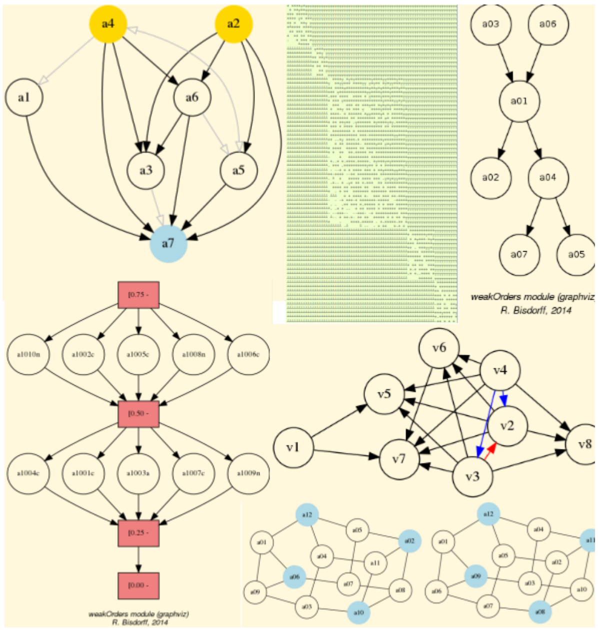
Documentation of the DIGRAPH3 software collection



Tutorials and Advanced Topics

Raymond BISDORFF

Luxembourg, 2020

Last updated : October 20, 2023

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

More documents are freely available here

A. 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. Finally, special tutorials are devoted to *perfect* graphs, like *split*, *interval* and *permutation* graphs, and to *tree-graphs* and *forests*.

Contents

---

1 Working with digraphs and outranking digraphs

1.1 Working with the *Digraph3* software resources

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.

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).

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

1. (*Recommended*) With a browser access, download and extract the latest distribution zip archive from

   https://github.com/rbisdorff/Digraph3 or, from

   https://sourceforge.net/projects/digraph3

2. By using a git client either, cloning from github

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

3. Or, from sourceforge.net

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

Starting a Python3 terminal session

You may start an interactive Python3 terminal session in the *Digraph3* directory.

```
$HOME/.../Digraph3$ python3
Python 3.10.0 (default, Oct 21 2021, 10:53:53)
[GCC 11.2.0] on linux Type "help", "copyright",
"credits" or "license" for more information.
>>>
```

For exploring the classes and methods provided by the *Digraph3* modules (see the Reference manual) enter the *Python3* commands following the session prompts marked with >>> or ... . The lines without the prompt are console output from the Python3 interpreter.

Listing 1.1: Generating a random digraph instance

```
>>> from randomDigraphs import RandomDigraph
>>> dg = RandomDigraph(order=5,arcProbability=0.5,seed=101)
>>> dg
 *------- Digraph instance description ------*
 Instance class   : RandomDigraph
 Instance name    : randomDigraph
 Digraph Order      : 5
 Digraph Size       : 12
 Valuation domain : [-1.00; 1.00]
 Determinateness  : 100.000
```

(continues on next page)

(continued from previous page)

```
11  Attributes         : ['actions', 'valuationdomain', 'relation',
12                        'order', 'name', 'gamma', 'notGamma',
13                        'seed', 'arcProbability', ]
```

In Listing 1.1 we import, for instance, from the `randomDigraphs` module the `RandomDigraph` class in order to generate a random *digraph* object *dg* of order 5 - number of nodes called (decision) *actions* - and arc probability of 50%. We may directly inspect the content of python object *dg* (Line 3).

> **Note:** For convenience of redoing the computations, all python code-blocks show in the upper right corner a specific **copy button** which allows to both copy *only* code lines, i.e. lines starting with '>>>' or '. . . ', and stripping the console prompts. The copied code lines may hence be right away *pasted* into a Python console session.

Digraph object structure

All `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 ordered dictionary of digraph nodes called **actions** (decision alternatives) 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** 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. A double dictionary called **relation** and 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** attribute, a dictionary containing the direct successors, respectively predecessors of each action, automatically added by the object constructor;
6. Its associated **notGamma** attribute, a dictionary containing the actions that are not direct successors respectively predecessors of each action, automatically added by the object constructor.

Permanent storage

The `save()` method stores the digraph object *dg* in a file named 'tutorialDigraph.py',

```
>>> dg.save('tutorialDigraph')
 *--- Saving digraph in file: <tutorialDigraph.py> ---*
```

with the following content

```
from decimal import Decimal
from collections import OrderedDict
actions = OrderedDict([
 ('a1', {'shortName': 'a1', 'name': 'random decision action'}),
 ('a2', {'shortName': 'a2', 'name': 'random decision action'}),
 ('a3', {'shortName': 'a3', 'name': 'random decision action'}),
 ('a4', {'shortName': 'a4', 'name': 'random decision action'}),
 ('a5', {'shortName': 'a5', 'name': 'random decision action'}),
 ])
valuationdomain = {'min': Decimal('-1.0'),
                   'med': Decimal('0.0'),
                   'max': Decimal('1.0'),
                   'hasIntegerValuation': True, # repr. format
                   }
relation = {
 'a1': {'a1':Decimal('-1.0'), 'a2':Decimal('-1.0'),
        'a3':Decimal('1.0'), 'a4':Decimal('-1.0'),
        'a5':Decimal('-1.0'),},
 'a2': {'a1':Decimal('1.0'), 'a2':Decimal('-1.0'),
        'a3':Decimal('-1.0'), 'a4':Decimal('1.0'),
        'a5':Decimal('1.0'),},
 'a3': {'a1':Decimal('1.0'), 'a2':Decimal('-1.0'),
        'a3':Decimal('-1.0'), 'a4':Decimal('1.0'),
        'a5':Decimal('-1.0'),},
 'a4': {'a1':Decimal('1.0'), 'a2':Decimal('1.0'),
        'a3':Decimal('1.0'), 'a4':Decimal('-1.0'),
        'a5':Decimal('-1.0'),},
 'a5': {'a1':Decimal('1.0'), 'a2':Decimal('1.0'),
        'a3':Decimal('1.0'), 'a4':Decimal('-1.0'),
        'a5':Decimal('-1.0'),},
 }
```

Inspecting a *Digraph* object

We may reload (see Listing 1.2) the previously saved digraph object from the file named 'tutorialDigraph.py' with the `Digraph` class constructor and different *show* methods (see Listing 1.2 below) reveal us that *dg* is a *crisp*, *irreflexive* and *connected* digraph of *order* five.

Listing 1.2: Random crisp digraph example

```
>>> from digraphs import Digraph
>>> dg = Digraph('tutorialDigraph')
>>> dg.showShort()
 *----- show short -------------*
 Digraph          : tutorialDigraph
 Actions          : OrderedDict([
  ('a1', {'shortName': 'a1', 'name': 'random decision action'}),
  ('a2', {'shortName': 'a2', 'name': 'random decision action'}),
  ('a3', {'shortName': 'a3', 'name': 'random decision action'}),
  ('a4', {'shortName': 'a4', 'name': 'random decision action'}),
  ('a5', {'shortName': 'a5', 'name': 'random decision action'})
  ])
 Valuation domain : {
  'min': Decimal('-1.0'),
  'max': Decimal('1.0'),
  'med': Decimal('0.0'), 'hasIntegerValuation': True
  }
>>> dg.showRelationTable()
 * ---- Relation Table -----
   S   |  'a1'  'a2'  'a3'  'a4'  'a5'
 ------|-------------------------------
  'a1' |   -1    -1     1    -1    -1
  'a2' |    1    -1    -1     1     1
  'a3' |    1    -1    -1     1    -1
  'a4' |    1     1     1    -1    -1
  'a5' |    1     1     1    -1    -1
 Valuation domain: [-1;+1]
>>> dg.showComponents()
 *--- Connected Components ---*
 1: ['a1', 'a2', 'a3', 'a4', 'a5']
>>> dg.showNeighborhoods()
 Neighborhoods:
   Gamma     :
 'a1': in => {'a2', 'a4', 'a3', 'a5'}, out => {'a3'}
 'a2': in => {'a5', 'a4'}, out => {'a1', 'a4', 'a5'}
 'a3': in => {'a1', 'a4', 'a5'}, out => {'a1', 'a4'}
 'a4': in => {'a2', 'a3'}, out => {'a1', 'a3', 'a2'}
 'a5': in => {'a2'}, out => {'a1', 'a3', 'a2'}
   Not Gamma :
```

(continues on next page)

(continued from previous page)

```
'a1': in => set(), out => {'a2', 'a4', 'a5'}
'a2': in => {'a1', 'a3'}, out => {'a3'}
'a3': in => {'a2'}, out => {'a2', 'a5'}
'a4': in => {'a1', 'a5'}, out => {'a5'}
'a5': in => {'a1', 'a4', 'a3'}, out => {'a4'}
```

The `exportGraphViz()` method generates in the current working directory a 'tutorialDigraph.dot' file and a 'tutorialdigraph.png' picture of the tutorial digraph *dg* (see Fig. 1.1), if the graphviz (https://graphviz.org/) tools are installed on your system[1].

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

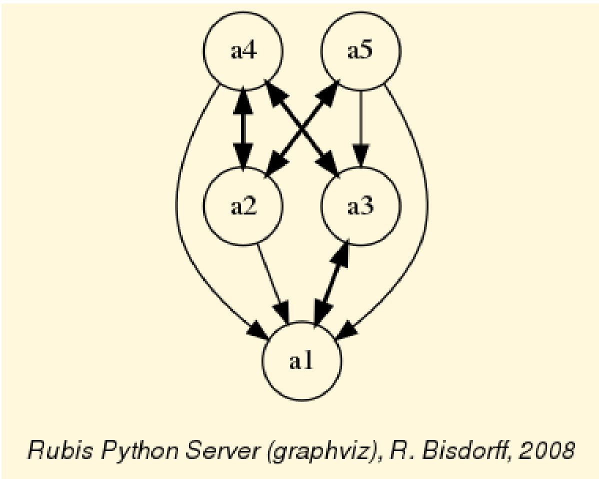


Fig. 1.1: The tutorial crisp digraph

Further methods are provided for inspecting this `Digraph` object *dg* , like the following `showStatistics()` method.

Listing 1.3: Inspecting a Digraph object

```
>>> dg.showStatistics()
 *----- general statistics -------------*
 for digraph              : <tutorialDigraph.py>
 order                    : 5 nodes
 size                     : 12 arcs
 # undetermined           : 0 arcs
 determinateness (%)      : 100.0
 arc density              : 0.60
```

(continues on next page)

[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.

(continued from previous page)

```
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 (%)  : 60.0
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
```

These *show* methods usually rely upon corresponding *compute* methods, like the `computeSize()`, the `computeDeterminateness()` or the `computeTransitivityDegree()` method (see Listing 1.3 Line 5,7,16).

```
>>> dg.computeSize()
12
>>> dg.computeDeterminateness(InPercents=True)
Decimal('100.00')
>>> dg.computeTransitivityDegree(InPercents=True)
Decimal('60.00')
```

Mind that *show* methods output their results in the Python console. We provide also some *showHTML* methods which output their results in a system browser's window.

```
>>> dg.showHTMLRelationMap(relationName='r(x,y)',rankingRule=None)
```

Relation Map

Ranking rule: Alphabetic

r(x,y)	a1	a2	a3	a4	a5
a1	—	—	+	—	—
a2	+	—	—	+	+
a3	+	—	—	+	—
a4	+	+	+	—	—
a5	+	+	+	—	—

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

Fig. 1.2: Browsing the relation map of the tutorial digraph

In Fig. 1.2 we find confirmed again that our random digraph instance *dg*, is indeed a crisp, i.e. 100% determined digraph instance.

Special *Digraph* instances

Some constructors for universal digraph instances, like the `CompleteDigraph`, the `EmptyDigraph` or the *circular oriented* `GridDigraph` constructor, are readily available (see Fig. 1.3).

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

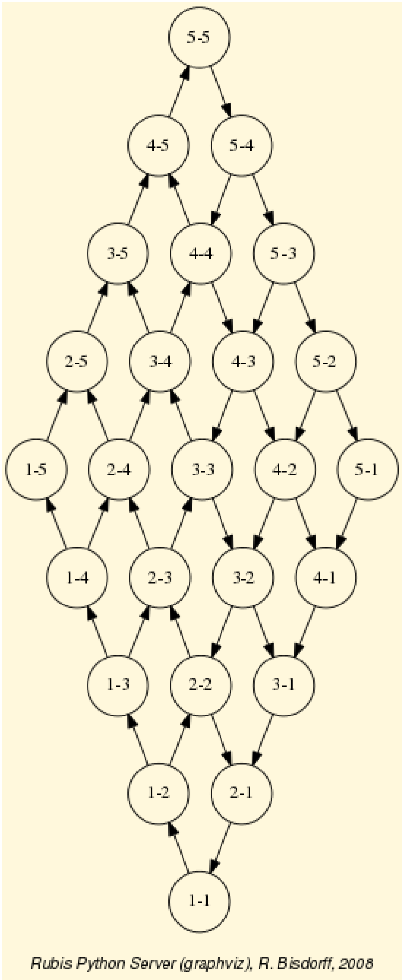


Fig. 1.3: The 5x5 grid graph median split oriented

Back to *Content Table* (page 1)

1.2 Working with the `digraphs` module

Random digraphs

We are starting this tutorial with generating a uniformly random [-1.0; +1.0]-valued digraph of order 7, denoted *rdg* and modelling, for instance, a binary relation ($x\ S\ y$) defined on the set of nodes of *rdg*. For this purpose, the *Digraph3* collection contains a `randomDigraphs` module providing a specific `RandomValuationDigraph` constructor.

Listing 1.4: Random bipolar-valued digraph instance

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

With the `save()` method (see Listing 1.4 Line 3) we may keep a backup version for future use of *rdg* which will be stored in a file called *tutRandValDigraph.py* in the current working directory. The genuine `Digraph` class constructor may restore the *rdg* object from the stored file (Line 4). We may easily inspect the content of *rdg* (Lines 5). The digraph size 22 indicates the number of positively valued arcs. The valuation domain is uniformly distributed in the interval $[-1.0; 1.0]$ and the mean absolute arc valuation is $(0.7524 \times 2) - 1.0 = 0.5048$ (Line 12) .

All `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 containing the direct neighbourhood of each action.

As mentioned previously, the `Digraph` class provides some generic *show…* methods for exploring a given *Digraph* object, like the `showShort()`, `showAll()`, `showRelationTable()` and the `showNeighborhoods()` methods.

Listing 1.5: Example of random valuation digraph

```
1 >>> rdg.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** links by considering that they are **indeterminate**, i.e. the characteristic value $r(x\,S\,x)$ for all action x is set to the *median*, i.e. *indeterminate* value 0.0 in this case (see Listing 1.5 Lines 12-18 and [BIS-2004a]).

Graphviz drawings

We may even get a better insight into the `Digraph` object *rdg* by looking at a graphviz (https://graphviz.org/) drawing[Page 7, 1] .

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

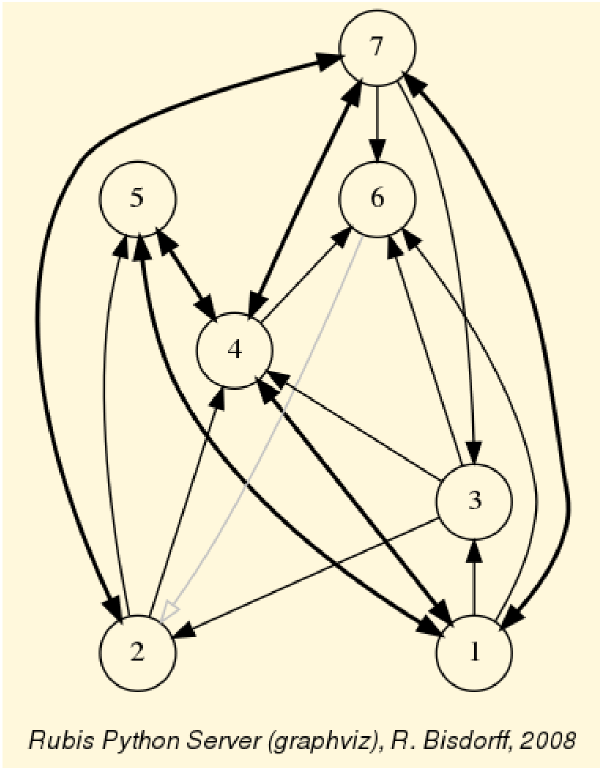


Fig. 1.4: The tutorial random valuation digraph

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. 1.4).

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. 1.5).

```
>>> from digraphs import AsymmetricPartialDigraph,
...                        SymmetricPartialDigraph

>>> asymDg = AsymmetricPartialDigraph(rdg)
>>> asymDg.exportGraphViz()
>>> symDg = SymmetricPartialDigraph(rdg)
>>> symDg.exportGraphViz()
```

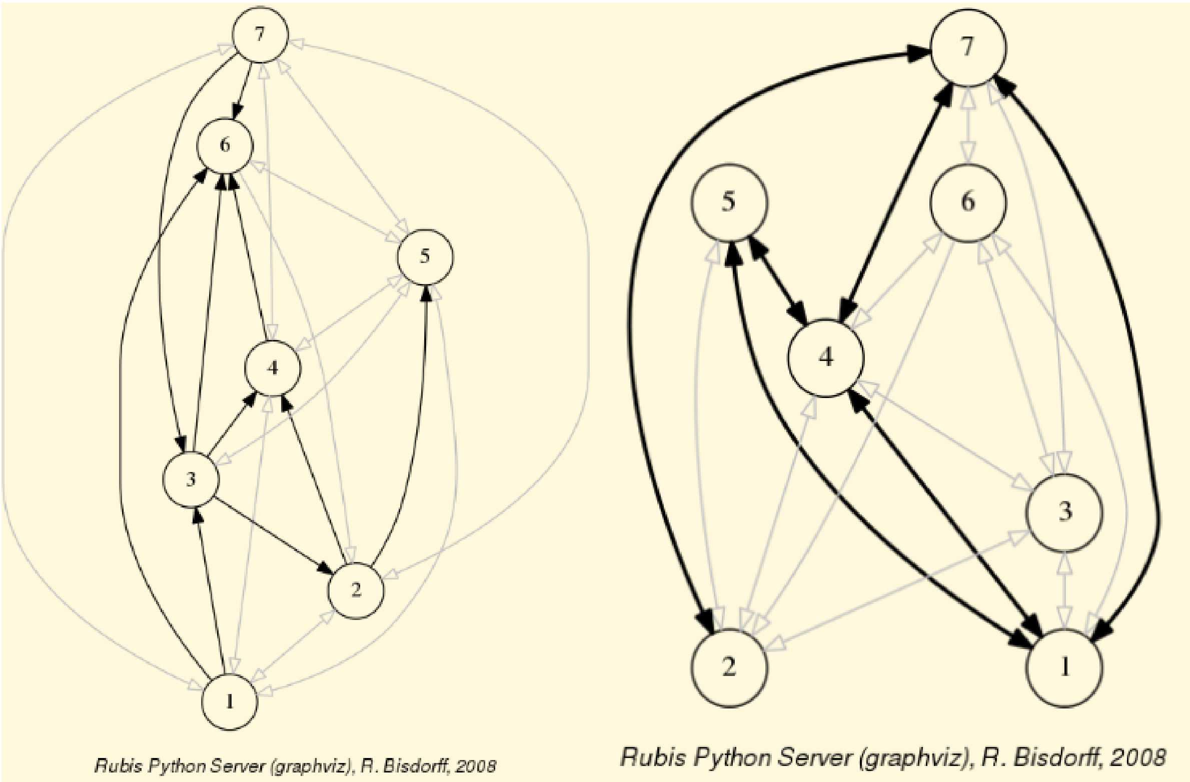


Fig. 1.5: Asymmetric and symmetric part of the tutorial random valuation digraph

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. 1.5).

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

```
def _constructRelation(self):
    actions = self.actions
```

(continues on next page)

(continued from previous page)

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

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 Kernel-Tutorial-label)- as well as, the inner part -the *complement* of the border- with the help of two corresponding class constructors: `GraphBorder` and `GraphInner` (see Listing 1.6).

Let us illustrate these parts on a linear ordering obtained from the tutorial random valuation digraph *rdg* with the *NetFlows ranking rule* (page 78) (see Listing 1.6 Line 2-3).

Listing 1.6: Border and inner part of a linear order

```
1 >>> from digraphs import GraphBorder, GraphInner
2 >>> from linearOrders import NetFlowsOrder
3 >>> nf = NetFlowsOrder(rdg)
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'])
```

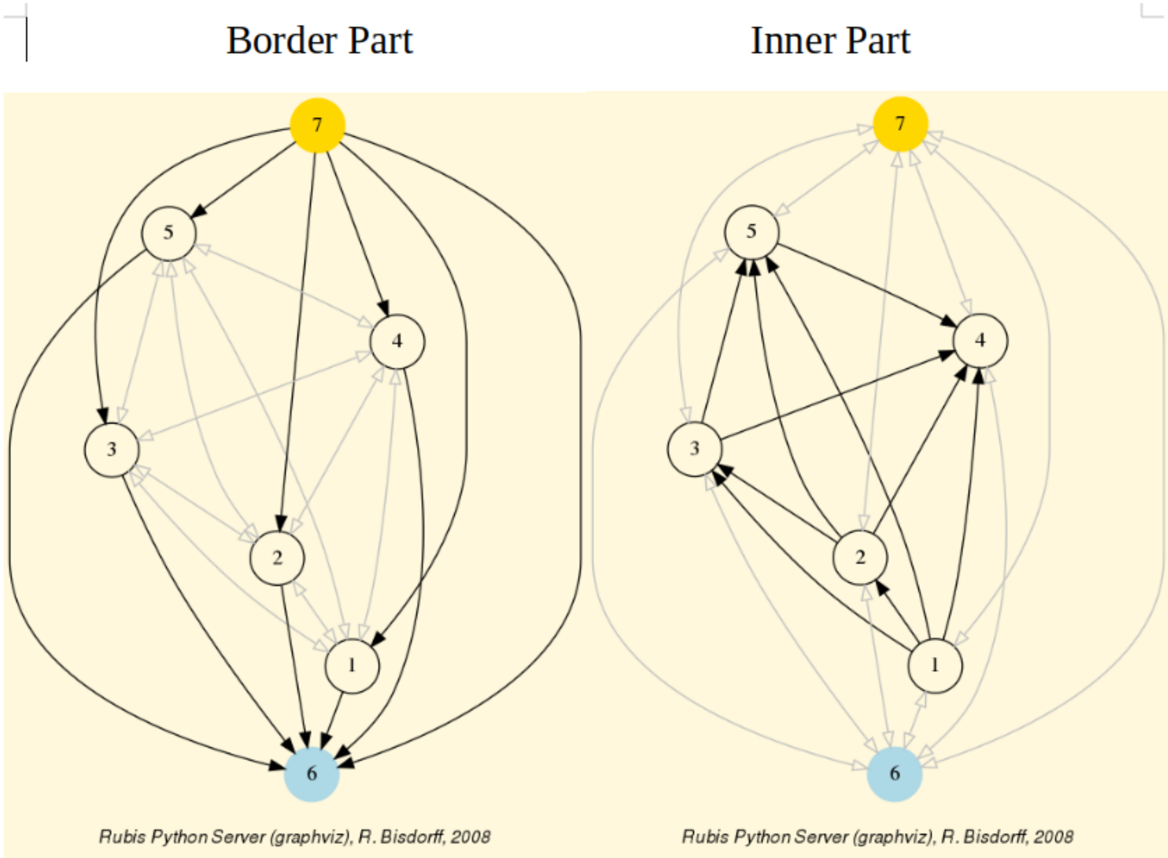


Fig. 1.6: *Border* and *inner* part of a linear order oriented by *terminal* and *initial* kernels

We may orient the graphviz drawings in Fig. 1.6 with the terminal node 6 (*worstChoice* parameter) and initial node 7 (*bestChoice* parameter), see Listing 1.6 Lines 7 and 9).

Note: The constructor of the partial digraphs *bnf* and *inf* (see Listing 1.6 Lines 3 and 6) puts to the *indeterminate* characteristic value all links *not* in the *border*, respectively *not* in the *inner* part (see Fig. 1.7).

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. 1.7).

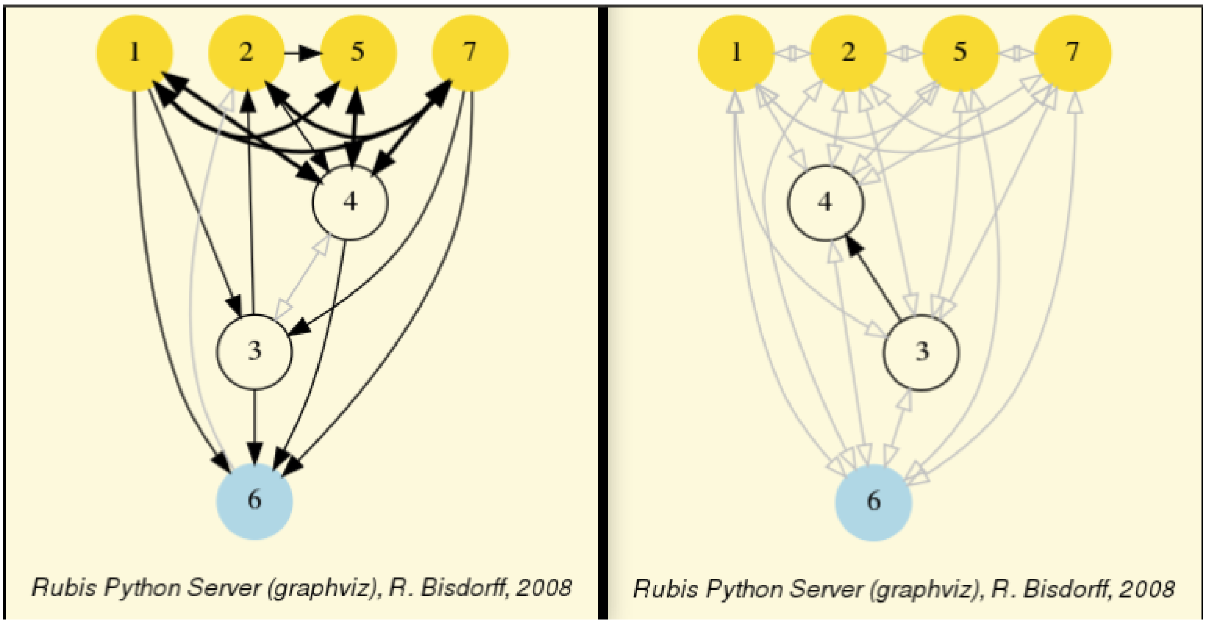


Fig. 1.7: Border and inner part of the tutorial random valuation digraph *rdg*

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.

Fusion by epistemic disjunction

We may recover object *rdg* 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 `FusionDigraph` class (see Listing 1.4 Lines 12-21).

Listing 1.7: Epistemic fusion of partial diagraphs

```
1  >>> from digraphs import FusionDigraph
2  >>> fusDg = FusionDigraph(asymDg,symDg,operator='o-max')
3  >>> # fusDg = FusionDigraph(bg,ig,operator='o-max')
4  >>> fusDg.showRelationTable()
5   * ---- Relation Table -----
6   r(xSy) |  '1'    '2'   '3'  '4'   '5'    '6'  '7'
7   -------|-----------------------------------------
8   '1'    |  0.00 -0.48  0.70  0.86  0.30  0.38  0.44
9   '2'    | -0.22  0.00 -0.38  0.50  0.80 -0.54  0.02
10  '3'    | -0.42  0.08  0.00  0.70 -0.56  0.84 -1.00
11  '4'    |  0.44 -0.40 -0.62  0.00  0.04  0.66  0.76
12  '5'    |  0.32 -0.48 -0.46  0.64  0.00 -0.22 -0.52
13  '6'    | -0.84  0.00 -0.40 -0.96 -0.18  0.00 -0.22
14  '7'    |  0.88  0.72  0.82  0.52 -0.84  0.04  0.00
```

The *epistemic fusion* (page 17) operator **o-max** (see Listing 1.7 Line 2) works as follows.

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

- o-max(*r*, *r'*) = max(*r*, *r'*) when both *r* and *r'* are more or less valid or indeterminate;
- o-max(*r*, *r'*) = min(*r*, *r'*) when both *r* and *r'* are more or less invalid or indeterminate;
- o-max(*r*, *r'*) = *indeterminate* otherwise.

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 *rdg*.

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

(continues on next page)

[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.

(continued from previous page)

```
 '2'      |  0.48  0.00 -0.08  0.40  0.48  0.00 -0.72
 '3'      | -0.70  0.38  0.00  0.62  0.46  0.40 -0.82
 '4'      | -0.86 -0.50 -0.70  0.00 -0.64  0.96 -0.52
 '5'      | -0.30 -0.80  0.56 -0.04  0.00  0.18  0.84
 '6'      | -0.38  0.54 -0.84 -0.66  0.22  0.00 -0.04
 '7'      | -0.44 -0.02  1.00 -0.76  0.52  0.22  0.00
```

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

Listing 1.8: Computing the *dual*, the *converse* and the *codual* of a digraph

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

Symmetric and transitive closures

Symmetric and transitive closures, by default in-site constructors, are also available (see Fig. 1.8). Note that it is a good idea, before going ahead with these in-site operations, who irreversibly modify the original *rdg* object, to previously make a backup version of *rdg*. The simplest storage method, always provided by the generic `save()`, writes out in a named file the python content of the Digraph object in string representation.

Listing 1.9: Symmetric and transitive in-site closures

```
1 >>> rdg.save('tutRandValDigraph')
2 >>> rdg.closeSymmetric(InSite=True)
3 >>> rdg.closeTransitive(InSite=True)
4 >>> rdg.exportGraphViz('strongComponents')
```

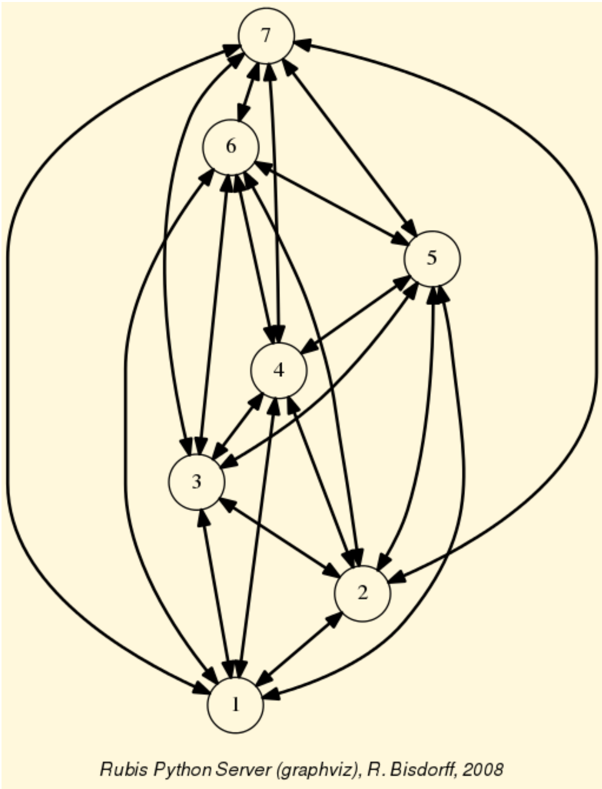


Fig. 1.8: Symmetric and transitive in-site closures

The `closeSymmetric()` method (see Listing 1.9 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 `closeTransitive()` method (see Listing 1.9 Line 3), implements the *Roy-Warshall* transitive closure algorithm of complexity $\mathcal{O}(n^3)$. ([17])

Note: The same `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 `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.

Strong components

As the original digraph *rdg* was connected (see above the result of the `showShort()` command), both the symmetric and the 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 *collapsed* digraph. Using the `StrongComponentsCollapsedDigraph` constructor here will render a single hyper-node gathering all the original nodes (see Line 7 below).

