	55.5	45.7	50.5	52.2	61.6	57.4	NA	54.9	NA	NA	51.6	49.0
chem	54.3	57.1	60.8	53.3	NA	NA	52.7	NA	NA	NA	57.7	57.5	53.6	NA	NA
erl	57.9	55.1	58.7	55.4	42.9	NA	55.6	60.3	54.0	60.5	54.6	55.9	55.1	51.8	49.3
goet	58.7	56.3	59.8	53.5	44.8	53.6	52.6	60.4	53.9	63.1	NA	NA	NA	50.5	48.9
tri	54.1	58.0	58.3	54.9	46.3	52.8	52.8	NA	NA	60.7	52.3	NA	NA	NA	NA
koel	51.7	57.6	58.9	56.1	46.1	56.1	54.6	58.7	54.0	56.5	NA	NA	NA	51.8	50.7
bon	54.1	57.3	60.3	56.0	47.2	53.6	NA	59.9	53.1	59.4	53.7	NA	NA	50.1	3.0
main	54.2	57.9	56.9	55.7	46.5	50.7	53.1	60.8	56.3	54.7	NA	NA	NA	50.0	49.2
brem	56.9	55.5	52.5	54.5	40.9	NA	55.4	59.7	NA	NA	54.1	50.1	NA	53.3	NA
darm	1.0	59.7	58.6	52.0	NA	NA	NA	62.5	2.0	59.4	3.0	NA	56.1	1.0	NA
gie	53.0	59.0	58.0	NA	41.9	NA	51.2	57.6	NA	NA	NA	NA	NA	50.4	50.0
bertu	51.4	NA	57.7	59.1	NA	49.6	54.0	58.9	52.0	56.8	55.4	56.1	54.3	NA	NA
ham	57.0	60.2	57.3	53.6	44.1	52.1	49.8	56.4	54.2	54.9	54.7	NA	NA	52.7	49.2
aach	53.3	50.8	62.7	51.0	NA	NA	49.6	59.1	53.6	58.6	54.6	57.2	54.4	52.2	49.5
fran	51.7	53.1	58.0	51.5	41.9	53.5	52.0	62.1	55.5	57.0	52.4	NA	NA	51.3	51.2
brau	53.5	54.0	NA	51.5	NA	NA	53.4	59.8	50.1	54.7	52.6	54.5	55.2	53.1	NA
han	50.4	52.8	NA	49.9	41.2	NA	NA	57.5	53.6	56.6	58.8	53.5	53.6	53.8	NA
dsd	53.5	NA	57.5	48.8	44.9	NA	50.5	NA	53.5	NA	NA	NA	NA	47.3	50.5
kiel	51.9	52.2	58.4	NA	45.1	50.5	52.8	59.7	52.8	NA	54.9	54.2	NA	52.7	49.6
boc	53.9	NA	55.2	NA	39.1	NA	NA	56.8	53.3	57.6	NA	54.2	54.4	48.0	49.8
bie	51.4	NA	54.4	55.6	41.9	NA	50.7	53.9	54.2	56.3	55.8	NA	NA	49.7	NA
duis	50.6	52.5	NA	47.9	NA	NA	47.5	54.6	52.8	51.6	56.8	53.6	51.9	NA	48.0

Color legend:

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

Fig. 1: Quality of enroled students per academic discipline

All fifteen academic disciplines are considered *equally significant* for the rating decisions

(see Fig. 1 row 2). The average enrolment quality scores are coloured along a 7-tiling scheme and the universities are ranked with the *NetFlows* rule applied to the corresponding pairwise *outranking* situations.

In Fig. 1 we may furthermore notice that the Technical University *Dresden* is the only Institution which offers all fifteen academic disciplines. Whereas, on the one side, the Technical University of *München* and the University of *Kaiserslautern* only offer *Sciences* and *Technology* disciplines. On the other side, the University of *Mannheim*, is only offering *Humanities* and *Law Studies*.

Details of the rating criteria may be consulted in a browser view (see Fig. 2 below).

```
>>> t.showHTMLCriteria()
```

#	Identifyer	Name	Comment	Weight	Scale			Thresholds (ax + b)		
					direction	min	max	indifference	preference	veto
1	**bio**	Biology	Faculty STM	1.00	max	45.00	65.00	0.00x + 0.00	0.00x + 0.50	
2	**chem**	Chemistry	Faculty STM	1.00	max	45.00	65.00	0.00x + 0.00	0.00x + 0.50	
3	**eco**	Economics	Faculty HLM	1.00	max	45.00	65.00	0.00x + 0.00	0.00x + 0.50	
4	**elec**	Electrical Engineering	Faculty STM	1.00	max	45.00	65.00	0.00x + 0.00	0.00x + 0.50	
5	**germ**	German Studies	Faculty HLM	1.00	max	45.00	65.00	0.00x + 0.00	0.00x + 0.50	
6	**info**	Informatics	Faculties STM & HLM	1.00	max	45.00	65.00	0.00x + 0.00	0.00x + 0.50	
7	**law**	Law Studies	Faculty HLM	1.00	max	35.00	65.00	0.00x + 0.00	0.00x + 0.50	
8	**math**	Mathematics	Faculty STM	1.00	max	45.00	65.00	0.00x + 0.00	0.00x + 0.50	
9	**mec**	Mechanical Engineering	Faculty STM	1.00	max	45.00	65.00	0.00x + 0.00	0.00x + 0.50	
10	**med**	Medecine	Faculty STM	1.00	max	45.00	65.00	0.00x + 0.00	0.00x + 0.50	
11	**mgt**	Management	Faculties HLM & STM	1.00	max	40.00	80.00	0.00x + 0.00	0.00x + 0.50	
12	**phys**	Physics	Faculty STM	1.00	max	45.00	65.00	0.00x + 0.00	0.00x + 0.50	
13	**pol**	Politology	Faculty HLM	1.00	max	50.00	70.00	0.00x + 0.00	0.00x + 0.50	
14	**psy**	Psychology	Faculty HLM	1.00	max	50.00	70.00	0.00x + 0.00	0.00x + 0.50	
15	**soc**	Sociology	Faculty HLM	1.00	max	45.00	65.00	0.00x + 0.00	0.00x + 0.50	

Fig. 2: Detals of the rating criteria

The computing of a quality score for a student actually depends on the enroled discipline. The average quality measurement scales may thus differ from discipline to discipline, like *Law Studies* and *Politology* (see Fig. 2). The average enrolment quality scores shown in Fig. 2, hence, appear to be **incommensurable** between disciplines. To take furthermore into account a potential *imprecision* of the individual scores' computation, we shall assume that, for all disciplines, a difference of 0.1 is *insignificant* and a difference of 0.5 *positively attests* a better average students' quality.

Let us now inspect the resulting bipolar-valued outranking digraph.

11.2 The outranking digraph

Listing 11.2: Inspecting the outranking digraph

```
1  >>> g = BipolarOutrankingDigraph(t)
2  >>> g
3   *------- Object instance description ------*
4    Instance class       : BipolarOutrankingDigraph
5    Instance name        : rel_studentenSpiegel04
6    # Actions            : 41
7    # Criteria           : 15
8    Size                 : 863
9    Determinateness (%)  : 63.75
10   Valuation domain     : [-1.00;1.00]
11  >>> g.computeTransitivityDegree(Comments=True)
12   Transitivity degree of digraph <rel_studentenSpiegel04>:
13    #triples x>y>z: 57837, #closed: 32009, #open: 25828
14    (#closed/#triples) =  0.553
15  >>>  g.computeSymmetryDegree(Comments=True)
16   Symmetry degree of digraph <rel_studentenSpiegel04>:
17    #arcs x>y: 793, #symmetric: 70, #asymmetric: 723
18    #arcs/#symmetric =  0.088
```

The actual outranking digraph obtained with the average quality scores per academic discipline shows 863 validatesd pairwise outranking situations between the Universities. Unfortunately, the transitivity of the outranking digraph is far from being satisfied: nearly half of the transitive closure is missing. Despite the rather large preference discrimination threshold we have assumed, there does not occur many indifference situations.

We may furthermore check if there exists any *cyclic* outranking situations.

Listing 11.3: Computing the chordlessCircuits

```
1  >>> g.computeChordlessCircuits()
2  >>> g.showChordlessCircuits()
3   *---- Chordless circuits ----*
4    46 circuits.
5     1:  ['aach', 'bertu', 'brau'] , credibility : 0.200
6     2:  ['aach', 'bertu', 'brem'] , credibility : 0.067
7     3:  ['aach', 'bie', 'darm', 'brau'] , credibility : 0.067
8     4:  ['aug', 'saar', 'mnst'] , credibility : 0.133
9     5:  ['aug', 'main', 'mnh'] , credibility : 0.067
10    6:  ['aug', 'mu', 'mnst'] , credibility : 0.133
11    7:  ['aug', 'mu', 'hei'] , credibility : 0.067
12    8:  ['berf', 'ksl', 'chem'] , credibility : 0.067
13    9:  ['berf', 'ksl', 'dres'] , credibility : 0.067
14   10:  ['berf', 'tri', 'marb'] , credibility : 0.067
15   11:  ['berh', 'kons', 'stu'] , credibility : 0.133
16   12:  ['berh', 'kons', 'hei'] , credibility : 0.067
17   13:  ['bie', 'darm', 'boc'] , credibility : 0.067
18   14:  ['bie', 'darm', 'gie'] , credibility : 0.067
```

```
19    15:  ['bon', 'tri', 'marb'] , credibility : 0.067
20    16:  ['brau', 'dsd', 'darm'] , credibility : 0.067
21    17:  ['chem', 'marb', 'koel'] , credibility : 0.067
22    18:  ['chem', 'marb', 'ksl'] , credibility : 0.133
23    19:  ['chem', 'wrzb', 'koel'] , credibility : 0.067
24    20:  ['chem', 'wrzb', 'reg'] , credibility : 0.133
25    21:  ['chem', 'wrzb', 'ksl'] , credibility : 0.133
26    22:  ['chem', 'saar', 'koel'] , credibility : 0.067
27    23:  ['chem', 'saar', 'ksl'] , credibility : 0.133
28    24:  ['chem', 'saar', 'dres'] , credibility : 0.133
29    25:  ['chem', 'saar', 'mnst'] , credibility : 0.133
30    26:  ['chem', 'jena', 'koel'] , credibility : 0.067
31    27:  ['chem', 'jena', 'ksl'] , credibility : 0.133
32    28:  ['chem', 'jena', 'dres'] , credibility : 0.133
33    29:  ['dres', 'marb', 'ksl'] , credibility : 0.067
34    30:  ['dres', 'marb', 'saar'] , credibility : 0.067
35    31:  ['dres', 'main', 'mnh'] , credibility : 0.067
36    32:  ['erl', 'tri', 'marb'] , credibility : 0.067
37    33:  ['goet', 'tri', 'marb'] , credibility : 0.067
38    34:  ['jena', 'saar', 'mnst'] , credibility : 0.133
39    35:  ['jena', 'main', 'mnh'] , credibility : 0.067
40    36:  ['koel', 'main', 'mnh'] , credibility : 0.067
41    37:  ['koel', 'tri', 'marb'] , credibility : 0.067
42    38:  ['ksl', 'main', 'mnh'] , credibility : 0.067
43    39:  ['ksl', 'tri', 'marb'] , credibility : 0.067
44    40:  ['main', 'mnh', 'marb'] , credibility : 0.067
45    41:  ['main', 'mnh', 'wrzb'] , credibility : 0.067
46    42:  ['main', 'mnh', 'mu'] , credibility : 0.067
47    43:  ['marb', 'saar', 'wrzb'] , credibility : 0.133
48    44:  ['marb', 'saar', 'tri'] , credibility : 0.067
49    45:  ['marb', 'saar', 'mnst'] , credibility : 0.133
50    46:  ['mnh', 'mu', 'stu'] , credibility : 0.133
```

Here we observe 46 such outranking circuits, like: *Konstanz* $>=$ *Berlin-Humboldt* $>=$ *Stuttgart* $>=$ *Konstanz* (see Listing 11.3 circuit 6 above), With the *NetFlows* ranking rule, they appear in fact 6th, 7th and 8th ranked (see (Fig. 1). The occurence of so many outranking circuits makes any linear ranking doubtful, independently of the specific ranking rule we might apply.

In this case, it is more convincing to operate quantiles rating of the sutdents' quality scores, as is, indeed, proposed by the Spiegel magazine publication.

11.3 Nine-tiling the average enrolment quality scores

We are going to rate the Universities with a normed 9-tiles rating procedure. Therefore we need, first, to estimate 9-tiling score limits per academic discipline.

Listing 11.4: Computing the 9-tiling enrolment quality scores limits

```
>>> from performanceQuantiles import *
>>> pq = PerformanceQuantiles(t,numberOfBins=9)
>>> pq
 *------- PerformanceQuantiles instance description ------*
  Instance class    : PerformanceQuantiles
  Instance name     : 9-tiled_performances
  # Criteria        : 15
  # Quantiles       : 9
  # History sizes   : {'germ': 39, 'pol': 34, 'psy': 34, 'soc': 32,
                       'law': 32, 'eco': 21, 'mgt': 34, 'phys': 37,
                       'chem': 35, 'math': 27, 'info': 33,
                       'elec': 14, 'mec': 13, 'bio': 34, 'med': 28}
>>> pq.showHTMLLimitingQuantiles(Transposed=True,\
        ndigits=1,title='9-tiled quality score limits')
```

9-tiled quality score limits

Sampling sizes between 13 and 39.

criterion	0.00	0.11	0.22	0.33	0.44	0.56	0.67	0.78	0.89	1.00
bio	45.0	49.9	50.5	51.4	52.3	53.0	53.5	54.8	55.5	57.1
chem	45.0	52.8	53.5	54.0	54.4	55.6	56.4	57.1	57.8	58.8
eco	49.6	50.6	52.2	53.3	53.5	53.9	55.8	56.8	59.3	60.8
elec	50.1	53.6	54.2	54.4	55.9	56.1	57.3	57.5	59.1	60.2
germ	45.0	51.5	52.4	53.5	54.1	55.1	56.9	57.3	57.9	61.4
info	45.0	52.5	54.6	54.9	55.7	56.2	57.2	58.0	58.7	59.8
law	39.1	41.6	43.0	44.9	45.4	46.1	46.4	47.2	48.5	51.1
math	51.6	54.9	56.6	57.0	57.9	59.4	60.5	60.7	62.2	63.1
mec	51.9	53.6	54.2	54.4	54.7	55.1	55.8	56.4	57.4	57.8
med	45.0	49.0	49.2	49.6	50.2	51.0	51.4	52.3	54.0	60.1
mgt	47.5	50.7	52.2	52.8	53.5	54.6	55.5	55.7	56.8	68.0
phys	53.9	56.9	58.9	59.7	60.0	60.7	61.6	61.8	62.3	62.8
pol	50.8	53.0	54.9	55.8	56.7	57.6	58.3	59.6	60.4	65.9
psy	52.5	56.8	57.7	58.3	58.6	59.7	59.8	60.8	62.2	64.1
soc	45.0	50.5	52.0	53.4	54.5	55.0	55.6	56.2	59.1	59.8

Fig. 3: 9-tiling score limits per academic discipline

The history sizes reported in Listing 11.4 indicate the number of Universities offering each one of the fifteen disciplines. *German Studies*, for instance, are thus offered in 39 out of 41 Universities, whereas *Electrical* and *Mechanical Engineering* are only offered in 14, respectively 13 Instituions. None of the fifteen disciplines are offered in all the 41 Universities.

We add, now, these nine-tiling score limits to the 41 Universities and sort out, with the help of the `sortingDigraphs.NormedQuantilesRatingDigraph` class, the 9-tiles quality equivalence classes.

Listing 11.5: Rating the Universities into enrolment quality 9-tiles

```
1  >>> from sortingDigraphs import NormedQuantilesRatingDigraph
2  >>> nqr = NormedQuantilesRatingDigraph(pq,t,rankingRule='NetFlows')
3  >>> nqr.showQuantilesRating()
4   *-------- Quantiles rating result ---------
5    [0.67 - 0.78[ ['frei', 'tum', 'mu']
6    [0.56 - 0.67[ ['hei', 'kons', 'leip', 'berh', 'stu',
7                   'tueb', 'aug', 'mnst', 'reg', 'mnh']
8    [0.44 - 0.56[ ['jena', 'dres', 'wrzb', 'marb', 'saar',
9                   'ksl', 'berf', 'chem', 'erl', 'goet',
10                  'tri', 'koel']
11   [0.33 - 0.44[ ['bon', 'main', 'brem', 'darm', 'gie',
12                  'bertu', 'ham', 'aach']
13   [0.22 - 0.33[ ['fran', 'dsd', 'brau', 'han', 'kiel',
14                  'boc', 'bie']
15   [0.11 - 0.22[ ['duis']
```

The average enrolment quality at the Universities of *Freiburg* and *München*, as well as at the Technical University of *München* are first rated (see Listing 11.5 Line 5). Last-rated appears the average enrolment quality at the University of *Duisburg* (Line 15). Midfield appear to be the Universities of *Jena, Dresden, Würzburg, Marburg, Saarbrücken, Kaiserslautern, Berlin* (Frei), *Chemnitz, Erlangen-Nürnberg, Göttingen, Trier* and *Köln*.

We may nicely illustrate this rating result with the help of a corresponding heatmap view.

```
>>> nqr.showHTMLRatingHeatmap(colorLevels=7,\
                ndigits=1,rankingRule='NetFlows')
```

criteria	soc	psy	pol	phys	mgt	med	mec	math	law	info	germ	elec	eco	chem	bio
weights	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
[0.89 -	59.14	62.24	60.42	62.29	56.80	54.04	57.40	62.20	48.52	58.67	57.90	59.09	59.33	57.82	55.47
[0.78 -	56.19	60.84	59.59	61.83	55.73	52.28	56.37	60.70	47.16	57.97	57.33	57.50	56.80	57.09	54.78
frei	57.50	64.10	60.50	61.60	NA	54.20	NA	60.60	50.70	58.10	57.80	NA	53.30	57.00	55.40
tum	NA	NA	NA	62.80	68.00	60.10	56.90	62.60	NA	58.20	NA	58.20	NA	58.80	53.60
mu	54.00	60.90	60.10	62.00	57.50	52.60	NA	59.60	47.30	57.10	57.20	NA	55.80	58.10	50.40
[0.67 -	55.63	59.80	58.27	61.57	55.47	51.40	55.80	60.50	46.40	57.20	56.90	57.30	55.80	56.40	53.50
hei	52.20	59.80	59.50	60.90	NA	55.50	NA	61.30	51.10	NA	61.40	NA	54.40	56.70	55.20
kons	54.70	59.10	65.90	59.90	NA	NA	NA	NA	48.30	59.70	56.90	NA	59.00	58.00	55.70
leip	59.50	62.40	60.40	62.20	54.60	51.50	NA	NA	46.30	3.00	57.40	NA	NA	57.00	56.90
berh	59.20	59.80	58.50	61.90	55.50	52.30	NA	57.90	48.80	55.60	57.30	NA	59.50	53.20	55.00
stu	NA	NA	58.40	61.50	56.60	NA	57.80	60.60	NA	59.80	52.50	60.20	NA	55.20	57.10
tueb	1.00	58.40	57.70	61.60	54.40	52.10	NA	NA	46.80	55.20	57.90	NA	60.80	57.50	53.70
aug	54.80	NA	54.30	62.30	54.30	NA	NA	61.20	45.60	58.10	57.90	NA	NA	NA	NA
mnst	56.30	62.20	56.70	55.20	56.30	52.80	NA	56.90	46.40	57.70	55.40	NA	54.10	55.00	51.20
reg	NA	62.10	55.40	60.50	55.50	50.90	NA	59.20	46.10	NA	54.80	NA	52.30	55.80	54.60
mnh	55.00	61.10	57.20	NA	59.70	NA	NA	NA	45.00	58.60	52.20	NA	57.00	NA	NA
[0.56 -	54.98	59.72	57.57	60.67	54.58	51.01	55.12	59.40	46.10	56.20	55.07	56.10	53.93	55.63	52.97
jena	52.80	58.50	55.80	61.60	56.20	51.10	NA	NA	45.30	57.20	56.50	NA	NA	57.80	52.70
dres	56.20	60.60	55.90	59.90	54.80	49.20	54.80	57.80	44.00	56.20	55.20	56.10	56.70	55.80	55.30
wrzb	NA	59.80	56.00	60.20	52.80	52.20	NA	NA	46.40	55.90	56.90	NA	53.30	56.60	53.00
marb	59.80	57.60	54.70	62.80	55.50	51.10	NA	NA	40.30	55.60	53.60	NA	NA	57.80	53.20
saar	NA	56.50	NA	61.20	52.20	49.60	NA	NA	48.10	57.60	57.90	NA	NA	56.20	NA
ksl	NA	NA	NA	59.70	55.90	NA	56.30	62.20	NA	56.20	NA	57.50	NA	54.60	50.80
berf	55.50	59.80	61.40	61.60	52.20	49.00	NA	NA	45.70	54.90	54.70	NA	50.50	57.40	51.60
chem	53.30	60.80	57.10	NA	52.70	NA	53.60	NA	NA	57.70	54.30	57.50	NA	NA	NA
erl	55.40	58.70	55.10	60.30	55.60	49.30	55.10	60.50	42.90	54.60	57.90	55.90	NA	54.00	51.80
goet	53.50	59.80	56.30	60.40	52.60	48.90	NA	63.10	44.80	NA	58.70	NA	53.60	53.90	50.50
tri	54.90	58.30	58.00	NA	52.80	NA	NA	60.70	46.30	52.30	54.10	NA	52.80	NA	NA
koel	56.10	58.90	57.60	58.70	54.60	50.70	NA	56.50	46.10	NA	51.70	NA	56.10	54.00	51.80
[0.44 -	54.54	58.61	56.74	60.03	53.47	50.22	54.71	57.90	45.37	55.73	54.13	55.94	53.53	54.42	52.26
bon	56.00	60.30	57.30	59.90	NA	3.00	NA	59.40	47.20	53.70	54.10	NA	53.60	53.10	50.10
main	55.70	56.90	57.90	60.80	53.10	49.20	NA	54.70	46.50	NA	54.20	NA	50.70	56.30	50.00
brem	54.50	52.50	55.50	59.70	55.40	NA	NA	NA	40.90	54.10	56.90	50.10	NA	NA	53.30
darm	52.00	58.60	59.70	62.50	NA	NA	56.10	59.40	NA	3.00	1.00	NA	NA	2.00	1.00
gie	NA	58.00	59.00	57.60	51.20	50.00	NA	NA	41.90	NA	53.00	NA	NA	NA	50.40
bertu	59.10	57.70	NA	58.90	54.00	NA	54.30	56.80	NA	55.40	51.40	56.10	49.60	52.00	NA
ham	53.60	57.30	60.20	56.40	49.80	49.20	NA	54.90	44.10	54.70	57.00	NA	52.10	54.20	52.70
aach	51.00	62.70	50.80	59.10	49.60	49.50	54.40	58.60	NA	54.60	53.30	57.20	NA	53.60	52.20
[0.33 -	53.43	58.33	55.83	59.73	52.80	49.60	54.40	57.00	44.87	54.90	53.50	54.40	53.30	53.97	51.40
fran	51.50	58.00	53.10	62.10	52.00	51.20	NA	57.00	41.90	52.40	51.70	NA	53.50	55.50	51.30
dsd	48.80	57.50	NA	NA	50.50	50.50	NA	NA	44.90	NA	53.50	NA	NA	53.50	47.30
brau	51.50	NA	54.00	59.80	53.40	NA	55.20	54.70	NA	52.60	53.50	54.50	NA	50.10	53.10
han	49.90	NA	52.80	57.50	NA	NA	53.60	56.60	41.20	58.80	50.40	53.50	NA	53.60	53.80
kiel	NA	58.40	52.20	59.70	52.80	49.60	NA	NA	45.10	54.90	51.90	54.20	50.50	52.80	52.70
boc	NA	55.20	NA	56.80	NA	49.80	54.40	57.60	39.10	NA	53.90	54.20	NA	53.30	48.00
bie	55.60	54.40	NA	53.90	50.70	NA	NA	56.30	41.90	55.80	51.40	NA	NA	54.20	49.70
[0.22 -	52.02	57.66	54.92	58.94	52.20	49.22	54.22	56.60	43.02	54.60	52.40	54.20	52.23	53.46	50.46
duis	47.90	NA	52.50	54.60	47.50	48.00	51.90	51.60	NA	56.80	50.60	53.60	NA	52.80	NA
[0.11 -	50.51	56.81	53.03	56.88	50.60	49.02	53.60	54.90	41.59	52.53	51.50	53.56	50.57	52.80	49.93
[0.00 -	1.00	52.50	50.60	53.90	47.50	3.00	51.90	51.60	39.10	3.00	1.00	50.10	49.60	2.00	1.00

Color legend:

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

Fig. 4: Heatmap view of the 9-tiles rating result

The ranking correlation between the pairwise outranking situations and the shown *Net-Flows* ranking is quite high (+0.923, see Fig. 4 first row).