```
1  >>> from digraphs import StrongComponentsCollapsedDigraph
2  >>> sc = StrongComponentsCollapsedDigraph(dg)
3  >>> sc.showAll()
4   *----- show detail -----*
5   Digraph          : tutRandValDigraph_Scc
6   *---- Actions ----*
7   ['_7_1_2_6_5_3_4_']
8   * ---- Relation Table -----
9     S      |  'Scc_1'
10   -------|---------
11  'Scc_1' |  0.00
12  short        content
13  Scc_1        _7_1_2_6_5_3_4_
14  Neighborhoods:
15    Gamma     :
16  'frozenset({'7', '1', '2', '6', '5', '3', '4'})': in => set(), out => set()
17    Not Gamma :
18  'frozenset({'7', '1', '2', '6', '5', '3', '4'})': in => set(), out => set()
```

CSV storage

Sometimes it is required to exchange the graph valuation data in CSV format with a statistical package like R (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.

```
>>> rdg = Digraph('tutRandValDigraph')
>>> rdg.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
```

(continues on next page)

(continued from previous page)

```
"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.

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

It is as well possible to show a colored version of the valued relation table in a system browser window tab (see Fig. 1.9).

```
>>> rdgcsv.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. 1.9: 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*.

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 **CompleteDigraph**, the **EmptyDigraph** and the **IndeterminateDigraph** classes, which put all characteristic values respectively to the *maximum*, the *minimum* or the median *indeterminate* characteristic value.

Listing 1.10: Complete, empty and indeterminate digraphs

```
>>> from digraphs import CompleteDigraph,EmptyDigraph,
...                            IndeterminateDigraph

>>> e = EmptyDigraph(order=5)
>>> e.showRelationTable()
 * ---- Relation Table -----
   S   |    '1'    '2'    '3'    '4'    '5'
 ---- -|---------------------------------
 '1'   |  -1.00  -1.00  -1.00  -1.00  -1.00
 '2'   |  -1.00  -1.00  -1.00  -1.00  -1.00
 '3'   |  -1.00  -1.00  -1.00  -1.00  -1.00
 '4'   |  -1.00  -1.00  -1.00  -1.00  -1.00
 '5'   |  -1.00  -1.00  -1.00  -1.00  -1.00
 >>> e.showNeighborhoods()
 Neighborhoods:
   Gamma     :
 '1': in => set(), out => set()
 '2': in => set(), out => set()
 '5': in => set(), out => set()
 '3': in => set(), out => set()
 '4': in => set(), out => set()
   Not Gamma :
 '1': in => {'2', '4', '5', '3'}, out => {'2', '4', '5', '3'}
 '2': in => {'1', '4', '5', '3'}, out => {'1', '4', '5', '3'}
 '5': in => {'1', '2', '4', '3'}, out => {'1', '2', '4', '3'}
 '3': in => {'1', '2', '4', '5'}, out => {'1', '2', '4', '5'}
 '4': in => {'1', '2', '5', '3'}, out => {'1', '2', '5', '3'}
>>> i = IndeterminateDigraph()
 * ---- Relation Table -----
   S   |   '1'   '2'   '3'   '4'   '5'
 ------|-----------------------------
 '1'   |  0.00  0.00  0.00  0.00  0.00
 '2'   |  0.00  0.00  0.00  0.00  0.00
 '3'   |  0.00  0.00  0.00  0.00  0.00
 '4'   |  0.00  0.00  0.00  0.00  0.00
 '5'   |  0.00  0.00  0.00  0.00  0.00
>>> i.showNeighborhoods()
 Neighborhoods:
```

(continues on next page)

(continued from previous page)

```
39      Gamma      :
40    '1': in => set(), out => set()
41    '2': in => set(), out => set()
42    '5': in => set(), out => set()
43    '3': in => set(), out => set()
44    '4': in => set(), out => set()
45      Not Gamma :
46    '1': in => set(), out => set()
47    '2': in => set(), out => set()
48    '5': in => set(), out => set()
49    '3': in => set(), out => set()
50    '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 1.10 Lines 22-27) whereas, in the latter, *nothing is known* about the actual neighborhoods of the nodes (see Listing 1.10 Lines 45-50). These two cases illustrate why in the case of **bipolar-valued** digraphs, we may need both a *gamma* **and** a *notGamma* attribute.

Back to *Content Table* (page 1)

1.3 Working with the `outrankingDigraphs` module

> "*The rule for the combination of independent concurrent arguments takes a very simple form when expressed in terms of the intensity of belief … It is this: Take the sum of all the feelings of belief which would be produced separately by all the arguments pro, subtract from that the similar sum for arguments con, and the remainder is the feeling of belief which ought to have the whole. This is a proceeding which men often resort to, under the name of balancing reasons.*"
>
> —C.S. Peirce, The probability of induction (1878)

Outranking digraph model

In this *Digraph3* module, the `BipolarOutrankingDigraph` class from the `outrankingDigraphs` module provides our standard **outranking digraph** constructor. Such an instance represents a **hybrid** object of both, the `PerformanceTableau` type and the `OutrankingDigraph` type. A given object consists hence in:

1. an ordered dictionary of decision **actions** describing the potential decision actions or alternatives with 'name' and 'comment' attributes,
2. a possibly empty ordered dictionary of decision **objectives** with 'name' and 'comment attributes, describing the multiple preference dimensions involved in the decision problem,
3. a dictionary of performance **criteria** describing *preferentially independent* and *non-redundant* decimal-valued functions used for measuring the performance of each potential decision action with respect to a decision objective,
4. a double dictionary **evaluation** gathering performance grades for each decision action or alternative on each criterion function.
5. the digraph **valuationdomain**, a dictionary with three entries: the *minimum* (-1.0, certainly outranked), the *median* (0.0, indeterminate) and the *maximum* characteristic value (+1.0, certainly outranking),
6. 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.

Let us construct, for instance, a random bipolar-valued outranking digraph with seven decision actions denotes *a1*, *a2*, . . . , *a7*. We need therefore to first generate a corresponding random performance tableaux (see below).

```
>>> from outrankingDigraphs import *
>>> pt = RandomPerformanceTableau(numberOfActions=7,
...                               seed=100)

>>> pt
*------- PerformanceTableau instance description ------*
 Instance class     : RandomPerformanceTableau
 Seed               : 100
 Instance name      : randomperftab
 # Actions          : 7
 # Criteria         : 7
 NaN proportion (%) : 6.1
>>> pt.showActions()
    *----- show digraphs actions --------------*
    key:  a1
    name:       action #1
    comment:    RandomPerformanceTableau() generated.
```

(continues on next page)

(continued from previous page)

```
    key:  a2
    name:       action #2
    comment:    RandomPerformanceTableau() generated.
    ...
    ...
    key:  a7
    name:       action #7
    comment:    RandomPerformanceTableau() generated.
```

In this example we consider furthermore a family of seven **equisignificant cardinal criteria functions** *g1*, *g2*, . . . , *g7*, measuring the performance of each alternative on a rational scale from 0.0 (worst) to 100.00 (best). In order to capture the grading procedure's potential uncertainty and imprecision, each criterion function *g1* to *g7* admits three performance **discrimination thresholds** of 2.5, 5.0 and 80 pts for warranting respectively any indifference, preference or considerable performance difference situation.

```
1  >>> pt.showCriteria()
2   *----  criteria -----*
3   g1 'RandomPerformanceTableau() instance'
4     Scale = [0.0, 100.0]
5     Weight = 1.0
6     Threshold ind : 2.50 + 0.00x ; percentile: 4.76
7     Threshold pref : 5.00 + 0.00x ; percentile: 9.52
8     Threshold veto : 80.00 + 0.00x ; percentile: 95.24
9   g2 'RandomPerformanceTableau() instance'
10    Scale = [0.0, 100.0]
11    Weight = 1.0
12    Threshold ind : 2.50 + 0.00x ; percentile: 6.67
13    Threshold pref : 5.00 + 0.00x ; percentile: 6.67
14    Threshold veto : 80.00 + 0.00x ; percentile: 100.00
15  ...
16  ...
17  g7 'RandomPerformanceTableau() instance'
18    Scale = [0.0, 100.0]
19    Weight = 1.0
20    Threshold ind : 2.50 + 0.00x ; percentile: 0.00
21    Threshold pref : 5.00 + 0.00x ; percentile: 4.76
22    Threshold veto : 80.00 + 0.00x ; percentile: 100.00
```

On criteria function *g1* (see Lines 6-8 above) we observe, for instance, about 5% of **indifference**, about 90% of **preference** and about 5% of **considerable** performance difference situations. The individual performance evaluation of all decision alternative on each criterion are gathered in a *performance tableau*.

```
>>> pt.showPerformanceTableau()
 *----  performance tableau -----*
  criteria |  'a1'  'a2'  'a3'  'a4'  'a5'  'a6'  'a7'
```

(continues on next page)

(continued from previous page)

```
4    ---------|-----------------------------------------
5     'g1'    |  15.2  44.5  57.9  58.0  24.2  29.1  96.6
6     'g2'    |  82.3  43.9   NA   35.8  29.1  34.8  62.2
7     'g3'    |  44.2  19.1  27.7  41.5  22.4  21.5  56.9
8     'g4'    |  46.4  16.2  21.5  51.2  77.0  39.4  32.1
9     'g5'    |  47.7  14.8  79.7  67.5   NA   90.7  80.2
10    'g6'    |  69.6  45.5  22.0  33.8  31.8   NA   48.8
11    'g7'    |  82.9  41.7  12.8  21.9  75.7  15.4   6.0
```

It is noteworthy to mention the three **missing data** (*NA*) cases: action *a3* is missing, for instance, a grade on criterion *g2* (see Line 6 above).

The bipolar-valued outranking digraph

Given the previous random performance tableau *pt*, the `BipolarOutrankingDigraph` constructor computes the corresponding **bipolar-valued outranking digraph**.

Listing 1.11: Example of random bipolar-valued outranking digraph

```
1  >>> odg = BipolarOutrankingDigraph(pt)
2  >>> odg
3   *------- Object instance description ------*
4    Instance class       : BipolarOutrankingDigraph
5    Instance name        : rel_randomperftab
6    # Actions            : 7
7    # Criteria           : 7
8    Size                 : 20
9    Determinateness (%)  : 63.27
10   Valuation domain     : [-1.00;1.00]
11   Attributes           : [
12      'name', 'actions',
13      'criteria', 'evaluation', 'NA',
14      'valuationdomain', 'relation',
15      'order', 'gamma', 'notGamma', ...
16      ]
```

The resulting digraph contains 20 positive (valid) outranking realtions. And, the mean majority criteria significance support of all the pairwise outranking situations is 63.3% (see Listing 1.11 Lines 8-9). We may inspect the complete [-1.0,+1.0]-valued adjacency table as follows.

```
>>> odg.showRelationTable()
 * ---- Relation Table -----
  r(x,y)|  'a1'   'a2'   'a3'   'a4'   'a5'   'a6'   'a7'
  ------|------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
5     'a1' | +1.00  +0.71  +0.29  +0.29  +0.29  +0.29  +0.00
6     'a2' | -0.71  +1.00  -0.29  -0.14  +0.14  +0.29  -0.57
7     'a3' | -0.29  +0.29  +1.00  -0.29  -0.14  +0.00  -0.29
8     'a4' | +0.00  +0.14  +0.57  +1.00  +0.29  +0.57  -0.43
9     'a5' | -0.29  +0.00  +0.14  +0.00  +1.00  +0.29  -0.29
10    'a6' | -0.29  +0.00  +0.14  -0.29  +0.14  +1.00  +0.00
11    'a7' | +0.00  +0.71  +0.57  +0.43  +0.29  +0.00  +1.00
12     Valuation domain: [-1.0; 1.0]
```

Considering the given performance tableau *pt*, the `BipolarOutrankingDigraph` class constructor computes the characteristic value $r(x, y)$ of a **pairwise outranking** relation "$x \succsim y$" (see [BIS-2013], [ADT-L7]) in a default *normalised* **valuation domain** [-1.0,+1.0] with the *median value* 0.0 acting as **indeterminate** characteristic value. The semantics of $r(x, y)$ are the following.

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

Pairwise comparisons

From above given semantics, we may consider (see Line 5 above) that *a1* outranks *a2* ($r(a_1, a_2) > 0.0$), but not *a7* ($r(a_1, a_7) = 0.0$). In order to comprehend the characteristic values shown in the relation table above, we may furthermore inspect the details of the pairwise multiple criteria comparison between alternatives *a1* and *a2*.

```
>>> odg.showPairwiseComparison('a1','a2')
 *------------  pairwise comparison ----*
  Comparing actions : (a1, a2)
  crit. wght.  g(x)  g(y)    diff     | ind   pref    r()
  -------------------------------      --------------------
   g1    1.00  15.17  44.51  -29.34   | 2.50  5.00   -1.00
   g2    1.00  82.29  43.90  +38.39   | 2.50  5.00   +1.00
   g3    1.00  44.23  19.10  +25.13   | 2.50  5.00   +1.00
   g4    1.00  46.37  16.22  +30.15   | 2.50  5.00   +1.00
   g5    1.00  47.67  14.81  +32.86   | 2.50  5.00   +1.00
   g6    1.00  69.62  45.49  +24.13   | 2.50  5.00   +1.00
   g7    1.00  82.88  41.66  +41.22   | 2.50  5.00   +1.00
```

(continues on next page)

(continued from previous page)

```
  ----------------------------------------
  Valuation in range: -7.00 to +7.00; r(x,y): +5/7 = +0.71
```

The outranking characteristic value $r(a_1 \succsim a_2)$ represents the **majority margin** resulting from the difference between the weights of the criteria in favor and the weights of the criteria in disfavor of the statement that alternative *a1* is at least as well performing as alternative *a2*. No considerable performance difference being observed above, no veto or counter-veto situation is triggered in this pairwise comparison. Such a situation is, however, observed for instance when we pairwise compare the performances of alternatives *a1* and *a7*.

```
>>> odg.showPairwiseComparison('a1','a7')
 *------------  pairwise comparison ----*
  Comparing actions : (a1, a7)
  crit. wght.  g(x)  g(y)    diff   | ind   pref    r()   |  v     veto
   -------------------------------   ------------------   -----------
   g1   1.00  15.17  96.58  -81.41 | 2.50  5.00   -1.00 | 80.00 -1.00
   g2   1.00  82.29  62.22  +20.07 | 2.50  5.00   +1.00 |
   g3   1.00  44.23  56.90  -12.67 | 2.50  5.00   -1.00 |
   g4   1.00  46.37  32.06  +14.31 | 2.50  5.00   +1.00 |
   g5   1.00  47.67  80.16  -32.49 | 2.50  5.00   -1.00 |
   g6   1.00  69.62  48.80  +20.82 | 2.50  5.00   +1.00 |
   g7   1.00  82.88   6.05  +76.83 | 2.50  5.00   +1.00 |
          ----------------------------------------
  Valuation in range: -7.00 to +7.00; r(x,y)= +1/7 => 0.0
```

This time, we observe a 57.1% majority of criteria significance $[(1/7 + 1)/2 = 0.571]$ warranting an *as well as performing* situation. Yet, we also observe a considerable negative performance difference on criterion *g1* (see first row in the relation table above). Both contradictory facts trigger eventually an *indeterminate* outranking situation [BIS-2013].

Recoding the digraph valuation

All outranking digraphs, being of root type `Digraph`, inherit the methods available under this latter class. The characteristic valuation domain of a digraph may, for instance, be recoded with the `recodeValutaion()` method below to the *integer* range [-7,+7], 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,ReflexiveTerms=False)
 * ---- Relation Table -----
  r(x,y)  |  'a1'  'a2'  'a3'  'a4'  'a5'  'a6'  'a7'
 ---------|------------------------------------------
   'a1'   |    0     5     2     2     2     2     0
```

(continues on next page)

(continued from previous page)

```
    'a2'   |   -5     0    -1    -1     1     2    -4
    'a3'   |   -1     2     0    -1    -1     0    -1
    'a4'   |    0     1     4     0     2     4    -3
    'a5'   |   -1     0     1     0     0     2    -1
    'a6'   |   -1     0     1    -1     1     0     0
    'a7'   |    0     5     4     3     2     0     0
 Valuation domain: [-7;+7]
```

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

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 $r(x, y) < 0.0$ then $r(y, x) \geq 0.0$. A bipolar-valued outranking relation verifies furthermore the **coduality** principle: the **dual** (*strict negation* -[Page 18, 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[Page 7, 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
```

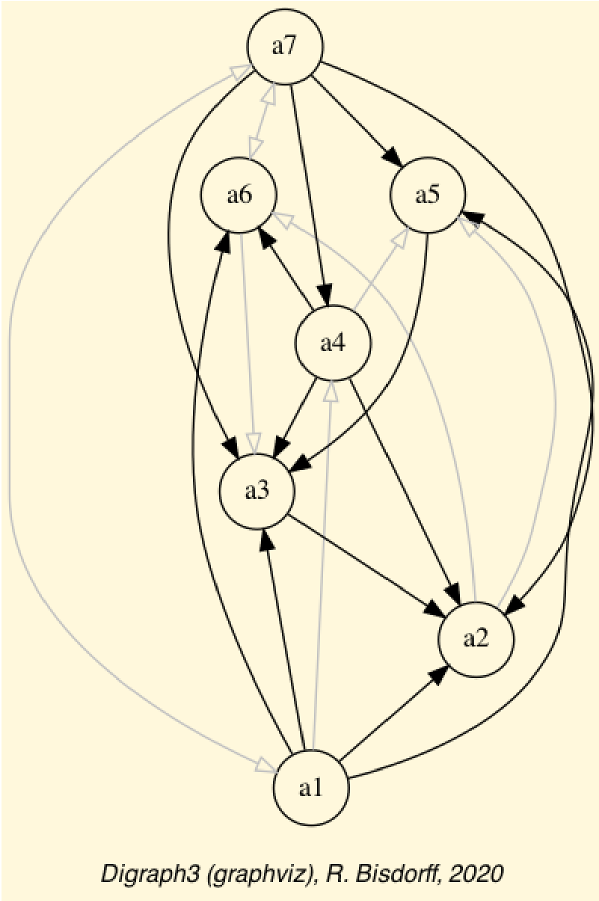


Fig. 1.10: Codual digraph

It becomes readily clear now from the picture above that both alternatives *a1* and *a7* are *not outranked* by any other alternatives. Hence, *a1* and *a7* appear as **weak Condorcet winner** and may be recommended as potential *best decision actions* in this illustrative preference modelling exercise.

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).

In this tutorial we have constructed a random outranking digraph with the help of a random performance tableau instance. The next *Digraph3* tutorial presents now different models of random performance tableaux illustrating various types of decision problems.

Back to *Content Table* (page 1)

---

2 Evaluation and decision methods and tools

2.1 Generating random performance tableaux with the `randPerfTabs` module

Introduction

The `randomPerfTabs` module provides several constructors for generating 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); 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 corresponding performance tableau model, called **RandomAcademicPerformanceTableau**, 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 with the votingProfiles module* (page 59)) 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).

Random standard performance tableaux

The `RandomPerformanceTableau` class, the simplest of the kind, specializes the generic `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).
- NA := <Decimal> (default = -999); missing data symbol.

Code example.

Listing 2.1: 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
               ...
               },
       ...
      }
>>> 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. 2.1: Browser view on random performance tableau instance

Note: Missing (NA) evaluation are registered in a performance tableau by default as *Decimal('-999')* value (see Listing 2.1 Line 24). Best and worst performance on each criterion are marked in *light green*, respectively in *light red*.

Random Cost-Benefit performance tableaux

We provide the `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.
- missingDataProbability := 0 <= float <= 1.0 ; probability of missing performance evaluation on a criterion for an alternative (default 0.025).
- NA := <Decimal> (default = -999); missing data symbol.

Warning: Minimal number of decision actions required is 3 !

Example Python session

Listing 2.2: Generating a random Cost-Benefit performance tableau

```
>>> from randomPerfTabs import RandomCBPerformanceTableau
>>> t = RandomCBPerformanceTableau(
...       numberOfActions=7,
...       numberOfCriteria=5,
...       weightDistribution='equiobjectives',
...       commonPercentiles={'ind':0.05,'pref':0.10,'veto':0.95},
...       seed=100)
```

(continues on next page)

(continued from previous page)

```
8
9  >>> t.showActions()
10  *----- show decision action --------------*
11  key:  a1
12    short name: a1
13    name:       random cheap decision action
14  key:  a2
15    short name: a2
16    name:       random neutral decision action
17  ...
18  key:  a7
19    short name: a7
20    name:       random advantageous decision action
21 >>> t.showCriteria()
22  *----  criteria -----*
23  g1 'random ordinal benefit criterion'
24    Scale = (0, 10)
25    Weight = 2
26  ...
27  g2 'random cardinal cost criterion'
28    Scale = (0.0, 100.0)
29    Weight = 3
30    Threshold ind  :  1.76 + 0.00x ; percentile:   9.5
31    Threshold pref :  2.16 + 0.00x ; percentile:  14.3
32    Threshold veto : 73.19 + 0.00x ; percentile:  95.2
33  ...
```

In the example above, we may notice the three types of decision actions (Listing 2.2 Lines 10-20), as well as the two types (Lines 22-32) of criteria with either an **ordinal** or a **cardinal** performance measuring scale. In the latter case, by default about 5% of the random performance differences will be below the **indifference** and 10% below the **preference discriminating threshold**. About 5% will be considered as **considerably large**. More statistics about the generated performances is available as follows.

```
>>> t.showStatistics()
 *-------- Performance tableau summary statistics -------*
 Instance name      : randomCBperftab
 #Actions           : 7
 #Criteria          : 5
  Criterion name       : g1
    Criterion weight     : 2
    criterion scale    : 0.00 - 10.00
    mean evaluation    : 5.14
    standard deviation : 2.64
    maximal evaluation : 8.00
    quantile Q3 (x_75) : 8.00
```

(continues on next page)

(continued from previous page)

```
    median evaluation  : 6.50
    quantile Q1 (x_25) : 3.50
    minimal evaluation : 1.00
    mean absolute difference      : 2.94
    standard difference deviation : 3.74
  Criterion name       : g2
    Criterion weight     : 3
    criterion scale    : -100.00 - 0.00
    mean evaluation    : -49.32
    standard deviation : 27.59
    maximal evaluation : 0.00
    quantile Q3 (x_75) : -27.51
    median evaluation  : -35.98
    quantile Q1 (x_25) : -54.02
    minimal evaluation : -91.87
    mean absolute difference      : 28.72
    standard difference deviation : 39.02
  ...
```

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

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

Heatmap of performance tableau

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

Color legend:

quantile	0.20%	0.40%	0.60%	0.80%	1.00%

Fig. 2.2: Unranked heatmap of a random Cost-Benefit performance tableau

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

```
>>> t.save('temp')
 *----- saving performance tableau in XMCDA 2.0 format  -------------*
```

(continues on next page)

(continued from previous page)

```
 File: temp.py saved !
>>> from perfTabs import PerformanceTableau
>>> t = PerformanceTableau('temp')
```

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

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

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

Back to *Content Table* (page 1)

Random three objectives performance tableaux

We provide the `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,
- NA := <Decimal> (default = -999); missing data symbol.
- 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 2.3: Generating a random 3 Objectives performance tableau

```
>>> from randomPerfTabs import Random3ObjectivesPerformanceTableau
>>> t = Random3ObjectivesPerformanceTableau(
...                 numberOfActions=31,
...                 numberOfCriteria=13,
...                 weightDistribution='equiobjectives',
...                 seed=120)

>>> t.showObjectives()
 *------ show objectives -------"
 Eco: Economical aspect
    ec04 criterion of objective Eco 20
    ec05 criterion of objective Eco 20
    ec08 criterion of objective Eco 20
    ec11 criterion of objective Eco 20
   Total weight: 80.00 (4 criteria)
 Soc: Societal aspect
    so06 criterion of objective Soc 16
    so07 criterion of objective Soc 16
    so09 criterion of objective Soc 16
    s010 criterion of objective Soc 16
    s013 criterion of objective Soc 16
   Total weight: 80.00 (5 criteria)
 Env: Environmental aspect
    en01 criterion of objective Env 20
```

(continues on next page)

(continued from previous page)

```
25     en02 criterion of objective Env 20
26     en03 criterion of objective Env 20
27     en12 criterion of objective Env 20
28   Total weight: 80.00 (4 criteria)
```

In Listing 2.3 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 16-22). 4 *equisignificant* criteria do the same from an **economical** (Lines 10-15), respectively an **environmental** point of view (Lines 23-28). The *equiobjectives* directive results hence in a balanced total weight (80.00) for each decision objective.

```
>>> t.showActions()
 key:  p01
   name:       random public policy Eco+ Soc- Env+
   profile:    {'Eco': 'good', 'Soc': 'weak', 'Env': 'good'}
 key:  p02
 ...
 key:  p26
   name:       random public policy Eco+ Soc+ Env-
   profile:    {'Eco': 'good', 'Soc': 'good', 'Env': 'weak'}
 ...
 key:  p30
   name:       random public policy Eco- Soc- Env-
   profile:    {'Eco': 'weak', 'Soc': 'weak', 'Env': 'weak'}
 ...
```

Variable triangular modes (0.3, 0.5 or 0.7 of the span of the measure scale) for each objective result in different performance status for each 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 `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.

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

>>> tsoc = PartialPerformanceTableau(t,criteriaSubset=\
...                              t.objectives['Soc']['criteria'])

>>> tenv = PartialPerformanceTableau(t,criteriaSubset=\
...                              t.objectives['Env']['criteria'])
```

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

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

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

```
>>> from digraphs import FusionLDigraph
>>> gfus = FusionLDigraph([geco,gsoc,genv])
>>> gfus.strongComponents()
 {frozenset({'p30'}),
  frozenset({'p10', 'p03', 'p19', 'p08', 'p07', 'p04', 'p21', 'p20',
             'p13', 'p23', 'p16', 'p12', 'p24', 'p02', 'p31', 'p29',
             'p05', 'p09', 'p28', 'p25', 'p17', 'p14', 'p15', 'p06',
             'p01', 'p27', 'p11', 'p18', 'p22'}),
  frozenset({'p26'})}
>>> from digraphs import StrongComponentsCollapsedDigraph
>>> scc = StrongComponentsCollapsedDigraph(gfus)
>>> scc.showActions()
 *----- show digraphs actions --------------*
 key:  frozenset({'p30'})
   short name: Scc_1
   name:       _p30_
   comment:    collapsed strong component
 key:  frozenset({'p10', 'p03', 'p19', 'p08', 'p07', 'p04', 'p21', 'p20',
↪ 'p13',
                   'p23', 'p16', 'p12', 'p24', 'p02', 'p31', 'p29', 'p05',
↪ 'p09', 'p28', 'p25',
                   'p17', 'p14', 'p15', 'p06', 'p01', 'p27', 'p11', 'p18',
↪ 'p22'})
   short name: Scc_2
   name:       _p10_p03_p19_p08_p07_p04_p21_p20_p13_p23_p16_p12_p24_p02_
↪p31_\
                p29_p05_p09_p28_p25_p17_p14_p15_p06_p01_p27_p11_p18_p22_
   comment:    collapsed strong component
 key:  frozenset({'p26'})
   short name: Scc_3
   name:       _p26_
   comment:    collapsed strong component
```

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

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

(continues on next page)

(continued from previous page)

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

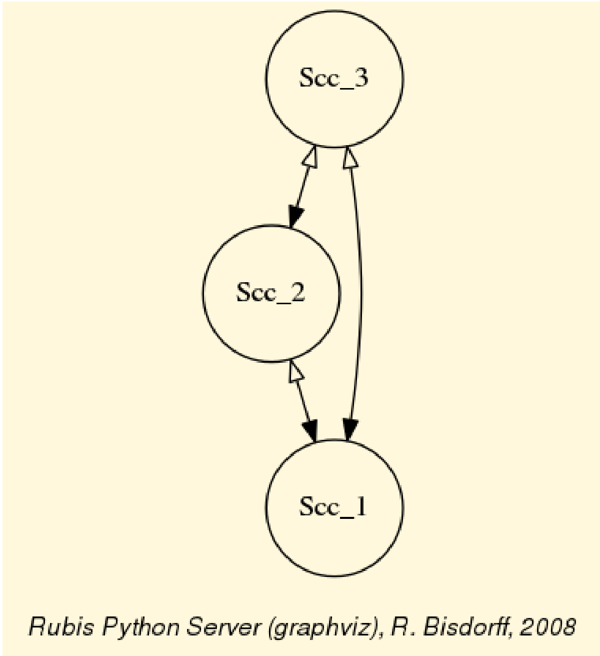


Fig. 2.3: Strong components digraph

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.

Random academic performance tableaux

The `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)
- NA := <Decimal> (default = -999); missing data symbol.

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 2.4: 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 |   'm1'  'm2'  'm3'  'm4'  'm5'  'm6'  'm7'
    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
```

(continues on next page)

(continued from previous page)

```
32 >>> t.weightPreorder
33  [['m2', 'm5', 'm6'], ['m1'], ['m3'], ['m4'], ['m7']]
```

The example tableau, generated for instance above with *missingDataProbability* = 0.03, *WithTypes* = True and *seed* = 100 (see Listing 2.4 Lines 2-5), 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 *m5* and one in course *m3* and course *m6* (see Lines 21-31).

We may show a statistical summary of the students' grades obtained in the heighest weighted course, namely *m7*, followed by a performance heatmap browser view showing a global ranking of the students' performances from best to weakest.

Listing 2.5: Student performance summary statistics per course

```
>>> t.showCourseStatistics('m7')
 *----- 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. 2.4: Ranking the students with a performance heatmap view

The ranking shown here in Fig. 2.4 is produced with the default *NetFlows ranking rule* (page 78). With a mean marginal correlation of +0.361 (see Listing 2.6 Lines 17-) associated with a low standard deviation (0.248), the result represents a rather *fair weighted consensus* made between the individual courses' marginal rankings.

Listing 2.6: Consensus quality of the students's ranking

```
1  >>> from outrankingDigraphs import BipolarOutrankingDigraph
2  >>> g = BipolarOutrankingDigraph(t)
3  >>> t.showRankingConsensusQuality(g.computeNetFlowsRanking())
4   Consensus quality of ranking:
5    ['s07', 's02', 's09', 's05', 's06', 's03', 's10',
6     's01', 's08', 's04', 's11']
7   criterion (weight): correlation
8   -------------------------------
9    m7 (0.294): +0.727
10   m4 (0.235): +0.309
11   m2 (0.059): +0.291
12   m3 (0.176): +0.200
13   m1 (0.118): +0.109
14   m6 (0.059): +0.091
15   m5 (0.059): +0.073
16  Summary:
17   Weighted mean marginal correlation (a): +0.361
```

(continues on next page)

(continued from previous page)

```
Standard deviation (b)              : +0.248
Ranking fairness (a)-(b)            : +0.113
```

Random linearly ranked performance tableaux

Finally, we provide the `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 `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 1)

2.2 How to create a new performance tableau instance

In this tutorial we illustrate a way of creating a new `PerformanceTableau` instance by editing a template with 5 decision alternatives, 3 decision objectives and 6 performance criteria.

Editing a template file

For this purpose we provide the following perfTab_Template.py file in the *examples* directory of the **Digraph3** resources.

Listing 2.7: PerformanceTableau object template

```
##########################################################
# Digraph3 documentation
# Template for creating a new PerformanceTableau instance
# (C) R. Bisdorff Mar 2021
# Digraph3/examples/perfTab_Template.py
###########################################################
from decimal import Decimal
from collections import OrderedDict
#####
# edit the decision actions
# avoid special characters, like '_', '/' or ':',
# in action identifiers and short names
actions = OrderedDict([
 ('a1', {
  'shortName': 'action1',
  'name': 'decision alternative a1',
  'comment': 'some specific features of this alternative',
   }),
  ...
  ...
])
#####
# edit the decision objectives
# adjust the list of performance criteria
# and the total weight (sum of the criteria weights)
# per objective
objectives = OrderedDict([
 ('obj1', {
  'name': 'decision objective obj1',
  'comment': "some specific features of this objective",
  'criteria': ['g1', 'g2'],
  'weight': Decimal('6'),
  }),
  ...
  ...
 ])
```

(continues on next page)

(continued from previous page)

```
#####
# edit the performance criteria
# adjust the objective reference
# Left Decimal of a threshold = constant part and
#  right Decimal = proportional part of the threshold
criteria = OrderedDict([
 ('g1', {
  'shortName': 'crit1',
  'name': "performance criteria 1",
  'objective': 'obj1',
  'preferenceDirection': 'max',
  'comment': 'measurement scale type and unit',
  'scale': (Decimal('0.0'), Decimal('100.0'),
  'thresholds': {'ind':  (Decimal('2.50'), Decimal('0.0')),
                 'pref': (Decimal('5.00'), Decimal('0.0')),
                 'veto': (Decimal('60.00'), Decimal('0.0'))
                },
  'weight': Decimal('3'),
  }),
  ...
  ...
 ])
#####
# default missing data symbol = -999
NA = Decimal('-999')
#####
# edit the performance evaluations
# criteria to be minimized take negative grades
evaluation = {
 'g1': {
    'a1':Decimal("41.0"),
    'a2':Decimal("100.0"),
    'a3':Decimal("63.0"),
    'a4':Decimal('23.0'),
    'a5': NA,
   },
 # g2 is of ordinal type and scale 0-10
 'g2': {
    'a1':Decimal("4"),
    'a2':Decimal("10"),
    'a3':Decimal("6"),
    'a4':Decimal('2'),
    'a5':Decimal('9'),
   },
 # g3 has preferenceDirection = 'min'
 'g3': {
```

(continues on next page)

(continued from previous page)

```
83         'a1':Decimal("-52.2"),
84         'a2':NA,
85         'a3':Decimal("-47.3"),
86         'a4':Decimal('-35.7'),
87         'a5':Decimal('-68.00'),
88       },
89     ...
90     ...
91   }
92   ####################
```

The template file, shown in Listing 2.7, contains first the instructions to import the required *Decimal* and *OrderedDict* classes (see Lines 7-8). Four main sections are following: the potential decision **actions**, the decision **objectives**, the performance **criteria**, and finally the performance **evaluation**.

Editing the decision alternatives

Decision alternatives are stored in attribute **actions** under the *OrderedDict* format (see the OrderedDict (https://docs.python.org/3/library/collections.html) description in the Python documentation).

Required attributes of each decision alternative, besides the object **identifier**, are: **shortName**, **name** and **comment** (see Lines 15-17). The *shortName* attribute is essentially used when showing the performance tableau or the performance heatmap in a browser view.

Note: Mind that graphviz drawings require digraph actions' (nodes) identifier strings without any special characters like _ or /.

Decision actions descriptions are stored in the order of which they appear in the stored instance file. The *OrderedDict* object keeps this given order when iterating over the decision alternatives.

The random performance tableau models presented in the previous tutorial use the *actions* attribute for storing special features of the decision alternatives. The *Cost-Benefit* model, for instance, uses a **type** attribute for distinguishing between *advantageous*, *neutral* and *cheap* alternatives. The *3-Objectives* model keeps a detailed record of the performance profile per decision objective and the corresponding random generators per performance criteria (see Lines 7- below).

```
1 >>> t = Random3ObjectivesPerformanceTableau()
2 >>> t.actions
3  OrderedDict([
4   ('p01', {'shortName': 'p01',
5            'name': 'action p01 Eco~ Soc- Env+',
```

(continues on next page)

(continued from previous page)

```
            'comment': 'random public policy',
            'Eco': 'fair',
            'Soc': 'weak',
            'Env': 'good',
            'profile': {'Eco':'fair',
                        'Soc':'weak',
                        'Env':'good'}
            'generators': {'ec01': ('triangular', 50.0, 0.5),
                           'so02': ('triangular', 30.0, 0.5),
                           'en03': ('triangular', 70.0, 0.5),
                           ...
                           },
            }
        ),
    ...
    ])
```

The second section of the template file concerns the decision *objectives*.

Editing the decision objectives

The minimal required attributes (see Listing 2.7 Lines 27-33) of the ordered decision **objectives** dictionary, besides the individual objective identifiers, are **name**, **comment**, **criteria** (the list of significant performance criteria) and **weight** (the importance of the decision objective). The latter attribute contains the sum of the *significance* weights of the objective's criteria list.

The **objectives** attribute is methodologically useful for specifying the performance criteria significance in building decision recommendations. Mostly, we assume indeed that decision objectives are all equally important and the performance criteria are equi-significant per objective. This is, for instance, the default setting in the random *3-Objectives* performance tableau model.

Listing 2.8: Example of decision objectives' description

```
>>> t = Random3ObjectivesPerformanceTableau()
>>> t.objectives
 OrderedDict([
  ('Eco',
   {'name': 'Economical aspect',
    'comment': 'Random3ObjectivesPerformanceTableau generated',
    'criteria': ['ec01', 'ec06', 'ec09'],
    'weight': Decimal('48')}),
  ('Soc',
   {'name': 'Societal aspect',
    'comment': 'Random3ObjectivesPerformanceTableau generated',
```

(continues on next page)

(continued from previous page)

```
12     'criteria': ['so02', 'so12'],
13     'weight': Decimal('48')}),
14   ('Env',
15    {'name': 'Environmental aspect',
16     'comment': 'Random3ObjectivesPerformanceTableau generated',
17     'criteria': ['en03', 'en04', 'en05', 'en07',
18                  'en08', 'en10', 'en11', 'en13'],
19     'weight': Decimal('48')})
20  ])
```

The importance weight sums up to 48 for each one of the three example decision objectives shown in Listing 2.8 (Lines 8,13 and 19), so that the significance of each one of the 3 economic criteria is set to 16, of both societal criteria is set to 24, and of each one of the 8 environmental criteria is set to 8.

Note: Mind that the **objectives** attribute is always present in a *PerformanceTableau* object instance, even when empty. In this case, we consider that each performance criterion canonically represents in fact its own decision objective. The criterion significance equals in this case the corresponding decision objective's importance weight.

The third section of the template file concerns now the *performance criteria*.

Editing the family of performance criteria

In order to assess how well each potential decision alternative is satisfying a given decision objective, we need *performance criteria*, i.e. decimal-valued grading functions gathered in an ordered **criteria** dictionary. The required attributes (see Listing 2.9), besides the criteria identifiers, are the usual **shortName**, **name** and **comment**. Specific for a criterion are furthermore the **objective** reference, the significance **weight**, the grading **scale** (minimum and maximum performance values), the **preferenceDirection** ('max' or 'min') and the performance discrimination **thresholds**.

Listing 2.9: Example of performance criteria description

```
criteria = OrderedDict([
 ('g1', {
  'shortName': 'crit1',
  'name': "performance criteria 1",
  'comment': 'measurement scale type and unit',
  'objective': 'obj1',
  'weight': Decimal('3'),
  'scale': (Decimal('0.0'), Decimal('100.0'),
  'preferenceDirection': 'max',
  'thresholds': {'ind':  (Decimal('2.50'), Decimal('0.0')),
                 'pref': (Decimal('5.00'), Decimal('0.0')),
```

(continues on next page)

(continued from previous page)

```
12                   'veto': (Decimal('60.00'), Decimal('0.0'))
13                  },
14  }),
15  ...
16  ...])
```

In our bipolar-valued outranking approach, all performance criteria implement *decimal-valued* grading functions, where preferences are either *increasing* or *decreasing* with measured performances.

Note: In order to model a **coherent** performance tableau, the decision criteria must satisfy two methodological requirements:

1. **Independance**: Each decision criterion implements a grading that is *functionally independent* of the grading of the other decision criteria, i.e. the performance measured on one of the criteria does not *constrain* the performance measured on any other criterion.
2. **Non redundancy**: Each performance criterion is only *significant* for a *single* decision objective.

In order to take into account any, usually *unavoidable*, **imprecision** of the performance grading procedures, we may specify three performance **discrimination thresholds**: an **indifference** ('ind'), a **preference** ('pref') and a **considerable performance difference** ('veto') threshold (see Listing 2.9 Lines 10-12). The left decimal number of a threshold description tuple indicates a *constant part*, whereas the right decimal number indicates a proportional part.

On the template performance criterion *g1*, shown in Listing 2.9, we observe for instance a grading scale from 0.0 to 100.0 with a constant *indifference* threshold of 2.5, a constant *preference* threshold of 5.0, and a constant *considerable performance difference* threshold of 60.0. The latter theshold will trigger, the case given, a *polarisation* of the outranking statement [BIS-2013] .

In a random *Cost-Benefit* performance tableau model we may obtain by default the following content.

Listing 2.10: Example of cardinal Costs criterion

```
>>> tcb = RandomCBPerformanceTableau()
>>> tcb.showObjectives()
 *------ decision objectives -------"
  C: Costs
   c1 random cardinal cost criterion 6
  Total weight: 6.00 (1 criteria)
 ...
 ...
```

(continues on next page)

(continued from previous page)

```
>>> tcb.criteria
 OrderedDict([
  ('c1', {'preferenceDirection': 'min',
          'scaleType': 'cardinal',
          'objective': 'C',
          'shortName': 'c1',
          'name': 'random cardinal cost criterion',
          'scale': (0.0, 100.0),
          'weight': Decimal('6'),
          'randomMode': ['triangular', 50.0, 0.5],
          'comment': 'Evaluation generator: triangular law ...',
          'thresholds':
           OrderedDict([
            ('ind', (Decimal('1.49'), Decimal('0'))),
            ('pref', (Decimal('3.7'), Decimal('0'))),
            ('veto', (Decimal('67.71'), Decimal('0')))
            ])
          }
   ...
   ...
   ])
```

Criterion *c1* appears here (see Listing 2.10) to be a cardinal criterion to be minimized and significant for the *Costs* (C) decision objective. We may use the `showCriteria()` method for printing the corresponding performance discrimination thresholds.

```
1 >>> tcb.showCriteria(IntegerWeights=True)
2  *----  criteria -----*
3   c1 'Costs/random cardinal cost criterion'
4    Scale = (0.0, 100.0)
5    Weight = 6
6    Threshold ind : 1.49 + 0.00x ; percentile: 5.13
7    Threshold pref : 3.70 + 0.00x ; percentile: 10.26
8    Threshold veto : 67.71 + 0.00x ; percentile: 96.15
```

The *indifference* threshold on this criterion amounts to a constant value of 1.49 (Line 6 above). More or less 5% of the observed performance differences on this criterion appear hence to be **insignificant**. Similarly, with a preference threshold of 3.70, about 90% of the observed performance differences are preferentially **significant** (Line 7). Furthermore, 100.0 - 96.15 = 3.85% of the observed performance differences appear to be **considerable** (Line 8) and will trigger a *polarisation* of the corresponding outranking statements.

After the performance criteria description, we are ready for recording the actual *performance table*.

Editing the performance table

The individual grades of each decision alternative on each decision criterion are recorded in a double *criterion* x *action* dictionary called **evaluation** (see Listing 2.11). As we may encounter missing data cases, we previously define a *missing data* symbol **NA** which is set here to a value disjoint from all the measurement scales, by default *Decimal('-999')* (Line 2).

Listing 2.11: Editing performance grades

```
1  #----------
2  NA = Decimal('-999')
3  #----------
4  evaluation = {
5   'g1': {
6      'a1':Decimal("41.0"),
7      'a2':Decimal("100.0"),
8      'a3':Decimal("63.0"),
9      'a4':Decimal('23.0'),
10     'a5': NA,  # missing data
11    },
12   ...
13   ...
14  # g3 has preferenceDirection = 'min'
15  'g3': {
16     'a1':Decimal("-52.2"), # negative grades
17     'a2':NA,
18     'a3':Decimal("-47.3"),
19     'a4':Decimal('-35.7'),
20     'a5':Decimal('-68.00'),
21    },
22  ...
23  ...
24  }
```

Notice in Listing 2.11 (Lines 16-) that on a criterion with *preferenceDirection* = '**min**' all performance grades are recorded as **negative** values.

We may now inspect the eventually recorded complete template performance table.

```
>>> from perfTabs import PerformanceTableau
>>> t = PerformanceTableau('perfTab_Template')
>>> t.showPerformanceTableau(ndigits=1)
 *----  performance tableau -----*
  Criteria  |  'g1'   'g2'  'g3'  'g4'   'g5'   'g6'
  Actions   |    3      3     6     2      2      2
   ---------|-----------------------------------------
  'action1' |  41.0   4.0  -52.2  71.0   63.0   22.5
  'action2' | 100.0  10.0    NA   89.0   30.7   75.0
```

(continues on next page)

(continued from previous page)

```
  'action3' |  63.0   6.0  -47.3  55.4   63.5    NA
  'action4' |  23.0   2.0  -35.7  83.5   37.5   54.9
  'action5' |   NA    9.0  -68.0  10.0   88.0   75.0
```

We may furthermore compute the associated outranking digraph and check if we observe any polarised outranking situtations.

```
>>> from outrankingDigraphs import BipolarOutrankingDigraph
>>> g = BipolarOutrankingDigraph(t)
>>> g.showVetos()
 *----  Veto situations ---
  number of veto situations : 1
  1: r(a4 >= a2) = -0.44
  criterion: g1
  Considerable performance difference : -77.00
  Veto discrimination threshold       : -60.00
  Polarisation: r(a4 >= a2) = -0.44 ==> -1.00
 *----  Counter-veto situations ---
  number of counter-veto situations : 1
  1: r(a2 >= a4) = 0.56
  criterion: g1
  Considerable performance difference : 77.00
  Counter-veto threshold              : 60.00
  Polarisation: r(a2 >= a4) = 0.56 ==> +1.00
```

Indeed, due to the considerable performance difference (77.00) oberved on performance criterion *g1*, alternative *a2* **for sure** *outranks* alternative *a4*, respectively *a4* **for sure** *does not outrank a2*.

Inspecting the template outranking relation

Let us have a look at the outranking relation table.

Listing 2.12: The template outranking relation

```
1 >>> g.showRelationTable()
2  * ---- Relation Table -----
3     r   |  'a1'   'a2'   'a3'   'a4'   'a5'
4    -----|-----------------------------------
5    'a1' | +1.00  -0.44  -0.22  -0.11  +0.06
6    'a2' | +0.44  +1.00  +0.33  +1.00  +0.28
7    'a3' | +0.67  -0.33  +1.00  +0.00  +0.17
8    'a4' | +0.11  -1.00  +0.00  +1.00  +0.06
9    'a5' | -0.06  -0.06  -0.17  -0.06  +1.00
```

We may notice in the outranking relation table above (see Listing 2.12) that decision alternative *a2* positively **outranks** all the other four alternatives (Line 6). Similarly, alternative *a5* is positively **outranked** by all the other alternatives (see Line 9). We may orient this way the *graphviz* drawing of the template outranking digraph.

```
>>> g.exportGraphViz(fileName= 'template',
...                  firstChoice =['a2'],
...                  lastChoice=['a5'])
 *---- exporting a dot file for GraphViz tools ---------*
  Exporting to template.dot
  dot -Grankdir=BT -Tpng template.dot -o template.png
```

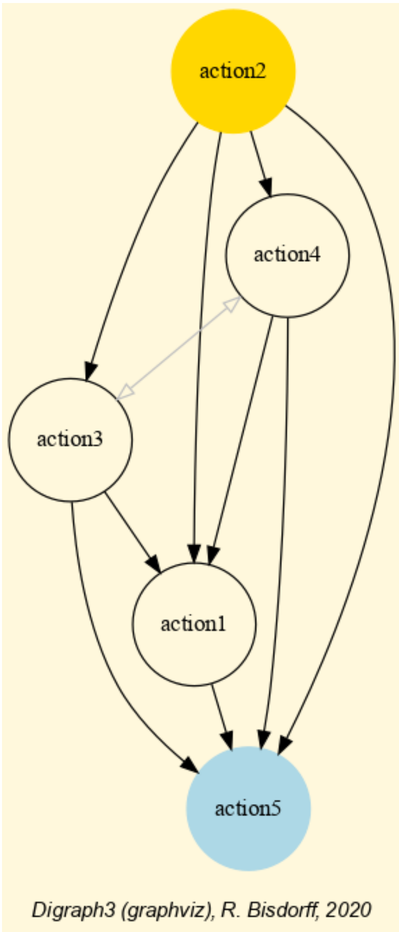


Fig. 2.5: The template outranking digraph

In Fig. 2.5 we may notice that the template outranking digraph models in fact a **partial order** on the five potential decision alternatives. Alternatives *action3* ('a3') and *action4* ('a4') appear actually **incomparable**. In Listing 2.12 their pairwise outranking chracteritics show indeed the **indeterminate** value 0.00 (Lines 7-8). We may check their pairwise comparison as follows.

```
>>> g.showPairwiseComparison('a3','a4')
 *------------  pairwise comparison ----*
 Comparing actions : (a3, a4)
  crit. wght.  g(x)   g(y)   diff    | ind   pref   r()   |
```
(continues on next page)

(continued from previous page)

```
------------------------------     ------------------
g1   3.00  63.00   23.00  +40.00   | 2.50  5.00  +3.00 |
g2   3.00   6.00    2.00   +4.00   | 0.00  1.00  +3.00 |
g3   6.00 -47.30  -35.70  -11.60   | 0.00 10.00  -6.00 |
g4   2.00  55.40   83.50  -28.10   | 2.09  4.18  -2.00 |
g5   2.00  63.50   37.50  +26.00   | 0.00 10.00  +2.00 |
g6   NA    54.90
Outranking characteristic value:   r(a3 >= a4) = +0.00
Valuation in range: -18.00 to +18.00
```

The incomparability situation between 'a3' and 'a4' results here from a perfect balancing of positive (+8) and negative (-8) criteria significance weights.

Ranking the template peformance tableau

We may eventually rank the five decision alternatives with a heatmap browser view following the *Copeland* ranking rule which consistently reproduces the partial outranking order shown in Fig. 2.5.

```
>>> g.showHTMLPerformanceHeatmap(ndigits=1,colorLevels=5,
...     Correlations=True,rankingRule='Copeland',
...     pageTitle='Heatmap of the template performance tableau')
```

Heatmap of the template performance tableau

criteria	crit4	crit1	crit3	crit2	crit6	crit5
weights	+2.00	+3.00	+6.00	+3.00	+2.00	+2.00
tau(*)	+0.60	+0.40	+0.35	+0.20	+0.10	-0.60
action2	89.0	100.0	NA	10.0	75.0	30.7
action3	55.4	63.0	-47.3	6.0	NA	63.5
action4	83.5	23.0	-35.7	2.0	54.9	37.5
action1	71.0	41.0	-52.2	4.0	22.5	63.0
action5	10.0	NA	-68.0	9.0	75.0	88.0

Color legend:

quantile	20.00%	40.00%	60.00%	80.00%	100.00%

(*) tau: *Ordinal (Kendall) correlation between marginal criterion and global ranking relation*
Outranking model: **standard**, *Ranking rule*: **Copeland**
Ordinal (Kendall) correlation between global ranking and global outranking relation: **+1.000**
Mean marginal correlation (a) : **+0.228**
Standard marginal correlation deviation (b) : **+0.322**
Ranking fairness (a) - (b) : **-0.094**

Due to a 11 against 7 **plurality tyranny** effect, the *Copeland* ranking rule, essentially based on crisp majority outranking counts, puts here alternative *action5* (*a5*) last, despite its excellent grades observed on criteria *g2*, *g5* and *g6*. A slightly **fairer** ranking result may be obtained with the *NetFlows* ranking rule.