A corresponding graphviz drawing gives the following result.

```
>>> nqr.exportRatingGraphViz(fileName='ratingResult',\
                             graphSize='12,12')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to ratingResult.dot
dot -Grankdir=TB -Tpdf dot -o ratingResult.pdf
```
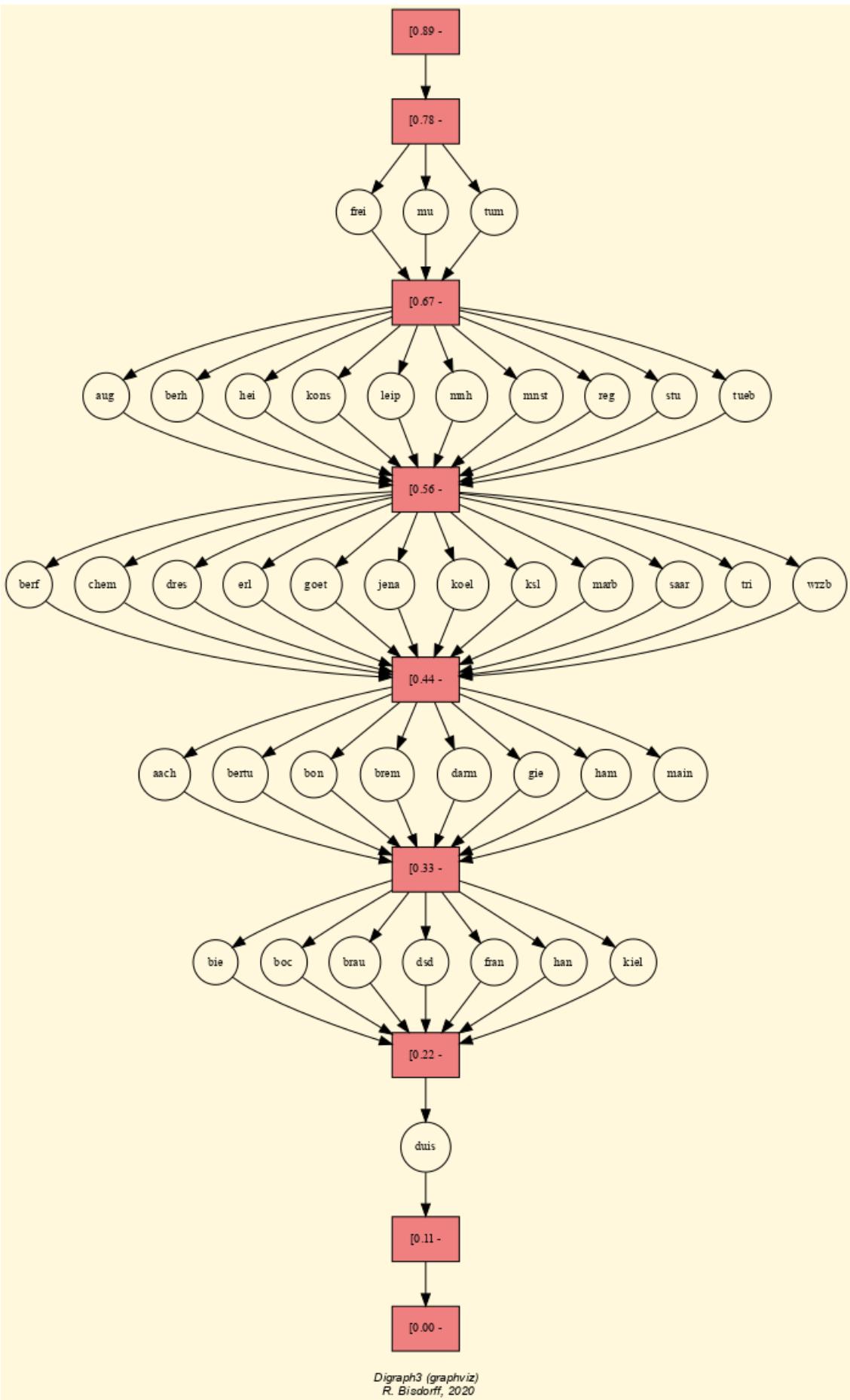
Fig. 5: Graphviz drawing of the 9-tiles rating result

12 HPC ranking with big outranking digraphs

12.1 C-compiled Python modules

The Digraph3 collection provides cythonized[6], i.e. C-compiled and optimised versions of the main python modules for tackling multiple criteria decision problems facing very large sets of decision alternatives (> 10000). Such problems appear usually with a combinatorial organisation of the potential decision alternatives, as is frequently the case in bioinformatics for instance. If HPC facilities with nodes supporting numerous cores ($>$ 20) and big RAM (> 50GB) are available, ranking up to several millions of alternatives (see [BIS-2016]) becomes effectively tractable.

Four cythonized Digraph3 modules, prefixed with the letter c and taking a *pyx* extension, are provided with their corresponding setup tools in the *Digraph3/cython* directory, namely

- *cRandPerfTabs.pyx*

- *cIntegerOutrankingDigraphs.pyx*

- *cIntegerSortingDigraphs.pyx*

- *cSparseIntegerOutrankingDigraphs.pyx*

Their automatic compilation and installation, alongside the standard Digraph3 python3 modules, requires the *cython* compiler[6] (...\$ pip3 install cython) and a C compiler (...\$ sudo apt install gcc on Ubuntu).

Warning: These cythonized modules, specifically designed for being run on HPC clusters (see https://hpc.uni.lu), require the Unix *forking* start

---

[6] See https://cython.org/

method of subprocesses (see start methods of the multiprocessing module (https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods)) and therefore, due to forking problems on Mac OS platforms, may only operate safely on Linux platforms.

12.2 Big Data performance tableaux

In order to efficiently type the C variables, the `cRandPerfTabs` module provides the usual random performance tableau models, but, with **integer** action keys, **float** performance evaluations, **integer** criteria weights and **float** discrimination thresholds. And, to limit as much as possible memory occupation of class instances, all the usual verbose comments are dropped from the description of the *actions* and *criteria* dictionaries.

```
1  >>> from cRandPerfTabs import *
2  >>> t = cRandomPerformanceTableau(numberOfActions=4,numberOfCriteria=2)
3  >>> t
4      *------- PerformanceTableau instance description ------*
5      Instance class   : cRandomPerformanceTableau
6      Seed             : None
7      Instance name    : cRandomperftab
8      # Actions        : 4
9      # Criteria       : 2
10     Attributes       : ['randomSeed', 'name', 'actions', 'criteria',
11                         'evaluation', 'weightPreorder']
12 >>> t.actions
13     OrderedDict([(1, {'name': '#1'}), (2, {'name': '#2'}),
14                 (3, {'name': '#3'}), (4, {'name': '#4'})])
15 >>> t.criteria
16     OrderedDict([
17     ('g1', {'name': 'RandomPerformanceTableau() instance',
18             'comment': 'Arguments: ; weightDistribution=equisignificant;
19                         weightScale=(1, 1); commonMode=None',
20             'thresholds': {'ind': (10.0, 0.0),
21                            'pref': (20.0, 0.0),
22                            'veto': (80.0, 0.0)},
23             'scale': (0.0, 100.0),
24             'weight': 1,
25             'preferenceDirection': 'max'}),
26     ('g2', {'name': 'RandomPerformanceTableau() instance',
27             'comment': 'Arguments: ; weightDistribution=equisignificant;
28                         weightScale=(1, 1); commonMode=None',
29             'thresholds': {'ind': (10.0, 0.0),
30                            'pref': (20.0, 0.0),
31                            'veto': (80.0, 0.0)},
32             'scale': (0.0, 100.0),
33             'weight': 1,
34             'preferenceDirection': 'max'})])
```

(continues on next page)

```
35  >>> t.evaluation
36      {'g1': {1: 35.17, 2: 56.4, 3: 1.94, 4: 5.51},
37       'g2': {1: 95.12, 2: 90.54, 3: 51.84, 4: 15.42}}
38  >>> t.showPerformanceTableau()
39      Criteria |  'g1'     'g2'
40      Actions  |   1        1
41      ---------|---------------
42         '#1'  |  91.18   90.42
43         '#2'  |  66.82   41.31
44         '#3'  |  35.76   28.86
45         '#4'  |   7.78   37.64
```

Conversions from the Big Data model to the standard model and vice versa are provided.

```
1   >>> t1 = t.convert2Standard()
2   >>> t1.convertWeight2Decimal()
3   >>> t1.convertEvaluation2Decimal()
4   >>> t1
5    *------- PerformanceTableau instance description ------*
6    Instance class    : PerformanceTableau
7    Seed              : None
8    Instance name     : std_cRandomperftab
9    # Actions         : 4
10   # Criteria        : 2
11   Attributes        : ['name', 'actions', 'criteria', 'weightPreorder',
12                        'evaluation', 'randomSeed']
```

12.3 C-implemented integer-valued outranking digraphs

The C compiled version of the bipolar-valued digraph models takes integer relation characteristic values.

```
1   >>> t = cRandomPerformanceTableau(numberOfActions=1000,numberOfCriteria=2)
2   >>> from cIntegerOutrankingDigraphs import *
3   >>> g = IntegerBipolarOutrankingDigraph(t,Threading=True,nbrCores=4)
4   >>> g
5     *------- Object instance description ------*
6     Instance class    : IntegerBipolarOutrankingDigraph
7     Instance name     : rel_cRandomperftab
8     # Actions         : 1000
9     # Criteria        : 2
10    Size              : 465024
11    Determinateness   : 56.877
12    Valuation domain  : {'min': -2, 'med': 0, 'max': 2,
13                        'hasIntegerValuation': True}
14    ----  Constructor run times (in sec.) ----
15    Total time        : 4.23880
16    Data input        : 0.01203
```

```
17      Compute relation : 3.60788
18      Gamma sets       : 0.61889
19      #Threads         : 4
20      Attributes       : ['name', 'actions', 'criteria', 'totalWeight',
21                          'valuationdomain', 'methodData', 'evaluation',
22                          'order', 'runTimes', 'nbrThreads', 'relation',
23                          'gamma', 'notGamma']
```

On a classic intel-i7 equipped PC with four single threaded cores, the `cIntegerOutrankingDigraphs.IntegerBipolarOutrankingDigraph` constructor takes about four seconds for computing a **million** pairwise outranking characteristic values. In a similar setting, the standard `outrankingDigraphs.BipolarOutrankingDigraph` class constructor operates more than two times slower.

```
1   >>> from outrankingDigraphs import BipolarOutrankingDigraph
2   >>> g1 = BipolarOutrankingDigraph(t1,Threading=True,nbrCores=4)
3   >>> g1
4       *------- Object instance description ------*
5       Instance class   : BipolarOutrankingDigraph
6       Instance name    : rel_std_cRandomperftab
7       # Actions        : 1000
8       # Criteria       : 2
9       Size             : 465024
10      Determinateness  : 56.817
11      Valuation domain : {'min': Decimal('-100.0'),
12                          'med': Decimal('0.0'),
13                          'max': Decimal('100.0'),
14                          'precision': Decimal('0')}
15      ----  Constructor run times (in sec.) ----
16      Total time       : 8.63340
17      Data input       : 0.01564
18      Compute relation : 7.52787
19      Gamma sets       : 1.08987
20      #Threads         : 4
```

By far, most of the run time is in each case needed for computing the individual pairwise outranking characteristic values. Notice also below the memory occupations of both outranking digraph instances.

```
1   >>> from digraphsTools import total_size
2   >>> total_size(g)
3    108662777
4   >>> total_size(g1)
5    212679272
6   >>> total_size(g.relation)/total_size(g)
7    0.34
8   >>> total_size(g.gamma)/total_size(g)
9    0.45
```

About 103MB for *g* and 202MB for *g1*. The standard *Decimal* valued

BipolarOutrankingDigraph instance *g1* thus nearly doubles the memory occupation of the corresponding IntegerBipolarOutrankingDigraph *g* instance (see Line 3 and 5 above). 3/4 of this memory occupation is due to the *g.relation* (34%) and the *g.gamma* (45%) dictionaries. And these ratios quadratically grow with the digraph order. To limit the object sizes for really big outranking digraphs, we need to abandon the complete implementation of adjacency tables and gamma functions.

12.4 The sparse outranking digraph implementation

The idea is to first decompose the complete outranking relation into an ordered collection of equivalent quantile performance classes. Let us consider for this illustration a random performance tableau with 100 decision alternatives evaluated on 7 criteria.

```
1  >>> from cRandPerfTabs import *
2  >>> t = cRandomPerformanceTableau(numberOfActions=100,
3                                     numberOfCriteria=7,seed=100)
```

We sort the 100 decision alternatives into overlapping quartile classes and rank with respect to the average quantile limits.

```
1  >>> from cSparseIntegerOutrankingDigraphs import *
2  >>> sg = SparseIntegerOutrankingDigraph(t,quantiles=4)
3  >>> sg
4   *----- Object instance description --------------*
5   Instance class     : SparseIntegerOutrankingDigraph
6   Instance name      : cRandomperftab_mp
7   # Actions          : 100
8   # Criteria         : 7
9   Sorting by         : 4-Tiling
10  Ordering strategy  : average
11  Ranking rule       : Copeland
12  # Components        : 6
13  Minimal order      : 1
14  Maximal order      : 35
15  Average order      : 16.7
16  fill rate          : 24.970%
17   *----  Constructor run times (in sec.) ----
18  Nbr of threads     : 1
19  Total time         : 0.08212
20  QuantilesSorting   : 0.01481
21  Preordering        : 0.00022
22  Decomposing        : 0.06707
23  Ordering           : 0.00000
24  Attributes         : ['runTimes', 'name', 'actions', 'criteria',
25                        'evaluation', 'order', 'dimension',
26                        'sortingParameters', 'nbrOfCPUs',
27                        'valuationdomain', 'profiles', 'categories',
28                        'sorting', 'minimalComponentSize',
29                        'decomposition', 'nbrComponents', 'nd',
```

(continues on next page)

```
30                    'components', 'fillRate',
31                    'maximalComponentSize', 'componentRankingRule',
32                    'boostedRanking']
```

We obtain in this example here a decomposition into 6 linearly ordered components with a maximal component size of 35 for component *c3*.

```
1    >>> sg.showDecomposition()
2     *--- quantiles decomposition in decreasing order---*
3     c1. ]0.75-1.00] : [3, 22, 24, 34, 41, 44, 50, 53, 56, 62, 93]
4     c2. ]0.50-1.00] : [7, 29, 43, 58, 63, 81, 96]
5     c3. ]0.50-0.75] : [1, 2, 5, 8, 10, 11, 20, 21, 25, 28, 30, 33,
6                        35, 36, 45, 48, 57, 59, 61, 65, 66, 68, 70,
7                        71, 73, 76, 82, 85, 89, 90, 91, 92, 94, 95, 97]
8     c4. ]0.25-0.75] : [17, 19, 26, 27, 40, 46, 55, 64, 69, 87, 98, 100]
9     c5. ]0.25-0.50] : [4, 6, 9, 12, 13, 14, 15, 16, 18, 23, 31, 32,
10                        37, 38, 39, 42, 47, 49, 51, 52, 54, 60, 67, 72,
11                        74, 75, 77, 78, 80, 86, 88, 99]
12    c6. ]<-0.25] : [79, 83, 84]
```

A restricted outranking relation is stored for each component with more than one alternative. The resulting global relation map of the first ranked 75 alternatives looks as follows.

```
>>> sg.showRelationMap(toIndex=75)
```
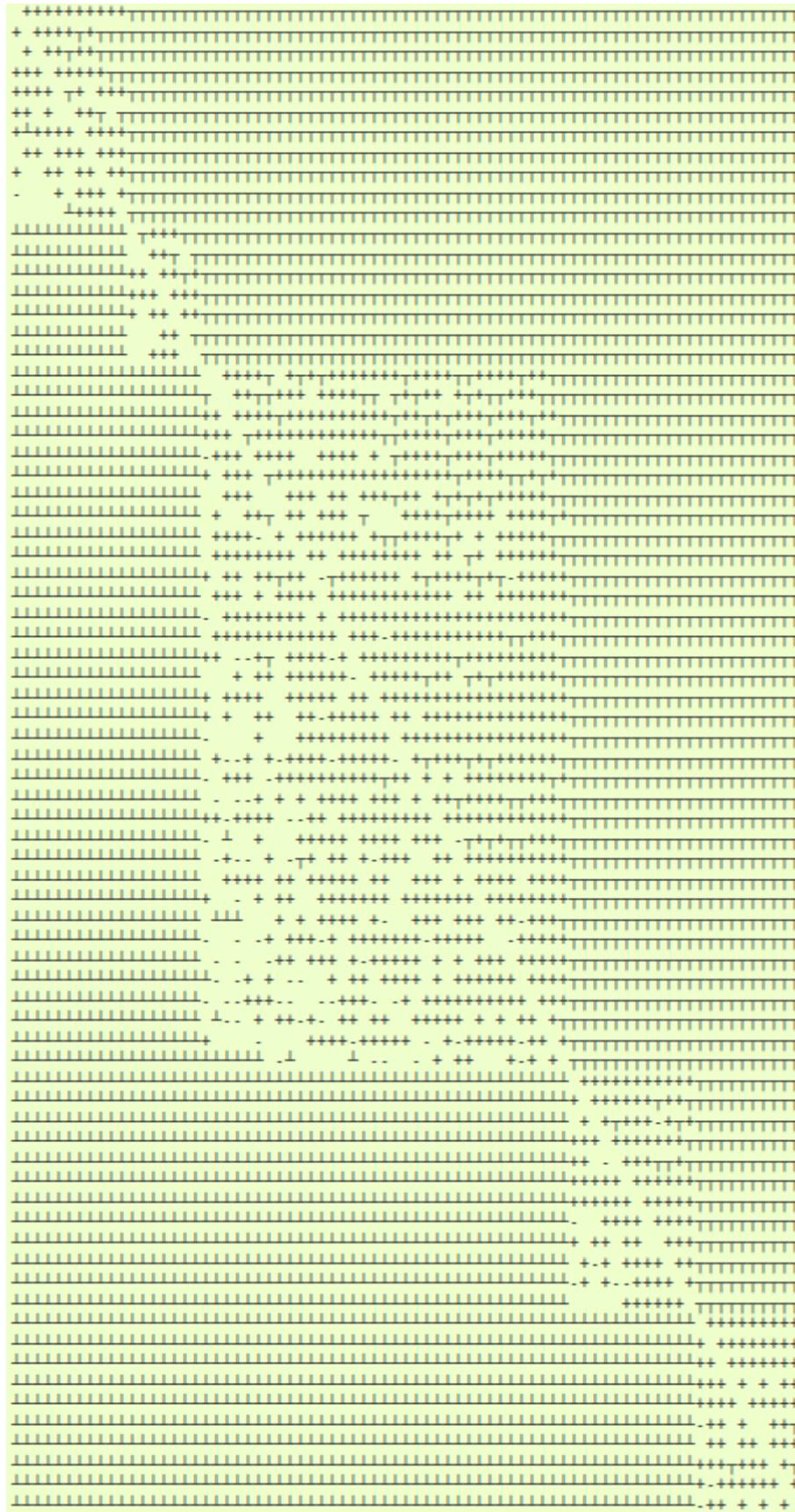
Fig. 1: Sparse quartiles-sorting decomposed outranking relation (extract).

Legend:

- *outranking* for certain (⊤)

- *outranked* for certain (⊥)

- more or less *outranking* (+)

- more or less *outranked* (-)

- *indeterminate* ( )

With a fill rate of 25%, the memory occupation of this sparse outranking digraph *sg* instance takes now only 769kB, compared to the 1.7MB required by a corresponding standard IntegerBipolarOutrankingDigraph instance.

```
>>> print('%.0f kB' % (total_size(sg)/1024) )
769kB
```

For sparse outranking digraphs, the adjacency table is implemented as a dynamic `self.relation(x,y)` function instead of a double dictionary `self.relation[x][y]`.

```python
def relation(self, int x, int y):
    """
    *Parameters*:
        * x (int action key),
        * y (int action key).
    Dynamic construction of the global outranking
    characteristic function *r(x S y)*.
    """
    cdef int Min, Med, Max, rx, ry
    Min = self.valuationdomain['min']
    Med = self.valuationdomain['med']
    Max = self.valuationdomain['max']
    if x == y:
        return Med
    cx = self.actions[x]['component']
    cy = self.actions[y]['component']
    #print(self.components)
    rx = self.components[cx]['rank']
    ry = self.components[cy]['rank']
    if rx == ry:
        try:
            rxpg = self.components[cx]['subGraph'].relation
            return rxpg[x][y]
        except AttributeError:
            componentRanking = self.components[cx]['componentRanking']
            if componentRanking.index(x) < componentRanking.index(x):
                return Max
            else:
                return Min
    elif rx > ry:
        return Min
    else:
        return Max
```

12.5 Ranking big sets of decision alternatives

We may now rank the complete set of 100 decision alternatives by locally ranking with the *Copeland* or the *NetFlows* rule, for instance, all these individual components.

```
1  >>> sg.boostedRanking
2   [22, 53, 3, 34, 56, 62, 24, 44, 50, 93, 41, 63, 29, 58,
3    96, 7, 43, 81, 91, 35, 25, 76, 66, 65, 8, 10, 1, 11, 61,
4    30, 48, 45, 68, 5, 89, 57, 59, 85, 82, 73, 33, 94, 70,
5    97, 20, 92, 71, 90, 95, 21, 28, 2, 36, 87, 40, 98, 46, 55,
6    100, 64, 17, 26, 27, 19, 69, 6, 38, 4, 37, 60, 31, 77, 78,
7    47, 99, 18, 12, 80, 54, 88, 39, 9, 72, 86, 42, 13, 23, 67,
8    52, 15, 32, 49, 51, 74, 16, 14, 75, 79, 83, 84]
```

When actually computing linear rankings of a set of alternatives, the local outranking relations are of no practical usage, and we may furthermore reduce the memory occupation of the resulting digraph by

1. refining the ordering of the quantile classes by taking into account how well an alternative is outranking the lower limit of its quantile class, respectively the upper limit of its quantile class is *not* outranking the alternative;

2. dropping the local outranking digraphs and keeping for each quantile class only a locally ranked list of alternatives.

We provide therefore the `cSparseIntegerOutrankingDigraphs.cQuantilesRankingDigraph` class.

```
1  >>> qr = cQuantilesRankingDigraph(t,4)
2  >>> qr
3   *----- Object instance description --------------*
4   Instance class     : cQuantilesRankingDigraph
5   Instance name      : cRandomperftab_mp
6   # Actions          : 100
7   # Criteria         : 7
8   Sorting by         : 4-Tiling
9   Ordering strategy  : optimal
10  Ranking rule       : Copeland
11  # Components       : 47
12  Minimal order      : 1
13  Maximal order      : 10
14  Average order      : 2.1
15  fill rate          : 2.566%
16  *----  Constructor run times (in sec.) ----*
17  Nbr of threads     : 1
18  Total time         : 0.03702
19  QuantilesSorting   : 0.01785
20  Preordering        : 0.00022
21  Decomposing        : 0.01892
22  Ordering           : 0.00000
23  Attributes         : ['runTimes', 'name', 'actions', 'order',
```

```
24                    'dimension', 'sortingParameters', 'nbrOfCPUs',
25                    'valuationdomain', 'profiles', 'categories',
26                    'sorting', 'minimalComponentSize',
27                    'decomposition', 'nbrComponents', 'nd',
28                    'components', 'fillRate', 'maximalComponentSize',
29                    'componentRankingRule', 'boostedRanking']
```

With this *optimised* quantile ordering strategy, we obtain now 47 performance equivalence classes.

```
1  >>> qr.components
2   OrderedDict([
3   ('c01', {'rank': 1,
4           'lowQtileLimit': ']0.75',
5           'highQtileLimit': '1.00]',
6           'componentRanking': [53]}),
7   ('c02', {'rank': 2,
8           'lowQtileLimit': ']0.75',
9           'highQtileLimit': '1.00]',
10          'componentRanking': [3, 23, 63, 50]}),
11  ('c03', {'rank': 3,
12          'lowQtileLimit': ']0.75',
13          'highQtileLimit': '1.00]',
14          'componentRanking': [34, 44, 56, 24, 93, 41]}),
15  ...
16  ...
17  ...
18  ('c45', {'rank': 45,
19          'lowQtileLimit': ']0.25',
20          'highQtileLimit': '0.50]',
21          'componentRanking': [49]}),
22  ('c46', {'rank': 46,
23          'lowQtileLimit': ']0.25',
24          'highQtileLimit': '0.50]',
25          'componentRanking': [52, 16, 86]}),
26  ('c47', {'rank': 47,
27          'lowQtileLimit': ']<',
28          'highQtileLimit': '0.25]',
29          'componentRanking': [79, 83, 84]})])
30  >>> print('%.0f kB' % (total_size(qr)/1024) )
31   208kB
```

We observe an even more considerably less voluminous memory occupation: 208kB compared to the 769kB of the SparseIntegerOutrankingDigraph instance. It is opportune, however, to measure the loss of quality of the resulting *Copeland* ranking when working with sparse outranking digraphs.

```
1  >>> from cIntegerOutrankingDigraphs import *
2  >>> ig = IntegerBipolarOutrankingDigraph(t)
```

```
3  >>> print('Complete outranking : %+.4f'\
4          % (ig.computeOrderCorrelation(ig.computeCopelandOrder())\
5             ['correlation']))
6   Complete outranking : +0.7474
7  >>> print('Sparse 4-tiling : %+.4f'\
8          % (ig.computeOrderCorrelation(\
9             list(reversed(sg.boostedRanking)))['correlation']))
10  Sparse 4-tiling        : +0.7172
11  >>> print('Optimzed sparse 4-tiling: %+.4f'\
12           % (ig.computeOrderCorrelation(\
13              list(reversed(qr.boostedRanking)))['correlation']))
14  Optimzed sparse 4-tiling: +0.7051
```

The best ranking correlation with the pairwise outranking situations ($+0.75$) is naturally given when we apply the *Copeland* rule to the complete outranking digraph. When we apply the same rule to the sparse 4-tiled outranking digraph, we get a correlation of $+0.72$, and when applying the *Copeland* rule to the optimised 4-tiled digraph, we still obtain a correlation of $+0.71$. These results actually depend on the number of quantiles we use as well as on the given model of random performance tableau. In case of Random3ObjectivesPerformanceTableau instances, for instance, we would get in a similar setting a complete outranking correlation of $+0.86$, a sparse 4-tiling correlation of $+0.82$, and an optimzed sparse 4-tiling correlation of $+0.81$.

12.6 HPC quantiles ranking records

Following from the separability property of the q-tiles sorting of each action into each q-tiles class, the q-sorting algorithm may be safely split into as much threads as are multiple processing cores available in parallel. Furthermore, the ranking procedure being local to each diagonal component, these procedures may as well be safely processed in parallel threads on each component restricted outrankingdigraph.