```
>>> g.showHTMLPerformanceHeatmap(ndigits=1,colorLevels=5,
...     Correlations=True,rankingRule='NetFlows',
...     pageTitle='Heatmap of the template performance tableau')
```

Heatmap of the template performance tableau

criteria	crit2	crit6	crit1	crit4	crit3	crit5
weights	+3.00	+2.00	+3.00	+2.00	+6.00	+2.00
tau(*)	+0.60	+0.50	+0.40	+0.20	-0.05	-0.20
action2	10.0	75.0	100.0	89.0	NA	30.7
action3	6.0	NA	63.0	55.4	-47.3	63.5
action5	9.0	75.0	NA	10.0	-68.0	88.0
action4	2.0	54.9	23.0	83.5	-35.7	37.5
action1	4.0	22.5	41.0	71.0	-52.2	63.0

Color legend:

quantile	20.00%	40.00%	60.00%	80.00%	100.00%

(*) tau: *Ordinal (Kendall) correlation between marginal criterion and global ranking relation*
Outranking model: **standard**, *Ranking rule:* **NetFlows**
Ordinal (Kendall) correlation between global ranking and global outranking relation: **+0.920**
Mean marginal correlation (a) : **+0.206**
Standard marginal correlation deviation (b) : **+0.286**
Ranking fairness (a) - (b) : **-0.081**

It might be opportun to furthermore study the robustness of the apparent outranking situations when assuming only *ordinal* or *uncertain* criteria significance weights. If interested in mainly objectively *unopposed* (multipartisan) outranking situations, one might also try the `UnOpposedOutrankingDigraph` constructor. (see the advanced topics of the Digraph3 documentation).

Back to *Content Table* (page 1)

2.3 Computing the winner of an election with the `votingProfiles` module

Linear voting profiles

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

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

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

Listing 2.13: Example of random linear voting profile

```
1  >>> from votingProfiles import RandomLinearVotingProfile
2  >>> v = RandomLinearVotingProfile(numberOfVoters=5,
3  ...                                 numberOfCandidates=3,
4  ...                                 RandomWeights=True)
5
6  >>> v.candidates
7   OrderedDict([ ('a1',{'name':'a1}), ('a2',{'name':'a2'}),
8                 ('a3',{'name':'a3'}) ])
9  >>> v.voters
10  OrderedDict([('v1',{'weight': 2}), ('v2':{'weight': 3}),
11   ('v3',{'weight': 1}), ('v4':{'weight': 5}),
12   ('v5',{'weight': 4})])
13 >>> v.linearBallot
14  {'v1': ['a1', 'a2', 'a3',],
15   'v2': ['a3', 'a2', 'a1',],
16   'v3': ['a1', 'a3', 'a2',],
17   'v4': ['a1', 'a3', 'a2',],
18   'v5': ['a2', 'a3', 'a1',]}
```

Notice that in this random example, the five voters are weighted (see Listing 2.13 Lines 10-12). Their linear ballots can be viewed with the `showLinearBallots()` method.

```
>>> v.showLinearBallots()
```

(continues on next page)

(continued from previous page)

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

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

```
>>> v.save(fileName='tutorialLinearVotingProfile1')
 *--- Saving linear profile in file: <tutorialLinearVotingProfile1.py> --
↪-*
>>> from votingProfiles import LinearVotingProfile
>>> v = LinearVotingProfile('tutorialLinearVotingProfile1')
```

Computing the winner

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

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

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

Listing 2.14: Example Instant Run Off Winner

```
>>> v.computeInstantRunoffWinner(Comments=True)
 Half of the Votes =  7.50
 ==> stage =  1
     remaining candidates ['a1', 'a2', 'a3']
     uninominal votes {'a1': 6, 'a2': 2, 'a3': 7}
     minimal number of votes =  2
     maximal number of votes =  7
     candidate to remove =  a2
     remaining candidates =  ['a1', 'a3']
 ==> stage =  2
     remaining candidates ['a1', 'a3']
     uninominal votes {'a1': 8, 'a3': 7}
     minimal number of votes =  7
     maximal number of votes =  8
```

(continues on next page)

(continued from previous page)

```
    candidate a1 obtains an absolute majority
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 2.14).

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 2.15: Example of *Borda* rank scores

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

Candidate *a1* obtains the minimal *Borda* score, followed by candidate *a3* and finally candidate *a2* (see Listing 2.15). The corresponding *Borda* **rank analysis table** may be printed out with a corresponding `show()` command.

Listing 2.16: Rank analysis example

```
>>> v.showRankAnalysisTable()
 *----  Borda rank analysis tableau -----*
 candi- | alternative-to-rank |     Borda
 dates  |   1     2     3     | score  average
 -------|-------------------------------------
  'a1'  |   6     9     0     | 24/15   1.60
  'a3'  |   7     1     7     | 30/15   2.00
  '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 2.14 and Listing 2.16). 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.

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 **majority margins digraph**. The `MajorityMarginsDigraph` class (a specialization of the `Digraph` class) is available for handling such kind of digraphs.

Listing 2.17: Example of *Majority Margins* digraph

```
>>> from votingProfiles import MajorityMarginsDigraph
>>> cdg = MajorityMarginsDigraph(v,IntegerValuation=True)
>>> cdg
 *------- Digraph instance description ------*
 Instance class      : MajorityMarginsDigraph
 Instance name       : rel_randomLinearVotingProfile1
 Digraph Order       : 3
 Digraph Size        : 3
 Valuation domain    : [-15.00;15.00]
 Determinateness (%) : 64.44
 Attributes          : ['name', 'actions', 'voters',
                        'ballot', 'valuationdomain',
                        'relation', 'order',
                        'gamma', 'notGamma']
>>> cdg.showAll()
 *----- show detail -------------*
 Digraph          : rel_randLinearVotingProfile1
 *---- Actions ----*
 ['a1', 'a2', 'a3']
 *---- Characteristic valuation domain ----*
 {'max': Decimal('15.0'), 'med': Decimal('0'),
```

(continues on next page)

(continued from previous page)

```
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]
```

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 2.17 Lines 26-28). This is not true in general for arbitrary voting profiles. The *majority margins* 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 *majority margins* 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 2.17 Line 26).

```
>>> cdg.computeCondorcetWinners()
 ['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 2)), and especially its `exportGraphViz()` method[Page 7, 1], for visualizing an election result.

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

[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.

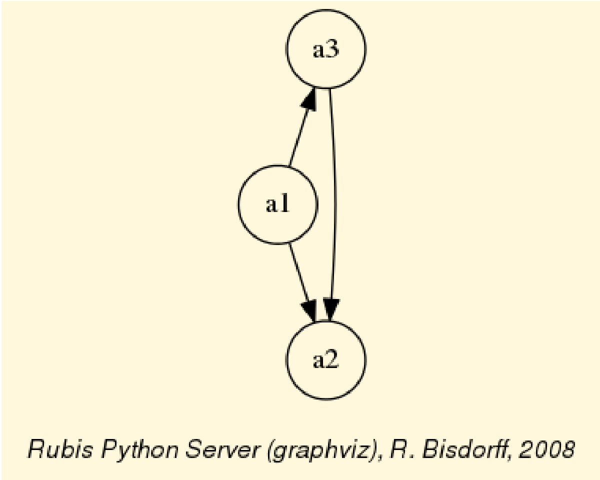


Fig. 2.6: Visualizing an election result

In Fig. 2.6 we notice that the *majority margins* digraph from our example linear voting profile gives a linear order of the candidates: [‘a1’, ‘a3’, ‘a2], the same actually as given by the *Borda* scores (see Listing 2.15). This is by far not given in general. Usually, when aggregating linear ballots, there appear cyclic social preferences.

Cyclic social preferences

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

Listing 2.18: Example of cyclic social preferences

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

(continues on next page)

(continued from previous page)

```
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 2.18 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 *majority margins* digraph, we may observe *cyclic* preferences, like ($a1$ > $a2$ > $a3$ > $a1$) for instance (see Fig. 2.7).

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

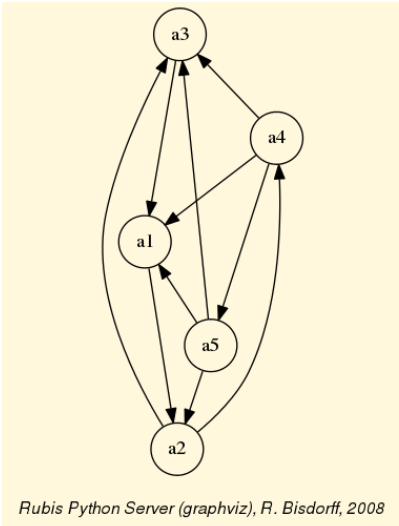


Fig. 2.7: Cyclic social preferences

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

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

Condorcet ‘s approach for determining the winner of an election is hence *not decisive* in all circumstances and we need to exploit more sophisticated approaches for finding the winner of the election on the basis of the majority margins of the given linear ballots (see the tutorial on *ranking with multiple incommensurable criteria* (page 72) 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).

Listing 2.19: Example linear voting heatmap

```
1 :linenos:
2 
3 >>> v.showHTMLVotingHeatmap(rankingRule='NetFlows',
4 ...                          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. 2.8: 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. 2.8) 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 72)). 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 2.19 Line 2)

On generating realistic random linear voting profiles

By default, the `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 `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 2.20: Generating a linear voting profile with random polls

```
>>> from votingProfiles import RandomLinearVotingProfile
>>> lvp = RandomLinearVotingProfile(numberOfCandidates=15,
...                                 numberOfVoters=1000,
...                                 WithPolls=True,
...                                 partyRepartition=0.5,
...                                 other=0.1,
...                                 seed=0.9189670954954139)

>>> lvp
 *------- VotingProfile instance description ------*
 Instance class    : RandomLinearVotingProfile
 Instance name     : randLinearProfile
 # Candidates      : 15
 # Voters          : 1000
 Attributes        : ['name', 'seed', 'candidates',
                      'voters', 'RandomWeights',
                      'sumWeights', 'poll1', 'poll2',
                      'bipartisan', 'linearBallot', 'ballot']
>>> lvp.showRandomPolls()
 Random repartition of voters
  Party_1 supporters : 460 (46.0%)
  Party_2 supporters : 436 (43.6%)
  Other voters       : 104 (10.4%)
 *---------------- random polls ---------------
  Party_1(46.0%) | Party_2(43.6%)|  expected
 -----------------------------------------------
    a06 : 19.91%  | a11 : 22.94%  | a06 : 15.00%
    a07 : 14.27%  | a08 : 15.65%  | a11 : 13.08%
    a03 : 10.02%  | a04 : 15.07%  | a08 : 09.01%
    a13 : 08.39%  | a06 : 13.40%  | a07 : 08.79%
    a15 : 08.39%  | a03 : 06.49%  | a03 : 07.44%
    a11 : 06.70%  | a09 : 05.63%  | a04 : 07.11%
    a01 : 06.17%  | a07 : 05.10%  | a01 : 05.06%
    a12 : 04.81%  | a01 : 05.09%  | a13 : 05.04%
    a08 : 04.75%  | a12 : 03.43%  | a15 : 04.23%
    a10 : 04.66%  | a13 : 02.71%  | a12 : 03.71%
    a14 : 04.42%  | a14 : 02.70%  | a14 : 03.21%
    a05 : 04.01%  | a15 : 00.86%  | a09 : 03.10%
    a09 : 01.40%  | a10 : 00.44%  | a10 : 02.34%
```

(continues on next page)

```
40    a04 : 01.18%  | a05 : 00.29%  | a05 : 01.97%
41    a02 : 00.90%  | a02 : 00.21%  | a02 : 00.51%
```

In this example (see Listing 2.20 Lines 19-), 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 2.21).

Listing 2.21: The uninominal election winner

```
>>> lvp.computeSimpleMajorityWinner()
 ['a06']
>>> lvp.computeInstantRunoffWinner()
 ['a06']
>>> lvp.computeBordaWinners()
 ['a06']
```

Is it also a *Condorcet* winner ? To verify, we start by creating the corresponding *majority margins* digraph *cdg* with the help of the `MajorityMarginsDigraph` class. The created digraph instance contains 15 *actions* -the candidates- and 105 *oriented* arcs -the *positive* majority margins- (see Listing 2.22 Lines 7-8).

Listing 2.22: A majority margins digraph constructed from a linear voting profile

```
1  >>> from votingProfiles import MajorityMarginsDigraph
2  >>> cdg = MajorityMarginsDigraph(lvp)
3  >>> cdg
4   *------- Digraph instance description ------*
5   Instance class       : MajorityMarginsDigraph
6   Instance name        : rel_randLinearProfile
7   Digraph Order        : 15
8   Digraph Size         : 104
9   Valuation domain     : [-1000.00;1000.00]
10  Determinateness (%)  : 67.08
11  Attributes           : ['name', 'actions', 'voters',
12                          'ballot', 'valuationdomain',
13                          'relation', 'order',
14                          '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. 2.9: Browsing the majority margins

In Fig. 2.9, *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* **loser**. We recover again candidate *a06* as *Condorcet* winner ([15]), whereas the obvious *Condorcet* loser is here candidate *a02*, the candidate with the lowest support in both parties (see Listing 2.20 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 2.23: 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']
```

(continues on next page)

[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 Kernel-Tutorial-label).

(continued from previous page)

```
12              6th first ranked ['a13']
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 2.23 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 2.23 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 2.21 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 2.20 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 *majority margins* digraph, operates the selection procedure by extracting at each step *initial* and *terminal* kernels, i.e. NP-hard operational problems (see tutorial on computing kernels 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* (page 72) 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 1)

---

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

2.4 Ranking with multiple incommensurable criteria

The ranking problem

We need to rank without ties a set X of items (usually decision alternatives) that are evaluated on multiple incommensurable performance criteria; yet, for which we may know their pairwise bipolar-valued *strict outranking* characteristics, i.e. $r(x \gnsim y)$ 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 2.24: 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
```

(continues on next page)

(continued from previous page)

```
 '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 2.25: Median cut polarised strict outranking relation characteristics

```
>>> ccd = PolarisedOutrankingDigraph(gcd,
...                      level=g.valuationdomain['med'],
...                      KeepValues=False,StrictCut=True)

>>> ccd.showRelationTable(ReflexiveTerms=False,IntegerValues=True)
 *---- Relation Table -----
  r(>)_med | 'a1' 'a2' 'a3' 'a4' 'a5' 'a6' 'a7' 'a8' 'a9'
  ---------|-------------------------------------------
     'a1'  |   -    0   +1   -1   -1   -1   -1   +1    0
     'a2'  |  -1    -   +0    0   -1   -1   -1   -1   -1
     'a3'  |  -1    0    -   -1   -1   -1   -1   -1    0
     'a4'  |   0    0   -1    -   -1   -1   -1   -1   -1
     'a5'  |  +1   +1   +1   +1    -   +1   +1   -1   -1
     'a6'  |  -1   -1   +1   +1   -1    -   +1   -1   -1
     'a7'  |  -1   +1   -1   -1   -1   -1    -   +1   +1
     'a8'  |  -1   -1   -1   -1   -1   +1   -1    -   +1
     '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 *linear voting profiles* (page 59)).

To estimate how *difficult* this ranking problem here may be, we may have a look at the corresponding strict outranking digraph *graphviz* drawing ([Page 7, 1]).

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

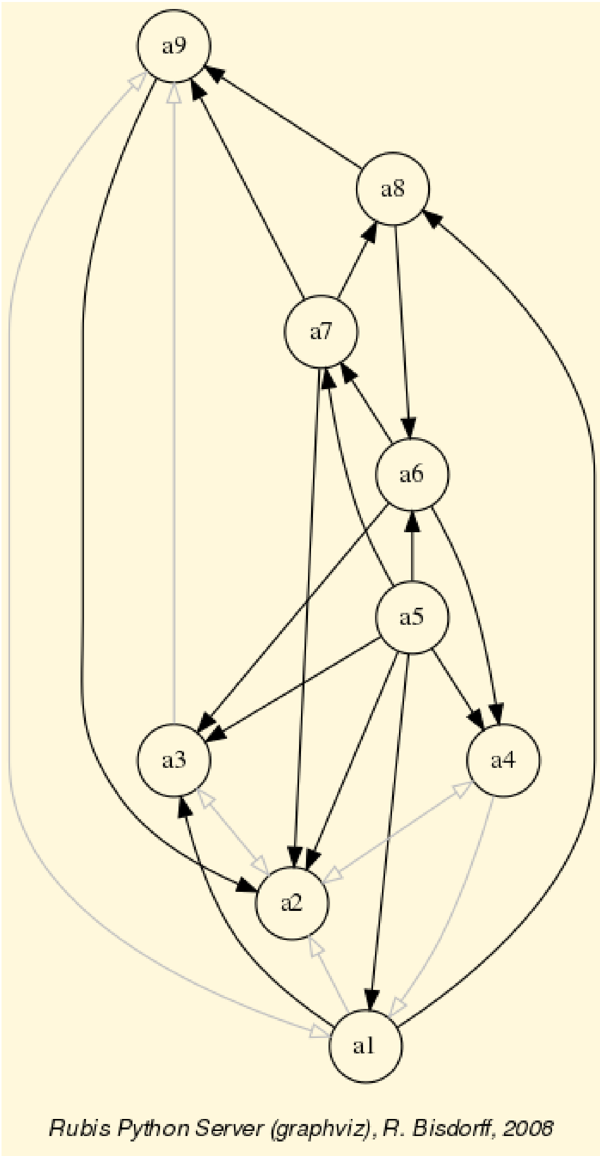


Fig. 2.10: The *strict outranking* digraph

The strict outranking relation $\gnsim$ 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. 2.10). 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.

```
>>> gcd.computeChordlessCircuits()
>>> gcd.showChordlessCircuits()
 1 circuit(s).
 *---- Chordless circuits ----*
 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. 2.10). 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.

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 $r(x \gnsim y)_{>0}$ and the crisp **strict outranked** characteristics $r(y \gnsim x)_{>0}$ 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 2.26: 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 2.26 Lines 4-12).

Copeland's ranking rule appears in fact **invariant** under the *codual transform* (page 18) and renders a same linear order indifferently from digraphs *g* or *gcd* . The resulting ranking (see Listing 2.26 Line 14) is rather correlated (+0.463) with the given pairwise outranking relation in the ordinal *Kendall* sense (see Listing 2.27).

Listing 2.27: 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 `WeakCopelandOrder` class.

Listing 2.28: 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 2.28 above, the ranking with ties delivered by the *Copeland* scores (see Listing 2.26). We may draw its corresponding *Hasse* diagram (see Listing 2.29).

Listing 2.29: 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; }
```

(continues on next page)

(continued from previous page)

```
2 { rank = same; a8; }
3 { rank = same; a4; a9}
4 { rank = same; a3; }
5 { rank = same; a2; }
dot -Grankdir=TB -Tpng weakCopelandRanking.dot\
    -o weakCopelandRanking.png
```

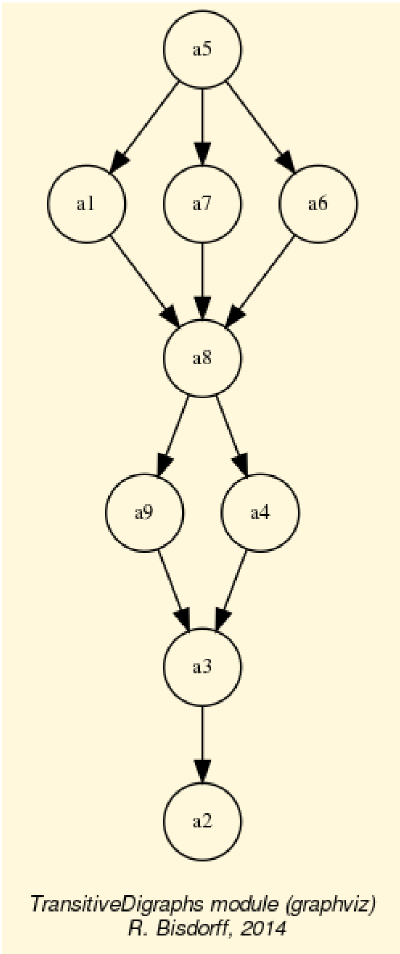


Fig. 2.11: A weak Copeland ranking

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

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 $r(x \succnsim y)$ and the **strict outranked** characteristics $r(y \succnsim x)$ for all pairs of alternatives where y is different from x.

Listing 2.30: Computing a *NetFlows* ranking

```
1  :linenos:
2
3  >>> from linearOrders import NetFlowsRanking
4  >>> nf = NetFlowsRanking(gcd,Comments=True)
5   Net Flows :
6   a5 : 3.600
7   a7 : 2.800
8   a6 : 1.300
9   a3 : 0.033
10  a1 : -0.400
11  a8 : -0.567
12  a4 : -1.283
13  a9 : -2.600
14  a2 : -2.883
15  NetFlows Ranking:
16  ['a5', 'a7', 'a6', 'a3', 'a1', 'a8', 'a4', 'a9', 'a2']
17 >>> cop.copelandRanking
18  ['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 2.30 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 2.30 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 `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 2.27 Lines 4-6).

Listing 2.31: Checking the quality of the *NetFlows* Ranking

```
>>> corr = gcd.computeOrdinalCorrelation(nf)
>>> gcd.showCorrelation(corr)
```

(continues on next page)

(continued from previous page)

```
Correlation indexes:
 Extended Kendall tau       : +0.638
 Epistemic determination    :  0.230
 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. 2.11).

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

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 2.32: 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 2.32 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 2.33: Optimal *Kemeny* rankings

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

(continues on next page)

(continued from previous page)

```
 ['a5', 'a6', 'a7', 'a3', 'a9', 'a4', 'a1', 'a8', 'a2']]
>>> ke.maxKemenyIndex
Decimal('12.9166667')
```

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

Listing 2.34: Computing the epistemic disjunction of all optimal *Kemeny* rankings

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

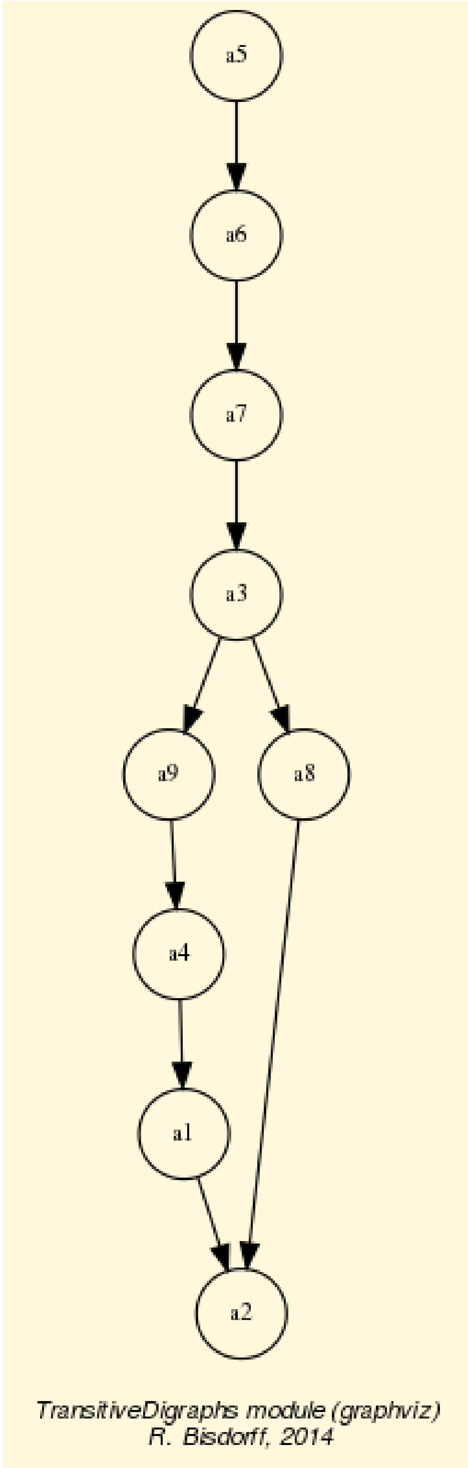


Fig. 2.12: Epistemic disjunction of optimal *Kemeny* rankings

It is interesting to notice in Fig. 2.12 and Listing 2.33, 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 `showRankingConsensusQuality()` method, the *most consensual* one.

Listing 2.35: Computing Consensus Quality of Rankings

```
>>> g.showRankingConsensusQuality(ke.maximalRankings[0])
  Consensus quality of ranking:
   ['a5', 'a6', 'a7', 'a3', 'a8', 'a9', 'a4', 'a1', 'a2']
  criterion (weight): correlation
  -------------------------------
   b09 (0.050): +0.361
   b04 (0.050): +0.333
   b08 (0.050): +0.292
   b01 (0.050): +0.264
   c01 (0.167): +0.250
   b03 (0.050): +0.222
   b07 (0.050): +0.194
   b05 (0.050): +0.167
   c02 (0.167): +0.000
   b10 (0.050): +0.000
   b02 (0.050): -0.042
   b06 (0.050): -0.097
   c03 (0.167): -0.167
  Summary:
   Weighted mean marginal correlation (a): +0.099
   Standard deviation (b)                : +0.177
   Ranking fairness (a)-(b)              : -0.079
>>> g.showRankingConsensusQuality(ke.maximalRankings[1])
  Consensus quality of ranking:
   ['a5', 'a6', 'a7', 'a3', 'a9', 'a4', 'a1', 'a8', 'a2']
  criterion (weight): correlation
  -------------------------------
   b09 (0.050): +0.306
   b08 (0.050): +0.236
   c01 (0.167): +0.194
   b07 (0.050): +0.194
   c02 (0.167): +0.167
   b04 (0.050): +0.167
   b03 (0.050): +0.167
   b01 (0.050): +0.153
   b05 (0.050): +0.056
   b02 (0.050): +0.014
   b06 (0.050): -0.042
   c03 (0.167): -0.111
   b10 (0.050): -0.111
  Summary:
   Weighted mean marginal correlation (a): +0.099
   Standard deviation (b)                : +0.132
   Ranking fairness (a)-(b)              : -0.033
```

Both Kemeny rankings show the same *weighted mean marginal correlation* (+0.099, see Listing 2.35 Lines 19-22, 42-44) with all thirteen performance criteria. However, the second ranking shows a slightly lower *standard deviation* (+0.132 vs +0.177), resulting in a slightly **fairer** ranking result (-0.033 vs -0.079).

When several rankings with maximal Kemeny index are given, the `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 2.32 Line 4).

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 2.36: 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 2.36 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 2.36 Line 11). The corresponding *epistemic disjunction* (page 17) gives the following partial ordering.

Listing 2.37: Computing the epistemic disjunction of optimal *Slater* rankings

```
>>> slw = RankingsFusion(sl,sl.maximalRankings)
>>> slw.exportGraphViz(fileName='tutorialSlater')
 *---- exporting a dot file for GraphViz tools ---------*
 Exporting to tutorialSlater.dot
 0 { rank = same; a5; }
 1 { rank = same; a6; }
 2 { rank = same; a7; a4; }
 3 { rank = same; a1; }
 4 { rank = same; a8; a3; }
```

(continues on next page)

(continued from previous page)

```
5 { rank = same; a9; }
6 { rank = same; a2; }
dot -Grankdir=TB -Tpng tutorialSlater.dot -o tutorialSlater.png
```

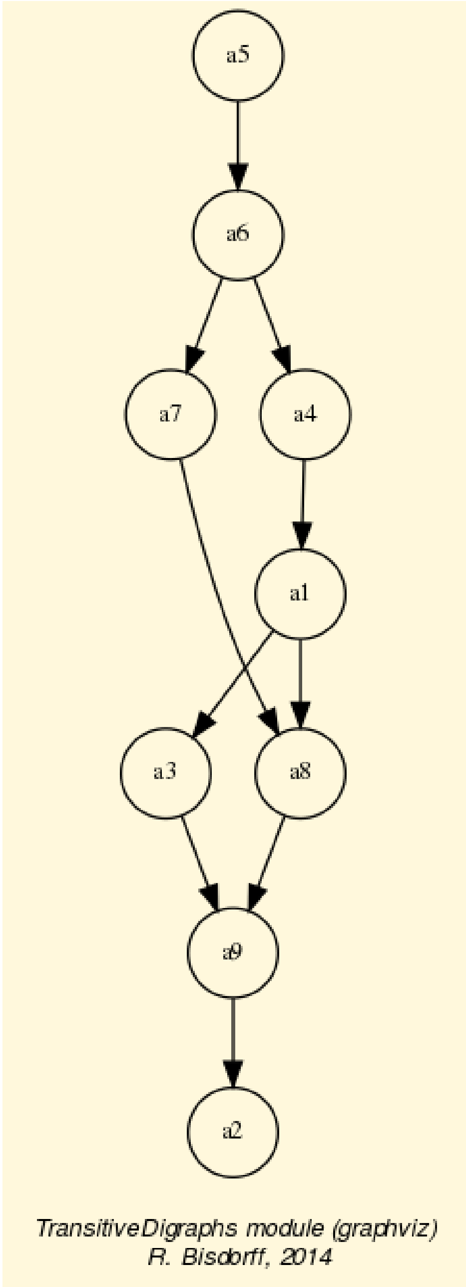


Fig. 2.13: Epistemic disjunction of optimal *Slater* rankings

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.