Using the HPC platform of the University of Luxembourg (https://hpc.uni.lu/), the following run times for very big ranking problems could be achieved both:

- on Iris -skylake nodes with 28 cores[7], and

- on the 3TB -bigmem Gaia-183 node with 64 cores[8],

by running the cythonized python modules in an Intel compiled virtual Python 3.6.5 environment [GCC Intel(R) 17.0.1 –enable-optimizations c++ gcc 6.3 mode] on Debian 8 Linux.

---

[7] See https://hpc.uni.lu/systems/iris/
[8] See https://hpc.uni.lu/systems/gaia/

Fig. 2: HPC-UL Ranking Performance Records (Spring 2018)

Example python session on the HPC-UL Iris-126 -skylake node[7]

```
(myPy365ICC) [rbisdorff@iris-126 Test]$ python
Python 3.6.5 (default, May  9 2018, 09:54:28)
[GCC Intel(R) C++ gcc 6.3 mode] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```
>>> from cRandPerfTabs import\
            cRandom3ObjectivesPerformanceTableau as cR3ObjPT
>>> pt = cR3ObjPT(numberOfActions=1000000,
            numberOfCriteria=21,
            weightDistribution='equiobjectives',
            commonScale = (0.0,1000.0),
            commonThresholds = [(2.5,0.0),(5.0,0.0),(75.0,0.0)],
            commonMode = ['beta','variable',None],
            missingDataProbability=0.05,
            seed=16)
>>> import cSparseIntegerOutrankingDigraphs as iBg
>>> qr = iBg.cQuantilesRankingDigraph(pt,quantiles=10,
            quantilesOrderingStrategy='optimal',
            minimalComponentSize=1,
            componentRankingRule='NetFlows',
            LowerClosed=False,
            Threading=True,
            tempDir='/tmp',
            nbrOfCPUs=28)
>>> qr
 *----- Object instance description --------------*
 Instance class    : cQuantilesRankingDigraph
 Instance name     : random3ObjectivesPerfTab_mp
 # Actions         : 1000000
 # Criteria        : 21
 Sorting by        : 10-Tiling
 Ordering strategy : optimal
 Ranking rule      : NetFlows
 # Components      : 233645
 Minimal order     : 1
```

```
31   Maximal order      : 153
32   Average order      : 4.3
33   fill rate          : 0.001%
34   *----  Constructor run times (in sec.) ----*
35   Nbr of threads     : 28
36   Total time         : 177.02770
37   QuantilesSorting   : 99.55377
38   Preordering        : 5.17954
39   Decomposing        : 72.29356
```

On this 2x14c Intel Xeon Gold 6132 @ 2.6 GHz equipped HPC node with 132GB RAM[7], deciles sorting and locally ranking a **million** decision alternatives evaluated on 21 incommensurable criteria, by balancing an economic, an environmental and a societal decision objective, takes us about **3 minutes** (see Lines 37-42 above); with 1.5 minutes for the deciles sorting and, a bit more than one minute, for the local ranking of the individual components.

The optimised deciles sorting leads to 233645 components (see Lines 32-36 above) with a maximal order of 153. The fill rate of the adjacency table is reduced to 0.001%. Of the potential trillion (10^{12}) pairwise outrankings, we effectively keep only 10 millions (10^7). This high number of components results from the high number of involved performance criteria (21), leading in fact to a very refined epistemic discrimination of majority outranking margins.

A non-optimised deciles sorting would instead give at most 110 components with inevitably very big intractable local digraph orders. Proceeding with a more detailed quantiles sorting, for reducing the induced decomposing run times, leads however quickly to intractable quantiles sorting times. A good compromise is given when the quantiles sorting and decomposing steps show somehow equivalent run times; as is the case in our example session: 99.6 versus 77.3 seconds (see Lines 40 and 42 above).

Let us inspect the 21 marginal performances of the five best-ranked alternatives listed below.

```
1    >>> pt.showPerformanceTableau(\
2                    actionsSubset=qr.boostedRanking[:5],\
3                    Transposed=True)
4     *----  performance tableau -----*
5     criteria | weights |  #773909  #668947  #567308  #578560  #426464
6     ---------|-------------------------------------------------------
7      'Ec01'  |    42   |  969.81   844.71   917.00     NA     808.35
8      'So02'  |    48   |    NA     891.52   836.43     NA     899.22
9      'En03'  |    56   |  687.10     NA     503.38   873.90     NA
10     'So04'  |    48   |  455.05   845.29   866.16   800.39   956.14
11     'En05'  |    56   |  809.60   846.87   939.46   851.83   950.51
12     'Ec06'  |    42   |  919.62   802.45   717.39   832.44   974.63
13     'Ec07'  |    42   |  889.01   722.09   606.11   902.28   574.08
14     'So08'  |    48   |  862.19   699.38   907.34   571.18   943.34
15     'En09'  |    56   |  857.34   817.44   819.92   674.60   376.70
```

143

```
16    'Ec10' |    42   |    NA    874.86    NA     847.75   739.94
17    'En11' |    56   |    NA    824.24   855.76    NA     953.77
18    'Ec12' |    42   |  802.18  871.06   488.76  841.41   599.17
19    'En13' |    56   |  827.73  839.70   864.48  720.31   877.23
20    'So14' |    48   |  943.31  580.69   827.45  815.18   461.04
21    'En15' |    56   |  794.57  801.44   924.29  938.70   863.72
22    'Ec16' |    42   |  581.15  599.87   949.84  367.34   859.70
23    'So17' |    48   |  881.55  856.05    NA     796.10   655.37
24    'Ec18' |    42   |  863.44  520.24   919.75  865.14   914.32
25    'So19' |    48   |    NA      NA      NA     790.43   842.85
26    'Ec20' |    42   |  582.52  831.93   820.92  881.68   864.81
27    'So21' |    48   |  880.87    NA     628.96  746.67   863.82
```

The given ranking problem involves 8 criteria assessing the economic performances, 7 criteria assessing the societal performances and 6 criteria assessing the environmental performances of the decision alternatives. The sum of criteria significance weights (336) is the same for all three decision objectives. The five best-ranked alternatives are, in decreasing order: #773909, #668947, #567308, #578560 and #426464.

Their random performance evaluations were obviously drawn on all criteria with a *good* (+) performance profile, i.e. a Beta($alpha = 5.8661$, $beta = 2.62203$) law (see the tutorial *Generating random performance tableaux* (page 32)).

```
1  >>> for x in qr.boostedRanking[:5]:
2          print(pt.actions[x]['name'],\
3              pt.actions[x]['profile'])
4   #773909 {'Eco': '+', 'Soc': '+', 'Env': '+'}
5   #668947 {'Eco': '+', 'Soc': '+', 'Env': '+'}
6   #567308 {'Eco': '+', 'Soc': '+', 'Env': '+'}
7   #578560 {'Eco': '+', 'Soc': '+', 'Env': '+'}
8   #426464 {'Eco': '+', 'Soc': '+', 'Env': '+'}
```

We consider now a partial performance tableau *best10*, consisting only, for instance, of the **ten best-ranked alternatives**, with which we may compute a corresponding integer outranking digraph valued in the range (-1008, +1008).

```
1  >>> best10 = cPartialPerformanceTableau(pt,qr.boostedRanking[:10])
2  >>> from cIntegerOutrankingDigraphs import *
3  >>> g = IntegerBipolarOutrankingDigraph(best10)
4  >>> g.valuationdomain
5   {'min': -1008, 'med': 0, 'max': 1008, 'hasIntegerValuation': True}
6  >>> g.showRelationTable(ReflexiveTerms=False)
7   * ---- Relation Table -----
8    r(x>y) | #773909 #668947 #567308 #578560 #426464 #298061 #155874 #815552
   ↪#279729 #928564
9    --------|---------------------------------------------------------------------
   ↪-------------
10   #773909 |    -      +390     +90     +270     -50     +340    +220     +60    ␣
   ↪+116     +222
```

```
11   #668947 |     +78       -        +42     +250      -22     +218      +56     +172     ␣
     ↪+74      +64
12   #567308 |     +70     +418       -       +180     +156     +174     +266      +78     ␣
     ↪+256     +306
13   #578560 |      -4      +78      +28       -        -12     +100      -48     +154      -
     ↪110      -10
14   #426464 |    +202     +258     +284     +138       -       +416     +312     +382     ␣
     ↪+534     +278
15   #298061 |     -48      +68     +172      +32      -42       -        +54      +48     ␣
     ↪+248     +374
16   #155874 |     +72     +378     +322     +174     +274     +466       -       +212     ␣
     ↪+308     +418
17   #815552 |     +78     +126     +272     +318      +54     +194     +172       -        -
     ↪14       +22
18   #279729 |    +240     +230     -110     +290      +72     +140     +388      +62      -␣
     ↪      +250
19   #928564 |     +22     +228      -14     +246      +36      +78      +56     +110     ␣
     ↪+318       -
20   r(x>y) image range := [-1008;+1008]
21   >>> g.condorcetWinners()
22   [155874, 426464, 567308]
23   >>> g.computeChordlessCircuits()
24   []
25   >>> g.computeTransitivityDegree()
26   0.78
```

Three alternatives -#155874, #426464 and #567308- qualify as Condorcet winners, i.e. they each **positively outrank** all the other nine alternatives. No chordless outranking circuits are detected, yet the transitivity of the apparent outranking relation is not given. And, no clear ranking alignment hence appears when inspecting the *strict* outranking digraph (i.e. the codual ˜(-g) of g) shown in Fig. 3.

```
1   >>> (˜(-g)).exportGraphViz()
2   *---- exporting a dot file for GraphViz tools ---------*
3   Exporting to converse-dual_rel_best10.dot
4   dot -Tpng converse-dual_rel_best10.dot -o converse-dual_rel_best10.png
```

Rubis Python Server (graphviz), R. Bisdorff, 2008
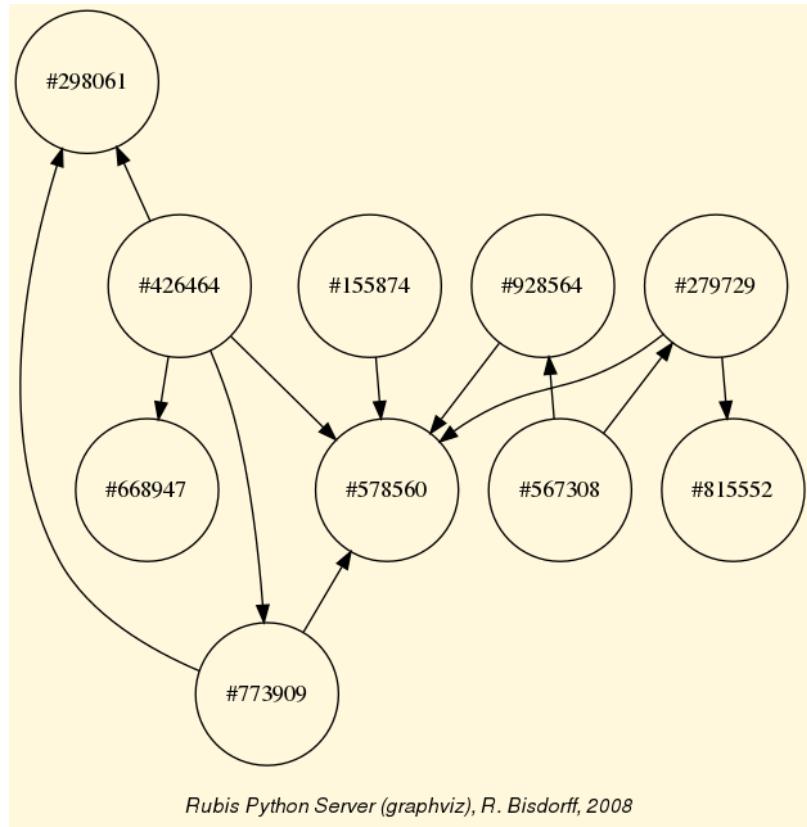
Fig. 3: Validated *strict* outranking situations between the ten best-ranked alternatives

Restricted to these ten best-ranked alternatives, the *Copeland*, the *NetFlows* as well as the *Kemeny* ranking rule will all rank alternative #426464 first and alternative #578560 last. Otherwise the three ranking rules produce in this case more or less different rankings.

```
>>> g.computeCopelandRanking()
 [426464, 567308, 155874, 279729, 773909, 928564, 668947, 815552, 298061,␣
↪578560]
>>> g.computeNetFlowsRanking()
 [426464, 155874, 773909, 567308, 815552, 279729, 928564, 298061, 668947,␣
↪578560]
>>> from linearOrders import *
>>> ke = KemenyOrder(g,orderLimit=10)
>>> ke.kemenyRanking
 [426464, 773909, 155874, 815552, 567308, 298061, 928564, 279729, 668947,␣
↪578560]
```

Note: It is therefore *important* to always keep in mind that, based on pairwise outranking situations, there **does not exist** any **unique optimal ranking**; especially when we face such big data problems. Changing the number of quantiles, the component ranking rule, the optimised quantile ordering strategy, all this will indeed produce, sometimes even substantially, diverse global ranking results.

Back to *Content Table* (page 3)

13 Working with the `graphs` module

See also the technical documentation of the graphs module.

13.1 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 vertex, automatically added by the object constructor.

See the technical documentation of the graphs module.

Example Python3 session

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

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

```
1  # Graph instance saved in Python format
2  vertices = {
3  'v1': {'shortName': 'v1', 'name': 'random vertex'},
```

147

```
4    'v2': {'shortName': 'v2', 'name': 'random vertex'},
5    'v3': {'shortName': 'v3', 'name': 'random vertex'},
6    'v4': {'shortName': 'v4', 'name': 'random vertex'},
7    'v5': {'shortName': 'v5', 'name': 'random vertex'},
8    'v6': {'shortName': 'v6', 'name': 'random vertex'},
9    'v7': {'shortName': 'v7', 'name': 'random vertex'},
10   }
11   valuationDomain = {'min':-1,'med':0,'max':1}
12   edges = {
13   frozenset(['v1','v2']) : -1,
14   frozenset(['v1','v3']) : -1,
15   frozenset(['v1','v4']) : -1,
16   frozenset(['v1','v5']) : 1,
17   frozenset(['v1','v6']) : -1,
18   frozenset(['v1','v7']) : -1,
19   frozenset(['v2','v3']) : 1,
20   frozenset(['v2','v4']) : 1,
21   frozenset(['v2','v5']) : -1,
22   frozenset(['v2','v6']) : 1,
23   frozenset(['v2','v7']) : -1,
24   frozenset(['v3','v4']) : -1,
25   frozenset(['v3','v5']) : -1,
26   frozenset(['v3','v6']) : -1,
27   frozenset(['v3','v7']) : -1,
28   frozenset(['v4','v5']) : 1,
29   frozenset(['v4','v6']) : -1,
30   frozenset(['v4','v7']) : 1,
31   frozenset(['v5','v6']) : 1,
32   frozenset(['v5','v7']) : -1,
33   frozenset(['v6','v7']) : -1,
34   }
```

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

```
1    >>> g = Graph('tutorialGraph')
2    >>> g.exportGraphViz()
3     *---- exporting a dot file for GraphViz tools ---------*
4     Exporting to tutorialGraph.dot
5     fdp -Tpng tutorialGraph.dot -o tutorialGraph.png
```
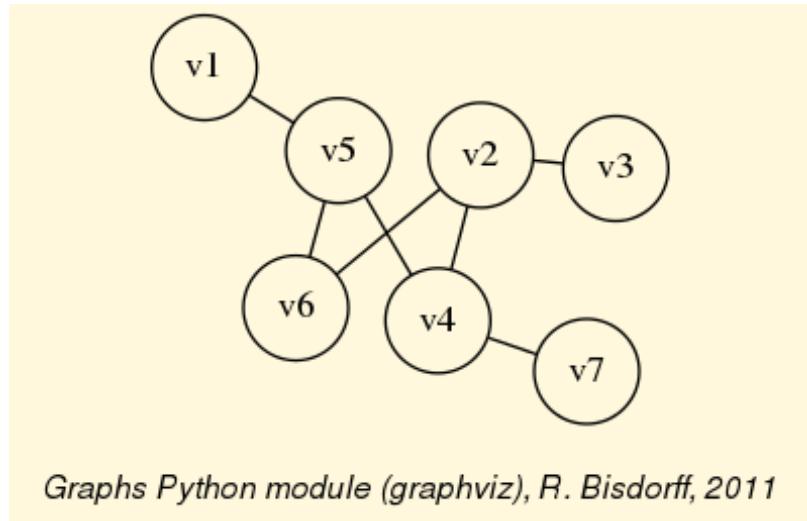
Fig. 1: Tutorial graph instance

Properties, like the gamma function and vertex degrees and neighbourhood depths 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'
 -----|------------------------------------------
```

```
6    'v1' |    -    -1   -1   -1    1   -1   -1
7    'v2' |   -1    -    1    1   -1    1   -1
8    'v3' |   -1    1    -   -1   -1   -1   -1
9    'v4' |   -1    1   -1    -    1   -1    1
10   'v5' |    1   -1   -1    1    -    1   -1
11   'v6' |   -1    1   -1   -1    1    -   -1
12   'v7' |   -1   -1   -1    1   -1   -1    -
13   >>> g1 = dg.digraph2Graph()
14   >>> g1.showShort()
15   *---- short description of the graph ----*
16   Name              : 'tutorialGraph'
17   Vertices          : ['v1', 'v2', 'v3', 'v4', 'v5', 'v6', 'v7']
18   Valuation domain :  {'med': 0, 'min': -1, 'max': 1}
19   Gamma function    :
20   v1 -> ['v5']
21   v2 -> ['v3', 'v6', 'v4']
22   v3 -> ['v2']
23   v4 -> ['v5', 'v7', 'v2']
24   v5 -> ['v6', 'v1', 'v4']
25   v6 -> ['v5', 'v2']
26   v7 -> ['v4']
27   degrees       :  [0, 1, 2, 3, 4, 5, 6]
28   distribution :  [0, 3, 1, 3, 0, 0, 0]
29   nbh depths    :  [0, 1, 2, 3, 4, 5, 6, 'inf.']
30   distribution :  [0, 0, 1, 4, 2, 0, 0, 0]
```

13.2 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.

```
1    >>> from graphs import Q_Coloring
2    >>> qc = Q_Coloring(g)
3     Running a Gibbs Sampler for 42 step !
4     The q-coloring with 3 colors is feasible !!
5    >>> qc.showConfiguration()
6     v5 lightblue
7     v3 gold
8     v7 gold
9     v2 lightblue
10    v4 lightcoral
11    v1 gold
12    v6 lightcoral
13   >>> qc.exportGraphViz('tutorial-3-coloring')
14   *---- exporting a dot file for GraphViz tools ---------*
15    Exporting to tutorial-3-coloring.dot
16    fdp -Tpng tutorial-3-coloring.dot -o tutorial-3-coloring.png
```
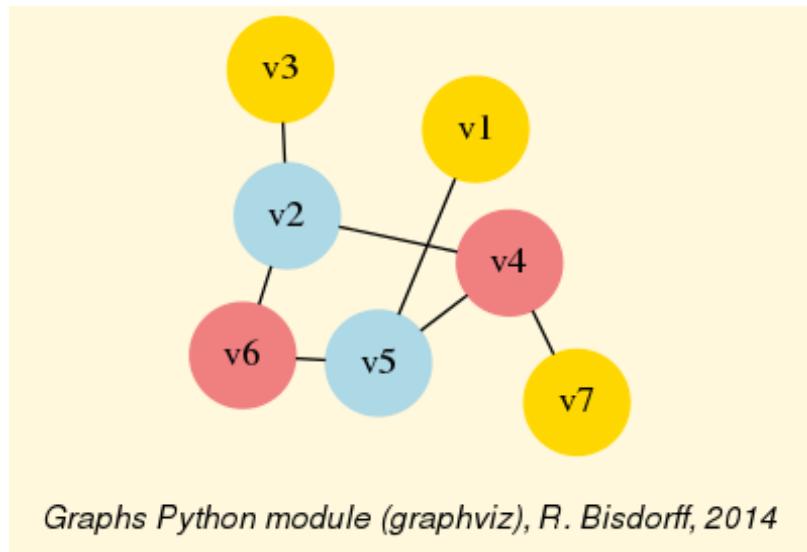
Fig. 2: 3-Coloring of the tutorial graph

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

```
1  >>> qc = Q_Coloring(g,colors=['gold','coral'])
2   Running a Gibbs Sampler for 42 step !
3   The q-coloring with 2 colors is feasible !!
4  >>> qc.showConfiguration()
5   v5 gold
6   v3 coral
7   v7 gold
8   v2 gold
9   v4 coral
10  v1 coral
11  v6 coral
12  >>> qc.exportGraphViz('tutorial-2-coloring')
13  Exporting to tutorial-2-coloring.dot
14  fdp -Tpng tutorial-2-coloring.dot -o tutorial-2-coloring.png
```
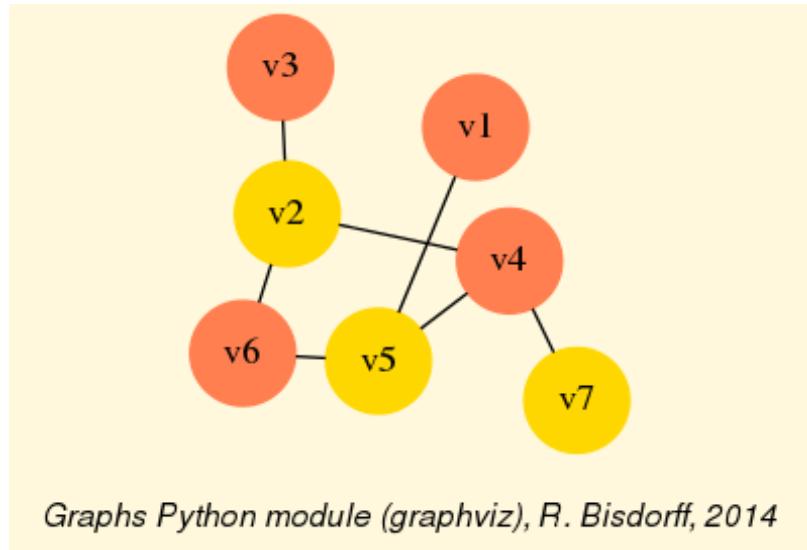
Fig. 3: 2-coloring of the tutorial graph

13.3 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 $-g^{14}$), corresponds to a maximal **clique**, i.e. a maximal complete subgraph in g. Maximal cliques may be directly enumerated with the *graphs.Graph.showCliques()* method.

```
1  >>> g.showCliques()
2   *---  Maximal Cliques ---*
3   ['v2', 'v3']
4   ['v4', 'v7']
5   ['v2', 'v4']
6   ['v4', 'v5']
7   ['v1', 'v5']
8   ['v2', 'v6']
9   ['v5', 'v6']
10  number of solutions:  7
11  cardinality distribution
12  card.:  [0, 1, 2, 3, 4, 5, 6, 7]
13  freq.:  [0, 0, 7, 0, 0, 0, 0, 0]
14  execution time: 0.00049 sec.
15  Results in self.cliques
16  >>> g.cliques
17  [frozenset({'v2', 'v3'}), frozenset({'v4', 'v7'}),
18   frozenset({'v2', 'v4'}), frozenset({'v4', 'v5'}),
19   frozenset({'v1', 'v5'}), frozenset({'v6', 'v2'}),
20   frozenset({'v6', 'v5'})]
```

13.4 Line graphs and maximal matchings

The module also provides a `graphs.LineGraph` constructor. A **line graph** represents the **adjacencies between edges** of the given graph instance. We may compute for instance the line graph of the 5-cycle graph.

```
1  >>> g = CycleGraph(order=5)
2  >>> g
3   *------- Graph instance description ------*
4   Instance class   : CycleGraph
5   Instance name    : cycleGraph
6   Graph Order      : 5
7   Graph Size       : 5
8   Valuation domain : [-1.00; 1.00]
9   Attributes       : ['name', 'order', 'vertices', 'valuationDomain',
10                      'edges', 'size', 'gamma']
11  >>> lg = LineGraph(g)
12  >>> lg
13   *------- Graph instance description ------*
14   Instance class   : LineGraph
15   Instance name    : line-cycleGraph
16   Graph Order      : 5
17   Graph Size       : 5
18   Valuation domain : [-1.00; 1.00]
19   Attributes       : ['name', 'graph', 'valuationDomain', 'vertices',
20                      'order', 'edges', 'size', 'gamma']
21  >>> lg.showShort()
```

153

```
22   *---- short description of the graph ----*
23   Name             : 'line-cycleGraph'
24   Vertices         : [frozenset({'v1', 'v2'}), frozenset({'v1', 'v5'}),␣
     ↪frozenset({'v2', 'v3'}),
25                      frozenset({'v3', 'v4'}), frozenset({'v4', 'v5'})]
26   Valuation domain : {'min': Decimal('-1'), 'med': Decimal('0'), 'max': Decimal(
     ↪'1')}
27   Gamma function   :
28   frozenset({'v1', 'v2'}) -> [frozenset({'v2', 'v3'}), frozenset({'v1', 'v5'})]
29   frozenset({'v1', 'v5'}) -> [frozenset({'v1', 'v2'}), frozenset({'v4', 'v5'})]
30   frozenset({'v2', 'v3'}) -> [frozenset({'v1', 'v2'}), frozenset({'v3', 'v4'})]
31   frozenset({'v3', 'v4'}) -> [frozenset({'v2', 'v3'}), frozenset({'v4', 'v5'})]
32   frozenset({'v4', 'v5'}) -> [frozenset({'v4', 'v3'}), frozenset({'v1', 'v5'})]
33   degrees      : [0, 1, 2, 3, 4]
34   distribution : [0, 0, 5, 0, 0]
35   nbh depths   : [0, 1, 2, 3, 4, 'inf.']
36   distribution : [0, 0, 5, 0, 0, 0]
```

Iterated line graph constructions are usually expanding, except for *chordless cycles*, where the same cycle is repeated, and for *non-closed paths*, where iterated line graphs progressively reduce one by one the number of vertices and edges and become eventually an empty graph.

Notice that the MISs in the line graph provide **maximal matchings** - *maximal sets of independent edges* - of the original graph.