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 2.24) 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 2.38: Computing a *Kohler* ranking

```
>>> from linearOrders import KohlerRanking
>>> kocd = KohlerRanking(gcd)
>>> kocd.showRanking()
 ['a5', 'a7', 'a6', 'a3', 'a9', 'a8', 'a4', 'a1', 'a2']
>>> corr = gcd.computeOrdinalCorrelation(kocd)
>>> gcd.showCorrelation(corr)
 Correlation indexes:
  Extended Kendall tau       : +0.747
  Epistemic determination    :  0.230
  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 2.33). 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.

```
>>> ko = KohlerRanking(g)
>>> corr = g.computeOrdinalCorrelation(ko)
>>> g.showCorrelation(corr)
 Correlation indexes:
  Crisp ordinal correlation  : +0.483
  Epistemic determination    :  0.230
  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 2.24) the corresponding *net flow score* (page 78) ;
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 `IteratedNetFlowsRanking` class constructor (see Listing 2.39 Lines 12-13).

Listing 2.39: Ordering-by-choosing with iterated minimal *NetFlows* scores

```
1  >>> from linearOrders import IteratedNetFlowsRanking
2  >>> inf = IteratedNetFlowsRanking(g)
3  >>> inf
4   *------- Digraph instance description ------*
5    Instance class       : IteratedNetFlowsRanking
6    Instance name        : rel_randomCBperftab_ranked
7    Digraph Order        : 9
8    Digraph Size         : 36
9    Valuation domain     : [-1.00;1.00]
10   Determinateness (%)  : 100.00
11   Attributes           : ['valuedRanks', 'valuedOrdering',
12                           'iteratedNetFlowsRanking',
13                           'iteratedNetFlowsOrdering',
14                           'name', 'actions', 'order',
15                           'valuationdomain', 'relation',
16                           'gamma', 'notGamma']
17 >>> inf.iteratedNetFlowsOrdering
18  ['a2', 'a9', 'a1', 'a4', 'a3', 'a8', 'a7', 'a6', 'a5']
19 >>> corr = g.computeOrderCorrelation(inf.iteratedNetFlowsOrdering)
20 >>> g.showCorrelation(corr)
21  Correlation indexes:
22   Crisp ordinal correlation  : +0.751
23   Epistemic determination    :  0.230
24   Bipolar-valued equivalence : +0.173
25 >>> inf.iteratedNetFlowsRanking
```

(continues on next page)

(continued from previous page)

```
26  ['a5', 'a7', 'a6', 'a3', 'a4', 'a1', 'a8', 'a9', 'a2']
27 >>> corr = g.computeRankingCorrelation(inf.iteratedNetFlowsRanking)
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 2.39 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 2.38 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 2.34 and Listing 2.37), 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.

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 2.40: Computing a *RankedPairs* ranking

```
>>> from linearOrders import RankedPairsRanking
>>> rp = RankedPairsRanking(g)
>>> rp.showRanking()
 ['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.

```
>>> ke.maximalRankings
 [['a5', 'a6', 'a7', 'a3', 'a8', 'a9', 'a4', 'a1', 'a2'],
  ['a5', 'a6', 'a7', 'a3', 'a9', 'a4', 'a1', 'a8', 'a2']]
>>> corr = g.computeOrdinalCorrelation(rp)
>>> g.showCorrelation(corr)
 Correlation indexes:
  Extended Kendall tau       : +0.779
  Epistemic determination    :  0.230
  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 1)

2.5 Computing a first choice recommendation

See also:

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

What site to choose ?

A SME, specialized in printing and copy services, has to move into new offices, and its CEO has gathered seven **potential office sites** (see Table 2.1).

Table 2.1: The potential new office sites

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].

Table 2.2: The coherent family of performance criteria

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

[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.

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

Table 2.3: 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 2.3 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 2.3 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 2.3, what is now the office site we may recommend to the CEO as **best choice** ?

The 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.

```
>>> from perfTabs import *
>>> t = PerformanceTableau('officeChoice')
>>> t
 *------- PerformanceTableau instance description ------*
  Instance class      : PerformanceTableau
  Instance name       : officeChoice
  # Actions           : 7
  # Objectives        : 3
  # Criteria          : 7
  NaN proportion (%)  : 0.0
  Attributes          : ['name', 'actions', 'objectives',
                         'criteria', 'weightPreorder',
                         'NA', 'evaluation']
>>> t.showPerformanceTableau()
 *----  performance tableau -----*
   Criteria |  'C'         'Cf'    'P'    'Pr'     'St'     'V'     'W'
   Weights  |  45.00       6.00    3.00  32.00    23.00   26.00   10.00
   ---------|-----------------------------------------------------------
    'Ave'   | -35000.00   0.00   90.00  100.00  100.00   60.00   75.00
    'Bon'   | -17800.00 100.00   30.00   20.00   10.00   80.00   30.00
    'Ces'   |  -6700.00  10.00  100.00   80.00    0.00   70.00    0.00
    'Dom'   | -14100.00  30.00   90.00   70.00   30.00   50.00   55.00
    'Bel'   | -34800.00  60.00   70.00   40.00   90.00   60.00  100.00
    'Fen'   | -18600.00  80.00    0.00    0.00   70.00    0.00    0.00
    'Gar'   | -12000.00  50.00   80.00   60.00   20.00  100.00   50.00
```

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

```
>>> t.showCriteria(IntegerWeights=True)
 *----  criteria -----*
 C 'Costs'
 Scale = (Decimal('0.00'), Decimal('50000.00'))
 Weight = 45
 Threshold ind : 1000.00 + 0.00x ;  percentile:  9.5
 Threshold pref : 2500.00 + 0.00x ; percentile: 14.3
 Cf 'Comfort'
 Scale = (Decimal('0.00'), Decimal('100.00'))
 Weight = 6
 Threshold ind : 10.00 + 0.00x ;  percentile:   9.5
```

(continues on next page)

(continued from previous page)

```
12 Threshold pref : 20.00 + 0.00x ; percentile:  28.6
13 Threshold veto : 80.00 + 0.00x ; percentile:  90.5
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. 2.14 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. 2.14: 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.

The 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-2004a]).

For computing such a bipolar-valued outranking digraph from the given performance tableau t, we use the `BipolarOutrankingDigraph` constructor from the outrankingDigraphs module. The `showHTMLRelationTable` method shows here the resulting bipolar-valued adjacency matrix in a system browser window (see Fig. 2.15).

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

Valued Adjacency Matrix

r(x S y)	A	B	C	D	E	F	G
A	-	0.00	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. 2.15: The office choice outranking digraph

In Fig. 2.15 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.computeCondorcetWinners()
 ['D']
>>> g.computeWeakCondorcetWinners()
 ['A', 'C', 'D']
```

We may get even more insight in the apparent outranking situations when looking at the Condorcet digraph (see Fig. 2.16).

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

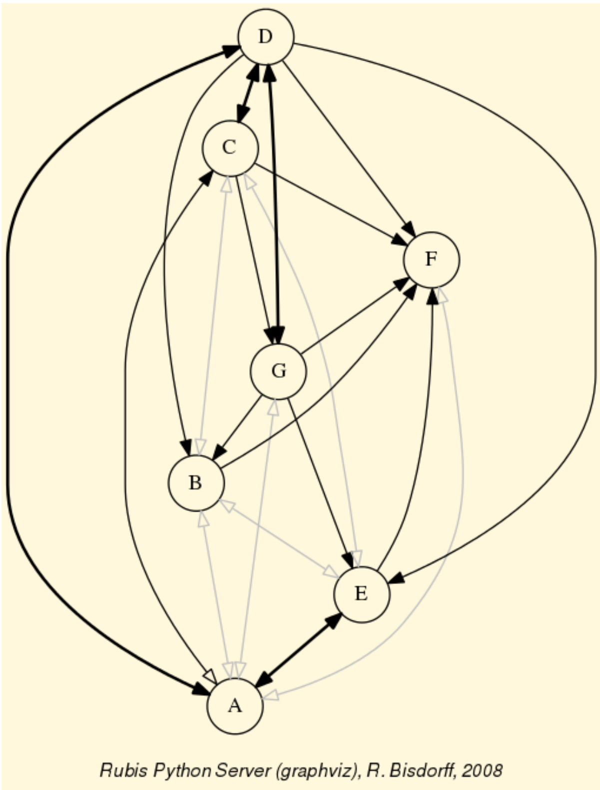


Fig. 2.16: The office choice outranking digraph

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

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

The *Rubis* best choice recommendation

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

```
>>> from digraphs import BrokenCocsDigraph
>>> bcg = BrokenCocsDigraph(g)
>>> bcg.brokenLinks
 set()
```

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

Listing 2.41: Computing outranking and outranked prekernels

```
>>> g.showPreKernels()
 *--- Computing preKernels ---*
 Dominant preKernels :
 ['D']
    independence :  1.0
    dominance    :  0.02
    absorbency   :  -1.0
    covering     :  1.000
 ['B', 'E', 'C']
    independence :  0.00
    dominance    :  0.10
    absorbency   :  -1.0
    covering     :  0.500
 ['A', 'G']
    independence :  0.00
    dominance    :  0.78
    absorbency   :  0.00
    covering     :  0.700
 Absorbent preKernels :
 ['F', 'A']
    independence :  0.00
    dominance    :  0.00
    absorbency   :  1.0
    covering     :  0.700
 *----- statistics -----
 graph name:  rel_officeChoice.xml
 number of solutions
  dominant kernels :  3
  absorbent kernels:  1
```

(continues on next page)

(continued from previous page)

```
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 2.41 three potential first 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()` method. Notice that this method actually works by default on the broken chords digraph *bcg*.

Listing 2.42: Computing a best choice recommendation

```
>>> g.showBestChoiceRecommendation(CoDual=False)
*****************************************
Rubis best choice recommendation(s) (BCR)
 (in decreasing order of determinateness)
Credibility domain: [-1.00,1.00]
=== >> potential first choice(s)
* choice              : ['D']
  independence        : 1.00
  dominance           : 0.02
  absorbency          : -1.00
  covering (%)        : 100.00
  determinateness (%) : 51.03
  - most credible action(s) = { 'D': 0.02, }
=== >> potential first choice(s)
* choice              : ['A', 'G']
  independence        : 0.00
  dominance           : 0.78
  absorbency          : 0.00
  covering (%)        : 70.00
  determinateness (%) : 50.00
  - most credible action(s) = { }
=== >> potential first choice(s)
* choice              : ['B', 'C', 'E']
  independence        : 0.00
  dominance           : 0.10
  absorbency          : -1.00
  covering (%)        : 50.00
  determinateness (%) : 50.00
  - most credible action(s) = { }
=== >> potential last choice(s)
```

(continues on next page)

(continued from previous page)

```
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 2.42 (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 first choice recommendations, as well as the potential last 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 first and a last choice recommendation (see Lines 15 and 31) and seams hence not actually comparable to its competitors.

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.

```
>>> g.showPairwiseComparison('G','D')
 *------------  pairwise comparison ----*
 Comparing actions : (G, D)
 crit. wght.  g(x)      g(y)    diff.  |  ind     pref    concord    |
=========================================================================
 C   45.00 -12000.00 -14100.00 +2100.00 | 1000.00 2500.00   +45.00    |
 Cf   6.00     50.00     30.00   +20.00 |   10.00   20.00    +6.00    |
 P    3.00     80.00     90.00   -10.00 |   10.00   20.00    +3.00    |
 Pr  32.00     60.00     70.00   -10.00 |   10.00   20.00   +32.00    |
 St  23.00     20.00     30.00   -10.00 |   10.00   20.00   +23.00    |
 V   26.00    100.00     50.00   +50.00 |   10.00   20.00   +26.00    |
 W   10.00     50.00     55.00    -5.00 |   10.00   20.00   +10.00    |
=========================================================================
 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.

```
>>> g.showPairwiseComparison('C','G')
 *------------  pairwise comparison ----*
 Comparing actions : (C, G)/(G, C)
 crit. wght.   g(x)     g(y)      diff.  |   ind.   pref.    (C,G)/(G,C) |
 ==========================================================================
 C    45.00 -6700.00 -12000.00  +5300.00 | 1000.00 2500.00  +45.00/-45.00 |
 Cf    6.00   10.00     50.00    -40.00 |   10.00   20.00   -6.00/ +6.00 |
 P     3.00  100.00     80.00    +20.00 |   10.00   20.00   +3.00/ -3.00 |
 Pr   32.00   80.00     60.00    +20.00 |   10.00   20.00  +32.00/-32.00 |
 St   23.00    0.00     20.00    -20.00 |   10.00   20.00  -23.00/+23.00 |
 V    26.00   70.00    100.00    -30.00 |   10.00   20.00  -26.00/+26.00 |
 W    10.00    0.00     50.00    -50.00 |   10.00   20.00  -10.00/+10.00 |
 ==========================================================================
 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 (-)[Page 18, 14], of the outranking digraph instance g (see [BIS-2013]), as follows.

Listing 2.43: Computing the strict best choice recommendation

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

 * --- First and last choice recommendation(s) ---*
  (in decreasing order of determinateness)
 Credibility domain: [-1.00,1.00]
 === >> potential first choice(s)
 * choice              : ['A', 'C', 'D']
   independence        : 0.00
   dominance           : 0.10
   absorbency          : 0.00
   covering (%)        : 41.67
   determinateness (%) : 50.59
```

(continues on next page)

(continued from previous page)

```
15    - characteristic vector = { 'D': 0.02, 'G': 0.00, 'C': 0.00,
16                                'A': 0.00, 'F': -0.02, 'E': -0.02,
17                                'B': -0.02, }
18  === >> potential last choice(s)
19  * choice                : ['A', 'F']
20    independence          : 0.00
21    dominance             : -0.52
22    absorbency            : 1.00
23    covered (%)           : 50.00
24    determinateness (%)   : 50.00
25    - characteristic vector = { 'G': 0.00, 'F': 0.00, 'E': 0.00,
26                                'D': 0.00, 'C': 0.00, 'B': 0.00,
27                                'A': 0.00, }
```

It is interesting to notice in Listing 2.43 (Line 9) 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 15-17), 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 first 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 first choice candidate.

We may also notice (see Line 17 and Line 21) that both alternatives A and F are reported as certainly strict outranked choices, hence as **potential last choice recommendation** . This confirms again the global incomparability status of alternative A (see Fig. 2.17).

```
>>> gcd = ~(-g) # codual of g
>>> gcd.exportGraphViz(fileName='bestChoiceChoice',
...                    fistChoice=['A','C','D'],
...                    lastChoice=['F'])
 *---- exporting a dot file for GraphViz tools ---------*
  Exporting to bestOfficeChoice.dot
  dot -Grankdir=BT -Tpng bestOfficeChoice.dot -o bestOfficeChoice.png
```

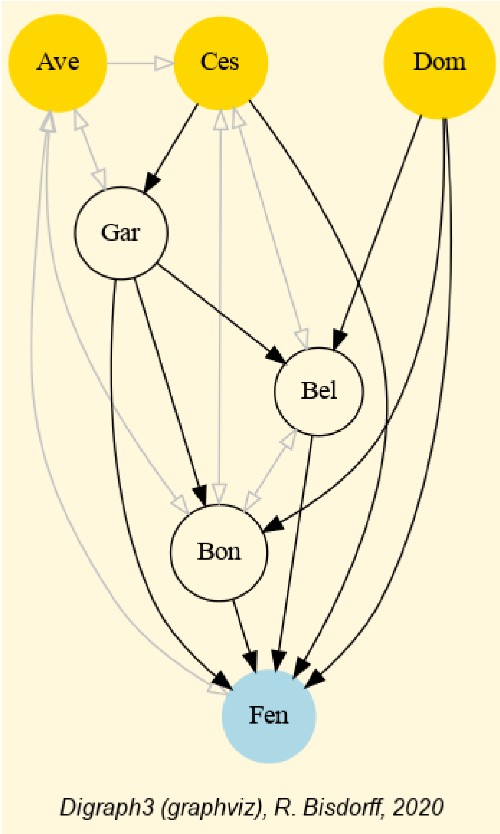


Fig. 2.17: Best office choice recommendation from strict outranking digraph

Weakly ordering the outranking digraph

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

```
>>> from transitiveDigraphs import RankingByChoosingDigraph
>>> rbc = RankingByChoosingDigraph(gcd)
 Threading ...  ## multiprocessing if 2 cores are available
 Exiting computing threads
>>> rbc.showRankingByChoosing()
 Ranking by Choosing and Rejecting
 1st ranked ['D']
    2nd ranked ['C', 'G']
    2nd last ranked ['B', 'C', 'E']
 1st last ranked ['A', 'F']
>>> rbc.exportGraphViz('officeChoiceRanking')
 *---- exporting a dot file for GraphViz tools ---------*
 Exporting to officeChoiceRanking.dot
 0 { rank = same; A; C; D; }
 1 { rank = same; G; }
```

(continues on next page)

(continued from previous page)

```
2 { rank = same; E; B; }
3 { rank = same; F; }
dot -Grankdir=TB -Tpng officeChoiceRanking.dot -o officeChoiceRanking.
↪png
```

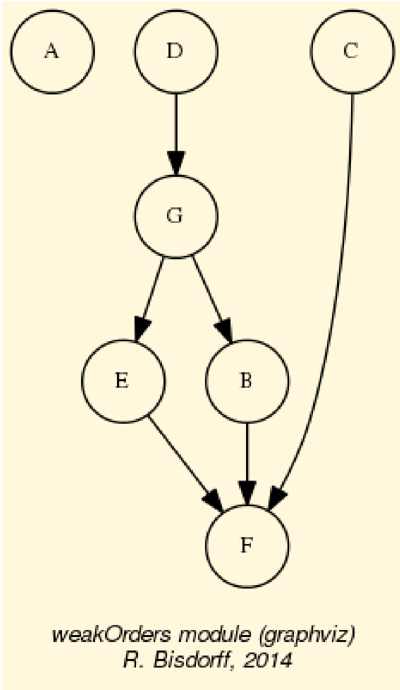


Fig. 2.18: Ranking-by-choosing from the office choice outranking digraph

In this **ranking-by-choosing** method, where we operate the *epistemic fusion* of iterated (strict) first and last choices, compromise alternative D is now ranked before compromise alternative G. If the computing node supports multiple processor cores, first and last 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 of the ranking-by-choosing relation (see Fig. 2.18).

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:

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

Back to *Content Table* (page 1)

2.6 Rating into relative performance quantiles

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 $O(nmq)$.

Performance quantile 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 $\geqslant$ ($\ngeqslant$), it will be sufficient to check that $x \geqslant m_k$ as well as $x \ngeqslant M_k$ 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 $m_k \ngeqslant x$ 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: $[Q_0; Q_1[$, $[Q_1; Q_2[$, ..., $[Q_{q-1}; \infty[$, 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 $]-\infty; Q_1]$, $]Q_1; Q_2]$, ..., $]Q_{q-1}; Q_q]$. 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 $K_4 = \{a_1, a_4, a_8, a_{20}\}$.

Rating-by-sorting into relative multicriteria performance quantiles

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, ..., 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-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.

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

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 `ratingDigraphs.RatingByRelativeQuantilesDigraph` class.

Listing 2.44: Computing a quintiles rating result

```
>>> from randomPerfTabs import *
>>> t = RandomPerformanceTableau(numberOfActions=50,seed=5)
>>> from ratingDigraphs import RatingByRelativeQuantilesDigraph
>>> rqr = RatingByRelativeQuantilesDigraph(t,quantiles=5)
>>> rqr
    *-----  Object instance description -----------*
     Instance class       : RatingByRelativeQuantilesDigraph
     Instance name        : relative_rating_randomperftab
     Actions              : 55
     Criteria             : 7
     Quantiles            : 5
     Lowerclosed          : False
     Rankingrule          : NetFlows
     Size                 : 1647
     Valuation domain     : [-1.00;1.00]
     Determinateness (%): 67.40
     Attributes           : ['name', 'actions', 'actionsOrig',
       'criteria', 'evaluation', 'NA', 'runTimes',
       'quantilesFrequencies', 'LowerClosed', 'categories',
       'criteriaCategoryLimits', 'limitingQuantiles', 'profiles',
       'profileLimits', 'order', 'nbrThreads', 'relation',
       'valuationdomain', 'sorting', 'relativeCategoryContent',
       'sortingRelation', 'rankingRule', 'rankingScores',
       'rankingCorrelation', 'actionsRanking', 'ratingCategories']
    *------  Constructor run times (in sec.) ------*
     Threads              : 1
     Total time           : 0.19248
     Data input           : 0.00710
     Compute quantiles    : 0.00117
     Compute outrankings : 0.17415
     rating-by-sorting    : 0.00074
     rating-by-ranking    : 0.00932
>>> rqr.showSorting()
 *--- Sorting results in descending order ---*
  ]0.80 - 1.00]: ['a22']
  ]0.60 - 0.80]: ['a03', 'a07', 'a08', 'a11', 'a14', 'a17',
                  'a19', 'a20', 'a29', 'a32', 'a33', 'a37',
```

(continues on next page)

(continued from previous page)

```
38                    'a39', 'a41', 'a42', 'a49']
39   ]0.40 - 0.60]: ['a01', 'a02', 'a04', 'a05', 'a06', 'a08',
40                    'a09', 'a16', 'a17', 'a18', 'a19', 'a21',
41                    'a24', 'a27', 'a28', 'a30', 'a31', 'a35',
42                    'a36', 'a40', 'a43', 'a46', 'a47', 'a48',
43                    'a49', 'a50']
44   ]0.20 - 0.40]: ['a04', 'a10', 'a12', 'a13', 'a15', 'a23',
45                    'a25', 'a26', 'a34', 'a38', 'a43', 'a44',
46                    'a45', 'a49']
47   ]   < - 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 2.44 Lines XX-).

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

Listing 2.45: Bipolar-valued sorting characteristics (extract)

```
1  >>> rqr.valuationdomain
2   {'min': Decimal('-1.0'), 'med': Decimal('0'),
3    'max': Decimal('1.0')}
4  >>> rqr.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]       1.00          -0.86         -0.86
7   a22 in ]0.20 - 0.40]    0.86          -0.71         -0.71
8   a22 in ]0.40 - 0.60]    0.71          -0.71         -0.71
9   a22 in ]0.60 - 0.80]    0.71          -0.14         -0.14
10  a22 in ]0.80 - 1.00]    0.14           1.00          0.14
11  ...
12  ...
13  a44 in ]< - 0.20]       1.00           0.00          0.00
14  a44 in ]0.20 - 0.40]    0.00           0.57          0.00
15  a44 in ]0.40 - 0.60]   -0.57           0.86         -0.57
16  a44 in ]0.60 - 0.80]   -0.86           0.86         -0.86
17  a44 in ]0.80 - 1.00]   -0.86           0.86         -0.86
18  ...
19  ...
20  a49 in ]< - 0.20]       1.00          -0.43         -0.43
21  a49 in ]0.20 - 0.40]    0.43           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           0.57          0.00
24  a49 in ]0.80 - 1.00]   -0.57           0.86         -0.57
```

Alternative *a22* verifies indeed positively both sorting conditions only for the highest quintile $[0.80 - 1.00]$ class (see Listing 2.45 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 2.46: Weakly ranking the quintiles sorting result

```
1  >>> rqr.showRatingByQuantilesSorting(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 2.46, 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.

Rating-by-ranking with relative quantile limits

The *actions* attribute of a `RatingByRelativeQuantilesDigraph` class instance contains, besides the *decision actions* gathered from the given performance tableau (stored in the *actionsOrig* attribute, also the quantile limits observed on all the criteria (stored in the *limitingquantiles* attribute, see Listing 2.44 Line 20).

Listing 2.47: The quintiling of the performance evaluation data per criterion

```
>>> rqr.showCriteriaQuantileLimits()
  Quantile Class Limits (q = 5)
  Upper-closed classes
  crit.       0.20    0.40    0.60    0.80    1.00
  *------------------------------------------------
   g1         31.35   41.09   58.53   71.91   98.08
   g2         27.81   39.19   49.87   61.66   96.18
   g3         25.10   34.78   49.45   63.97   92.59
   g4         24.61   37.91   53.91   71.02   89.84
   g5         26.94   36.43   52.16   72.52   96.25
   g6         23.94   44.06   54.92   67.34   95.97
   g7         30.94   47.40   55.46   69.04   97.10
```

We may hence rank this extended actions attribute as follows with the *NetFlows* ranking rule –default with the `RatingByRelativeQuantilesDigraph` class.

Listing 2.48: Rating by ranking the quintiling of the performance tableau

```
>>> rqr.computeNetFlowsRanking()
 ['5-M', '4-M', 'a22', 'a42', 'a07', 'a33', 'a03', 'a01',
  'a39', 'a48', 'a37', 'a29', 'a41', 'a11', 'a27', 'a05',
  'a46', 'a02', 'a17', 'a32', '3-M', 'a14', 'a12', 'a20',
  'a13', 'a08', 'a06', 'a24', 'a47', 'a31', 'a09', 'a21',
  'a19', 'a43', 'a49', 'a50', 'a40', 'a28', 'a38', 'a25',
  'a45', 'a18', 'a16', 'a36', 'a35', 'a30', 'a23', 'a34',
  'a15', '2-M', 'a10', 'a26', 'a04', 'a44', '1-M']
  >>> rqr.showRatingByQuantilesRanking()
  *-------- rating by quantiles ranking result ---------
  ]0.60 - 0.80] ['a22', 'a42', 'a07', 'a33', 'a03', 'a01',
                 'a39', 'a48', 'a37', 'a29', 'a41', 'a11',
                 'a27', 'a05', 'a46', 'a02', 'a17', 'a32']
  ]0.40 - 0.60] ['a14', 'a12', 'a20', 'a13', 'a08', 'a06',
                 'a24', 'a47', 'a31', 'a09', 'a21', 'a19',
                 'a43', 'a49', 'a50', 'a40', 'a28', 'a38',
                 'a25', 'a45', 'a18', 'a16', 'a36', 'a35',
                 'a30', 'a23', 'a34', 'a15']
  ]0.20 - 0.40] ['a10', 'a26', 'a04', 'a44']
```

As we are rating into upperclosed quintiles, we obtain from the ranking above an immediate precise rating result. No performance record is rated in the lowest quintile]0.00 - 0.20] and in the highest quintile]0.80 - 1.00] and 28 out of the 50 records are rated in the midfiled, i.e. the median quintile]0.40 - 0.60].

The rating-by-ranking delivers thus a precise quantiling of a given performance tableau. One must however not forget that there does not exist a single optimal ranking rule, and various ranking heuristics may render also various more or less diverging rating results.

Back to *Content Table* (page 1)

2.7 Rating with learned performance quantile norms

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 5 **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.