```
1   >>> c8 = CycleGraph(order=8)
2   >>> lc8 = LineGraph(c8)
3   >>> lc8.showMIS()
4    *---  Maximal Independent Sets ---*
5    [frozenset({'v3', 'v4'}), frozenset({'v5', 'v6'}), frozenset({'v1', 'v8'})]
6    [frozenset({'v2', 'v3'}), frozenset({'v5', 'v6'}), frozenset({'v1', 'v8'})]
7    [frozenset({'v8', 'v7'}), frozenset({'v2', 'v3'}), frozenset({'v5', 'v6'})]
8    [frozenset({'v8', 'v7'}), frozenset({'v2', 'v3'}), frozenset({'v4', 'v5'})]
9    [frozenset({'v7', 'v6'}), frozenset({'v3', 'v4'}), frozenset({'v1', 'v8'})]
10   [frozenset({'v2', 'v1'}), frozenset({'v8', 'v7'}), frozenset({'v4', 'v5'})]
11   [frozenset({'v2', 'v1'}), frozenset({'v7', 'v6'}), frozenset({'v4', 'v5'})]
12   [frozenset({'v2', 'v1'}), frozenset({'v7', 'v6'}), frozenset({'v3', 'v4'})]
13   [frozenset({'v7', 'v6'}), frozenset({'v2', 'v3'}), frozenset({'v1', 'v8'}),
14    frozenset({'v4', 'v5'})]
15   [frozenset({'v2', 'v1'}), frozenset({'v8', 'v7'}), frozenset({'v3', 'v4'}),
16    frozenset({'v5', 'v6'})]
17   number of solutions:  10
18   cardinality distribution
19   card.:  [0, 1, 2, 3, 4, 5, 6, 7, 8]
20   freq.:  [0, 0, 0, 8, 2, 0, 0, 0, 0]
21   execution time: 0.00029 sec.
```

The two last MISs of cardinality 4 (see Lines 13-16 above) give **isomorphic perfect maximum matchings** of the 8-cycle graph. Every vertex of the cycle is adjacent to a

matching edge. Odd cycle graphs do not admit any perfect matching.

```
1  >>> maxMatching = c8.computeMaximumMatching()
2  >>> c8.exportGraphViz(fileName='maxMatchingcycleGraph',
3                        matching=maxMatching)
4   *---- exporting a dot file for GraphViz tools ---------*
5   Exporting to maxMatchingcycleGraph.dot
6   Matching:  {frozenset({'v1', 'v2'}), frozenset({'v5', 'v6'}),
7              frozenset({'v3', 'v4'}), frozenset({'v7', 'v8'}) }
8   circo -Tpng maxMatchingcyleGraph.dot -o maxMatchingcyleGraph.png
```
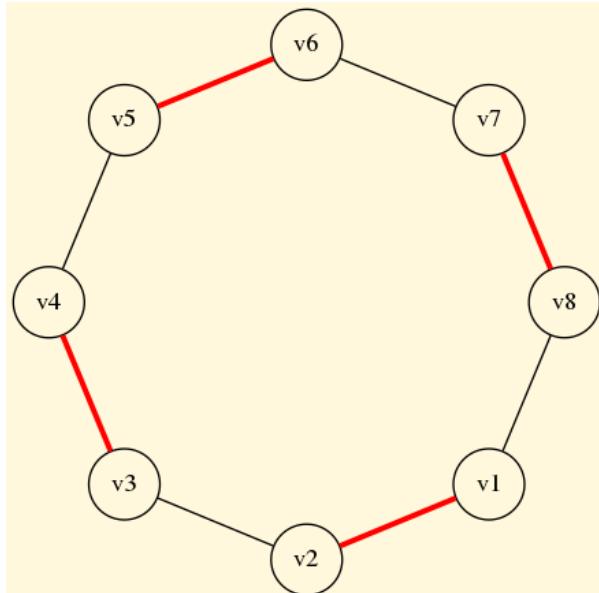

Graphs Python module (graphviz), R. Bisdorff, 2015

Fig. 4: A perfect maximum matching of the 8-cycle graph

13.5 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.

```
1  >>> from graphs import GridGraph, IsingModel
2  >>> g = GridGraph(n=15,m=15)
3  >>> g.showShort()
4   *----- show short --------------*
5   Grid graph     :  grid-6-6
6   n              :  6
7   m              :  6
8   order          :  36
9  >>> im = IsingModel(g,beta=0.3,nSim=100000,Debug=False)
10  Running a Gibbs Sampler for 100000 step !
11 >>> im.exportGraphViz(colors=['lightblue','lightcoral'])
```

```
12   *---- exporting a dot file for GraphViz tools ---------*
13   Exporting to grid-15-15-ising.dot
14   fdp -Tpng grid-15-15-ising.dot -o grid-15-15-ising.png
```
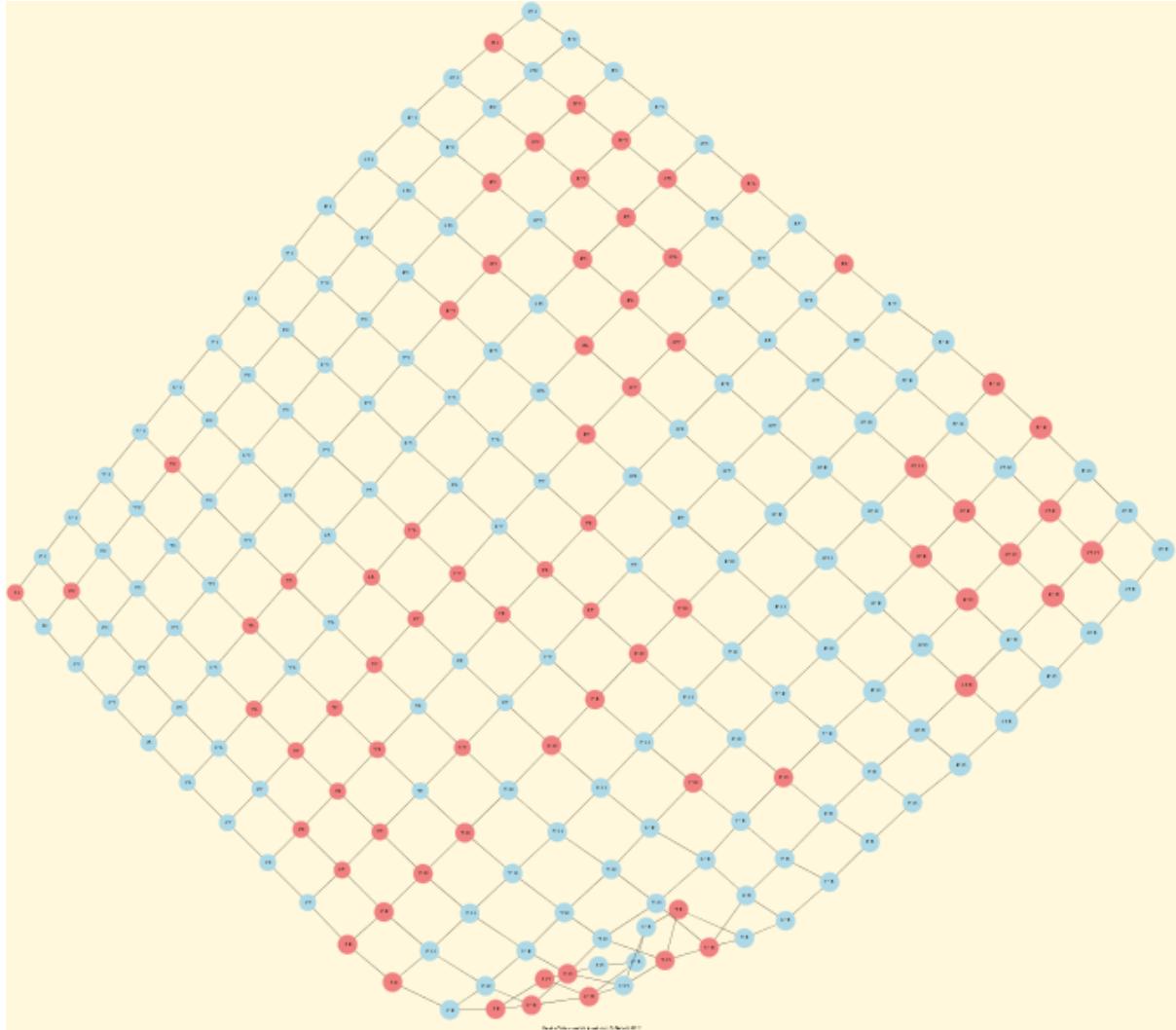

Fig. 5: Ising model of the 15x15 grid graph

13.6 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']
14   >>> probs = {}  # initialize a potential stationary probability vector
15   >>> n = g.order # for instance: probs[v_i] = n-i/Sum(1:n) for i in 1:n
16   >>> i = 0
17   >>> verticesList = [x for x in g.vertices]
18   >>> verticesList.sort()
19   >>> for v in verticesList:
20           probs[v] = (n - i)/(n*(n+1)/2)
21           i += 1
22   >>> met = MetropolisChain(g,probs)
23   >>> frequency = met.checkSampling(verticesList[0],nSim=30000)
24   >>> for v in verticesList:
25           print(v,probs[v],frequency[v])
26    v1 0.3333 0.3343
27    v2 0.2666 0.2680
28    v3 0.2    0.2030
29    v4 0.1333 0.1311
30    v5 0.0666 0.0635
31   >>> met.showTransitionMatrix()
32   * ---- Transition Matrix -----
33     Pij  | 'v1'    'v2'    'v3'    'v4'    'v5'
34     -----|-----------------------------------
35     'v1' |  0.23    0.33    0.30    0.13    0.00
36     'v2' |  0.42    0.42    0.00    0.17    0.00
37     'v3' |  0.50    0.00    0.33    0.00    0.17
38     'v4' |  0.33    0.33    0.00    0.08    0.25
39     '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].

Back to *Content Table* (page 3)

14 Computing the non isomorphic MISs of the 12-cycle graph

- *Introduction* (page 158)
- *Computing the maximal independent sets (MISs)* (page 159)
- *Computing the automorphism group* (page 161)
- *Computing the isomorphic MISs* (page 161)

14.1 Introduction

Due to the public success of our common 2008 publication with Jean-Luc Marichal [ISOMIS-08] , we present in this 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.

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

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

```
1  >>> from graphs import CycleGraph
2  >>> cg12 = CycleGraph(order=12)
3  >>> cg12
4   *------- Graph instance description ------*
5   Instance class    : CycleGraph
6   Instance name     : cycleGraph
7   Graph Order       : 12
8   Graph Size        : 12
9   Valuation domain  : [-1.0, 1.0]
10  Attributes        : ['name', 'order', 'vertices', 'valuationDomain',
```

```
11                      'edges', 'size', 'gamma']
12  >>> cg12.exportGraphViz('cg12')
```
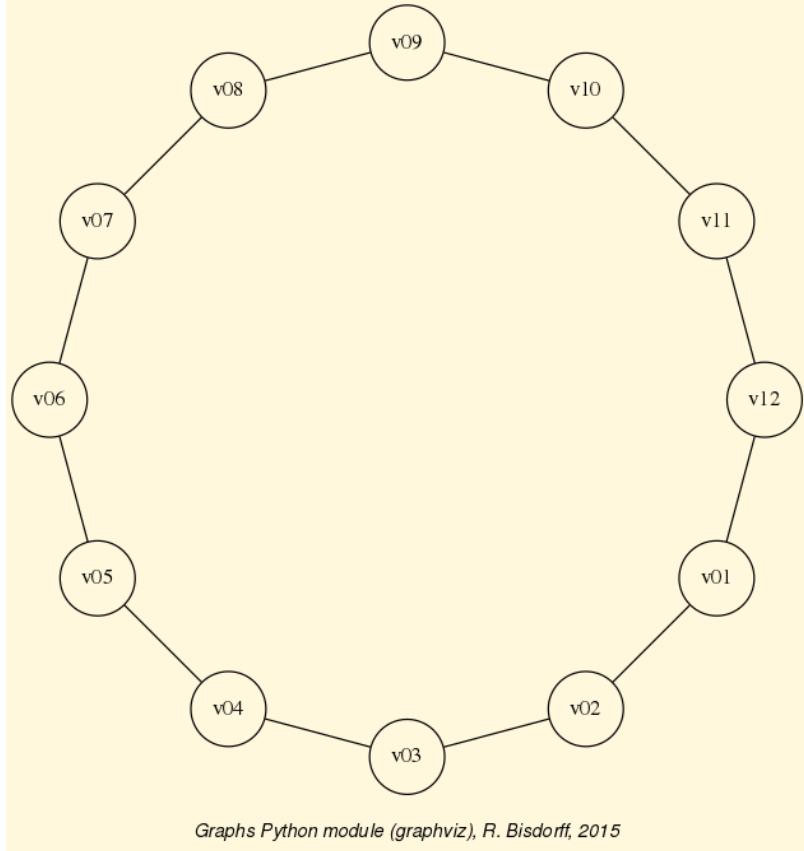

Fig. 1: The 12-cycle graph

14.2 Computing the maximal independent sets (MISs)

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.

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

159

C and installed[3] along with all the Digraphs3 python modules for computing the set of MISs observed in the graph.

```
1   ...$ echo 12 | /usr/local/bin/perrinMIS
2   # ------------------------------------ #
3   # Generating MIS set of Cn with the    #
4   # Perrin sequence algorithm.           #
5   # Temporary files used.                #
6   # even versus odd order optimised.     #
7   # RB December 2006                     #
8   # Current revision Dec 2018            #
9   # ------------------------------------ #
10  Input cycle order ? <-- 12
11  mis 1 : 100100100100
12  mis 2 : 010010010010
13  mis 3 : 001001001001
14  ...
15  ...
16  ...
17  mis 27 : 001001010101
18  mis 28 : 101010101010
19  mis 29 : 010101010101
20  Cardinalities:
21  0 : 0
22  1 : 0
23  2 : 0
24  3 : 0
25  4 : 3
26  5 : 24
27  6 : 2
28  7 : 0
29  8 : 0
30  9 : 0
31  10 : 0
32  11 : 0
33  12 : 0
34  Total: 29
35  execution time: 0 sec. and 2 millisec.
```

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

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

---

[3] The `perrinMIS` shell command may 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.

```
5    frozenset({'7', '2', '4', '10', '12'}),
6    ...
7    ...
8    ...
9    frozenset({'8', '4', '10', '1', '6'}),
10   frozenset({'11', '4', '1', '9', '6'}),
11   frozenset({'8', '2', '4', '10', '12', '6'})
12  }
```

14.3 Computing the automorphism group

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].

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

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

14.4 Computing the isomorphic MISs

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).

```
1  >>> c12.showOrbits(c12.misset,withListing=False)
2   ...
3     *---- Global result ----
4     Number of MIS:  29
```

---

[2] Dependency: The `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`.

```
5    Number of orbits :  4
6    Labelled representatives and cardinality:
7    1: ['2','4','6','8','10','12'], 2
8    2: ['2','5','8','11'], 3
9    3: ['2','4','6','9','11'], 12
10   4: ['1','4','7','9','11'], 12
11   Symmetry vector
12   stabilizer size: [1, 2, 3, ..., 8, 9, ..., 12, 13, ...]
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 Fig. 2 below.
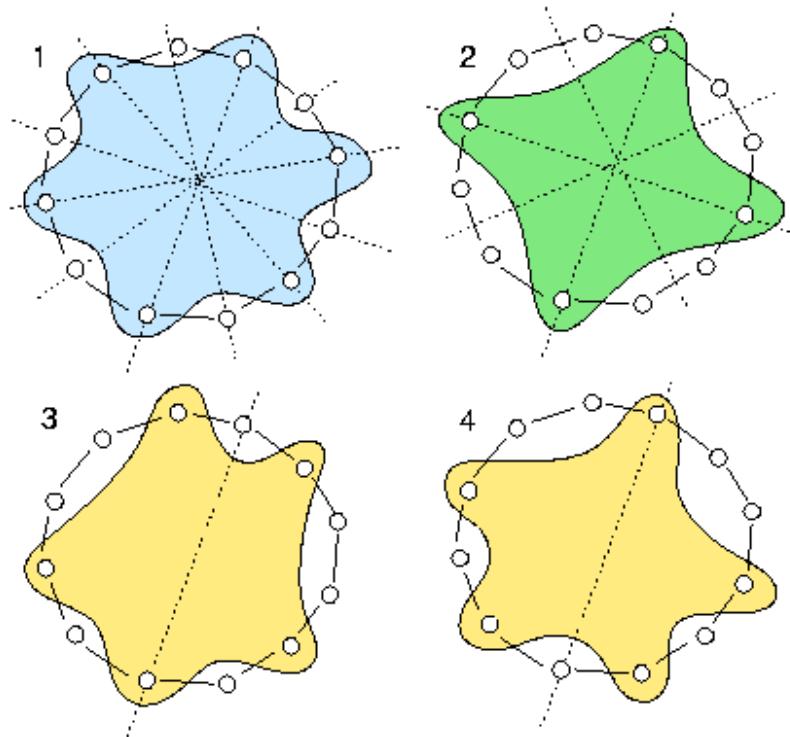
Fig. 2: The symmetry axes of the four non isomorphic MISs of the 12-cycle graph

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]).

Back to *Content Table* (page 3)

15 On computing digraph kernels

15.1 What is a graph kernel ?

We call **choice** in a graph, respectively a digraph, a subset of its vertices, resp. of its nodes or actions. A choice Y is called **internally stable** or **independent** when there exist **no links** (edges) or relations (arcs) between its members. Furthermore, a choice Y is called **externally stable** when for each vertex, node or action x not in Y, there exists at least a member y of Y such that x is linked or related to y. Now, an internally **and** externally stable choice is called a **kernel**.

A first trivial example is immediately given by the maximal independent vertices sets (MISs) of the n-cycle graph (see *Computing the non isomorphic MISs of the 12-cycle graph* (page 158)). Indeed, each MIS in the n-cycle graph is by definition independent, i.e. internally stable, and each non selected vertex in the n-cycle graph is in relation with either one or even two members of the MIS. See, for instance, the four non isomorphic MISs of the 12-cycle graph as shown in Fig. 2.

In all graph or symmetric digraph, the *maximality condition* imposed on the internal stability is equivalent to the external stability condition. Indeed, if there would exist a vertex or node not related to any of the elements of a choice, then we may safely add this vertex or node to the given choice without violating its internal stability. All kernels must hence be maximal independent choices. In fact, in a topological sense, they correspond to maximal **holes** in the given graph.

We may illustrate this coincidence between MISs and kernels in graphs and symmetric digraphs with the following random 3-regular graph instance (see Fig. 1).

```
1  >>> from graphs import RandomRegularGraph
2  >>> g = RandomRegularGraph(order=12,degree=3,seed=100)
3  >>> g.exportGraphViz('random3RegularGraph')
4   *---- exporting a dot file for GraphViz tools ---------*
5   Exporting to random3RegularGraph.dot
6   fdp -Tpng random3RegularGraph.dot -o random3RegularGraph.png
```
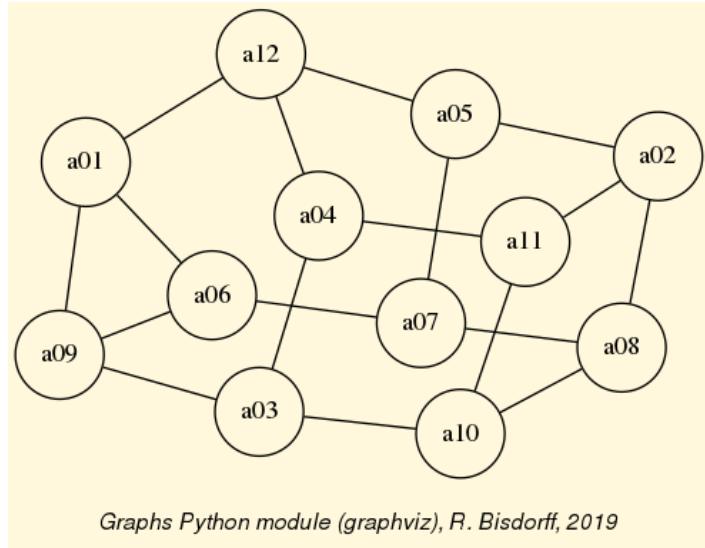
Fig. 1: A random 3-regular graph instance

A random MIS in this graph may be computed for instance by using the `graphs.MISModel` class.

```
1  >>> from graphs import MISModel
2  >>> mg = MISModel(g)
3   Iteration:  1
4   Running a Gibbs Sampler for 660 step !
5   {'a06', 'a02', 'a12', 'a10'}  is maximal !
6  >>> mg.exportGraphViz('random3RegularGraph_mis')
7   *---- exporting a dot file for GraphViz tools ---------*
8   Exporting to random3RegularGraph-mis.dot
9   fdp -Tpng random3RegularGraph-mis.dot -o random3RegularGraph-mis.png
```
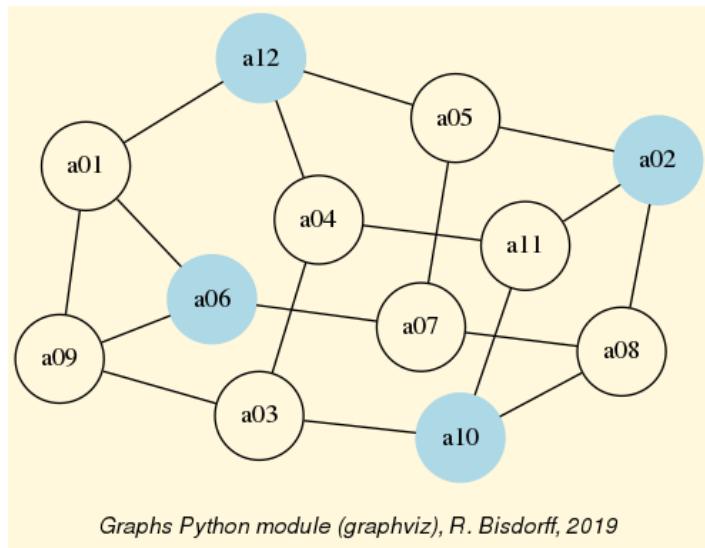

Fig. 2: A random MIS colored in the random 3-regular graph

It is easily verified in Fig. 2 above, that the computed MIS renders indeed a valid kernel

of the given graph. The complete set of kernels of this 3-regular graph instance coincides hence with the set of its MISs.

```
1  >>> g.showMIS()
2   *---  Maximal Independent Sets ---*
3   ['a01', 'a02', 'a03', 'a07']
4   ['a01', 'a04', 'a05', 'a08']
5   ['a04', 'a05', 'a08', 'a09']
6   ['a01', 'a04', 'a05', 'a10']
7   ['a04', 'a05', 'a09', 'a10']
8   ['a02', 'a03', 'a07', 'a12']
9   ['a01', 'a03', 'a07', 'a11']
10  ['a05', 'a08', 'a09', 'a11']
11  ['a03', 'a07', 'a11', 'a12']
12  ['a07', 'a09', 'a11', 'a12']
13  ['a08', 'a09', 'a11', 'a12']
14  ['a04', 'a05', 'a06', 'a08']
15  ['a04', 'a05', 'a06', 'a10']
16  ['a02', 'a04', 'a06', 'a10']
17  ['a02', 'a03', 'a06', 'a12']
18  ['a02', 'a06', 'a10', 'a12']
19  ['a01', 'a02', 'a04', 'a07', 'a10']
20  ['a02', 'a04', 'a07', 'a09', 'a10']
21  ['a02', 'a07', 'a09', 'a10', 'a12']
22  ['a01', 'a03', 'a05', 'a08', 'a11']
23  ['a03', 'a05', 'a06', 'a08', 'a11']
24  ['a03', 'a06', 'a08', 'a11', 'a12']
25   number of solutions:  22
26   cardinality distribution
27   card.:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
28   freq.:  [0, 0, 0, 0, 16, 6, 0, 0, 0, 0, 0, 0, 0]
29   execution time: 0.00045 sec.
30   Results in self.misset
31  >>> g.misset
32   [frozenset({'a02', 'a01', 'a07', 'a03'}),
33    frozenset({'a04', 'a01', 'a08', 'a05'}),
34    frozenset({'a09', 'a04', 'a08', 'a05'}),
35    ...
36    ...
37    frozenset({'a06', 'a02', 'a12', 'a10'}),
38    frozenset({'a06', 'a11', 'a08', 'a03', 'a05'}),
39    frozenset({'a03', 'a06', 'a11', 'a12', 'a08'})]
```

We cannot resist in looking in this 3-regular graph for non isomorphic kernels (MISs, see previous tutorial). To do so we must first, convert the given *graph* instance into a *digraph* instance. Then, compute its automorphism generators, and finally, identify the isomorphic kernel orbits.

```
1  >>> dg = g.graph2Digraph()
2  >>> dg.automorphismGenerators()
```

```
3    *----- saving digraph in nauty dre format  -------------*
4    Actions index:
5    1 :  a01
6    2 :  a02
7    3 :  a03
8    4 :  a04
9    5 :  a05
10   6 :  a06
11   7 :  a07
12   8 :  a08
13   9 :  a09
14   10 :  a10
15   11 :  a11
16   12 :  a12
17   {'1': 'a01',  '2': 'a02', '3': 'a03', '4': 'a04',  '5': 'a05',
18    '6': 'a06',  '7': 'a07', '8': 'a08', '9': 'a09', '10': 'a10',
19   '11': 'a11', '12': 'a12'}
20   # automorphisms extraction from dre file #
21   # Using input file: randomRegularGraph.dre
22   echo '<randomRegularGraph.dre -m p >randomRegularGraph.auto x' | dreadnaut
23   # permutation = 1['1', '11', '7', '5', '4', '9', '3', '10', '6', '8', '2', '12
     ↪']
24   >>> dg.showOrbits(g.misset)
25   *--- Isomorphic reduction of choices
26   ...
27   current representative:  frozenset({'a09', 'a11', 'a12', 'a08'})
28   length    :  4
29   number of isomorph choices 2
30   isormorph choices
31   ['a06', 'a02', 'a12', 'a10']  # <<== the random MIS shown above
32   ['a09', 'a11', 'a12', 'a08']
33   ---------------------------
34   *---- Global result ----
35   Number of choices:  22
36   Number of orbits :  11
37   Labelled representatives:
38   ['a06', 'a04', 'a10', 'a05']
39   ['a09', 'a07', 'a10', 'a04', 'a02']
40   ['a06', 'a11', 'a12', 'a08', 'a03']
41   ['a04', 'a01', 'a10', 'a05']
42   ['a07', 'a02', 'a12', 'a03']
43   ['a09', 'a11', 'a12', 'a07']
44   ['a06', 'a04', 'a08', 'a05']
45   ['a06', 'a04', 'a02', 'a10']
46   ['a01', 'a11', 'a07', 'a03']
47   ['a01', 'a11', 'a08', 'a03', 'a05']
48   ['a09', 'a11', 'a12', 'a08']
49   Symmetry vector
50   stabilizer size  :  [1, 2]
```

```
51  frequency        :  [11, 0]
```

In our random 3-regular graph instance (see Fig. 1), we may thus find eleven non isomorphic kernels with orbit sizes equal to two. We illustrate below the isomorphic twin of the random MIS example shown in Fig. 2 .
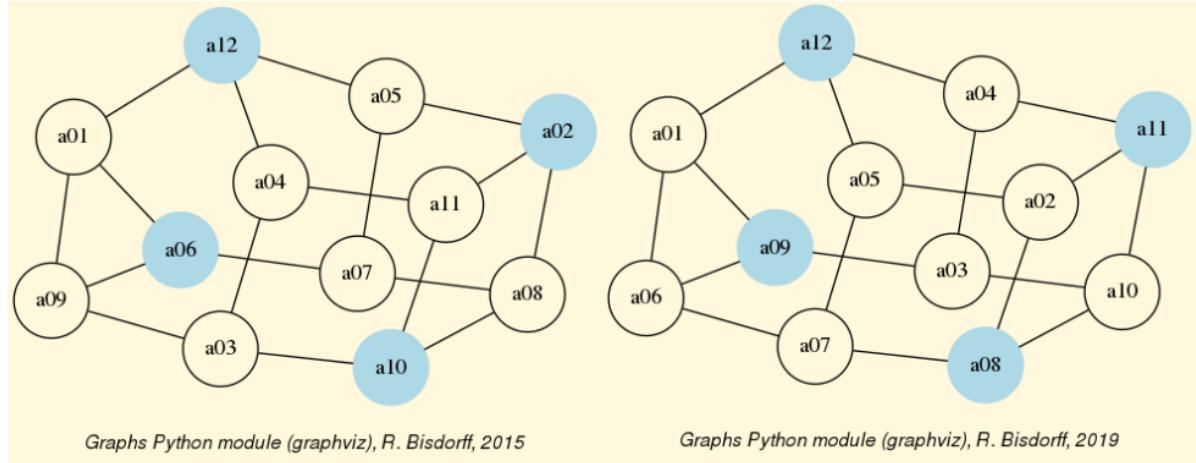
Fig. 3: Two isomorphic kernels of the random 3-regular graph instance

All graphs and symmetric digraphs admit MISs, hence also kernels.

It is worthwhile noticing that the **maximal matchings** of a graph correspond bijectively to its line graph's **kernels** (see the `graphs.LineGraph` class).

```
1   >>> from graphs import CycleGraph
2   >>> c8 = CycleGraph(order=8)
3   >>> maxMatching = c8.computeMaximumMatching()
4   >>> c8.exportGraphViz(fileName='maxMatchingcycleGraph',
5                           matching=maxMatching)
6    *---- exporting a dot file for GraphViz tools ---------*
7    Exporting to maxMatchingcyleGraph.dot
8    Matching:  {frozenset({'v1', 'v2'}), frozenset({'v5', 'v6'}),
9               frozenset({'v3', 'v4'}), frozenset({'v7', 'v8'}) }
10   circo -Tpng maxMatchingcyleGraph.dot -o maxMatchingcyleGraph.png
```

167

Graphs Python module (graphviz), R. Bisdorff, 2015
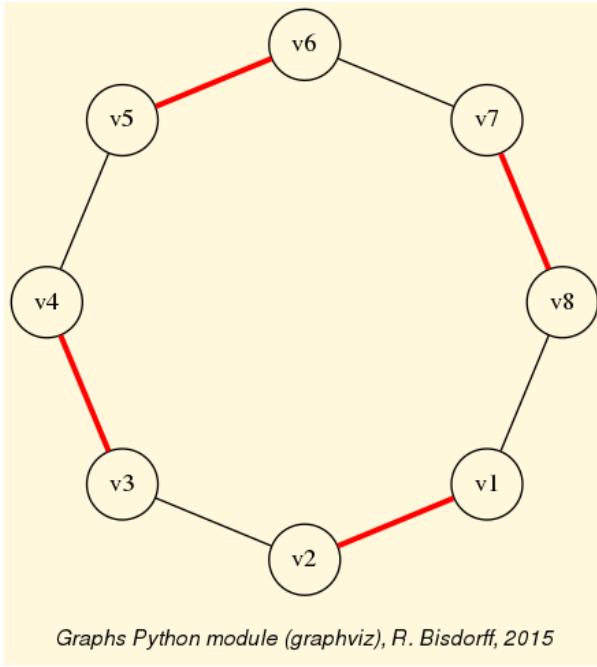
Fig. 4: Perfect maximum matching in the 8-cycle graph

In the context of digraphs, i.e. *oriented* graphs, the kernel concept gets much richer and separates from the symmetric MIS concept.

15.2 Initial and terminal kernels

In an oriented graph context, the internal stability condition of the kernel concept remains untouched; however, the external stability condition gets indeed split up by the *orientation* into two lateral cases:

1. A **dominant** stability condition, where each non selected node is *dominated* by at least one member of the kernel;

2. An **absorbent** stability condition, where each non selected node is *absorbed* by at least one member of the kernel.

A both *internally* **and** *dominant*, resp. *absorbent stable* choice is called a *dominant* or **initial**, resp. an *absorbent* or **terminal** kernel. From a topological perspective, the initial kernel concept looks from the outside of the digraph into its interior, whereas the terminal kernel looks from the interior of a digraph toward its outside. From an algebraic perspective, the initial kernel is a *prefix* operand, and the terminal kernel is a *postfix* operand in the *Berge* kernel equation systems (see Digraph3 advanced topic on bipolar-valued kernel membership characteristics).

Furthermore, as the kernel concept involves conjointly a **positive logical refutation** (the *internal stability*) and a **positive logical affirmation** (the *external stability*), it appeared rather quickly necessary in our operational developments to adopt a bipolar characteristic [-1,1] valuation domain, modelling *negation* by change of numerical sign and including explicitly a third **median** logical value (0) expressing logical **indeterminateness** (neither positive, nor negative, see [BIS-2000] and [BIS-2004]).

168

In such a bipolar-valued context, we call **prekernel** a choice which is **externally stable** and for which the **internal stability** condition is **valid or indeterminate**. We say that the independence condition is in this case only **weakly** validated. Notice that all kernels are hence prekernels, but not vice-versa.

In graphs or symmetric digraphs, where there is essentially no apparent ' *laterality* ', all prekernels are *initial* **and** *terminal* at the same time. They correspond to what we call *holes* in the graph. An *universal* example is given by the **complete** digraph.

```
>>> from digraphs import CompleteDigraph
>>> u = CompleteDigraph(order=5)
>>> u
 *------- Digraph instance description ------*
 Instance class   : CompleteDigraph
 Instance name    : complete
 Digraph Order     : 5
 Digraph Size      : 20
 Valuation domain : [-1.00 ; 1.00]
 --------------------------------
>>> u.showPreKernels()
 *--- Computing preKernels ---*
 Dominant kernels :
 ['1'] independence: 1.0; dominance : 1.0; absorbency : 1.0
 ['2'] independence: 1.0; dominance : 1.0; absorbency : 1.0
 ['3'] independence: 1.0; dominance : 1.0; absorbency : 1.0
 ['4'] independence: 1.0; dominance : 1.0; absorbency : 1.0
 ['5'] independence: 1.0; dominance : 1.0; absorbency : 1.0
 Absorbent kernels :
 ['1'] independence: 1.0; dominance : 1.0; absorbency : 1.0
 ['2'] independence: 1.0; dominance : 1.0; absorbency : 1.0
 ['3'] independence: 1.0; dominance : 1.0; absorbency : 1.0
 ['4'] independence: 1.0; dominance : 1.0; absorbency : 1.0
 ['5'] independence: 1.0; dominance : 1.0; absorbency : 1.0
 *----- statistics -----
 graph name:  complete
 number of solutions
  dominant kernels :   5
  absorbent kernels:   5
 cardinality frequency distributions
 cardinality      :  [0, 1, 2, 3, 4, 5]
 dominant kernel :  [0, 5, 0, 0, 0, 0]
 absorbent kernel:  [0, 5, 0, 0, 0, 0]
 Execution time  : 0.00004 sec.
 Results in sets: dompreKernels and abspreKernels.
```

In a complete digraph, each single node is indeed both an initial and a terminal prekernel candidate and there is no definite *begin* or *end* of the digraph to be detected. *Laterality* is here entirely *relative* to a specific singleton chosen as reference point of view. The same absence of laterality is apparent in two other universal digraph models, the **empty** and the **indeterminate** digraph.

```
1  >>> ed = EmptyDigraph(order=5)
2  >>> ed.showPreKernels()
3   *--- Computing preKernels ---*
4   Dominant kernel :
5   ['1', '2', '3', '4', '5']
6      independence :   1.0
7      dominance    :   1.0
8      absorbency   :   1.0
9   Absorbent kernel :
10  ['1', '2', '3', '4', '5']
11     independence :   1.0
12     dominance    :   1.0
13     absorbency   :   1.0
14  ...
```

In the empty digraph, the whole set of nodes gives indeed at the same time the **unique** *initial* **and** *terminal* prekernel. Similarly, for the **indeterminate** digraph.

```
1  >>> from digraphs import IndeterminateDigraph
2  >>> id = IndeterminateDigraph(order=5)
3  >>> id.showPreKernels()
4   *--- Computing preKernels ---*
5   Dominant prekernel :
6   ['1', '2', '3', '4', '5']
7      independence :   0.0   # <<== indeterminate
8      dominance    :   1.0
9      absorbency   :   1.0
10  Absorbent prekernel :
11  ['1', '2', '3', '4', '5']
12     independence :   0.0   # <<== indeterminate
13     dominance    :   1.0
14     absorbency   :   1.0
```

Both these results make sense, as in a completely empty or indeterminate digraph, there is no *interior* of the digraph defined, only a *border* which is hence at the same time an initial and terminal prekernel. Notice however, that in the latter indeterminate case, the complete set of nodes verifies only weakly the internal stability condition (see above).

Other common digraph models, although being clearly oriented, may show nevertheless no apparent laterality, like **odd chordless circuits**, i.e. *holes* surrounded by an *oriented cycle* -a circuit- of odd length. They do not admit in fact any initial or terminal prekernel.

```
1  >>> from digraphs import CirculantDigraph
2  >>> c5 = CirculantDigraph(order=5,circulants=[1])
3  >>> c5.showPreKernels()
4   *----- statistics -----
5   digraph name:  c5
6   number of solutions
7    dominant prekernels :   0
8    absorbent prekernels:   0
```

Chordless circuits of **even** length 2 x *k*, with $k > 1$, contain however two isomorphic prekernels of cardinality *k* which qualify conjointly as initial and terminal candidates.

```
>>> c6 = CirculantDigraph(order=6,circulants=[1])
>>> c6.showPreKernels()
 *--- Computing preKernels ---*
 Dominant preKernels :
 ['1', '3', '5'] independence: 1.0, dominance: 1.0, absorbency: 1.0
 ['2', '4', '6'] independence: 1.0, dominance: 1.0, absorbency: 1.0
 Absorbent preKernels :
 ['1', '3', '5'] independence: 1.0, dominance: 1.0, absorbency: 1.0
 ['2', '4', '6'] independence: 1.0, dominance: 1.0, absorbency: 1.0
```

Chordless circuits of even length may thus be indifferently oriented along two opposite directions. Notice by the way that the duals of **all** chordless circuits of *odd* **or** *even* length, i.e. *filled* circuits also called **anti-holes** (see Fig. 5), never contain any potential prekernel candidates.

```
>>> dc6 = -c6     # dc6 = DualDigraph(c6)
>>> dc6.showPreKernels()
 *----- statistics -----
 graph name:  dual_c6
 number of solutions
  dominant prekernels :  0
  absorbent prekernels:  0
>>> dc6.exportGraphViz(fileName='dualChordlessCircuit')
 *---- exporting a dot file for GraphViz tools ---------*
 Exporting to dualChordlessCircuit.dot
 circo -Tpng dualChordlessCircuit.dot -o dualChordlessCircuit.png
```
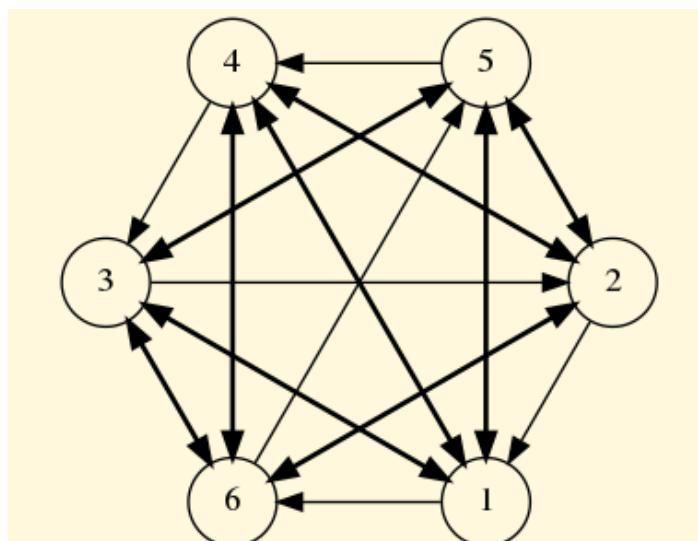

Rubis Python Server (graphviz), R. Bisdorff, 2008

Fig. 5: The dual of the chordless 6-circuit

171

We call **weak**, a *chordless circuit* with *indeterminate inner part*. The `digraphs.`
`CirculantDigraph` class provides a parameter for constructing such a kind of *weak chord-less* circuits.

```
>>> c6 = CirculantDigraph(order=6, circulants=[1],
                          IndeterminateInnerPart=True)
```

It is worth noticing that the *dual* version ([14]) of a *weak* circuit corresponds to its *converse* version, i.e. *-c6 = ~c6* (see Fig. 6).

```
>>> (-c6).exportGraphViz()
 *---- exporting a dot file for GraphViz tools ---------*
 Exporting to dual_c6.dot
 circo -Tpng dual_c6.dot -o dual_c6.png
>>> (~c6).exportGraphViz()
 *---- exporting a dot file for GraphViz tools ---------*
 Exporting to converse_c6.dot
 circo -Tpng converse_c6.dot -o converse_c6.png
```
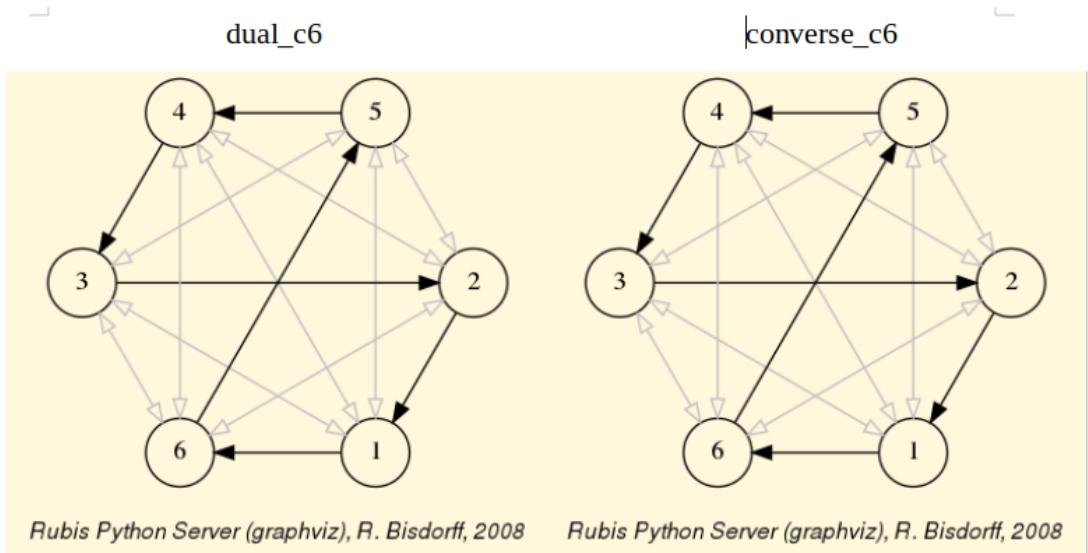

Fig. 6: Dual and converse of the weak 6-circuit

It immediately follows that weak chordless circuits are part of the class of digraphs that are **invariant** under the *codual* transform, $cn = - (\tilde{\ } \ cn\ ) = \tilde{\ } (\ -cn\ )$[13]. In the case, now, of an *odd* weak chordless circuit, *neither* the weak chordless circuit, *nor* its dual, converse, or codual versions will admit *any* initial or terminal prekernels.

15.3 Kernels in lateralized digraphs

Humans do live in an apparent physical space of plain transitive **lateral orientation**, fully empowered in finite geometrical 3D models with **linear orders**, where first, resp. last ranked, nodes deliver unique initial, resp. terminal, kernels. Similarly, in finite **preorders**, the first, resp. last, equivalence classes deliver the unique initial, resp. unique terminal, kernels. More generally, in finite **partial orders**, i.e. asymmetric and transitive digraphs, topological sort algorithms will easily reveal on the first, resp. last, level all unique initial, resp. terminal, kernels.

In genuine random digraphs, however, we may need to check for each of its MISs, whether *one*, *both*, or *none* of the lateralized external stability conditions may be satisfied. Consider, for instance, the following random digraph instance of order 7 and generated with an arc probability of 30%.

```
1  >>> from randomDigraphs import RandomDigraph
2  >>> rd = RandomDigraph(order=7,arcProbability=0.3,seed=5)
3  >>> rd.exportGraphViz('randomLaterality')
4   *---- exporting a dot file for GraphViz tools ---------*
5   Exporting to randomLaterality.dot
6   dot -Grankdir=BT -Tpng randomLaterality.dot -o randomLaterality.png
```



*Rubis Python Server (graphviz), R. Bisdorff, 2008*
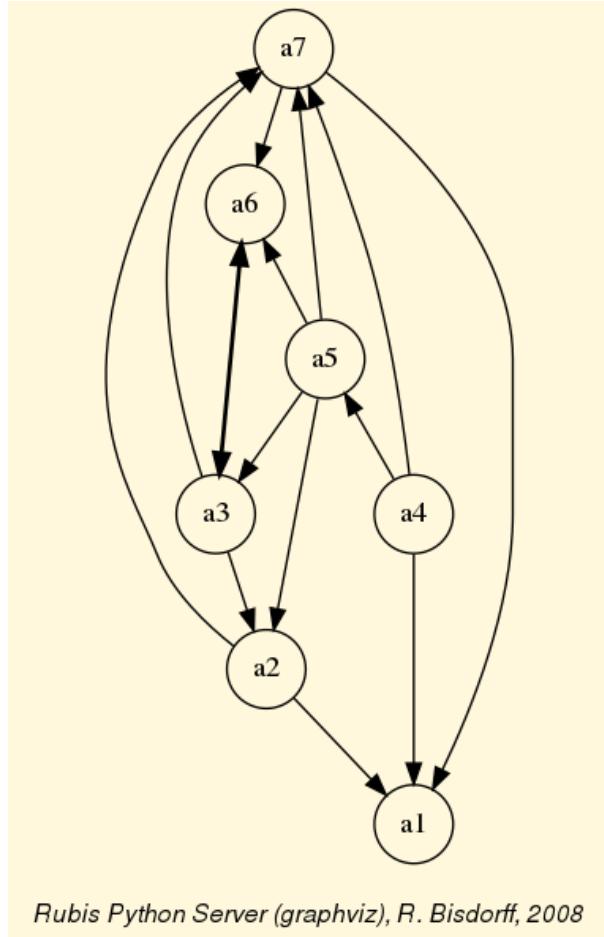
Fig. 7: A random digraph instance of order 7 and arc probability 0.3

The random digraph shown in Fig. 7 above has no apparent special properties, except from being connected (see Line 3 below).

```
1  >>> rd.showComponents()
2   *--- Connected Components ---*
3   1: ['a1', 'a2', 'a3', 'a4', 'a5', 'a6', 'a7']
4  >>> rd.computeSymmetryDegree(Comments=True,InPercents=True)
5   Symmetry degree (%) of digraph <randomDigraph>:
6    #arcs x>y: 14, #symmetric: 1, #asymmetric: 13
7    #symmetric/#arcs =  7.1
8  >>> rd.computeChordlessCircuits()
9   []  # no chordless circuits detected
10 >>> rd.computeTransitivityDegree(Comments=True,InPercents=True)
11  Transitivity degree (%) of graph <randomDigraph>:
12   #triples x>y>z: 23, #closed: 11, #open: 12
13   #closed/#triples =  47.8
```

The given digraph instance is neither asymmetric (a3 <-> a6) nor symmetric (a2 -> a1, a1 -/> a2) (see Line 6 above); there are no chordless circuits (see Line 9 above); and, the digraph is not transitive (a5 -> a2 -> a1, but a5 -/> a1). More than half of the required transitive closure is missing (see Line 12 above).

Now, we know that its potential prekernels must be among its set of maximal independent choices.

```
1  >>> rd.showMIS()
2   *---  Maximal independent choices ---*
3   ['a2', 'a4', 'a6']
4   ['a6', 'a1']
5   ['a5', 'a1']
6   ['a3', 'a1']
7   ['a4', 'a3']
8   ['a7']
9   ------
10 >>> rd.showPreKernels()
11  *--- Computing preKernels ---*
12  Dominant preKernels :
13  ['a2', 'a4', 'a6']
14     independence :  1.0
15     dominance    :  1.0
16     absorbency   :  -1.0
17     covering     :  0.500
18  ['a4', 'a3']
19     independence :  1.0
20     dominance    :  1.0
21     absorbency   :  -1.0
22     covering     :  0.600  # <<==
23  Absorbent preKernels :
24  ['a3', 'a1']
25     independence :  1.0
26     dominance    :  -1.0
```

```
27      absorbency    :   1.0
28      covering      :   0.500
29   ['a6', 'a1']
30      independence :   1.0
31      dominance    :  -1.0
32      absorbency    :   1.0
33      covering      :   0.600  # <<==
34   ...
```

Among the six MISs contained in this random digraph (see above Lines 3-8) we discover two initial and two terminal kernels (Lines 12-34). Notice by the way the covering values (between 0.0 and 1.0) shown by the `digraphs.Digraph.showPreKernels()` method (Lines 17, 22, 28 and 33). The higher this value, the more the corresponding kernel candidate makes apparent the digraph's *laterality*. We may hence redraw the same digraph in Fig. 8 by looking into its interior via the *best covering* initial kernel candidate: the dominant choice {'a3','4a'} (coloured in yellow), and looking out of it via the *best covered* terminal kernel candidate: the absorbent choice {'a1','a6'} (coloured in blue).

```
1   >>> rd.exportGraphViz(fileName='orientedLaterality',\
2                        bestChoice=set(['a3', 'a4']),\
3                        worstChoice=set(['a1', 'a6']))
4    *---- exporting a dot file for GraphViz tools ---------*
5    Exporting to orientedLaterality.dot
6    dot -Grankdir=BT -Tpng orientedLaterality.dot -o orientedLaterality.png
```
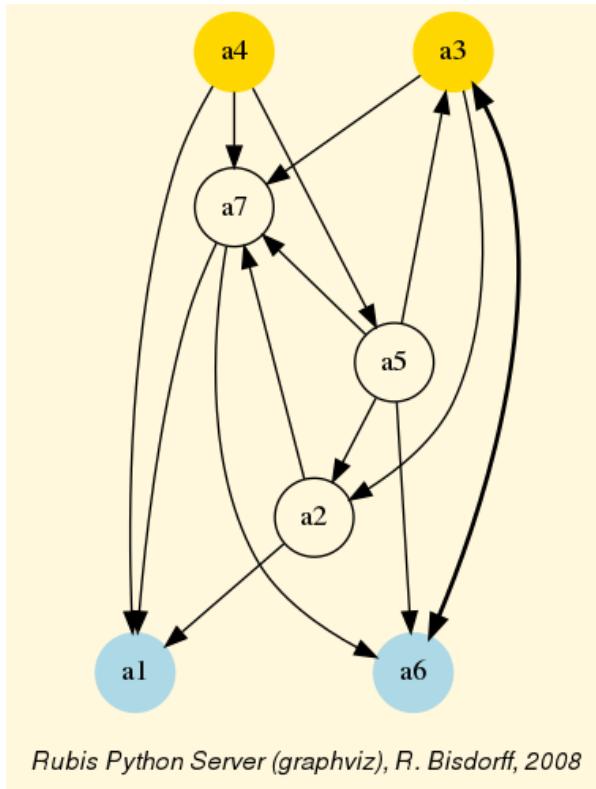

Fig. 8: A random digraph oriented by best covering initial and best covered terminal kernel

In algorithmic decision theory, initial and terminal prekernels may provide convincing best, resp. worst, choice recommendations (see *Computing a best choice recommendation* (page 74)).