Incremental learning of historical performance quantiles

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` 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 2.49: Computing performance quantiles from a given performance tableau

```
>>> from performanceQuantiles import PerformanceQuantiles
>>> from randomPerfTabs import RandomCBPerformanceTableau
>>> nbrActions=900
>>> nbrCrit = 7
>>> seed = 100
>>> tp = RandomCBPerformanceTableau(numberOfActions=nbrActions,
...                         numberOfCriteria=nbrCrit,seed=seed)

>>> pq = PerformanceQuantiles(tp,
```

(continues on next page)

(continued from previous page)

```
10 ...                    numberOfBins = 'quartiles',
11 ...                    LowerClosed=True)
12
13 >>> pq
14  *------- PerformanceQuantiles instance description ------*
15  Instance class    : PerformanceQuantiles
16  Instance name     : 4-tiled_performances
17  # Objectives      : 2
18  # Criteria        : 7
19  # Quantiles       : 4
20  # History sizes   : {'c1': 887, 'b1': 888, 'b2': 891, 'b3': 895,
21                       'b4': 892, 'c2': 893, 'b5': 887}
22  Attributes        : ['perfTabType', 'valueDigits', 'actionsTypeStatistics
↪',
23                       'objectives', 'BigData', 'missingDataProbability',
24                       'criteria', 'LowerClosed', 'name',
25                       'quantilesFrequencies', 'historySizes',
26                       'limitingQuantiles', 'cdf']
```

The `PerformanceQuantiles` class parameter *numberOfBins* (see Listing 2.49 Line 10 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 2.50: Printing out the estimated quartile limits

```
>>> pq.showLimitingQuantiles(ByObjectives=True)
 ----  Historical performance quantiles -----*
 Costs
 criteria | weights |  '0.00'   '0.25'   '0.50'   '0.75'   '1.00'
 ---------|---------------------------------------------------------
    'c1'  |    5    |   -10       -7       -5       -3        0
    'c2'  |    5    | -96.37   -70.65   -50.10   -30.00    -1.43
 Benefits
 criteria | weights | '0.00'   '0.25'   '0.50'   '0.75'    '1.00'
 ---------|---------------------------------------------------------
    'b1'  |    2    |  1.99    29.82    49,44     70.73    99.83
    'b2'  |    2    |   0        3        5         7        10
    'b3'  |    2    |   0        3        5         7        10
    'b4'  |    2    |  3.27    30.10    50.82     70.89    98.05
    '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 2.50 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 `RandomPerformanceTableau` model, one the for the two objectives `RandomCBPerformanceTableau` Cost-Benefit model, and one for the `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 2.51: 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 quartile norms shown above
5 >>> pq.updateQuantiles(newTab,historySize=None)
```

Parameter *historySize* (see Listing 2.51 Line 5) of the `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. 2.19).

```
>>> # showing the updated quantile limits in a browser view
>>> 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. 2.19: Showing the updated quartiles limits

Rating-by-ranking 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 `RatingByLearnedQuantilesDigraph` class from the `ratingDigraphs` module. The rating result is computed by **ranking** the new performance records together with the learned quantile limits. The constructor requires a valid `PerformanceQuantiles` instance.

Note: It is important to notice that the `RatingByLearnedQuantilesDigraph` class, contrary to the generic `OutrankingDigraph` class, does not only inherit from the generic `PerformanceTableau` class, but also from the `PerformanceQuantiles` class. The **actions** in such a `RatingByLearnedQuantilesDigraph` instance do not contain only the newly given decision alternatives, but also the historical quantile profiles obtained from a given `PerformanceQuantiles` 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 2.52: Computing an absolute rating of 10 new decision alternatives

```
>>> from ratingDigraphs import\
...                    RatingByLearnedQuantilesDigraph
>>> newActions = rpg.randomActions(10)
```

(continues on next page)

(continued from previous page)

```
4  >>> lqr = RatingByLearnedQuantilesDigraph(pq,newActions,
5  ...                                        rankingRule='best')
6  >>> lqr
7   *-----  Object instance description -----------*
8   Instance class      : RatingByLearnedQuantilesDigraph
9   Instance name       : learnedRatingDigraph
10  Actions             : 14
11  Criteria            : 7
12  Quantiles           : 4
13  Lowerclosed         : True
14  Rankingrule         : Copeland
15  Size                : 93
16  Valuation domain    : [-1.00;1.00]
17  Determinateness (%): 76.09
18  Attributes          : ['runTimes', 'objectives', 'criteria',
19     'LowerClosed', 'quantilesFrequencies', 'criteriaCategoryLimits',
20     'limitingQuantiles', 'historySizes', 'cdf', 'NA', 'name',
21     'newActions', 'evaluation', 'actionsOrig', 'actions',
22     'categories', 'profiles', 'profileLimits', 'order',
23     'nbrThreads', 'relation', 'valuationdomain', 'sorting',
24     'relativeCategoryContent', 'sortingRelation', 'rankingRule',
25     'rankingCorrelation', 'rankingScores', 'actionsRanking',
26     'ratingCategories']
27  *------  Constructor run times (in sec.) ------*
28  Threads             : 1
29  Total time          : 0.03680
30  Data input          : 0.00119
31  Compute quantiles   : 0.00014
32  Compute outrankings : 0.02771
33  rating-by-sorting   : 0.00033
34  rating-by-ranking   : 0.00742
```

Data input to the `RatingByLearnedQuantilesDigraph` class constructor (see Listing 2.52 Line 4) 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 2.53: Performance tableau of the new incoming decision alternatives

```
>>> lqr.showPerformanceTableau(actionsSubset=lqr.newActions)
 *----  performance tableau -----*
 criteria | a1001 a1002 a1003 a1004 a1005 a1006 a1007 a1008 a1009 a1010
 ---------|-----------------------------------------------------------
    'b1'  |  37.0  27.0  24.0  16.0  42.0  33.0  39.0  64.0  42.0  32.0
    'b2'  |   2.0   5.0   8.0   3.0   3.0   3.0   6.0   5.0   4.0   9.0
```

(continues on next page)

(continued from previous page)

```
    'b3'  |   2.0   4.0   2.0   1.0   6.0   3.0   2.0   6.0   6.0   6.0
    'b4'  |  61.0  54.0  74.0  25.0  28.0  20.0  20.0  49.0  44.0  55.0
    'b5'  |  31.0  63.0  61.0  48.0  30.0  39.0  16.0  96.0  57.0  51.0
    'c1'  |  -4.0  -6.0  -8.0  -5.0  -1.0  -5.0  -1.0  -6.0  -6.0  -4.0
    '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 2.53), we recognize alternatives *a1001* (see column 2) and *a1010* (see last column) we have mentioned in our introduction.

The `RatingByLearnedQuantilesDigraph` 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 2.54 below).

Listing 2.54: Showing the limiting profiles of the rating quantiles

```
1  >>> lqr.showPerformanceTableau(actionsSubset=lqr.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 2.52 Lines 27-) 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 `RatingByLearnedQuantilesDigraph` 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 2.54 Line 4) the `RatingByLearnedQuantilesDigraph` 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 2.55: Copeland ranking of new alternatives and historical quartile limits

```
>>> lqr.rankingRule
 'Copeland'
>>> lqr.actionsRanking
 ['m4', 'a1005', 'a1010', 'a1002', 'a1008', 'a1006', 'a1001',
  'a1003', 'm3', 'a1007', 'a1004', 'a1009', 'm2', 'm1']
>>> lqr.showCorrelation(lqr.rankingCorrelation)
 Correlation indexes:
  Crisp ordinal correlation  : +0.945
  Epistemic determination    :  0.522
  Bipolar-valued equivalence : +0.493
```

We achieve here (see Listing 2.55) 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*.

```
>>> lqr.ratingCategories
 OrderedDict([
 ('m2', ['a1007','a1004','a1009']),
 ('m3', ['a1005','a1010','a1002','a1008','a1006','a1001','a1003'])
 ])
```

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 2.56: Showing a quantiles rating result

```
>>> lqr.showRatingByQuantilesRanking()
 *-------- rating by quantiles ranking result ---------
 [0.50 - 0.75[ ['a1005', 'a1010', 'a1002', 'a1008',
                'a1006', 'a1001', 'a1003']
 [0.25 - 0.50[ ['a1004', 'a1007', 'a1009']
```

The same result may more conveniently be consulted in a browser view via a specialised rating heatmap format (see `showHTMLPerformanceHeatmap()` method (see Fig. 2.20).

```
>>> lqr.showHTMLRatingHeatmap(
...             pageTitle='Heatmap of Quartiles Rating',
...             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. 2.20: Heatmap of absolute quartiles ranking

Using furthermore a specialised version of the `exportGraphViz()` method allows drawing the same rating result in a Hasse diagram format (see Fig. 2.21).

```
>>> lqr.exportRatingByRankingGraphViz('normedRatingDigraph')
 *---- exporting a dot file for GraphViz tools ---------*
  Exporting to normedRatingDigraph.dot
  dot -Grankdir=TB -Tpng normedRatingDigraph.dot -o normedRatingDigraph.png
```

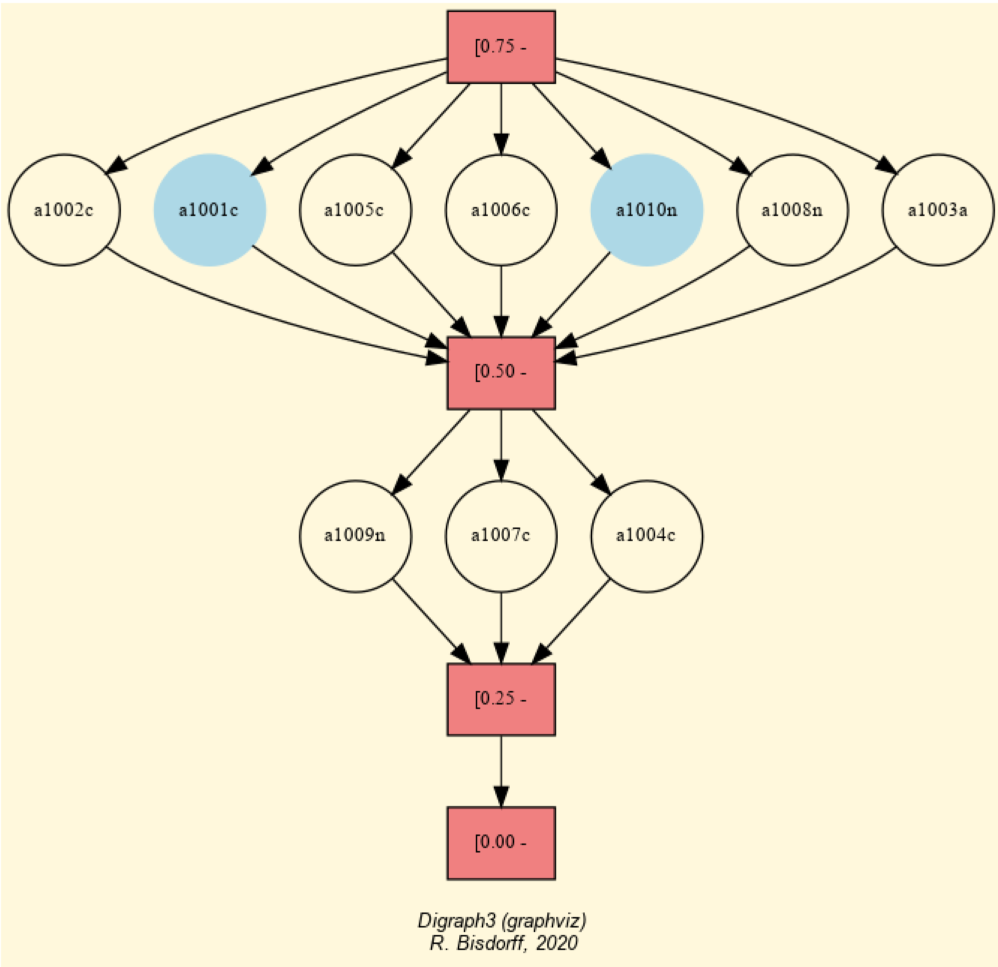


Fig. 2.21: Absolute quartiles rating digraph

We may now answer the **absolute 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. 2.21), even if the *Copeland* ranking, obtained from the underlying strict outranking digraph (see Fig. 2.20), 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 2.57: Absolute deciles rating result

```
>>> pq1 = PerformanceQuantiles(tp, numberOfBins = 'deciles',
...                     LowerClosed=True)

>>> pq1.updateQuantiles(newTab,historySize=None)
>>> lqr1 = RatingByLearnedQuantilesDigraph(pq1,newActions,rankingRule='best')
>>> lqr1.showRatingByQuantilesRanking()
 *-------- Deciles rating result ---------
 [0.60 - 0.70[ ['a1005', 'a1010', 'a1008', 'a1002']
 [0.50 - 0.60[ ['a1006', 'a1001', 'a1003']
 [0.40 - 0.50[ ['a1007', 'a1004']
 [0.30 - 0.40[ ['a1009']
```

Compared with the quartiles rating result, we notice in Listing 2.57 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. 2.22).

```
>>> lqr1.showHTMLRatingHeatmap(
...       pageTitle='Heatmap of the deciles rating',
...       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. 2.22: Heatmap of absolute 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 2.58: From deciles interpolated quartiles rating result

```
1 >>> lqr2 = RatingByLearnedQuantilesDigraph(pq1,newActions,
2 ...                     quantiles='quartiles')
3 >>> lqr2.showRatingByQuantilesRanking()
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 2.58 Line 2), we may recover by interpolation the same quartiles rating as obtained directly with historical performance quartiles (see Listing 2.56). 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 1)

2.8 Sparse bipolar-valued outranking digraphs

The `RatinbByRelativeQuantilesDigraph` constructor gives via the rating by relative quantiles a linearly ordered decomposition of the corresponding bipolar-valued outranking digraph (see Listing 2.46). This decomposition leads us to a new **sparse pre-ranked** outranking digraph model.

The sparse pre-ranked outranking digraph model

We may notice 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 tableau for linearly decomposing the set of potential decision alternatives into **ordered quantiles equivalence classes** by using the quantiles sorting technique seen in the previous Section.

In the coding example shown in Listing 2.59 below, we generate for instance, first (Lines 2-3), a simple performance tableau of 75 decision alternatives and, secondly (Lines 4), we construct the corresponding `PreRankedOutrankingDigraph` instance called *prg*. Notice by the way the *BigData* flag (Line 3) used here for generating a parsimoniously commented performance tableau.

Listing 2.59: Computing a *pre-ranked* sparse outranking digraph

```
1  >>> from sparseOutrankingDigraphs import \
2  ...                             PreRankedOutrankingDigraph
3  >>> tp = RandomPerformanceTableau(numberOfActions=75,
4  ...                                 BigData=True,seed=100)
5  >>> prg = PreRankedOutrankingDigraph(tp,quantiles=5)
6  >>> prg
7   *----- Object instance description ------*
8    Instance class     : PreRankedOutrankingDigraph
9    Instance name      : randomperftab_pr
10   # Actions          : 75
11   # Criteria         : 7
12   Sorting by         : 5-Tiling
13   Ordering strategy  : average
14   # Components       : 9
15   Minimal order      : 1
16   Maximal order      : 25
17   Average order      : 8.3
18   fill rate          : 20.432%
19   Attributes         : ['actions', 'criteria', 'evaluation', 'NA', 'name',
20       'order', 'runTimes', 'dimension', 'sortingParameters',
21       'valuationdomain', 'profiles', 'categories', 'sorting',
22       'decomposition', 'nbrComponents', 'components',
23       'fillRate', 'minimalComponentSize', 'maximalComponentSize', ... ]
```

The ordering of the 5-tiling result is following the **average** lower and upper quintile limits strategy (see previous section and Listing 2.59 Line 12). We obtain here 9 ordered components of minimal order 1 and maximal order 25. The corresponding **pre-ranked decomposition** may be visualized as follows.

Listing 2.60: The quantiles decomposition of a pre-ranked outranking digraph

```
>>> prg.showDecomposition()
 *--- quantiles decomposition in decreasing order---*
  c1. ]0.80-1.00] : [5, 42, 43, 47]
  c2. ]0.60-1.00] : [73]
  c3. ]0.60-0.80] : [1, 4, 13, 14, 22, 32, 34, 35, 40,
                     41, 45, 61, 62, 65, 68, 70, 75]
  c4. ]0.40-0.80] : [2, 54]
  c5. ]0.40-0.60] : [3, 6, 7, 10, 15, 18, 19, 21, 23, 24,
                     27, 30, 36, 37, 48, 51, 52, 56, 58,
```

(continues on next page)

(continued from previous page)

```
10                         63, 67, 69, 71, 72, 74]
11   c6. ]0.20-0.60] : [8, 11, 25, 28, 64, 66]
12   c7. ]0.20-0.40] : [12, 16, 17, 20, 26, 31, 33, 38, 39,
13                      44, 46, 49, 50, 53, 55]
14   c8. ]   <-0.40] : [9, 29, 60]
15   c9. ]   <-0.20] : [57, 59]
```

The highest quintile class (]80%-100%]) contains decision alternatives *5*, *42*, *43* and *47*. Lowest quintile class (]-20%]) gathers alternatives *57* and *59* (see Listing 2.60 Lines 3 and 15). We may inspect the resulting sparse outranking relation map as follows in a browser view.

```
>>> prg.showHTMLRelationMap()
```

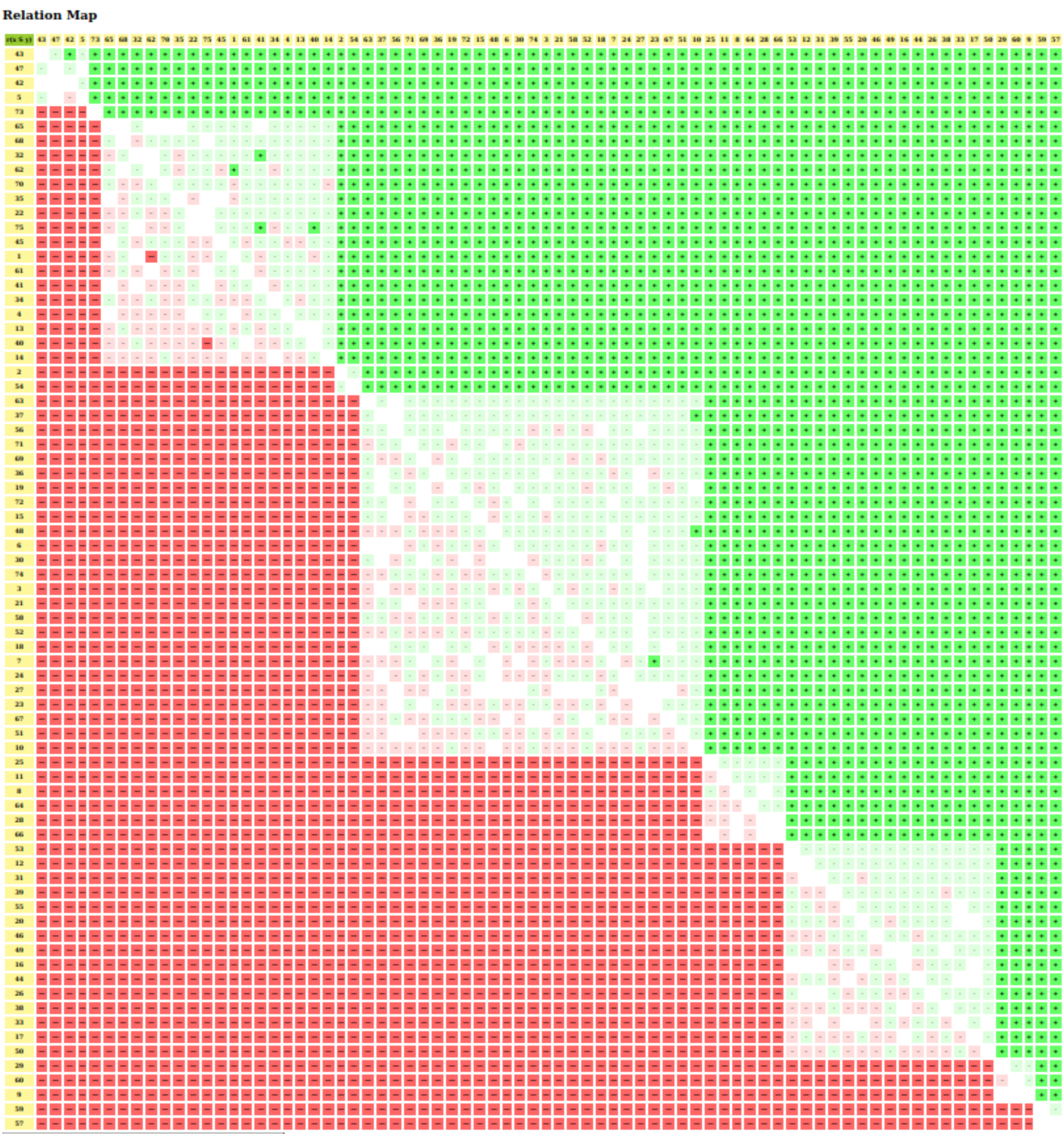


Fig. 2.23: The relation map of a sparse outranking digraph

In Fig. 2.23 we easily recognize the 9 linearly ordered quantile equivalence classes. *Green* and *light-green* show positive **outranking** situations, whereas positive **outranked** situations are shown in **red** and **light-red**. Indeterminate situations appear in white. In each one of the 9 quantile equivalence classes we recover in fact the corresponding bipolar-valued outranking *sub-relation*, which leads to an actual **fill-rate** of 20.4% (see Listing 2.59 Line 20).

We may now check how faithful the sparse model represents the complete outranking relation.

```
>>> g = BipolarOutrankingDigraph(tp)
>>> corr = prg.computeOrdinalCorrelation(g)
>>> g.showCorrelation(corr)
 Correlation indexes:
  Crisp ordinal correlation  : +0.863
  Epistemic determination    :  0.315
  Bipolar-valued equivalence : +0.272
```

The ordinal correlation index between the standard and the sparse outranking relations is quite high (+0.863) and their bipolar-valued equivalence is supported by a mean criteria significance majority of $(1.0{+}0.272)/2 = 64\%$.

It is worthwhile noticing in Listing 2.59 Line 18 that sparse pre-ranked outranking digraphs do not contain a *relation* attribute. The access to pairwise outranking characteristic values is here provided via a corresponding `relation()` function.

```
def relation(self,x,y):
    """
    Dynamic construction of the global
    outranking characteristic function r(x,y).
    """
    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']
    if cx == cy:
        return self.components[cx]['subGraph'].relation[x][y]
    elif self.components[cx]['rank'] > self.components[cy]['rank']:
        return Min
    else:
        return Max
```

All reflexive situations are set to the *indeterminate* value. When two decision alternatives belong to a same component -quantile equivalence class- we access the relation attribute of the corresponding outranking sub-digraph. Otherwise we just check the respective ranks of the components.

Ranking pre-ranked sparse outranking digraphs

Each one of these 9 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 *prg.boostedOrder* attribute. A reversed linear ranking (from the best to the worst) is stored in a *prg.boostedRanking* attribute.

Listing 2.61: Showing the component wise *Copeland* ranking

```
1 >>> prg.boostedRanking
2  [43, 47, 42, 5, 73, 65, 68, 32, 62, 70, 35, 22, 75, 45, 1,
3   61, 41, 34, 4, 13, 40, 14, 2, 54, 63, 37, 56, 71, 69, 36,
4   19, 72, 15, 48, 6, 30, 74, 3, 21, 58, 52, 18, 7, 24, 27,
5   23, 67, 51, 10, 25, 11, 8, 64, 28, 66, 53, 12, 31, 39, 55,
6   20, 46, 49, 16, 44, 26, 38, 33, 17, 50, 29, 60, 9, 59, 57]
```

Alternative *43* appears *first ranked*, whereas alternative *57* is *last ranked* (see Listing 2.61 Line 2 and 6). The quality of this ranking result may be assessed by computing its ordinal correlation with the standard outranking relation.

```
>>> corr = g.computeRankingCorrelation(prg.boostedRanking)
>>> g.showCorrelation(corr)
 Correlation indexes:
  Crisp ordinal correlation  : +0.807
  Epistemic determination    :  0.315
  Bipolar-valued equivalence : +0.254
```

We may also verify below that the *Copeland* ranking obtained from the standard outranking digraph is highly correlated (+0.822) with the one obtained from the sparse outranking digraph.

```
>>> from linearOrders import CopelandOrder
>>> cop = CopelandOrder(g)
>>> print(cop.computeRankingCorrelation(prg.boostedRanking))
 {'correlation': 0.822, 'determination': 1.0}
```

Noticing the computational efficiency of the quantiles sorting construction, coupled with the separability property of the quantile class membership characteristics computation, we will make usage of the `PreRankedOutrankingDigraph` constructor in the *cythonized Digraph3 modules* (page 125) for HPC ranking big and even huge performance tableaux.

Back to *Content Table* (page 1)

2.9 HPC ranking with big outranking digraphs

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[Page 125, 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 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.

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

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.

```
>>> from cRandPerfTabs import *
>>> t = cRandomPerformanceTableau(numberOfActions=4,numberOfCriteria=2)
>>> t
    *------- PerformanceTableau instance description ------*
    Instance class    : cRandomPerformanceTableau
    Seed              : None
    Instance name     : cRandomperftab
    # Actions         : 4
    # Criteria        : 2
    Attributes        : ['randomSeed', 'name', 'actions', 'criteria',
                          'evaluation', 'weightPreorder']
>>> t.actions
    OrderedDict([(1, {'name': '#1'}), (2, {'name': '#2'}),
                 (3, {'name': '#3'}), (4, {'name': '#4'})])
>>> t.criteria
    OrderedDict([
    ('g1', {'name': 'RandomPerformanceTableau() instance',
            'comment': 'Arguments: ; weightDistribution=equisignificant;
                         weightScale=(1, 1); commonMode=None',
            'thresholds': {'ind': (10.0, 0.0),
                            'pref': (20.0, 0.0),
                            'veto': (80.0, 0.0)},
            'scale': (0.0, 100.0),
            'weight': 1,
            'preferenceDirection': 'max'}),
    ('g2', {'name': 'RandomPerformanceTableau() instance',
            'comment': 'Arguments: ; weightDistribution=equisignificant;
                        weightScale=(1, 1); commonMode=None',
            'thresholds': {'ind': (10.0, 0.0),
                            'pref': (20.0, 0.0),
                            'veto': (80.0, 0.0)},
            'scale': (0.0, 100.0),
            'weight': 1,
            'preferenceDirection': 'max'})])
>>> t.evaluation
     {'g1': {1: 35.17, 2: 56.4, 3: 1.94, 4: 5.51},
      'g2': {1: 95.12, 2: 90.54, 3: 51.84, 4: 15.42}}
>>> t.showPerformanceTableau()
     Criteria |  'g1'    'g2'
```

(continues on next page)

(continued from previous page)

```
     Actions  |    1       1
     ---------|---------------
        '#1'  |  91.18   90.42
        '#2'  |  66.82   41.31
        '#3'  |  35.76   28.86
        '#4'  |   7.78   37.64
```

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

```
>>> t1 = t.convert2Standard()
>>> t1.convertWeight2Decimal()
>>> t1.convertEvaluation2Decimal()
>>> t1
 *------- PerformanceTableau instance description ------*
 Instance class   : PerformanceTableau
 Seed             : None
 Instance name    : std_cRandomperftab
 # Actions        : 4
 # Criteria       : 2
 Attributes       : ['name', 'actions', 'criteria', 'weightPreorder',
                     'evaluation', 'randomSeed']
```

C-implemented integer-valued outranking digraphs

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

```
>>> t = cRandomPerformanceTableau(numberOfActions=1000,
↪numberOfCriteria=2)
>>> from cIntegerOutrankingDigraphs import *
>>> g = IntegerBipolarOutrankingDigraph(t,Threading=True,nbrCores=4)
>>> g
   *------- Object instance description ------*
   Instance class   : IntegerBipolarOutrankingDigraph
   Instance name    : rel_cRandomperftab
   # Actions        : 1000
   # Criteria       : 2
   Size             : 465024
   Determinateness  : 56.877
   Valuation domain : {'min': -2, 'med': 0, 'max': 2,
                       'hasIntegerValuation': True}
   ----  Constructor run times (in sec.) ----
   Total time       : 4.23880
   Data input       : 0.01203
   Compute relation : 3.60788
```

(continues on next page)

(continued from previous page)

```
   Gamma sets       : 0.61889
   #Threads         : 4
   Attributes       : ['name', 'actions', 'criteria', 'totalWeight',
                       'valuationdomain', 'methodData', 'evaluation',
                       'order', 'runTimes', 'nbrThreads', 'relation',
                       'gamma', 'notGamma']
```

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

```
>>> from outrankingDigraphs import BipolarOutrankingDigraph
>>> t1 = t.convert2Standard()
>>> g1 = BipolarOutrankingDigraph(t1,Threading=True,nbrCores=4)
>>> g1
   *------- Object instance description ------*
   Instance class    : BipolarOutrankingDigraph
   Instance name     : rel_std_cRandomperftab
   # Actions         : 1000
   # Criteria        : 2
   Size              : 465024
   Determinateness   : 56.817
   Valuation domain : {'min': Decimal('-1.0'),
                       'med': Decimal('0.0'),
                       'max': Decimal('1.0'),
                       'precision': Decimal('0')}
   ----  Constructor run times (in sec.) ----
   Total time        : 8.63340
   Data input        : 0.01564
   Compute relation : 7.52787
   Gamma sets        : 1.08987
   #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.