15.4 Computing good and bad choice recommendations

To illustrate this idea, let us finally compute good and bad choice recommendations in the following random bipolar-valued **outranking** digraph.

```
1  >>> from outrankingDigraphs import *
2  >>> g = RandomBipolarOutrankingDigraph(seed=5)
3  >>> g
4   *------- Object instance description ------*
5   Instance class    : RandomBipolarOutrankingDigraph
6   Instance name     : randomOutranking
7   # Actions         : 7
8   # Criteria        : 7
9   Size              : 26
10  Determinateness   : 34.275
11  Valuation domain  : {'min': -100.0, 'med': 0.0, 'max': 100.0}
12  >>> g.showHTMLPerformanceTableau()
```

Performance table randomOutranking

criterion	g1	g2	g3	g4	g5	g6	g7
a1	64.90	1.31	13.88	98.24	94.10	14.57	31.00
a2	NA	NA	61.75	87.24	69.06	6.51	81.85
a3	11.32	27.95	12.67	28.93	96.66	30.14	48.07
a4	46.91	91.63	0.18	96.15	89.37	60.31	31.58
a5	NA	76.57	87.14	53.92	29.88	0.34	48.12
a6	54.38	15.96	20.95	67.78	36.12	67.79	70.47
a7	57.39	79.71	21.55	20.48	16.60	33.79	5.70

Fig. 9: The performance tableau of a random outranking digraph instance

The underlying random performance tableau (see Fig. 9) shows the performance grading of 7 potential decision actions with respect to 7 decision criteria supporting each an increasing performance scale from 0 to 100. Notice the missing performance data concerning decision actions 'a2' and 'a5'. The resulting **strict outranking** - i.e. a weighted majority supported - *better than without considerable counter-performance* - digraph is shown in Fig. 10 below.

```
1  >>> gcd = ~(-g)    # Codual: the converse of the negation
2  >>> gcd.exportGraphViz(fileName='tutOutRanking')
3   *---- exporting a dot file for GraphViz tools ---------*
```

(continues on next page)

```
4    Exporting to tutOutranking.dot
5    dot -Grankdir=BT -Tpng tutOutranking.dot -o tutOutranking.png
```
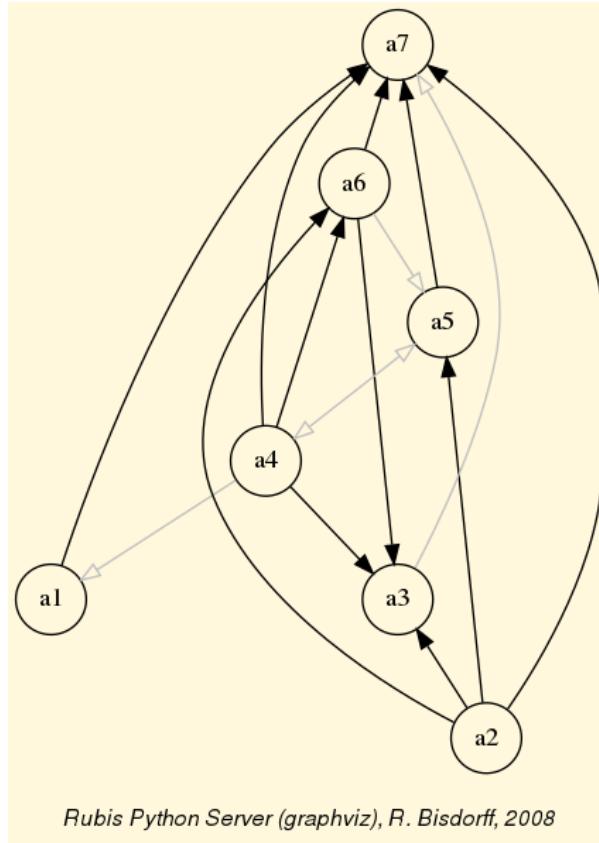

Fig. 10: A random strict outranking digraph instance

All decision actions appear strictly better performing than action 'a7'. We call it a **Condorcet looser** and it is an evident terminal prekernel candidate. On the other side, three actions: 'a1', 'a2' and 'a4' are not dominated. They give together an initial prekernel candidate.

```
1    >>> gcd.showPreKernels()
2    *--- Computing preKernels ---*
3    Dominant preKernels :
4    ['a1', 'a2', 'a4']
5        independence :  0.00
6        dominance    :  6.98
7        absorbency   :  -48.84
8        covering     :  0.667
9    Absorbent preKernels :
10   ['a3', 'a7']
11       independence :  0.00
12       dominance    :  -74.42
13       absorbency   :  16.28
14       covered      :  0.800
```

With such unique disjoint initial and terminal prekernels (see Line 4 and 10), the given digraph instance is hence clearly *lateralized*. Indeed, these initial and terminal prekernels of the codual outranking digraph reveal best, resp. worst, choice recommendations one may formulate on the basis of a given outranking digraph instance.

```
>>> g.showBestChoiceRecommendation()
 ***********************
 Rubis best choice recommendation(s) (BCR)
   (in decreasing order of determinateness)
 Credibility domain: [-100.00,100.00]
  === >> potential best choice(s)
 * choice              : ['a1', 'a2', 'a4']
   independence        : 0.00
   dominance           : 6.98
   absorbency          : -48.84
   covering (%)        : 66.67
   determinateness (%) : 57.97
   - most credible action(s) = { 'a4': 20.93, 'a2': 20.93, }
  === >> potential worst choice(s)
 * choice              : ['a3', 'a7']
   independence        : 0.00
   dominance           : -74.42
   absorbency          : 16.28
   covered (%)         : 80.00
   determinateness (%) : 64.62
   - most credible action(s) = { 'a7': 48.84, }
```

Notice that solving the valued *Berge* kernel equations (see Bipolar-Valued Kernels in the Advanced Topics) provides furthermore a positive characterization of the most credible decision actions in each respective choice recommendation (see Lines 14 and 23 above). Actions 'a2' and 'a4' are equivalent candidates for a unique best choice, and action 'a7' is clearly confirmed as the worst choice.

In Fig. 11 below, we orient the drawing of the strict outranking digraph instance with the help of these best and worst choice recommendations.

```
>>> gcd.exportGraphViz(fileName='bestWorstOrientation',
         bestChoice=['a2','a4'], worstChoice=['a7'])
 *---- exporting a dot file for GraphViz tools ---------*
 Exporting to bestWorstOrientation.dot
 dot -Grankdir=BT -Tpng bestWorstOrientation.dot -o bestWorstOrientation.png
```
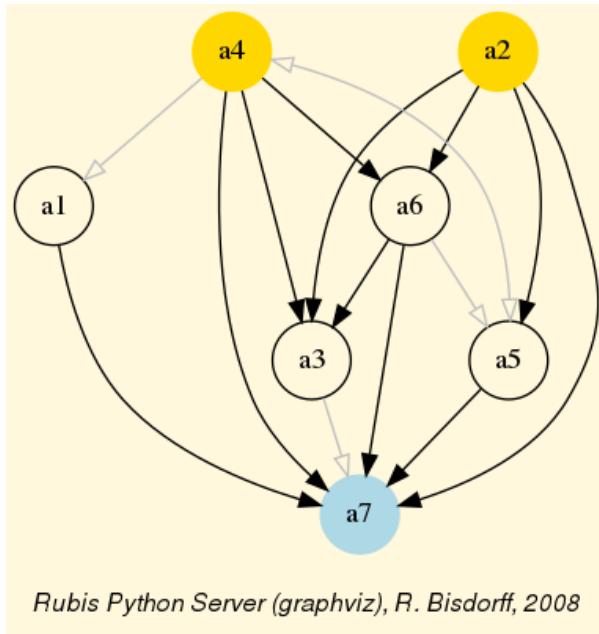
Fig. 11: The strict outranking digraph oriented by its best and worst choice recommendations

The gray arrows in Fig. 11, like the one between actions 'a4' and 'a1', represent indeterminate preferential situations. Action 'a1' appears hence to be rather incomparable to all the other, except action 'a7'. It may be interesting to compare this result with a *Copeland* ranking of the underlying performance tableau (see *Ranking with multiple incommensurable criteria* (page 59)).

```
1  >>> g.showHTMLPerformanceHeatmap(colorLevels=5, ndigits=0,
2               Correlations=True, rankingRule='Copeland')
```

Heatmap of Performance Tableau 'randomOutranking'

criteria	g4	g7	g5	g6	g1	g2	g3
weights	9	10	6	5	4	8	1
tau(*)	+0.64	+0.40	+0.29	+0.17	+0.02	-0.05	-0.10
a4	96	32	89	60	47	92	0
a2	87	82	69	7	NA	NA	62
a6	68	70	36	68	54	16	21
a1	98	31	94	15	65	1	14
a5	54	48	30	0	NA	77	87
a3	29	48	97	30	11	28	13
a7	20	6	17	34	57	80	22

Color legend:

quantile	20.00%	40.00%	60.00%	80.00%	100.00%

(*) tau: *Ordinal (Kendall) correlation between marginal criterion and global ranking relation*
Ranking rule: **Copeland**
Ordinal (Kendall) correlation between global ranking and global outranking relation: **+0.848**

Fig. 12: heatmap with Copeland ranking of the performance tableau

In the resulting linear ranking (see Fig. 12), action 'a4' is set at first rank, followed by action 'a2'. This makes sense as 'a4' shows three performances in the first quintile, whereas 'a2' is only partially evaluated and shows only two such excellent performances. But 'a4' also shows a very weak performance in the first quintile. Both decision actions, hence, don't show eventually a performance profile that would make apparent a clear preference situation in favour of one or the other. In this sense, the prekernels based best choice recommendations may appear more faithful with respect to the actually definite strict outranking relation than any 'forced' linear ranking result as shown in Fig. 12 above.

15.5 Tractability

Finally, let us give some hints on the **tractability** of kernel computations. Detecting all (pre)kernels in a digraph is a famously NP-hard computational problem. Checking external stability conditions for an independent choice is equivalent to checking its maximality and may be done in the linear complexity of the order of the digraph. However, checking all independent choices contained in a digraph may get hard already for tiny sparse digraphs of order $n > 20$ (see [BIS-2006b]). Indeed, the worst case is given by an empty or indeterminate digraph where the set of all potential independent choices to check is in fact the power set of the vertices.

```
1  >>> e = EmptyDigraph(order=20)
2  >>> e.showMIS()   # by visiting all 2^20 independent choices
3   *---  Maximal independent choices ---*
4   [ '1',  '2',  '3',  '4',  '5',  '6',  '7',  '8',  '9', '10',
5    '11', '12', '13', '14', '15', '16', '17', '18', '19', '20']
6   number of solutions:  1
7   execution time: 1.47640 sec.  # <<== !!!
8  >>> 2**20
9   1048576
```

Now, there exist more efficient specialized algorithms for directly enumerating MISs and dominant or absorbent kernels contained in specific digraph models without visiting all independent choices (see [BIS-2006b]). Alain Hertz provided kindly such a MISs enumeration algorithm for the Digraph3 project (see `digraphs.Digraph.showMIS_AH()`). When the number of independent choices is big compared to the actual number of MISs, like in very sparse or empty digraphs, the performance difference may be dramatic (see Line 7 above and Line 15 below).

```
1  >>> e.showMIS_AH()  # by visiting only maximal independent choices
2   *--------------------------------*
3   * Python implementation of Hertz's  *
4   * algorithm for generating all MISs *
5   * R.B. version 7(6)-25-Apr-2006     *
6   *--------------------------------*
7   ===>>> Initial solution :
8   [ '1',  '2',  '3',  '4',  '5',  '6',  '7',  '8',  '9', '10',
9    '11', '12', '13', '14', '15', '16', '17', '18', '19', '20']
```

(continues on next page)

```
10  *---- results ----*
11  [ '1',  '2',  '3',  '4',  '5',  '6',  '7',  '8',  '9', '10',
12   '11', '12', '13', '14', '15', '16', '17', '18', '19', '20']
13  *---- statistics ----*
14  mis solutions   :   1
15  execution time  : 0.00026 sec. # <<== !!!
16  iteration history:  1
```

For more or less dense strict outranking digraphs of modest order, as facing usually in algorithmic decision theory applications, enumerating all independent choices remains however in most cases tractable, especially by using a very efficient Python generator (see `digraphs.Digraph.independentChoices()` below).

```python
1   def independentChoices(self,U):
2       """
3       Generator for all independent choices with associated
4       dominated, absorbed and independent neighborhoods
5       of digraph instance self.
6       Initiate with U = self.singletons().
7       Yields [(independent choice, domnb, absnb, indnb)].
8       """
9       if U == []:
10          yield [(frozenset(),set(),set(),set(self.actions))]
11      else:
12          x = list(U.pop())
13          for S in self.independentChoices(U):
14              yield S
15              if x[0] <=  S[0][3]:
16                  Sxgamdom = S[0][1] | x[1]
17                  Sxgamabs = S[0][2] | x[2]
18                  Sxindep = S[0][3] &  x[3]
19                  Sxchoice = S[0][0] | x[0]
20                  Sx = [(Sxchoice,Sxgamdom,Sxgamabs,Sxindep)]
21                  yield Sx
```

And, checking maximality of independent choices via the external stability conditions during their enumeration (see `digraphs.Digraph.computePreKernels()` below) provides the effective advantage of computing all initial **and** terminal prekernels in a single loop (see Line 10 and [BIS-2006b]).

```python
1   def computePreKernels(self):
2       """
3       computing dominant and absorbent preKernels:
4       Result in self.dompreKernels and self.abspreKernels
5       """
6       actions = set(self.actions)
7       n = len(actions)
8       dompreKernels = set()
9       abspreKernels = set()
```

181

```
10        for choice in self.independentChoices(self.singletons()):
11            restactions = actions - choice[0][0]
12            if restactions <= choice[0][1]:
13                dompreKernels.add(choice[0][0])
14            if restactions <= choice[0][2]:
15                abspreKernels.add(choice[0][0])
16        self.dompreKernels = dompreKernels
17        self.abspreKernels = abspreKernels
```

Back to *Content Table* (page 3)

---

16 About split, interval and permutation graphs

16.1 A multiply *perfect* graph

Following Martin Golumbic (see [GOL-2004] p. 149), we call a given graph g:

- **Comparability graph** when g is *transitively orientable*;
- **Triangulated graph** when g does not contain any *chordless cycle* of length 4 and more;
- **Interval graph** when g is *triangulated* and its dual *-g* is a *comparability* graph;
- **Permutation graph** when g and its dual *-g* are both *comparability* graphs;
- **Split graph** when g and its dual *-g* are both *triangulated* graphs.

To illustrate these *perfect* graph classes, we will generate from 8 intervals, randomly chosen in the default integer range [0,10], a `graphs.RandomIntervalIntersectionsGraph` instance g (see Listing 16.1 Line 2 below).

Listing 16.1: A multiply perfect random interval intersection graph

```
1  >>> from graphs import RandomIntervalIntersectionsGraph
2  >>> g = RandomIntervalIntersectionsGraph(order=8,seed=100)
3  >>> g
```

```
4   *------- Graph instance description ------*
5   Instance class    : RandomIntervalIntersectionsGraph
6   Instance name     : randIntervalIntersections
7   Seed              : 100
8   Graph Order       : 8
9   Graph Size        : 23
10  Valuation domain  : [-1.0; 1.0]
11  Attributes        : ['seed', 'name', 'order', 'intervals',
12                       'vertices', 'valuationDomain',
13                       'edges', 'size', 'gamma']
14  >>> print(g.intervals)
15  [(2, 7), (2, 7), (5, 6), (6, 8), (1, 8), (1, 1), (4, 7), (0, 10)]
```

With seed = 100, we obtain here an *interval* graph, in fact a **perfect graph**, which is **conjointly** a *triangulated*, a *comparability*, a *split* and a *permutation* graph.

Listing 16.2: testing perfect graph categories

```
1   >>> g.isPerfectGraph(Comments=True)
2    Graph randIntervalIntersections is perfect !
3   >>> g.isIntervalGraph(Comments=True)
4    Graph 'randIntervalIntersections' is triangulated.
5    Graph 'dual_randIntervalIntersections' is transitively orientable.
6    => Graph 'randIntervalIntersections' is an interval graph.
7   >>> g.isSplitGraph(Comments=True)
8    Graph 'randIntervalIntersections' is triangulated.
9    Graph 'dual_randIntervalIntersections' is triangulated.
10   => Graph 'randIntervalIntersections' is a split graph.
11  >>> g.isPermutationGraph(Comments=True)
12   Graph 'randIntervalIntersections' is transitively orientable.
13   Graph 'dual_randIntervalIntersections' is transitively orientable.
14   => Graph 'randIntervalIntersections' is a permutation graph.
15  >>> print(g.computePermutation())
16   ['v5', 'v6', 'v4', 'v2', 'v1', 'v3', 'v7', 'v8']
17   ['v8', 'v6', 'v1', 'v2', 'v3', 'v4', 'v7', 'v5']
18   [8, 2, 6, 5, 7, 4, 3, 1]
19  >>> g.exportGraphViz('randomSplitGraph')
20   *---- exporting a dot file for GraphViz tools ---------*
21   Exporting to randomSplitGraph.dot
22   fdp -Tpng randomSplitGraph.dot -o randomSplitGraph.png
```

Graphs Python module (graphviz), R. Bisdorff, 2019
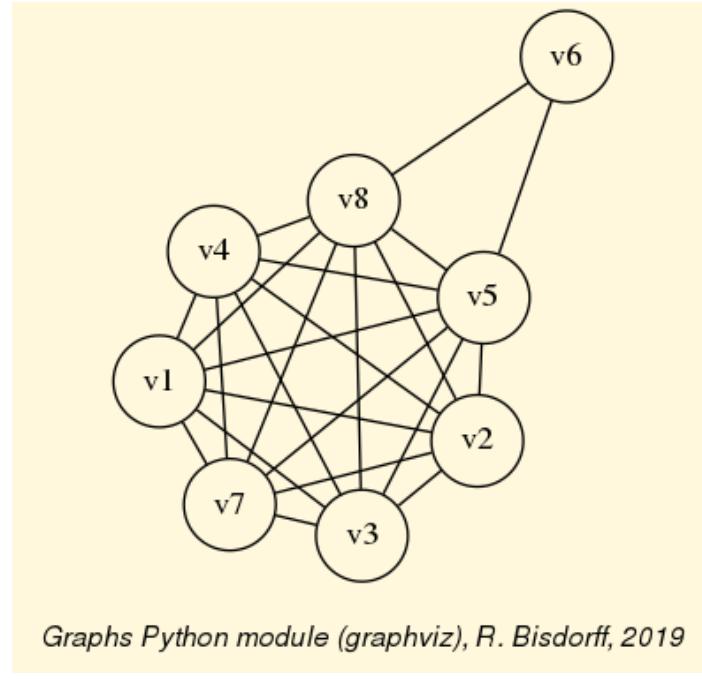
Fig. 1: A conjointly triangulated, comparability, interval, permutation and split graph

In Fig. 1 we may readily recognize the essential characteristic of **split graphs**, namely being always splitable into two disjoint sub-graphs: an *independent choice* (*v6*) and a *clique* (*v1*, *v2*, *v3*, *v4*, *v5*, *v7*, *v8*); which explains their name.

Notice however that the four properties:

1. *g* is a *comparability* graph;

2. *g* is a *cocomparability* graph, i.e. -*g* is a *comparability* graph;

3. *g* is a *triangulated* graph;

4. *g* is a *cotriangulated* graph, i.e. -*g* is a *comparability* graph;

are *independent* of one another (see [GOL-2004] p. 275).

16.2 Who is the liar ?

Claude Berge's famous mystery story (see [GOL-2004] p.20) may well illustrate the importance of being an **interval graph**.

Suppose that the file `berge.py`[18] contains the following `graphs.Graph` instance data:

```
1  vertices = {
2  'A': {'name': 'Abe', 'shortName': 'A'},
3  'B': {'name': 'Burt', 'shortName': 'B'},
4  'C': {'name': 'Charlotte', 'shortName': 'C'},
```

---

[18] A Digraph3 *graphs.Graph* encoded file is available in the `examples` directory of the Digraph3 software collection.

```
5  'D': {'name': 'Desmond', 'shortName': 'D'},
6  'E': {'name': 'Eddie', 'shortName': 'E'},
7  'I': {'name': 'Ida', 'shortName': 'I'},
8  }
9  valuationDomain = {'min':-1,'med':0,'max':1}
10 edges = {
11 frozenset(['A','B']) : 1,
12 frozenset(['A','C']) : -1,
13 frozenset(['A','D']) : 1,
14 frozenset(['A','E']) : 1,
15 frozenset(['A','I']) : -1,
16 frozenset(['B','C']) : -1,
17 frozenset(['B','D']) : -1,
18 frozenset(['B','E']) : 1,
19 frozenset(['B','I']) : 1,
20 frozenset(['C','D']) : 1,
21 frozenset(['C','E']) : 1,
22 frozenset(['C','I']) : 1,
23 frozenset(['D','E']) : -1,
24 frozenset(['D','I']) : 1,
25 frozenset(['E','I']) : 1,
26 }
```

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.

```
1  >>> from graphs import Graph
2  >>> g = Graph('berge')
3  >>> g.showShort()
4   *---- short description of the graph ----*
5   Name             : 'berge'
6   Vertices         :  ['A', 'B', 'C', 'D', 'E', 'I']
7   Valuation domain :  {'min': -1, 'med': 0, 'max': 1}
8   Gamma function   :
9   A -> ['D', 'B', 'E']
10  B -> ['E', 'I', 'A']
11  C -> ['E', 'D', 'I']
12  D -> ['C', 'I', 'A']
13  E -> ['C', 'B', 'I', 'A']
14  I -> ['C', 'E', 'B', 'D']
15 >>> g.exportGraphViz('berge1')
16  *---- exporting a dot file for GraphViz tools ---------*
17  Exporting to berge1.dot
18  fdp -Tpng berge1.dot -o berge1.png
```

Graphs Python module (graphviz), R. Bisdorff, 2011
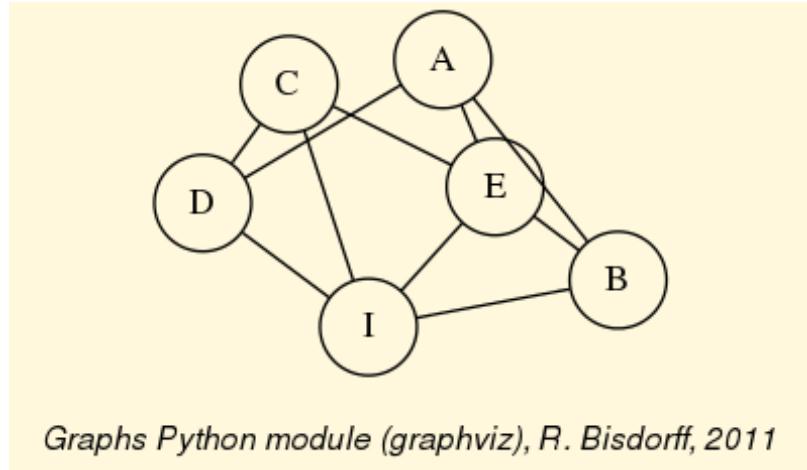
Fig. 2: Graph representation of the testimonies of the professors

From graph theory we know that time interval intersections graphs must in fact be interval graphs, i.e. *triangulated* and *co-comparative* graphs. The testimonies graph should therefore not contain any chordless cycle of four and more vertices. Now, the presence or not of such chordless cycles in the testimonies graph may be checked as follows.

```
1  >>> g.computeChordlessCycles()
2   Chordless cycle certificate -->>>  ['D', 'C', 'E', 'A', 'D']
3   Chordless cycle certificate -->>>  ['D', 'I', 'E', 'A', 'D']
4   Chordless cycle certificate -->>>  ['D', 'I', 'B', 'A', 'D']
5   [(['D', 'C', 'E', 'A', 'D'], frozenset({'C', 'D', 'E', 'A'})),
6    (['D', 'I', 'E', 'A', 'D'], frozenset({'D', 'E', 'I', 'A'})),
7    (['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 his testimony that he saw A (see Line 1 below), we obtain indeed a *triangulated* graph instance whose dual is a *comparability* graph.

```
1  >>> g.setEdgeValue( ('D','A'), 0)
2  >>> g.showShort()
3   *---- short description of the graph ----*
4   Name              : 'berge'
5   Vertices          : ['A', 'B', 'C', 'D', 'E', 'I']
6   Valuation domain :  {'med': 0, 'min': -1, 'max': 1}
7   Gamma function   :
8   A -> ['B', 'E']
9   B -> ['A', 'I', 'E']
10  C -> ['I', 'E', 'D']
11  D -> ['I', 'C']
12  E -> ['A', 'I', 'B', 'C']
13  I -> ['B', 'E', 'D', 'C']
14 >>> g.isIntervalGraph(Comments=True)
15  Graph 'berge' is triangulated.
```

```
16   Graph 'dual_berge' is transitively orientable.
17   => Graph 'berge' is an interval graph.
18   >>> g.exportGraphViz('berge2')
19   *---- exporting a dot file for GraphViz tools ---------*
20   Exporting to berge2.dot
21   fdp -Tpng berge2.dot -o berge2.png
```


Graphs Python module (graphviz), R. Bisdorff, 2011
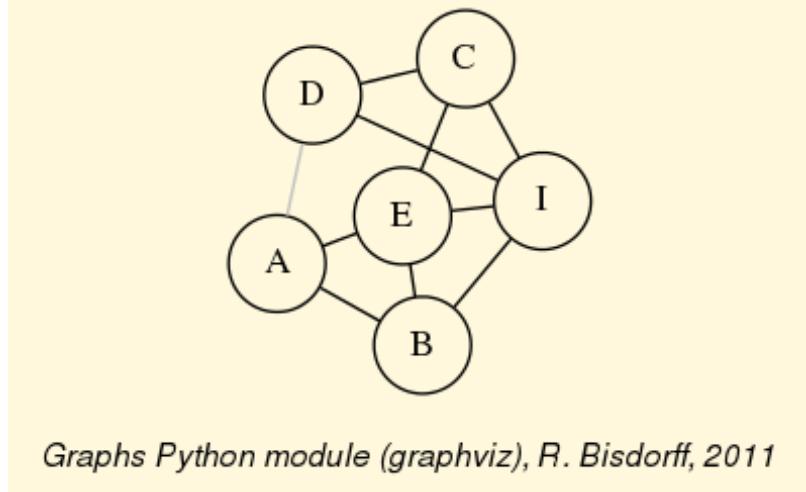
Fig. 3: The triangulated testimonies graph

16.3 Generating permutation graphs

A graph is called a **permutation** or *inversion* graph if there exists a permutation of its list of vertices such that the graph is isomorphic to the inversions operated by the permutation in this list (see [GOL-2004] Chapter 7, pp 157-170). This kind is also part of the class of perfect graphs.

```
1    >>> from graphs import PermutationGraph
2    >>> g = PermutationGraph(permutation = [4, 3, 6, 1, 5, 2])
3    >>> g
4     *------- Graph instance description ------*
5     Instance class    : PermutationGraph
6     Instance name     : permutationGraph
7     Graph Order       : 6
8     Permutation       : [4, 3, 6, 1, 5, 2]
9     Graph Size        : 9
10    Valuation domain  : [-1.00; 1.00]
11    Attributes        : ['name', 'vertices', 'order', 'permutation',
12                        'valuationDomain', 'edges', 'size', 'gamma']
13   >>> g.isPerfectGraph()
14    True
15   >>> g.exportGraphViz()
16    *---- exporting a dot file for GraphViz tools ---------*
```

```
17    Exporting to permutationGraph.dot
18    fdp -Tpng permutationGraph.dot -o permutationGraph.png
```
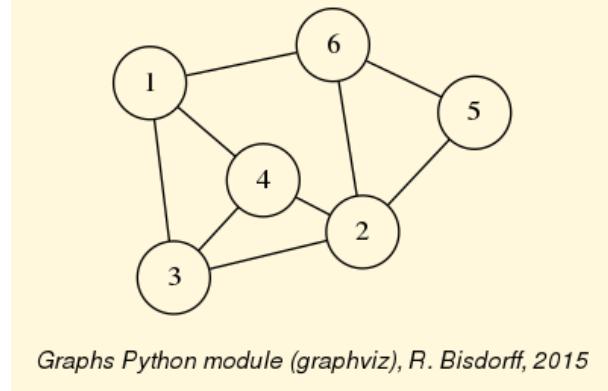

Fig. 4: The default permutation graph

By using color sorting queues, the minimal vertex coloring for a permutation graph is computable in $O(nlog(n))$ (see [GOL-2004]).

```
1    >>> g.computeMinimalVertexColoring(Comments=True)
2     vertex 1: lightcoral
3     vertex 2: lightcoral
4     vertex 3: lightblue
5     vertex 4: gold
6     vertex 5: lightblue
7     vertex 6: gold
8    >>> g.exportGraphViz(fileName='coloredPermutationGraph',\
9                    WithVertexColoring=True)
10    *---- exporting a dot file for GraphViz tools ---------*
11    Exporting to coloredPermutationGraph.dot
12    fdp -Tpng coloredPermutationGraph.dot -o coloredPermutationGraph.png
```
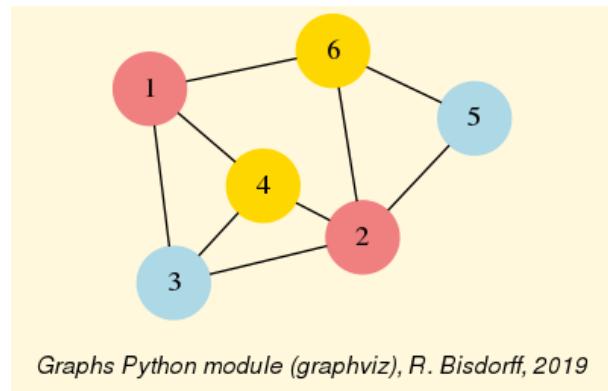

Fig. 5: Minimal vertex coloring of the permutation graph

The correspondingly colored **matching diagram** of the nine **inversions** -the actual *edges* of the permutation graph-, which are induced by the given permutation [4, 3, 6,

188

1, 5, 2], may as well be drawn with the graphviz *neato* layout and explicitly positioned horizontal lists of vertices (see Fig. 6).

```
1  >>> g.exportPermutationGraphViz(WithEdgeColoring=True)
2   *---- exporting a dot file for GraphViz tools ---------*
3   Exporting to perm_permutationGraph.dot
4   neato -n -Tpng perm_permutationGraph.dot -o perm_permutationGraph.png
```
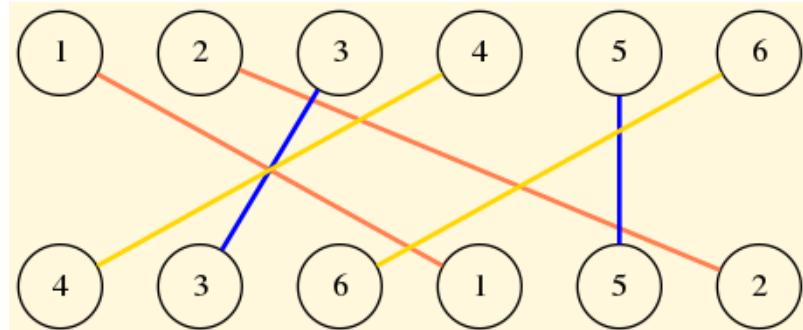

Graphs Python module (graphviz), R. Bisdorff, 2019

Fig. 6: Colored matching diagram of the permutation [4, 3, 6, 1, 5, 2]

As mentioned before, a permutation graph and its dual are **transitively orientable**. The `graphs.PermutationGraph.transitiveOrientation()` method constructs from a given permutation graph a digraph where each edge of the permutation graph is converted into an arc oriented in increasing alphabetic order of the adjacent vertices' keys (see [GOL-2004]). This orientation of the edges of a permutation graph is always transitive and delivers a *transitive ordering* of the vertices.

```
1  >>> dg = g.transitiveOrientation()
2  >>> dg
3   *------- Digraph instance description ------*
4   Instance class    : TransitiveDigraph
5   Instance name     : oriented_permutationGraph
6   Digraph Order       : 6
7   Digraph Size        : 9
8   Valuation domain : [-1.00; 1.00]
9   Determinateness   : 100.000
10  Attributes        : ['name', 'order', 'actions', 'valuationdomain',
11                       'relation', 'gamma', 'notGamma', 'size']
12 >>> print('Transitivity degree: %.3f' % dg.computeTransitivityDegree() )
13  Transitivity degree: 1.000
14 >>> dg.exportGraphViz()
15  *---- exporting a dot file for GraphViz tools ---------*
16  Exporting to oriented_permutationGraph.dot
17  0 { rank = same; 1; 2; }
18  1 { rank = same; 5; 3; }
19  2 { rank = same; 4; 6; }
20  dot -Grankdir=TB -Tpng oriented_permutationGraph.dot -o oriented_
   ↪permutationGraph.png
```
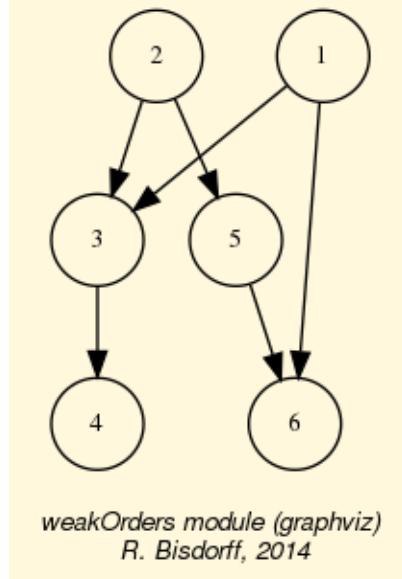
Fig. 7: Hasse diagram of the transitive orientation of the permutation graph

The dual of a permutation graph is *again* a permutation graph and as such also transitively orientable.

```
1  >>> dgd = (-g).transitiveOrientation()
2  >>> print('Dual transitivity degree: %.3f' %\
3                dgd.computeTransitivityDegree() )
4   Dual transitivity degree: 1.000
```

16.4 Recognizing permutation graphs

Now, a given graph g is a **permutation** graph **if and only if** both g **and** -g are *transitively orientable*. This property gives a polynomial test procedure (in $O(n^3)$ due to the transitivity check) for recognizing permutation graphs.

Let us consider, for instance, the following random graph of *order* 8 generated with an *edge probability* of 40% and a *random seed* equal to 4335.

```
1  >>> from graphs import *
2  >>> g = RandomGraph(order=8,edgeProbability=0.4,seed=4335)
3  >>> g
4   *------- Graph instance description ------*
5   Instance class    : RandomGraph
6   Instance name     : randomGraph
7   Seed              : 4335
8   Edge probability  : 0.4
9   Graph Order       : 8
```

```
10   Graph Size       : 10
11   Valuation domain : [-1.00; 1.00]
12   Attributes       : ['name', 'order', 'vertices', 'valuationDomain',
13                       'seed', 'edges', 'size',
14                       'gamma', 'edgeProbability']
15   >>> g.isPerfectGraph()
16   True
17   >>> g.exportGraphViz()
```


Graphs Python module (graphviz), R. Bisdorff, 2015
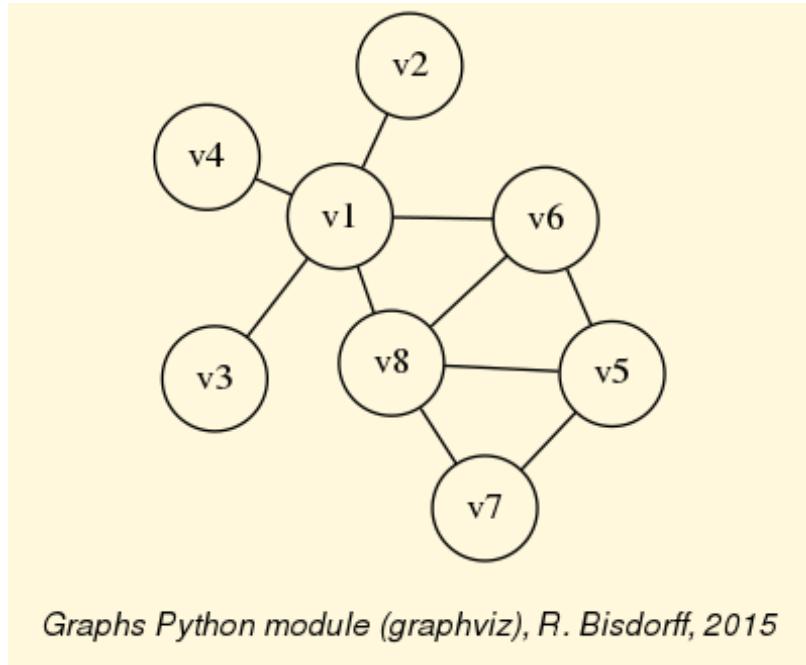
Fig. 8: Random graph of order 8 generated with edge probability 0.4

If the random perfect graph instance g (see Fig. 8) is indeed a permutation graph, g and its dual $-g$ should be *transitively orientable*, i.e. **comparability graphs** (see [GOL-2004]). With the `graphs.Graph.isComparabilityGraph()` test, we may easily check this fact. This method proceeds indeed by trying to construct a transitive neighbourhood decomposition of a given graph instance and, if successful, stores the resulting edge orientations into a *self.edgeOrientations* attribute (see [GOL-2004] p.129-132).

```
1   >>> if g.isComparabilityGraph():
2           print(g.edgeOrientations)
3    {('v1', 'v1'): 0, ('v1', 'v2'): 1, ('v2', 'v1'): -1, ('v1', 'v3'): 1,
4     ('v3', 'v1'): -1, ('v1', 'v4'): 1, ('v4', 'v1'): -1, ('v1', 'v5'): 0,
5     ('v5', 'v1'): 0, ('v1', 'v6'): 1, ('v6', 'v1'): -1, ('v1', 'v7'): 0,
6     ('v7', 'v1'): 0, ('v1', 'v8'): 1, ('v8', 'v1'): -1, ('v2', 'v2'): 0,
7     ('v2', 'v3'): 0, ('v3', 'v2'): 0, ('v2', 'v4'): 0, ('v4', 'v2'): 0,
8     ('v2', 'v5'): 0, ('v5', 'v2'): 0, ('v2', 'v6'): 0, ('v6', 'v2'): 0,
9     ('v2', 'v7'): 0, ('v7', 'v2'): 0, ('v2', 'v8'): 0, ('v8', 'v2'): 0,
10    ('v3', 'v3'): 0, ('v3', 'v4'): 0, ('v4', 'v3'): 0, ('v3', 'v5'): 0,
11    ('v5', 'v3'): 0, ('v3', 'v6'): 0, ('v6', 'v3'): 0, ('v3', 'v7'): 0,
```

```
12    ('v7', 'v3'): 0, ('v3', 'v8'): 0, ('v8', 'v3'): 0, ('v4', 'v4'): 0,
13    ('v4', 'v5'): 0, ('v5', 'v4'): 0, ('v4', 'v6'): 0, ('v6', 'v4'): 0,
14    ('v4', 'v7'): 0, ('v7', 'v4'): 0, ('v4', 'v8'): 0, ('v8', 'v4'): 0,
15    ('v5', 'v5'): 0, ('v5', 'v6'): 1, ('v6', 'v5'): -1, ('v5', 'v7'): 1,
16    ('v7', 'v5'): -1, ('v5', 'v8'): 1, ('v8', 'v5'): -1, ('v6', 'v6'): 0,
17    ('v6', 'v7'): 0, ('v7', 'v6'): 0, ('v6', 'v8'): 1, ('v8', 'v6'): -1,
18    ('v7', 'v7'): 0, ('v7', 'v8'): 1, ('v8', 'v7'): -1, ('v8', 'v8'): 0}
```


Graphs Python module (graphviz), R. Bisdorff, 2019
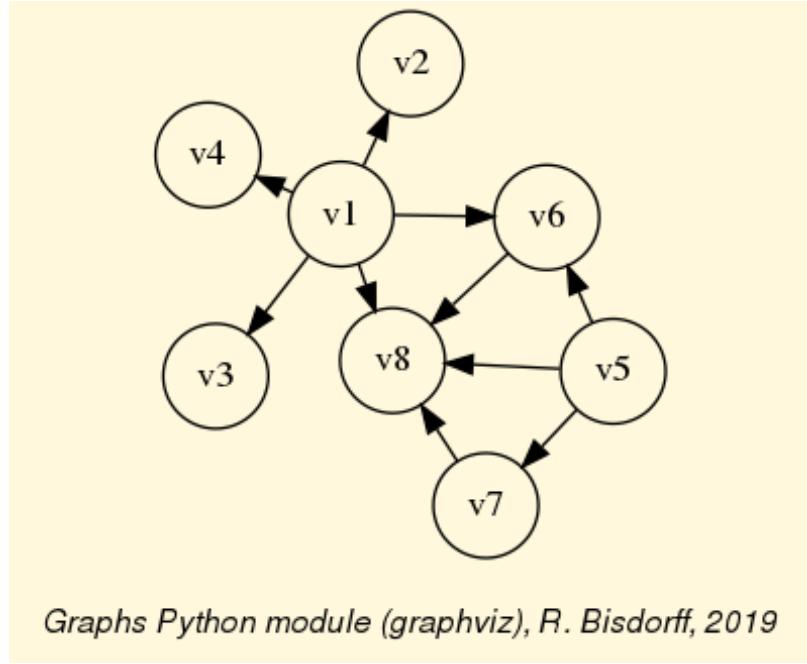
Fig. 9: Transitive neighbourhoods of the graph g

The resulting orientation of the edges of g (see Fig. 9) is indeed transitive. The same procedure applied to the dual graph $gd = $ -g gives a transitive orientation to the edges of -g.

```
1    >>> gd = -g
2    >>> if gd.isComparabilityGraph():
3            print(gd.edgeOrientations)
4     {('v1', 'v1'): 0, ('v1', 'v2'): 0, ('v2', 'v1'): 0, ('v1', 'v3'): 0,
5      ('v3', 'v1'): 0, ('v1', 'v4'): 0, ('v4', 'v1'): 0, ('v1', 'v5'): 1,
6      ('v5', 'v1'): -1, ('v1', 'v6'): 0, ('v6', 'v1'): 0, ('v1', 'v7'): 1,
7      ('v7', 'v1'): -1, ('v1', 'v8'): 0, ('v8', 'v1'): 0, ('v2', 'v2'): 0,
8      ('v2', 'v3'): -2, ('v3', 'v2'): 2, ('v2', 'v4'): -3, ('v4', 'v2'): 3,
9      ('v2', 'v5'): 1, ('v5', 'v2'): -1, ('v2', 'v6'): 1, ('v6', 'v2'): -1,
10     ('v2', 'v7'): 1, ('v7', 'v2'): -1, ('v2', 'v8'): 1, ('v8', 'v2'): -1,
11     ('v3', 'v3'): 0, ('v3', 'v4'): -3, ('v4', 'v3'): 3, ('v3', 'v5'): 1,
12     ('v5', 'v3'): -1, ('v3', 'v6'): 1, ('v6', 'v3'): -1, ('v3', 'v7'): 1,
13     ('v7', 'v3'): -1, ('v3', 'v8'): 1, ('v8', 'v3'): -1, ('v4', 'v4'): 0,
14     ('v4', 'v5'): 1, ('v5', 'v4'): -1, ('v4', 'v6'): 1, ('v6', 'v4'): -1,
15     ('v4', 'v7'): 1, ('v7', 'v4'): -1, ('v4', 'v8'): 1, ('v8', 'v4'): -1,
```

```
16    ('v5', 'v5'): 0, ('v5', 'v6'): 0, ('v6', 'v5'): 0, ('v5', 'v7'): 0,
17    ('v7', 'v5'): 0, ('v5', 'v8'): 0, ('v8', 'v5'): 0, ('v6', 'v6'): 0,
18    ('v6', 'v7'): 1, ('v7', 'v6'): -1, ('v6', 'v8'): 0, ('v8', 'v6'): 0,
19    ('v7', 'v7'): 0, ('v7', 'v8'): 0, ('v8', 'v7'): 0, ('v8', 'v8'): 0}
```


Graphs Python module (graphviz), R. Bisdorff, 2019
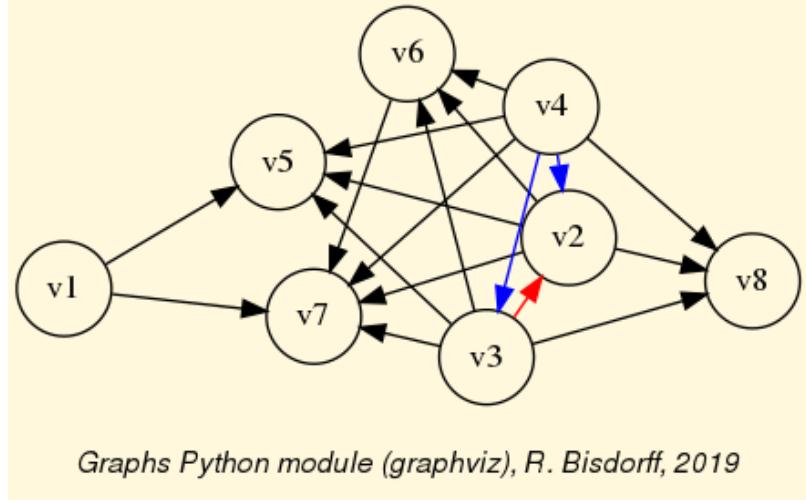
Fig. 10: Transitive neighbourhoods of the dual graph $-g$

It is worthwhile noticing that the orientation of g is achieved with a *single neighbourhood* decomposition, covering all the vertices. Whereas, the orientation of the dual $-g$ needs a decomposition into *three subsequent neighbourhoods* marked in black, red and blue (see Fig. 10).

Let us recheck these facts by explicitly constructing transitively oriented digraph instances with the `graphs.Graph.computeTransitivelyOrientedDigraph()` method.

```
1   >>> og = g.computeTransitivelyOrientedDigraph(PartiallyDetermined=True)
2   >>> print('Transitivity degree: %.3f' % (og.transitivityDegree))
3    Transitivity degree: 1.000
4   >>> ogd = (-g).computeTransitivelyOrientedDigraph(PartiallyDetermined=True)
5   >>> print('Transitivity degree: %.3f' % (ogd.transitivityDegree))
6    Transitivity degree: 1.000
```

The `PartiallyDetermined=True` flag (see Lines 1 and 5) is required here in order to orient *only* the actual edges of the graphs. Relations between vertices not linked by an edge will be put to the *indeterminate* characteristic value 0. This will allow us to compute, later on, convenient *disjunctive digraph fusions*.

As both graphs are indeed *transitively orientable* (see Lines 3 and 6 above), we may conclude that the given random graph g is actually a *permutation graph* instance. Yet, we still need to find now its corresponding *permutation*. We therefore implement a recipee given by Martin Golumbic [GOL-2004] p.159.

We will first **fuse** both *og* and *ogd* orientations above with an **epistemic disjunction** (see the `digraphsTools.omax()` operator), hence, the partially determined orientations requested above.

193

Listing 16.3: Fusing graph orientations

```
1  >>> from digraphs import FusionDigraph
2  >>> f1 = FusionDigraph(og,ogd,operator='o-max')
3  >>> s1 = f1.computeCopelandRanking()
4  >>> print(s1)
5   ['v5', 'v7', 'v1', 'v6', 'v8', 'v4', 'v3', 'v2']
```

We obtain by the *Copeland* ranking rule (see *Ranking with multiple incommensurable criteria* (page 59) and the `digraphs.Digraph.computeCopelandRanking()` method) a linear ordering of the vertices (see Listing 16.3 Line 5 above).

We reverse now the orientation of the edges in *og* (see -*og* in Line 1 below) in order to generate, again by *disjunctive fusion*, the *inversions* that are produced by the permutation we are looking for. Computing again a ranking with the *Copeland* rule, will show the correspondingly permuted list of vertices (see Line 4 below).

```
1  >>> f2 = FusionDigraph((-og),ogd,operator='o-max')
2  >>> s2 = f2.computeCopelandRanking()
3  >>> print(s2)
4   ['v8', 'v7', 'v6', 'v5', 'v4', 'v3', 'v2', 'v1']
```

Vertex *v8* is put from position 5 to position 1, vertex *v7* is put from position 2 to position 2, vertex *v6* from position 4 to position 3, 'vertex *v5* from position 1 to position 4, etc ... . We generate these position swaps for all vertices and obtain thus the required permutation (see Line 5 below).

```
1  >>> permutation = [0 for j in range(g.order)]
2  >>> for j in range(g.order):
3          permutation[s2.index(s1[j])] = j+1
4  >>> print(permutation)
5   [5, 2, 4, 1, 6, 7, 8, 3]
```

It is worthwhile noticing by the way that *transitive orientations* of a given graph and its dual are usually **not unique** and, so may also be the resulting permutations. However, they all correspond to isomorphic graphs (see [GOL-2004]). In our case here, we observe two different permutations and their reverses:

```
1  s1: ['v1', 'v4', 'v3', 'v2', 'v5', 'v6', 'v7', 'v8']
2  s2: ['v4', 'v3', 'v2', 'v8', 'v6', 'v1', 'v7', 'v5']
3  (s1 -> s2): [2, 3, 4, 8, 6, 1, 7, 5]
4  (s2 -> s1): [6, 1, 2, 3, 8, 5, 7, 4]
```

And:

```
1  s3: ['v5', 'v7', 'v1', 'v6', 'v8', 'v4', 'v3', 'v2']
2  s4: ['v8', 'v7', 'v6', 'v5', 'v4', 'v3', 'v2', 'v1']
3  (s3 -> s4): [5, 2, 4, 1, 6, 7, 8, 3]
4  (s4 -> s3) = [4, 2, 8, 3, 1, 5, 6, 7]
```

The `graphs.Graph.computePermutation()` method does directly operate all these steps:

- computing transitive orientations, - ranking their epistemic fusion and, - delivering a corresponding permutation.

```
1  >>> g.computePermutation(Comments=True)
2   ['v1', 'v2', 'v3', 'v4', 'v5', 'v6', 'v7', 'v8']
3   ['v2', 'v3', 'v4', 'v8', 'v6', 'v1', 'v7', 'v5']
4   [2, 3, 4, 8, 6, 1, 7, 5]
```

We may finally check that, for instance, the two permutations [2, 3, 4, 8, 6, 1, 7, 5] and [4, 2, 8, 3, 1, 5, 6, 7] observed above, will correctly generate corresponding *isomorphic permutation* graphs.

```
1  >>> gtesta = PermutationGraph(permutation=[2, 3, 4, 8, 6, 1, 7, 5])
2  >>> gtestb = PermutationGraph(permutation=[4, 2, 8, 3, 1, 5, 6, 7])
3  >>> gtesta.exportGraphViz('gtesta')
4  >>> gtestb.exportGraphViz('gtestb')
```
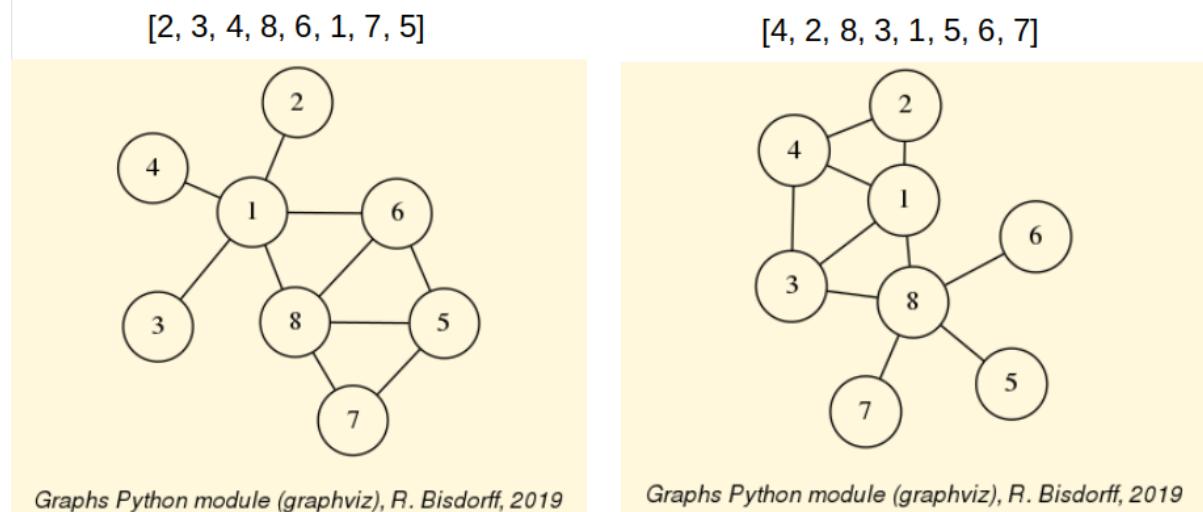

Fig. 11: Isomorphic permutation graphs

And, we recover indeed two *isomorphic copies* of the original random graph (compare Fig. 11 with Fig. 8).

Back to *Content Table* (page 3)

---

17 On tree graphs and graph forests

17.1 Generating random tree graphs

Using the `graphs.RandomTree` class, we may, for instance, generate a random tree graph with 9 vertices.