```
>>> from digraphsTools import total_size
>>> total_size(g)
 108662777
>>> total_size(g1)
 212679272
>>> total_size(g.relation)/total_size(g)
 0.34
>>> total_size(g.gamma)/total_size(g)
 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.

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.

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

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

```
>>> from cSparseIntegerOutrankingDigraphs import *
>>> sg = SparseIntegerOutrankingDigraph(t,quantiles=4)
>>> sg
 *----- Object instance description --------------*
 Instance class    : SparseIntegerOutrankingDigraph
 Instance name     : cRandomperftab_mp
 # Actions         : 100
 # Criteria        : 7
 Sorting by        : 4-Tiling
 Ordering strategy : average
 Ranking rule      : Copeland
 # Components      : 6
 Minimal order     : 1
 Maximal order     : 35
 Average order     : 16.7
 fill rate         : 24.970%
 *----  Constructor run times (in sec.) ----
 Nbr of threads    : 1
 Total time        : 0.08212
 QuantilesSorting  : 0.01481
 Preordering       : 0.00022
 Decomposing       : 0.06707
 Ordering          : 0.00000
 Attributes       : ['runTimes', 'name', 'actions', 'criteria',
                     'evaluation', 'order', 'dimension',
                     'sortingParameters', 'nbrOfCPUs',
```

(continues on next page)

(continued from previous page)

```
                    'valuationdomain', 'profiles', 'categories',
                    'sorting', 'minimalComponentSize',
                    'decomposition', 'nbrComponents', 'nd',
                    'components', 'fillRate',
                    'maximalComponentSize', 'componentRankingRule',
                    'boostedRanking']
```

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

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

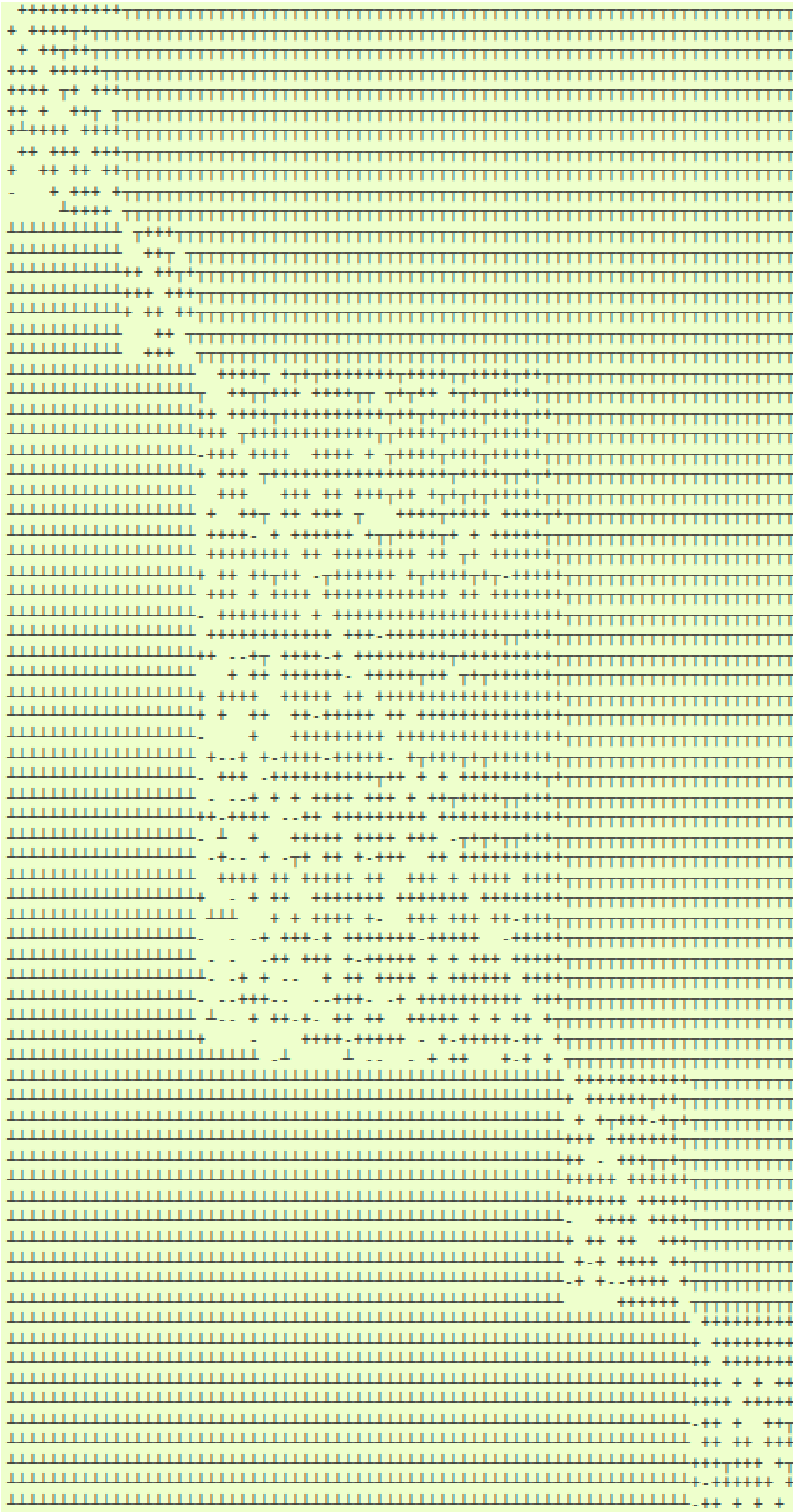


Fig. 2.24: Sparse quartiles-sorting decomposed outranking relation (extract). **Legend**: *outranking* for certain ($\top$); *outranked* for certain ($\bot$); more or less *outranking* ($+$); more or less *outranked* ($-$); *indeterminate* ( ).

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('%.0fkB' % (total_size(sg)/1024) )
769kB
```

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

```
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
```

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.

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

```
>>> qr = cQuantilesRankingDigraph(t,4)
>>> qr
 *----- Object instance description --------------*
 Instance class    : cQuantilesRankingDigraph
 Instance name     : cRandomperftab_mp
 # Actions         : 100
 # Criteria        : 7
 Sorting by        : 4-Tiling
 Ordering strategy : optimal
 Ranking rule      : Copeland
 # Components      : 47
 Minimal order     : 1
 Maximal order     : 10
 Average order     : 2.1
 fill rate         : 2.566%
 *----  Constructor run times (in sec.) ----*
 Nbr of threads    : 1
 Total time        : 0.03702
 QuantilesSorting  : 0.01785
 Preordering       : 0.00022
 Decomposing       : 0.01892
 Ordering          : 0.00000
```

(continues on next page)

(continued from previous page)

```
Attributes       : ['runTimes', 'name', 'actions', 'order',
                    'dimension', 'sortingParameters', 'nbrOfCPUs',
                    'valuationdomain', 'profiles', 'categories',
                    'sorting', 'minimalComponentSize',
                    'decomposition', 'nbrComponents', 'nd',
                    'components', 'fillRate', 'maximalComponentSize',
                    'componentRankingRule', 'boostedRanking']
```

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

```
>>> qr.components
 OrderedDict([
 ('c01', {'rank': 1,
          'lowQtileLimit': ']0.75',
          'highQtileLimit': '1.00]',
          'componentRanking': [53]}),
 ('c02', {'rank': 2,
          'lowQtileLimit': ']0.75',
          'highQtileLimit': '1.00]',
          'componentRanking': [3, 23, 63, 50]}),
 ('c03', {'rank': 3,
          'lowQtileLimit': ']0.75',
          'highQtileLimit': '1.00]',
          'componentRanking': [34, 44, 56, 24, 93, 41]}),
 ...
 ...
 ...
 ('c45', {'rank': 45,
          'lowQtileLimit': ']0.25',
          'highQtileLimit': '0.50]',
          'componentRanking': [49]}),
 ('c46', {'rank': 46,
          'lowQtileLimit': ']0.25',
          'highQtileLimit': '0.50]',
          'componentRanking': [52, 16, 86]}),
 ('c47', {'rank': 47,
          'lowQtileLimit': ']<',
          'highQtileLimit': '0.25]',
          'componentRanking': [79, 83, 84]})])
>>> print('%.0fkB' % (total_size(qr)/1024) )
 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.

```
>>> from cIntegerOutrankingDigraphs import *
>>> ig = IntegerBipolarOutrankingDigraph(t)
>>> print('Complete outranking : %+.4f'\
...         % (ig.computeOrderCorrelation(ig.computeCopelandOrder())\
...            ['correlation']))

 Complete outranking : +0.7474
>>> print('Sparse 4-tiling : %+.4f'\
...         % (ig.computeOrderCorrelation(\
...            list(reversed(sg.boostedRanking)))['correlation']))

 Sparse 4-tiling          : +0.7172
>>> print('Optimzed sparse 4-tiling: %+.4f'\
...          % (ig.computeOrderCorrelation(\
...             list(reversed(qr.boostedRanking)))['correlation']))

 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.

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/

$\gtrapprox^q$ outranking relation		q	fill	nbr.	run
order	size		rate	cores	time
5 000	25×10^6	4	0.005%	28	0.5"
10 000	1×10^8	4	0.001%	28	1"
100 000	1×10^{10}	5	0.002%	28	10"
1 000 000	1×10^{12}	6	0.001%	64	2'
3 000 000	9×10^{12}	15	0.004%	64	13'
6 000 000	36×10^{12}	15	0.002%	64	41'

Fig. 2.25: HPC-UL Ranking Performance Records (Spring 2018)

Example python session on the HPC-UL Iris-126 -skylake node[Page 135, 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
```

(continues on next page)

(continued from previous page)

```
27 # Actions           : 1000000
28 # Criteria          : 21
29 Sorting by          : 10-Tiling
30 Ordering strategy : optimal
31 Ranking rule        : NetFlows
32 # Components        : 233645
33 Minimal order       : 1
34 Maximal order       : 153
35 Average order       : 4.3
36 fill rate           : 0.001%
37 *----  Constructor run times (in sec.) ----*
38 Nbr of threads      : 28
39 Total time          : 177.02770
40 QuantilesSorting  : 99.55377
41 Preordering         : 5.17954
42 Decomposing         : 72.29356
```

On this 2x14c Intel Xeon Gold 6132 @ 2.6 GHz equipped HPC node with 132GB RAM[Page 135, 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.

```
>>> pt.showPerformanceTableau(
...              actionsSubset=qr.boostedRanking[:5],
...              Transposed=True)

*----  performance tableau -----*
 criteria | weights |  #773909  #668947  #567308  #578560  #426464
 ---------|---------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
'Ec01'  |    42   |   969.81   844.71   917.00     NA     808.35
'So02'  |    48   |     NA     891.52   836.43     NA     899.22
'En03'  |    56   |   687.10     NA     503.38   873.90     NA
'So04'  |    48   |   455.05   845.29   866.16   800.39   956.14
'En05'  |    56   |   809.60   846.87   939.46   851.83   950.51
'Ec06'  |    42   |   919.62   802.45   717.39   832.44   974.63
'Ec07'  |    42   |   889.01   722.09   606.11   902.28   574.08
'So08'  |    48   |   862.19   699.38   907.34   571.18   943.34
'En09'  |    56   |   857.34   817.44   819.92   674.60   376.70
'Ec10'  |    42   |     NA     874.86     NA     847.75   739.94
'En11'  |    56   |     NA     824.24   855.76     NA     953.77
'Ec12'  |    42   |   802.18   871.06   488.76   841.41   599.17
'En13'  |    56   |   827.73   839.70   864.48   720.31   877.23
'So14'  |    48   |   943.31   580.69   827.45   815.18   461.04
'En15'  |    56   |   794.57   801.44   924.29   938.70   863.72
'Ec16'  |    42   |   581.15   599.87   949.84   367.34   859.70
'So17'  |    48   |   881.55   856.05     NA     796.10   655.37
'Ec18'  |    42   |   863.44   520.24   919.75   865.14   914.32
'So19'  |    48   |     NA       NA       NA     790.43   842.85
'Ec20'  |    42   |   582.52   831.93   820.92   881.68   864.81
'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)).

```
>>> for x in qr.boostedRanking[:5]:
...     print(pt.actions[x]['name'],
...           pt.actions[x]['profile'])

 #773909 {'Eco': '+', 'Soc': '+', 'Env': '+'}
 #668947 {'Eco': '+', 'Soc': '+', 'Env': '+'}
 #567308 {'Eco': '+', 'Soc': '+', 'Env': '+'}
 #578560 {'Eco': '+', 'Soc': '+', 'Env': '+'}
 #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).

```
>>> best10 = cPartialPerformanceTableau(pt,qr.boostedRanking[:10])
```

(continues on next page)

(continued from previous page)

```
>>> from cIntegerOutrankingDigraphs import *
>>> g = IntegerBipolarOutrankingDigraph(best10)
>>> g.valuationdomain
 {'min': -1008, 'med': 0, 'max': 1008, 'hasIntegerValuation': True}
>>> g.showRelationTable(ReflexiveTerms=False)
 * ---- Relation Table -----
  r(x>y) | #773909 #668947 #567308 #578560 #426464 #298061 #155874 #815552 #279729 #928564
 --------|-----------------------------------------------------------------------------------
 #773909 |    -      +390     +90    +270     -50    +340    +220     +60    +116    +222
 #668947 |   +78      -       +42    +250     -22    +218     +56    +172     +74     +64
 #567308 |   +70     +418     -      +180    +156    +174    +266     +78    +256    +306
 #578560 |    -4      +78     +28     -       -12    +100     -48    +154    -110     -10
 #426464 |  +202     +258    +284    +138     -      +416    +312    +382    +534    +278
 #298061 |   -48      +68    +172     +32     -42     -       +54     +48    +248    +374
 #155874 |   +72     +378    +322    +174    +274    +466     -      +212    +308    +418
 #815552 |   +78     +126    +272    +318     +54    +194    +172     -       -14     +22
 #279729 |  +240     +230    -110    +290     +72    +140    +388     +62     -      +250
 #928564 |   +22     +228     -14    +246     +36     +78     +56    +110    +318     -
 r(x>y) image range := [-1008;+1008]
>>> g.condorcetWinners()
 [155874, 426464, 567308]
>>> g.computeChordlessCircuits()
 []
>>> g.computeTransitivityDegree()
 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. 2.26.

```
>>> (~(-g)).exportGraphViz()
*---- exporting a dot file for GraphViz tools ---------*
```

(continues on next page)

(continued from previous page)

```
Exporting to converse-dual_rel_best10.dot
dot -Tpng converse-dual_rel_best10.dot -o converse-dual_rel_best10.png
```

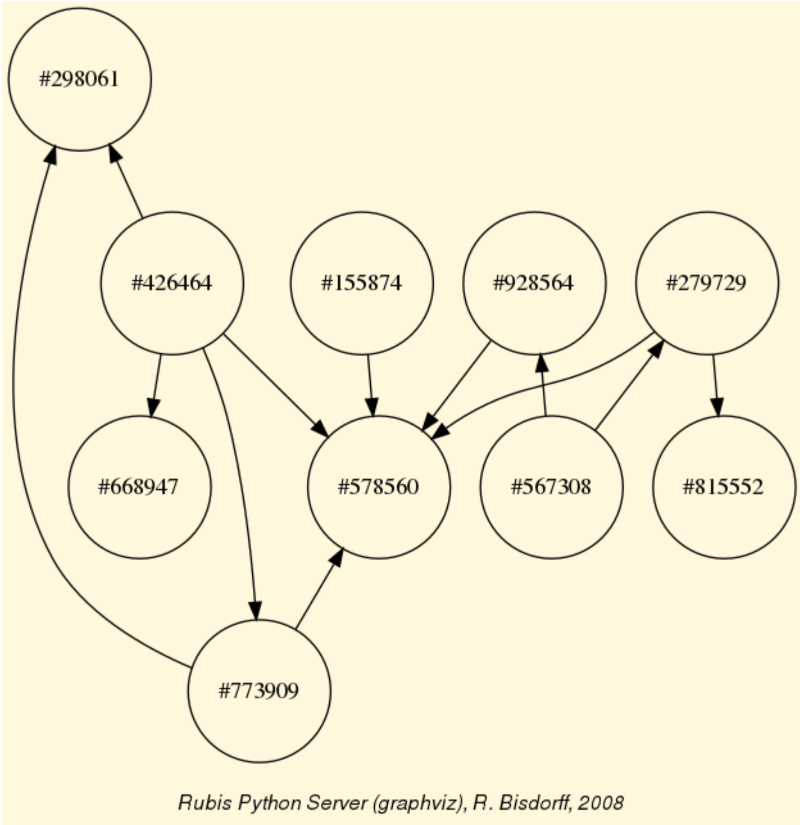


Fig. 2.26: Validated *strict* outranking situations between the ten best-ranked alternatives

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 1)

3 Evaluation and decision case studies

3.1 Alice's best choice: A *selection* case study[Page 141, 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 possible. Her parents however agree to financially support her study fees, as well as, her living costs during her studies.

[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.

The decision problem

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

Table 3.1: 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 3.1 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**[Page 89, 26].

Table 3.2: 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 3.2).

The performance tableau

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

Listing 3.1: Alice's performance tableau

```
>>> from perfTabs import PerformanceTableau
>>> t = PerformanceTableau('AliceChoice')
>>> t.showObjectives()
  *------ decision objectives -------"
  GEO: Geographical aspect
    DH Distance to parent's home 3
    BC Number of inhabitants     3
    Total weight: 6 (2 criteria)
  LEA: Learning aspect
    AS Attractiveness of the study program 6
    Total weight: 6.00 (1 criteria)
  FIN: Financial aspect
```

(continues on next page)

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

(continued from previous page)

```
    SF Annual registration fees 2
    LC Monthly living costs     2
    SL Study time               2
    Total weight: 6.00 (3 criteria)
  PRA: Professional aspect
    AP Attractiveness of the profession          2
    AI Annual professional income after studying 2
    OP Occupational Prestige                     2
    Total weight: 6.00 (3 criteria)
```

Details of the performance criteria may be consulted in a browser view (see Fig. 3.1 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. 3.1: Alice's performance criteria

It is worthwhile noticing in Fig. 3.1 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 *polarisation* of the outranking statement. 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. 3.2: 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. 3.2).

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. 3.2 (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 72).

-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].

Building a best choice recommendation

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

Listing 3.2: 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 3.2 Line 9-10).

Due to the poorly discriminating performance evaluations, nearly half of these outranking situations (see Line 13) 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. 3.3).

```
>>> 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. 3.3: '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. 3.3 above, these polarisations appear in the last column and last row. We may inspect the occurrence of such polarisations as follows.

Listing 3.3: Polarised outranking situations

```
>>> dg.showPolarisations()
 *----  Negative polarisations ----*
 number of negative polarisations : 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
```

(continues on next page)

(continued from previous page)

```
16 Considerable performance difference : -7.00
17 Veto discrimination threshold       : -7.00
18 Polarisation: r(S-HKK >= I-UHB) = -0.17 ==> -1.00
19 *----  Positive polarisations ----*
20 number of positive polarisations: 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 3.3, 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 8, 13, 18) 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. 3.3, that the four best-ranked study programs, *I-FHK*, *I-USB*, *I-UHB* and *T-FHK*, are in fact *Condorcet* winners (see Listing 3.4 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 3.4: 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 first choice(s)
8   choice              : ['I-FHK','I-UHB','I-USB','T-FHK']
9    independence        : 0.17
10   dominance           : 0.08
11   absorbency          : -0.83
```

(continues on next page)

(continued from previous page)

```
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 last 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 3.4 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. 3.3, 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 3.5 Line 2). Hence, a graphviz drawing of its *skeleton*, oriented by the previous *best*, respectively *worst* choice, may well illustrate our *best choice recommendation*.

[24] See also the Advanced Topic about computing best choice membership characteristics in the Digraph3 documentation.

Listing 3.5: Drawing the best choice recommendation

```
>>> dgcd = ~(-dg)
>>> dgcd.isTransitive()
 True
>>> dgcd.closeTransitive(Reverse=True,InSite=True)
>>> dgcd.exportGraphViz('aliceBestChoice',
...                        bestChoice=['I-FHK'],
...                        worstChoice=['S-HKK','C-HKK'])
 *---- exporting a dot file for GraphViz tools ---------*
  Exporting to aliceBestChoice.dot
  dot -Grankdir=BT -Tpng aliceBestChoice.dot -o aliceBestChoice.png
```

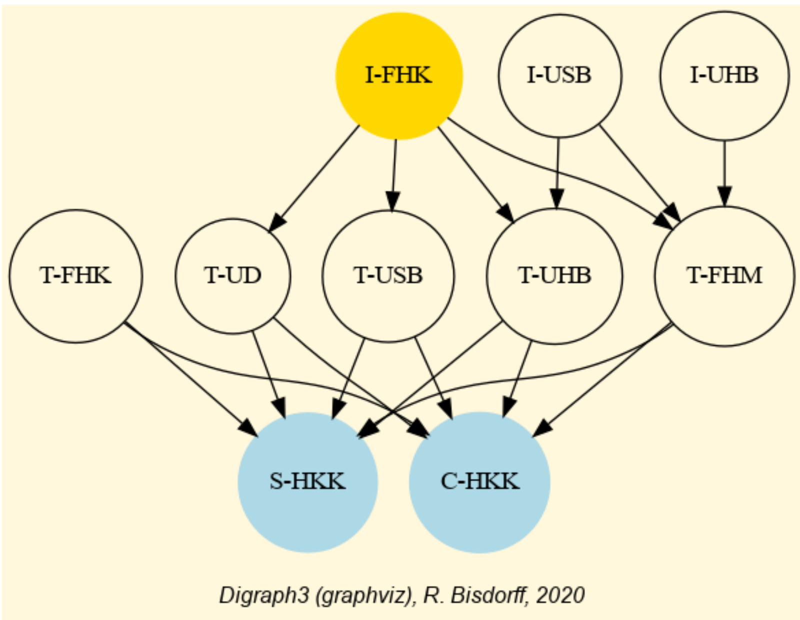


Fig. 3.4: Alice's best choice recommendation

In Fig. 3.4 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. 3.2).

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. 3.2).

```
>>> 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. 3.5: 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 `RankingByChoosingDigraph` constructor (see Listing 3.6 below), who computes a bipolar ranking by conjointly *best-choosing* and *last-rejecting* [BIS-1999].

Listing 3.6: Weakly ranking by bipolar best-choosing and last-rejecting

```
>>> from transitiveDigraphs import\
...                 RankingByChoosingDigraph

>>> rbc = RankingByChoosingDigraph(dg)
>>> rbc.showRankingByChoosing()
 Ranking by Choosing and Rejecting
  1st ranked ['I-FHK']
    2nd ranked ['I-USB']
      3rd ranked ['I-UHB']
        4th ranked ['T-FHK']
          5th ranked ['T-UD']
          5th last ranked ['T-UD']
        4th last ranked ['T-UHB', 'T-USB']
      3rd last ranked ['T-FHM']
    2nd last ranked ['C-HKK']
  1st last ranked ['S-HKK']
```

In Listing 3.6, 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 ?

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 `ConfidentBipolarOutrankingDigraph` constructor[22].

Listing 3.7: The 90% confident outranking digraph

```
>>> from outrankingDigraphs import\
...             ConfidentBipolarOutrankingDigraph
```

(continues on next page)

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

(continued from previous page)

```
3
4  >>> cdg = ConfidentBipolarOutrankingDigraph(t,
5  ...         distribution='triangular',confidence=90.0)
6
7  >>> cdg
8   *------- Object instance description ------*
9   Instance class       : ConfidentBipolarOutrankingDigraph
10  Instance name        : rel_AliceChoice_CLT
11  # Actions            : 10
12  # Criteria           : 9
13  Size                 : 44
14  Valuation domain     : [-1.00;1.00]
15  Uncertainty model    : triangular(a=0,b=2w)
16  Likelihood domain    : [-1.0;+1.0]
17  Confidence level     : 90.0%
18  Confident majority   : 14/24 (58.3%)
19  Determinateness (%)  : 68.19
```

Of the original 67 valid outranking situations, we retain 44 outranking situations as being 90%-*confident* (see Listing 3.7 Line 11). 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 3.8 below).

Listing 3.8: 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 first 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 9-10 and 16).

When previously comparing the two best-ranked study programs (see Fig. 3.5), we have observed that *I-FHK* actually positively outranks *I-USB* on all four decision objectives. When admitting equi-significant criteria significance weights 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 `UnOpposedBipolarOutrankingDigraph` constructor.

Listing 3.9: 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.isTransitive()
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 3.9 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 `TransitiveDigraph` class for drawing the corresponding partial ranking.

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

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

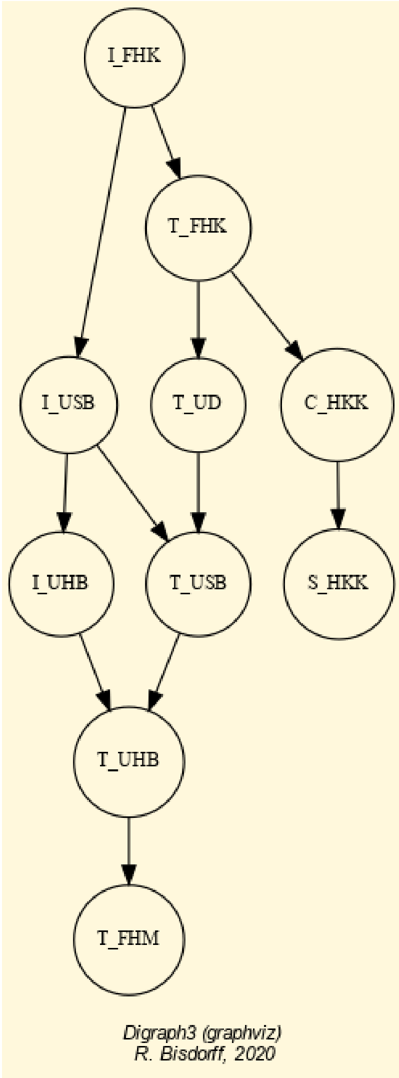


Fig. 3.6: Unopposed partial ranking of the potential study programs

Again, when *equi-signficant* performance criteria are assumed per decision objective, we observe in Fig. 3.6 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. 3.2, *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 1)

3.2 The best academic *Computer Science* Depts: a *ranking* case study

In this tutorial, we are studying a ranking decision problem based on published data from the *Times Higher Education* (THE) *World University Rankings* 2016 by *Computer Science* (CS) subject[36]. Several hundred academic CS Departments, from all over the world, were ranked that year following an overall numerical score based on the weighted average of five performance criteria: *Teaching* (the learning environment, 30%), *Research* (volume, income and reputation, 30%), *Citations* (research influence, 27.5%), *International outlook* (staff, students, and research, 7.5%), and *Industry income* (innovation, 5%).