```
1  >>> t = RandomTree(order=9,seed=100)
2  >>> t
3   *------- Graph instance description ------*
4   Instance class   : RandomTree
5   Instance name    : randomTree
6   Graph Order      : 9
7   Graph Size       : 8
8   Valuation domain : [-1.00; 1.00]
9   Attributes       : ['name', 'order', 'vertices', 'valuationDomain',
10                      'edges', 'prueferCode', 'size', 'gamma']
11  *---- RandomTree specific data ----*
12  Prüfer code  : ['v3', 'v8', 'v8', 'v3', 'v7', 'v6', 'v7']
13 >>> t.exportGraphViz('tutRandomTree')
14  *---- exporting a dot file for GraphViz tools ---------*
15  Exporting to tutRandomTree.dot
16  neato -Tpng tutRandomTree.dot -o tutRandomTree.png
```
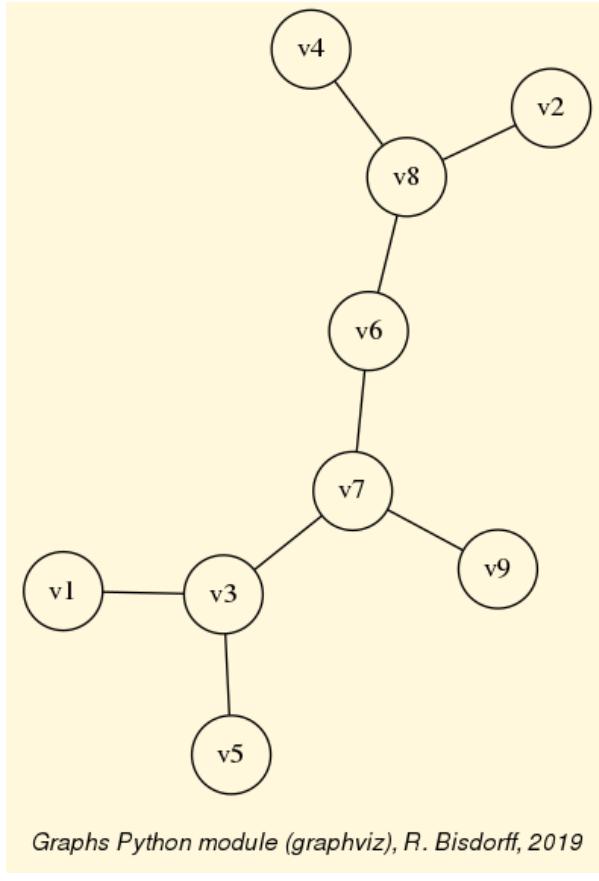
Fig. 1: Random Tree instance of order 9

A tree graph of order *n* contains *n-1* edges (see Line 8 and 9) and we may distinguish vertices like *v1*, *v2*, *v4*, *v5* or *v9* of degree 1, called the **leaves** of the tree, and vertices like *v3*, *v6*, *v7* or *v8* of degree 2 or more, called the **nodes** of the tree.

The structure of a tree of order $n > 2$ is entirely characterised by a corresponding *Prüfer* **code** -i.e. a *list of vertices keys*- of length *n-2*. See, for instance in Line 12 the code ['v3', 'v8', 'v8', 'v3', 'v7', 'v6', 'v7'] corresponding to our sample tree graph *t*.

Each position of the code indicates the parent of the remaining leaf with the smallest vertex label. Vertex *v3* is thus the parent of *v1* and we drop leaf *v1*, *v8* is now the parent of leaf *v2* and we drop *v2*, vertex *v8* is again the parent of leaf *v4* and we drop *v4*, vertex *v3* is the parent of leaf *v5* and we drop *v5*, *v7* is now the parent of leaf *v3* and we may drop *v3*, *v6* becomes the parent of leaf *v8* and we drop *v8*, *v7* becomes now the parent of leaf *v6* and we may drop *v6*. The two eventually remaining vertices, *v7* and *v9*, give the last link in the reconstructed tree (see [BAR-1991]).

It is as well possible to first, generate a random *Prüfer* code of length *n-2* from a set of *n* vertices and then, construct the corresponding tree of order *n* by reversing the procedure illustrated above (see [BAR-1991]).

```
1  >>> verticesList = ['v1','v2','v3','v4','v5','v6','v7']
2  >>> n = len(verticesList)
3  >>> from random import seed, choice
```

(continues on next page)

```
4  >>> seed(101)
5  >>> code = []
6  >>> for k in range(n-2):
7          code.append( choice(verticesList) )
8  >>> print(code)
9  ['v5', 'v7', 'v2', 'v5', 'v3']
10 >>> t = RandomTree(prueferCode=['v5', 'v7', 'v2', 'v5', 'v3'])
11 >>> t
12 *------- Graph instance description ------*
13 Instance class    : RandomTree
14 Instance name     : randomTree
15 Graph Order       : 7
16 Graph Size        : 6
17 Valuation domain : [-1.00; 1.00]
18 Attributes        : ['name', 'order', 'vertices', 'valuationDomain',
19                      'edges', 'prueferCode', 'size', 'gamma']
20 *---- RandomTree specific data ----*
21 Prüfer code  : ['v5', 'v7', 'v2', 'v5', 'v3']
22 >>> t.exportGraphViz('tutPruefTree')
23 *---- exporting a dot file for GraphViz tools ---------*
24 Exporting to tutPruefTree.dot
25 neato -Tpng tutPruefTree.dot -o tutPruefTree.png
```
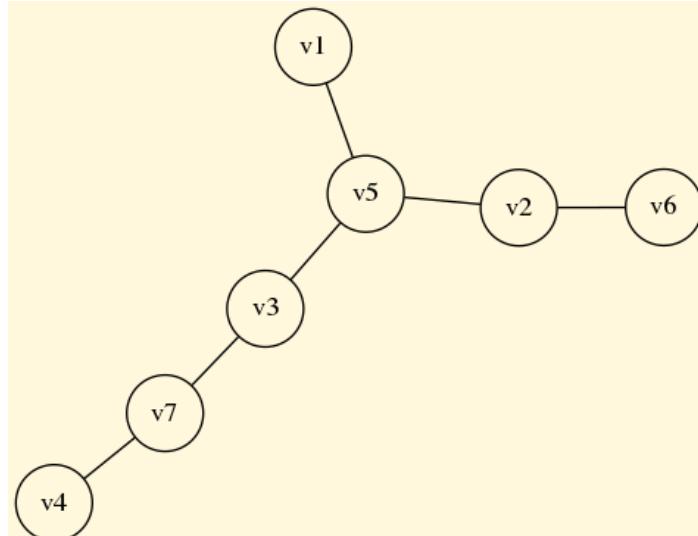

Graphs Python module (graphviz), R. Bisdorff, 2019

Fig. 2: Tree instance from a random Prüfer code

Following from the bijection between a labelled tree and its *Prüfer* code, we actually know that there exist n^{n-2} different tree graphs with the same n vertices.

Given a genuine graph, how can we recognize that it is in fact a tree instance ?

17.2 Recognizing tree graphs

Given a graph g of order n and size s, the following 5 assertions *A1*, *A2*, *A3*, *A4* and *A5* are all equivalent (see [BAR-1991]):

- *A1*: g is a tree;

- *A2*: g is without (chordless) cycles and $n = s + 1$;

- *A3*: g is connected and $n = s + 1$;

- *A4*: Any two vertices of g are always connected by a *unique path*;

- *A5*: g is connected and *dropping* any single edge will always disconnect *g*.

Assertion *A3*, for instance, gives a simple test for recognizing a tree graph. In case of a *lazy evaluation* of the test in Line 3 below, it is opportune, from a computational complexity perspective, to first, check the order and size of the graph, before checking its potential connectedness.

```
1  >>> from graphs import RandomGraph
2  >>> g = RandomGraph(order=6,edgeProbability=0.3,seed=62)
3  >>> if g.order == (g.size +1) and g.isConnected():
4         print('The graph is a tree ?', True)
5     else:
6         print('The graph is a tree ?',False)
7    The graph is a tree ? True
```

The random graph of order 6 and edge probability 30%, generated with seed 62, is actually a tree graph instance, as we may readily confirm from its *graphviz* drawing in Fig. 3 (see also the `graphs.Graph.isTree()` method for an implemented alternative test).

```
>>> g.exportGraphViz(
*---- exporting a dot file for GraphViz tools ---------*
Exporting to test62.dot
fdp -Tpng test62.dot -o test62.png
```
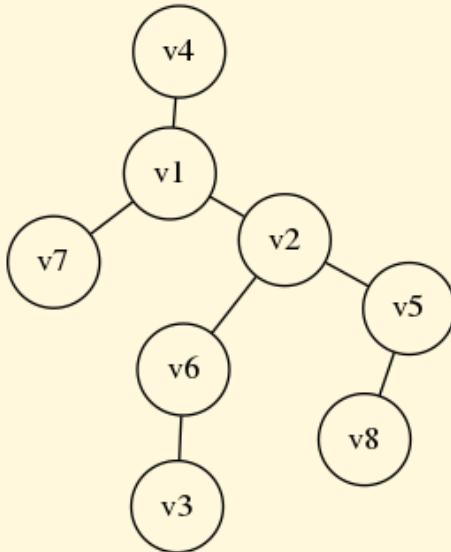
Graphs Python module (graphviz), R. Bisdorff, 2019

Fig. 3: Recognizing a tree instance

Yet, we still have to recover its corresponding *Prüfer* code. Therefore, we may use the `graphs.RandomTree.tree2Pruefer()` method.

```
>>> from graphs import RandomTree
>>> RandomTree.tree2Pruefer(g)
['v6', 'v1', 'v2', 'v1', 'v2', 'v5']
```

Let us now turn toward a major application of tree graphs, namely *spanning trees* and *forests* related to graph traversals.

17.3 Spanning trees and forests

With the `graphs.RandomSpanningTree` class we may generate, from a given **connected** graph *g* instance, **uniform random** instances of a **spanning tree** by using *Wilson*'s algorithm [WIL-1996]

Note: Wilson's algorithm *only* works for connected graphs[4].

```
1  >>> from graphs import *
2  >>> g = RandomGraph(order=9,edgeProbability=0.4,seed=100)
3  >>> spt = RandomSpanningTree(g)
4  >>> spt
5    *------- Graph instance description ------*
```

_____

[4] *Wilson*'s algorithm uses *loop-erased random walks.* See https://en.wikipedia.org/wiki/Loop-erased_random_walk .

```
6    Instance class    : RandomSpanningTree
7    Instance name     : randomGraph_randomSpanningTree
8    Graph Order       : 9
9    Graph Size        : 8
10   Valuation domain : [-1.00; 1.00]
11   Attributes        : ['name','vertices','order','valuationDomain',
12                        'edges','size','gamma','dfs','date',
13                        'dfsx','prueferCode']
14   *---- RandomTree specific data ----*
15   Prüfer code  : ['v7', 'v9', 'v5', 'v1', 'v8', 'v4', 'v9']
16   >>> spt.exportGraphViz(fileName='randomSpanningTree',\
17                       WithSpanningTree=True)
18   *---- exporting a dot file for GraphViz tools ---------*
19   Exporting to randomSpanningTree.dot
20   [['v1', 'v5', 'v6', 'v5', 'v1', 'v8', 'v9', 'v3', 'v9', 'v4',
21     'v7', 'v2', 'v7', 'v4', 'v9', 'v8', 'v1']]
22   neato -Tpng randomSpanningTree.dot -o randomSpanningTree.png
```
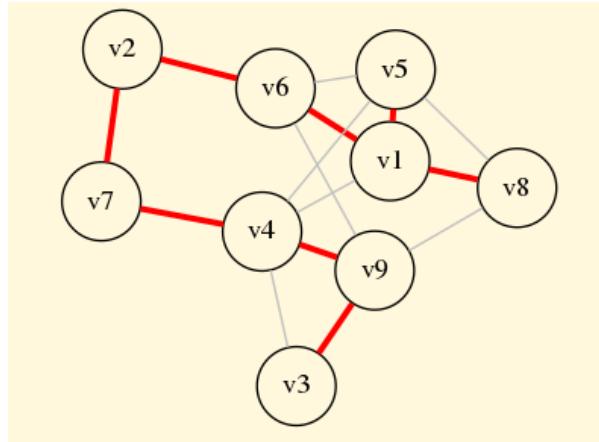

Graphs Python module (graphviz), R. Bisdorff, 2019

Fig. 4: Random spanning tree

More general, and in case of a not connected graph, we may generate with the `graphs.`
`RandomSpanningForest` class a *not necessarily uniform* random instance of a **spanning forest** -one or more random tree graphs- generated from a **random depth first search** of the graph components' traversals.

```
1    >>> g = RandomGraph(order=15,edgeProbability=0.1,seed=140)
2    >>> g.computeComponents()
3     [{'v12', 'v01', 'v13'}, {'v02', 'v06'},
4      {'v08', 'v03', 'v07'}, {'v15', 'v11', 'v10', 'v04', 'v05'},
5      {'v09', 'v14'}]
6    >>> spf = RandomSpanningForest(g,seed=100)
7    >>> spf.exportGraphViz(fileName='spanningForest',WithSpanningTree=True)
8     *---- exporting a dot file for GraphViz tools ---------*
```

```
9   Exporting to spanningForest.dot
10  [['v03', 'v07', 'v08', 'v07', 'v03'],
11   ['v13', 'v12', 'v13', 'v01', 'v13'],
12   ['v02', 'v06', 'v02'],
13   ['v15', 'v11', 'v04', 'v11', 'v15', 'v10', 'v05', 'v10', 'v15'],
14   ['v09', 'v14', 'v09']]
15  neato -Tpng spanningForest.dot -o spanningForest.png
```


Graphs Python module (graphviz), R. Bisdorff, 2019
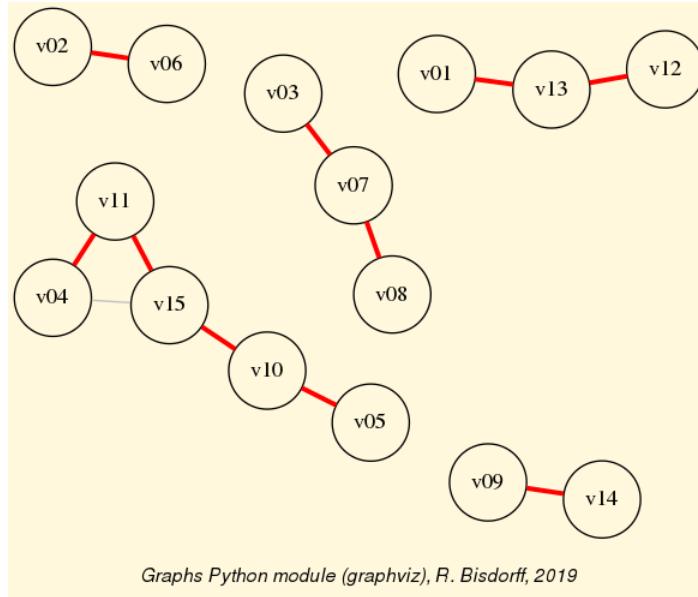
Fig. 5: Random spanning forest instance

17.4 Maximum determined spanning forests

In case of valued graphs supporting weighted edges, we may finally construct a **most determined** spanning tree (or forest if not connected) using *Kruskal's* *greedy* **minimum-spanning-tree algorithm**[5] on the *dual* valuation of the graph [KRU-1956].

We consider, for instance, a randomly valued graph with five vertices and seven edges bipolar-valued in [-1.0; 1.0].

```
1   >>> from graphs import *
2   >>> g = RandomValuationGraph(seed=2)
3   >>> print(g)
4   *------- Graph instance description ------*
5   Instance class    : RandomValuationGraph
6   Instance name     : randomGraph
7   Graph Order       : 5
8   Graph Size        : 7
```

---

[5] *Kruskal's* algorithm is a *minimum-spanning-tree* algorithm which finds an edge of the least possible weight that connects any two trees in the forest. See https://en.wikipedia.org/wiki/Kruskal%27s_algorithm .

```
9    Valuation domain : [-1.00; 1.00]
10   Attributes       : ['name', 'order', 'vertices', 'valuationDomain',
11                       'edges', 'size', 'gamma']
```

To inspect the edges' actual weights, we first transform the graph into a corresponding digraph (see Line 1 below) and use the `digraphs.Digraph.showRelationTable()` method (see Line 2 below) for printing its **symmetric adjacency matrix**.

```
1    >>> dg = g.graph2Digraph()
2    >>> dg.showRelationTable()
3     * ---- Relation Table -----
4      S  | 'v1'          'v2'     'v3'     'v4'     'v5'
5     ------|-------------------------------------------
6      'v1' |  0.00          0.91     0.90    -0.89    -0.83
7      'v2' |  0.91          0.00     0.67     0.47     0.34
8      'v3' |  0.90          0.67     0.00    -0.38     0.21
9      'v4' | -0.89          0.47    -0.38     0.00     0.21
10     'v5' | -0.83          0.34     0.21     0.21     0.00
11    Valuation domain: [-1.00;1.00]
```

To compute the most determined spanning tree or forest, we may use the `graphs.BestDeterminedSpanningForest` class constructor.

```
1    >>> mt = BestDeterminedSpanningForest(g)
2    >>> print(mt)
3     *------- Graph instance description ------*
4     Instance class   : BestDeterminedSpanningForest
5     Instance name    : randomGraph_randomSpanningForest
6     Graph Order      : 5
7     Graph Size       : 4
8     Valuation domain : [-1.00; 1.00]
9     Attributes       : ['name','vertices','order','valuationDomain',
10                        'edges','size','gamma','dfs',
11                        'date', 'averageTreeDetermination']
12    *---- best determined spanning tree specific data ----*
13    Depth first search path(s) :
14    [['v1', 'v2', 'v4', 'v2', 'v5', 'v2', 'v1', 'v3', 'v1']]
15    Average determination(s) : [Decimal('0.655')]
```

The given graph is connected and, hence, admits a single spanning tree (see Fig. 6) of **maximum mean determination** $= (0.47 + 0.91 + 0.90 + 0.34)/4 = \mathbf{0.655}$ (see Lines 9, 6 and 10 in the relation table above).

```
1    >>> mt.exportGraphViz(fileName='bestDeterminedspanningTree',\
2                          WithSpanningTree=True)
3     *---- exporting a dot file for GraphViz tools ---------*
4     Exporting to spanningTree.dot
5     [['v4', 'v2', 'v1', 'v3', 'v1', 'v2', 'v5', 'v2', 'v4']]
6     neato -Tpng bestDeterminedSpanningTree.dot -o bestDeterminedSpanningTree.png
```
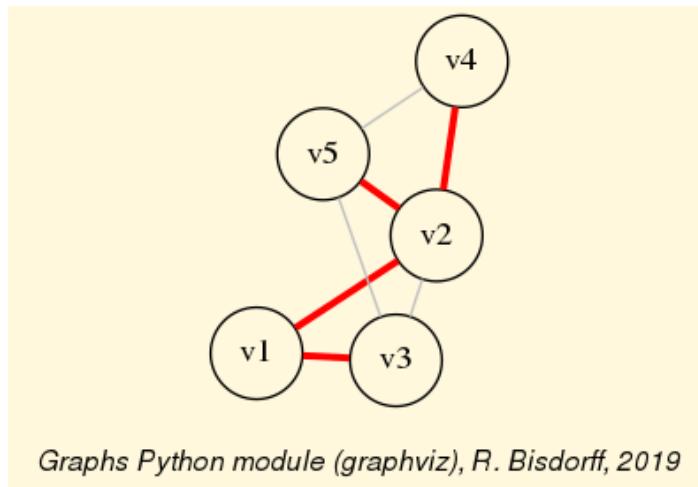
Fig. 6: Best determined spanning tree

One may easily verify that all other potential spanning trees, including instead the edges {*v3*, *v5*} and/or {*v4*, *v5*} - will show a lower average determination.

Back to

---

18 Appendices

References

[CPSTAT-L5]   Bisdorff R. (2017) "Simulating from abitrary empirical random distributions". MICS *Computational Statistics* course, Lecture 5. FSTC/ILIAS University of Luxembourg, Winter Semester 2017 (see http://hdl.handle.net/10993/37933).

[BIS-2016]   Bisdorff R. (2016). "Computing linear rankings from trillions of pairwise outranking situations". In Proceedings of DA2PL'2016 *From Multiple Criteria Decision Aid to Preference Learning*, R. Busa-Fekete, E. Hüllermeier, V. Mousseau and K. Pfannschmidt (Eds.), University of Paderborn (Germany), Nov. 7-8 2016: 1-6 (downloadable PDF file 451.4 kB (http://hdl.handle.net/10993/28613))

[BIS-2015]   Bisdorff R. (2015). "The EURO 2004 Best Poster Award: Choosing the Best Poster in a Scientific Conference". Chapter 5 in R. Bisdorff, L. Dias, P. Meyer, V. Mousseau, and M. Pirlot (Eds.), *Evaluation and Decision Models with Multiple Criteria: Case Studies.* Springer-Verlag Berlin Heidelberg, International Handbooks on Information Systems, DOI 10.1007/978-3-662-46816-6_1, pp. 117-166 (downloadable PDF file 754.7 kB (http://hdl.handle.net/10993/23714)).

[ADT-L2]   Bisdorff R. (2020) "Who wins the election?" MICS *Algorithmic Decision Theory* course, Lecture 2. FSTC/ILIAS University of Luxembourg, Summer Semester 2020 (see http://hdl.handle.net/10993/37933 and downloadable PDF file 199.5 kB).

[ADT-L7]   Bisdorff R.(2014) "Best multiple criteria choice: the Rubis outranking method". MICS *Algorithmic Decision Theory* course, Lecture 7. FSTC/ILIAS University of Luxembourg, Summer Semester 2014 (see http://hdl.handle.net/10993/37933 and downloadable PDF file 309.6 kB).

[BIS-2013] Bisdorff R. (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 (http://hdl.handle.net/10993/245)).

[BIS-2012] Bisdorff R. (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 (http://hdl.handle.net/10993/23909)).

[DIA-2010] Dias L.C. and Lamboray C. (2010). "Extensions of the prudence principle to exploit a valued outranking relation". *European Journal of Operational Research* Volume 201 Number 3 pp. 828-837.

[LAM-2009] Lamboray C. (2009) "A prudent characterization of the Ranked Pairs Rule". *Social Choice and Welfare* 32 pp. 129-155.

[BIS-2008] Bisdorff R., Meyer P. and Roubens M.(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 (http://hdl.handle.net/10993/23716)).

[ISOMIS-08] Bisdorff R. and Marichal J.-L. (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 (https://cs.uwaterloo.ca/journals/JIS/VOL11/Marichal/marichal.html)).

[NR3-2007] Press W.H., Teukolsky S.A., Vetterling W.T. and Flannery B.P. (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] Chambers J.M., James D.A., Lambert D. and Vander Wiel S. (2006) "Monitoring Networked Applications with Incremental Quantile Estimation". *Statistical Science*, Vol. 21, No.4, pp.463-475. DOI: 10 12140/088342306000000583.

[BIS-2006a] Bisdorff R., Pirlot M. and Roubens M. (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 (http://hdl.handle.net/10993/23720)).

[BIS-2006b] Bisdorff R. (2006). "On enumerating the kernels in a bipolar-valued digraph". Annales du Lamsade 6, Octobre 2006, pp. 1 - 38. Université Paris-Dauphine. ISSN 1762-455X (downloadable version PDF file 532.2 Kb (http://hdl.handle.net/10993/38741)).

[BIS-2004] Bisdorff R. (2004) "On a natural fuzzification of Boolean logic". In Erich Peter Klement and Endre Pap (editors), Proceedings of the 25th Linz Seminar on *Fuzzy Set Theory, Mathematics of Fuzzy Systems*. Bildungszentrum St. Magdalena, Linz (Austria), February 2004. pp. 20-26 (PDF file (133.4 Kb) for downloading (http://hdl.handle.net/10993/38740))

[GOL-2004] Golumbic M.Ch. (2004), *Agorithmic Graph Theory and Perfect Graphs* 2nd Ed., Annals of Discrete Mathematics 57, Elsevier.

[FMCAA] Häggström O. (2002) *Finite Markov Chains and Algorithmic Applications.* Cambridge University Press.

[BIS-2000] Bisdorff R. (2000), "Logical foundation of fuzzy preferential systems with application to the ELECTRE decision aid methods", *Computers and Operations Research*, 27:673-687 (downloadable version PDF file 159.1Kb (http://hdl.handle.net/10993/23724)).

[BIS-1999] Bisdorff R. (1999), "Bipolar ranking from pairwise fuzzy outrankings", JORBEL *Belgian Journal of Operations Research, Statistics and Computer Science*, Vol. 37 (4) 97 379-387. (PDF file (351.7 Kb) for downloading (http://hdl.handle.net/10993/38738))

[WIL-1996] Wilson D.B. (1996), *Generating random spanning trees more quickly than the cover time*, Proceedings of the Twenty-eighth Annual ACM *Symposium on the Theory of Computing* (Philadelphia, PA, 1996), 296-303, ACM, New York, 1996.

[BAR-1991] Barthélemy J.-P. and Guenoche A. (1991), *Trees and Proximities Representations*, Wiley, ISBN: 978-0471922636.

[KRU-1956] Kruskal J.B. (1956), *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proceedings of the American Mathematical Society. 7: 48–50.