To illustrate our *Digraph3* programming resources, we shall first have a look into the THE ranking data with short Python scripts. In a second Section, we shall relax the commensurability hypothesis of the ranking criteria and show how to similarly rank with multiple incommensurable performance criteria of ordinal significance. A third Section is finally devoted to introduce quality measures for qualifying ranking results.

The THE performance tableau

For our tutorial purpose, an extract of the published THE University rankings 2016 by computer science subject data, concerning the 75 first-ranked academic Institutions, is stored in a file named the_cs_2016.py of `PerformanceTableau` format[37].

Listing 3.10: The 2016 THE World University Ranking by CS subject

```
>>> from perfTabs import PerformanceTableau
>>> t = PerformanceTableau('the_cs_2016')
>>> t
 *------- PerformanceTableau instance description ------*
  Instance class      : PerformanceTableau
  Instance name       : the_cs_2016
  # Actions           : 75
  # Objectives        : 5
  # Criteria          : 5
  NaN proportion (%)  : 0.0
  Attributes          : ['name', 'description', 'actions',
```

(continues on next page)

[36] https://www.timeshighereducation.com/world-university-rankings/2017/subject-ranking/computer-science#!/page/0/length/25/sort_by/rank/sort_order/asc/cols/scores

[37] The performance tableau `the_cs_2016.py` is also available in the `examples` directory of the Digraph3 software collection.

(continued from previous page)

```
                              'objectives', 'criteria',
                              'weightPreorder', 'NA', 'evaluation']
```

Potential *decision actions*, in our case here, are the 75 THE best-ranked *CS Departments*, all of them located at world renowned Institutions, like *California Institute of Technology*, *Swiss Federal Institute of Technology Zurich*, *Technical University München*, *University of Oxford* or the *National University of Singapore* (see Listing 3.11 below).

Instead of using prefigured *Digraph3* **show** methods, readily available for inspecting *PerformanceTableau* instances, we will illustrate below how to write small Python scripts for printing out its content.

Listing 3.11: Printing the potential decision actions

```
>>> for x in t.actions:
...     print('%s:\t%s (%s)' %\
...         (x,t.actions[x]['name'],t.actions[x]['comment']) )

 albt:       University of Alberta (CA)
 anu:        Australian National University (AU)
 ariz:       Arizona State University (US)
 bju:        Beijing University (CN)
 bro:        Brown University (US)
 calt:       California Institute of Technology (US)
 cbu:        Columbia University (US)
 chku:       Chinese University of Hong Kong (HK)
 cihk:       City University of Hong Kong (HK)
 cir:        University of California at Irvine (US)
 cmel:       Carnegie Mellon University (US)
 cou:        Cornell University (US)
 csb:        University of California at Santa Barbara (US)
 csd:        University Of California at San Diego (US)
 dut:        Delft University of Technology (NL)
 eind:       Eindhoven University of Technology (NL)
 ens:        Superior Normal School at Paris (FR)
 epfl:       Swiss Federal Institute of Technology Lausanne (CH)
 epfr:       Polytechnic school of Paris (FR)
 ethz:       Swiss Federal Institute of Technology Zurich (CH)
 frei:       University of Freiburg (DE)
 git:        Georgia Institute of Technology (US)
 glas:       University of Glasgow (UK)
 hels:       University of Helsinki (FI)
 hkpu:       Hong Kong Polytechnic University (CN)
 hkst:       Hong Kong University of Science and Technology (HK)
 hku:        Hong Kong University (HK)
 humb:       Berlin Humboldt University (DE)
 icl:        Imperial College London (UK)
```

(continues on next page)

(continued from previous page)

```
indis:      Indian Institute of Science (IN)
itmo:       ITMO University (RU)
kcl:        King's College London (UK)
kist:       Korea Advances Institute of Science and Technology (KR)
kit:        Karlsruhe Institute of Technology (DE)
kth:        KTH Royal Institute of Technology (SE)
kuj:        Kyoto University (JP)
kul:        Catholic University Leuven (BE)
lms:        Lomonosov Moscow State University (RU)
man:        University of Manchester (UK)
mcp:        University of Maryland College Park (US)
mel:        University of Melbourne (AU)
mil:        Polytechnic University of Milan (IT)
mit:        Massachusetts Institute of Technology (US)
naji:       Nanjing University (CN)
ntu:        Nanyang Technological University of Singapore (SG)
ntw:        National Taiwan University (TW)
nyu:        New York University (US)
oxf:        University of Oxford (UK)
pud:        Purdue University (US)
qut:        Queensland University of Technology (AU)
rcu:        Rice University (US)
rwth:       RWTH Aachen University (DE)
shJi:       Shanghai Jiao Tong University (CN)
sing:       National University of Singapore (SG)
sou:        University of Southhampton (UK)
stut:       University of Stuttgart (DE)
tech:       Technion - Israel Institute of Technology (IL)
tlavu:      Tel Aviv University (IR)
tsu:        Tsinghua University (CN)
tub:        Technical University of Berlin (DE)
tud:        Technical University of Darmstadt (DE)
tum:        Technical University of München (DE)
ucl:        University College London (UK)
ued:        University of Edinburgh (UK)
uiu:        University of Illinois at Urbana-Champagne (US)
unlu:       University of Luxembourg (LU)
unsw:       University of New South Wales (AU)
unt:        University of Toronto (CA)
uta:        University of Texas at Austin (US)
utj:        University of Tokyo (JP)
utw:        University of Twente (NL)
uwa:        University of Waterloo (CA)
wash:       University of Washington (US)
wtu:        Vienna University of Technology (AUS)
zhej:       Zhejiang University (CN)
```

The THE authors base their ranking decisions on five objectives.

```
>>> for obj in t.objectives:
...     print('%s: %s (%.1f%%),\n\t%s' \
...          % (obj,t.objectives[obj]['name'],
...              t.objectives[obj]['weight'],
...              t.objectives[obj]['comment'])
...          )

 Teaching: Best learning environment (30.0%),
         Reputation survey; Staff-to-student ration;
         Doctorate-to-student ratio,
         Doctorate-to-academic-staff ratio, Institutional income.
 Research: Highest volume and repustation (30.0%),
         Reputation survey; Research income; Research productivity
 Citations: Highest research influence (27.5%),
         Impact.
 International outlook: Most international staff, students and research (7.5%),
         Proportions of international students; of international staff;
         international collaborations.
 Industry income: Best knowledge transfer (5.0%),
         Volume.
```

With a cumulated importance of 87% (see above), *Teaching*, *Research* and *Citations* represent clearly the **major** ranking objectives. *International outlook* and *Industry income* are considered of **minor** importance (12.5%).

THE does, unfortunately, not publish the detail of their performance assessments for grading CS Depts with respect to each one of the five ranking objectives[39]. The THE 2016 ranking publication reveals solely a compound assessment on a single *performance criteria* per ranking objective. The five retained performance criteria may be printed out as follows.

```
>>> for g in t.criteria:
...     print('%s:\t%s, %s (%.1f%%)' \
...        % (g,t.criteria[g]['name'],t.criteria[g]['comment'],
...           t.criteria[g]['weight']) )

 gtch:       Teaching, The learning environment (30.0%)
 gres:       Research, Volume, income and reputation (30.0%)
 gcit:       Citations, Research influence (27.5%)
 gint:       International outlook, In staff, students and research (7.5%)
 gind:       Industry income, knowledge transfer (5.0%)
```

[39] https://www.timeshighereducation.com/sites/default/files/styles/article785xauto/public/wur_graphic_1.jpg?itok=XS6NcZfL gives some insight on the subject and significance of the actual performance criteria used for grading along each ranking objective.

The largest part (87.5%) of criteria significance is, hence canonically, allocated to the major ranking criteria: *Teaching* (30%), *Research* (30%) and *Citations* (27.5%). The small remaining part (12.5%) goes to *International outlook* (7.5%) and *Industry income* (5%).

In order to render commensurable these performance criteria, the THE authors replace, per criterion, the actual performance grade obtained by each University with the corresponding **quantile** observed in the *cumulative distribution* of the performance grades obtained by all the surveyed institutions[40]. The THE ranking is eventually determined by an **overall score** per University which corresponds to the **weighted average** of these five criteria quantiles (see Listing 3.12 below).

Listing 3.12: Computing the THE overall scores

```
>>> theScores = []
>>> for x in t.actions:
...     xscore = Decimal('0')
...     for g in t.criteria:
...         xscore += t.evaluation[g][x] *\
...                    (t.criteria[g]['weight']/Decimal('100'))
...     theScores.append((xscore,x))
```

In Listing 3.13 Lines 15-16 below, we may thus notice that, in the 2016 edition of the *THE World University rankings* by CS subject, the *Swiss Federal Institute of Technology Zürich* is first-ranked with an overall score of 92.9; followed by the *California Institute of Technology* (overall score: 92.4)[38].

Listing 3.13: Printing the ranked performance table

```
1  >>> theScores.sort(reverse = True)
2  >>> print('##  Univ \tgtch  gres  gcit  gint  gind  overall')
3  >>> print('-------------------------------------------------')
4  >>> i = 1
5  >>> for it in theScores:
6  ...     x = it[1]
7  ...     xScore = it[0]
8  ...     print('%2d: %s' % (i,x), end=' \t')
9  ...     for g in t.criteria:
10 ...         print('%.1f ' % (t.evaluation[g][x]),end=' ')
11 ...     print(' %.1f' % xScore)
12 ...     i += 1
13
14  ##  Univ    gtch  gres  gcit  gint  gind  overall
15  -------------------------------------------------
16   1: ethz    89.2  97.3  97.1  93.6  64.1   92.9
17   2: calt    91.5  96.0  99.8  59.1  85.9   92.4
```

(continues on next page)

[40] https://www.timeshighereducation.com/world-university-rankings/methodology-world-university-rankings-2016-20

[38] The author's own Computer Science Dept at the *University of Luxembourg* was ranked on position 63 with an overall score of 58.0.

(continued from previous page)

```
 3: oxf      94.0  92.0  98.8  93.6  44.3   92.2
 4: mit      87.3  95.4  99.4  73.9  87.5   92.1
 5: git      87.2  99.7  91.3  63.0  79.5   89.9
 6: cmel     88.1  92.3  99.4  58.9  71.1   89.4
 7: icl      90.1  87.5  95.1  94.3  49.9   89.0
 8: epfl     86.3  91.6  94.8  97.2  42.7   88.9
 9: tum      87.6  95.1  87.9  52.9  95.1   87.7
10: sing     89.9  91.3  83.0  95.3  50.6   86.9
11: cou      81.6  94.1  99.7  55.7  45.7   86.6
12: ucl      85.5  90.3  87.6  94.7  42.4   86.1
13: wash     84.4  88.7  99.3  57.4  41.2   85.6
14: hkst     74.3  92.0  96.2  84.4  55.8   85.5
15: ntu      76.6  87.7  90.4  92.9  86.9   85.5
16: ued      85.7  85.3  89.7  95.0  38.8   85.0
17: unt      79.9  84.4  99.6  77.6  38.4   84.4
18: uiu      85.0  83.1  99.2  51.4  42.2   83.7
19: mcp      79.7  89.3  94.6  29.8  51.7   81.5
20: cbu      81.2  78.5  94.7  66.9  45.7   81.3
21: tsu      88.1  90.2  76.7  27.1  85.9   80.9
22: csd      75.2  81.6  99.8  39.7  59.8   80.5
23: uwa      75.3  82.6  91.3  72.9  41.5   80.0
24: nyu      71.1  77.4  99.4  78.0  39.8   79.7
25: uta      72.6  85.3  99.6  31.6  49.7   79.6
26: kit      73.8  85.5  84.4  41.3  76.8   77.9
27: bju      83.0  85.3  70.1  30.7  99.4   77.0
28: csb      65.6  70.9  94.8  72.9  74.9   76.2
29: rwth     77.8  85.0  70.8  43.7  89.4   76.1
30: hku      77.0  73.0  77.0  96.8  39.5   75.4
31: pud      76.9  84.8  70.8  58.1  56.7   75.2
32: kist     79.4  88.2  64.2  31.6  92.8   74.9
33: kcl      45.5  94.6  86.3  95.1  38.3   74.8
34: chku     64.1  69.3  94.7  75.6  49.9   74.2
35: epfr     81.7  60.6  78.1  85.3  62.9   73.7
36: dut      64.1  78.3  76.3  69.8  90.1   73.4
37: tub      66.2  82.4  71.0  55.4  99.9   73.3
38: utj      92.0  91.7  48.7  25.8  49.6   72.9
39: cir      68.8  64.6  93.0  65.1  40.4   72.5
40: ntw      81.5  79.8  66.6  25.5  67.6   72.0
41: anu      47.2  73.0  92.2  90.0  48.1   70.6
42: rcu      64.1  53.8  99.4  63.7  46.1   69.8
43: mel      56.1  70.2  83.7  83.3  50.4   69.7
44: lms      81.5  68.1  61.0  31.1  87.8   68.4
45: ens      71.8  40.9  98.7  69.6  43.5   68.3
46: wtu      61.8  73.5  73.7  51.9  62.2   67.9
47: tech     54.9  71.0  85.1  51.7  40.1   67.1
48: bro      58.5  54.9  96.8  52.3  38.6   66.5
```

(continues on next page)

(continued from previous page)

```
 49: man     63.5  71.9  62.9  84.1  42.1   66.3
 50: zhej    73.5  70.4  60.7  22.6  75.7   65.3
 51: frei    54.2  51.6  89.5  49.7  99.9   65.1
 52: unsw    60.2  58.2  70.5  87.0  44.3   63.6
 53: kuj     75.4  72.8  49.5  28.3  51.4   62.8
 54: sou     48.2  60.7  75.5  87.4  43.2   62.1
 55: shJi    66.9  68.3  62.4  22.8  38.5   61.4
 56: itmo    58.0  32.0  98.7  39.2  68.7   60.5
 57: kul     35.2  55.8  92.0  46.0  88.3   60.5
 58: glas    35.2  52.5  91.2  85.8  39.2   59.8
 59: utw     38.2  52.8  87.0  69.0  60.0   59.4
 60: stut    54.2  60.6  61.1  36.3  97.8   58.9
 61: naji    51.4  76.9  48.8  39.7  74.4   58.6
 62: tud     46.6  53.6  75.9  53.7  66.5   58.3
 63: unlu    35.2  44.2  87.4  99.7  54.1   58.0
 64: qut     45.5  42.6  82.8  75.2  63.0   58.0
 65: hkpu    46.8  36.5  91.4  73.2  41.5   57.7
 66: albt    39.2  53.3  69.9  91.9  75.4   57.6
 67: mil     46.4  64.3  69.2  44.1  38.5   57.5
 68: hels    48.8  49.6  80.4  50.6  39.5   57.4
 69: cihk    42.4  44.9  80.1  76.2  67.9   57.3
 70: tlavu   34.1  57.2  89.0  45.3  38.6   57.2
 71: indis   56.9  76.1  49.3  20.1  41.5   57.0
 72: ariz    28.4  61.8  84.3  59.3  42.0   56.8
 73: kth     44.8  42.0  83.6  71.6  39.2   56.4
 74: humb    48.4  31.3  94.7  41.5  45.5   55.3
 75: eind    32.4  48.4  81.5  72.2  45.8   54.4
```

It is important to notice that a ranking by weighted average scores requires *commensurable ranking criteria* of *precise decimal significance* and on wich a *precise decimal performance grading* is given. It is very unlikely that the THE 2016 performance assessments indeed verify these conditions. This tutorial shows how to relax these methodological requirements -precise commensurable criteria and numerical assessments- by following instead an epistemic bipolar-valued logic based ranking methodology.

Ranking with multiple incommensurable criteria of ordinal significance

Let us, first, have a critical look at the THE performance criteria.

```
>>> t.showHTMLCriteria(Sorted=False)
```

the_cs_2016: Family of Criteria

#	Identifyer	Name	Comment	Weight	Scale			Thresholds (ax + b)		
					direction	min	max	indifference	preference	veto
1	**gtch**	Teaching	The learning environment	30.00	max	0.00	100.00	0.00x + 2.50	0.00x + 5.00	0.00x + 60.00
2	**gres**	Research	Volume, income and reputation	30.00	max	0.00	100.00	0.00x + 2.50	0.00x + 5.00	0.00x + 60.00
3	**gcit**	Citations	Research influence	27.50	max	0.00	100.00	0.00x + 2.50	0.00x + 5.00	0.00x + 60.00
4	**gint**	International outlook	In staff, students and research	7.50	max	0.00	100.00	0.00x + 2.50	0.00x + 5.00	
5	**gind**	Industry income	Innovation	5.00	max	0.00	100.00	0.00x + 2.50	0.00x + 5.00	

Fig. 3.7: The THE ranking criteria

Considering a very likely imprecision of the performance grading procedure, followed by some potential violation of uniform distributed quantile classes, we assume here that a performance quantile difference of up to **abs(2.5)%** is **insignificant**, whereas a difference of **abs(5)%** warrants a **clearly better**, resp. **clearly less good**, performance. With quantiles 94%, resp. 87.3%, *Oxford*'s CS teaching environment, for instance, is thus clearly better evaluated than that of the *MIT* (see Listing 3.12 Lines 27-28). We shall furthermore assume that a **considerable** performance quantile difference of **abs(60)%**, observed on the three major ranking criteria: *Teaching*, *Research* and *Citations*, will trigger a **veto**, respectively a **counter-veto** against a *pairwise outranking*, respectively a *pairwise outranked* situation [BIS-2013].

The effect of these performance discrimination thresholds on the preference modelling may be inspected as follows.

Listing 3.14: Inspecting the performance discrimination thresholds

```
>>> t.showCriteria()
 *----  criteria -----*
 gtch 'Teaching'
   Scale = (Decimal('0.00'), Decimal('100.00'))
   Weight = 0.300
   Threshold ind : 2.50 + 0.00x ;   percentile:  8.07
   Threshold pref : 5.00 + 0.00x ;  percentile: 15.75
   Threshold veto : 60.00 + 0.00x ; percentile: 99.75
 gres 'Research'
   Scale = (Decimal('0.00'), Decimal('100.00'))
   Weight = 0.300
   Threshold ind : 2.50 + 0.00x ;   percentile:  7.86
   Threshold pref : 5.00 + 0.00x ;  percentile: 16.14
   Threshold veto : 60.00 + 0.00x ; percentile: 99.21
 gcit 'Citations'
   Scale = (Decimal('0.00'), Decimal('100.00'))
   Weight = 0.275
```

(continues on next page)

(continued from previous page)

```
    Threshold ind : 2.50 + 0.00x ;   percentile:  11.82
    Threshold pref : 5.00 + 0.00x ;  percentile:  22.99
    Threshold veto : 60.00 + 0.00x ; percentile: 100.00
  gint 'International outlook'
    Scale = (Decimal('0.00'), Decimal('100.00'))
    Weight = 0.075
    Threshold ind : 2.50 + 0.00x ;  percentile:  6.45
    Threshold pref : 5.00 + 0.00x ; percentile: 11.75
  gind 'Industry income'
    Scale = (Decimal('0.00'), Decimal('100.00'))
    Weight = 0.050
    Threshold ind : 2.50 + 0.00x ;  percentile: 11.82
    Threshold pref : 5.00 + 0.00x ; percentile: 21.51
```

Between 6% and 12% of the observed quantile differences are, thus, considered to be *insignificant*. Similarly, between 77% and 88% are considered to be *significant*. Less than 1% correspond to *considerable* quantile differences on both the *Teaching* and *Research* criteria; actually triggering an epistemic *polarisation* effect [BIS-2013].

Beside the likely imprecise performance discrimination, the **precise decimal** significance weights, as allocated by the THE authors to the five ranking criteria (see Fig. 3.7 Column *Weight*) are, as well, quite **questionable**. Significance weights may carry usually hidden strategies for rendering the performance evaluations commensurable in view of a numerical computation of the overall ranking scores. The eventual ranking result is thus as much depending on the precise values of the given criteria significance weights as, vice versa, the given precise significance weights are depending on the subjectively expected and accepted ranking results[42]. We will therefore drop such precise weights and, instead, only require a corresponding criteria significance preorder: $gtch = gres > gcit > gint > gind$. *Teaching environment* and *Research volume and reputation* are equally considered most important, followed by *Research influence*. Than comes *International outlook in staff, students and research* and, least important finally, *Industry income and innovation*.

Both these working hypotheses: performance *discrimitation* thresholds and solely *ordinal* criteria significance, give us way to a ranking methodology based on **robust pairwise outranking** situations [BIS-2004b]:

- We say that CS Dept x **robustly outranks** CS Dept y when x positively outranks y with **all** significance weight vectors that are **compatible** with the *significance preorder*: $gtch = gres > gcit > gint > gind$;
- We say that CS Dept x is **robustly outranked** by CS Dept y when x is positively outranked by y with **all** significance weight vectors that are **compatible** with the *significance preorder*: $gtch = gres > gcit > gint > gind$;
- Otherwise, CS Depts x and y are considered to be **incomparable**.

A corresponding digraph constructor is provided by the `RobustOutrankingDigraph` class.

[42] In a social choice context, this potential double bind between voting profiles and election result, corresponds to voting manipulation strategies.

Listing 3.15: Computing the robust outranking digraph

```
1  >>> from outrankingDigraphs import RobustOutrankingDigraph
2  >>> rdg = RobustOutrankingDigraph(t)
3  >>> rdg
4   *------- Object instance description ------*
5   Instance class       : RobustOutrankingDigraph
6   Instance name        : robust_the_cs_2016
7   # Actions            : 75
8   # Criteria           : 5
9   Size                 : 2993
10  Determinateness (%)  : 78.16
11  Valuation domain     : [-1.00;1.00]
12 >>> rdg.computeIncomparabilityDegree(Comments=True)
13  Incomparability degree (%) of digraph <robust_the_cs_2016>:
14   #links x<->y y: 2775, #incomparable: 102, #comparable: 2673
15   (#incomparable/#links) =  0.037
16 >>> rdg.computeTransitivityDegree(Comments=True)
17  Transitivity degree of digraph <robust_the_cs_2016>:
18   #triples x>y>z: 405150, #closed: 218489, #open: 186661
19   (#closed/#triples) =  0.539
20 >>> rdg.computeSymmetryDegree(Comments=True)
21  Symmetry degree (%) of digraph <robust_the_cs_2016>:
22   #arcs x>y: 2673, #symmetric: 320, #asymmetric: 2353
23   (#symmetric/#arcs) =  0.12
```

In the resulting digraph instance *rdg* (see Listing 3.15 Line 8), we observe 2993 such **robust pairwise outranking** situations validated with a mean significance of 78% (Line 9). Unfortunately, in our case here, they do not deliver any complete linear ranking relation. The robust outranking digraph *rdg* contains in fact 102 incomparability situations (3.7%, Line 13); nearly half of its transitive closure is missing (46.1%, Line 18) and 12% of the positive outranking situations correspond in fact to symmetric *indifference* situations (Line 22).

Worse even, the digraph *rdg* admits furthermore a high number of outranking circuits.

Listing 3.16: Inspecting outranking circuits

```
>>> rdg.computeChordlessCircuits()
>>> rdg.showChordlessCircuits()
 *---- Chordless circuits ----*
 145 circuits.
    1:  ['albt', 'unlu', 'ariz', 'hels'] , credibility : 0.300
    2:  ['albt', 'tlavu', 'hels'] , credibility : 0.150
    3:  ['anu', 'man', 'itmo'] , credibility : 0.250
    4:  ['anu', 'zhej', 'rcu'] , credibility : 0.250
 ...
 ...
```

(continues on next page)

(continued from previous page)

```
 82:  ['csb', 'epfr', 'rwth'] , credibility : 0.250
 83:  ['csb', 'epfr', 'pud', 'nyu'] , credibility : 0.250
 84:  ['csd', 'kcl', 'kist'] , credibility : 0.250
...
...
142:  ['kul', 'qut', 'mil'] , credibility : 0.250
143:  ['lms', 'rcu', 'tech'] , credibility : 0.300
144:  ['mil', 'stut', 'qut'] , credibility : 0.300
145:  ['mil', 'stut', 'tud'] , credibility : 0.300
```

Among the 145 detected robust outranking circuits reported in Listing 3.16, we notice, for instance, two outranking circuits of length 4 (see circuits #1 and #83). Let us explore below the bipolar-valued robust outranking characteristics $r(x \succsim y)$ of the first circuit.

Listing 3.17: Showing the relation table with stability denotation

```
>>> rdg.showRelationTable(actionsSubset= ['albt','unlu','ariz','hels'],
...                       Sorted=False)

 * ---- Relation Table -----
  r/(stab)|  'albt' 'unlu' 'ariz' 'hels'
     -----|------------------------------------------------------------
   'albt' |  +1.00  +0.30  +0.00  +0.00
          |   (+4)   (+2)   (-1)   (-1)
   'unlu' |  +0.00  +1.00  +0.40  +0.00
          |   (+0)   (+4)   (+2)   (-1)
   'ariz' |  +0.00  -0.12  +1.00  +0.40
          |   (+1)   (-2)   (+4)   (+2)
   'hels' |  +0.45  +0.00  -0.03  +1.00
          |   (+2)   (+1)   (-2)   (+4)
 Valuation domain: [-1.0; 1.0]
 Stability denotation semantics:
  +4|-4 : unanimous outranking | outranked situation;
  +2|-2 : outranking | outranked situation validated
          with all potential significance weights that are
          compatible with the given significance preorder;
  +1|-1 : validated outranking | outranked situation with
          the given significance weights;
    0   : indeterminate relational situation.
```

In Listing 3.17, we may notice that the robust outranking circuit ['albt', 'unlu', 'ariz', 'hels'] will reappear with all potential criteria significance weight vectors that are compatible with given preorder: $gtch = gres > gcit > gint > gind$. Notice also the (+1|-1) marked outranking situations, like the one between 'albt' and 'ariz'. The statement that "*Arizona State University* strictly outranks *University of Alberta*" is in fact valid with the precise THE weight vector, but not with all potential weight vectors compatible with the given significance preorder. All these outranking situations are hence put into **doubt** ($r(x \succsim y) = 0.00$) and the corresponding CS Depts, like *University of Alberta* and *Arizona State University*, become **incomparable** in a *robust outranking* sense.

Showing many incomparabilities and indifferences; not being transitive and containing many robust outranking circuits; all these relational characteristics, make that no ranking algorithm, applied to digraph *rdg*, does exist that would produce a *unique* optimal linear ranking result. Methodologically, we are only left with *ranking heuristics*. In the previous tutorial on *ranking with multiple criteria* (page 72) we have seen now several potential heuristic ranking rules that may be applied to rank from a pairwise outranking digraph; yet, delivering all potentially more or less diverging results. Considering the order of digraph *rdg* (75) and the largely unequal THE criteria significance weights, we rather opt, in this tutorial, for the *NetFlows ranking rule* (page 78)[41]. Its complexity in $O(n^2)$ is indeed quite tractable and, by avoiding potential *tyranny of short majority* effects, the *NetFlows* rule specifically takes the ranking criteria significance into a more fairly balanced account.

The *NetFlows* ranking result of the CS Depts may be computed explicitly as follows.

Listing 3.18: Computing the robust *NetFlows* ranking

```
>>> nfRanking = rdg.computeNetFlowsRanking()
>>> nfRanking
 ['ethz', 'calt', 'mit',  'oxf',   'cmel', 'git',  'epfl',
  'icl',  'cou',  'tum',  'wash',  'sing', 'hkst', 'ucl',
  'uiu',  'unt',  'ued',  'ntu',   'mcp',  'csd',  'cbu',
  'uta',  'tsu',  'nyu',  'uwa',   'csb',  'kit',  'utj',
  'bju',  'kcl',  'chku', 'kist',  'rwth', 'pud',  'epfr',
  'hku',  'rcu',  'cir',  'dut',   'ens',  'ntw',  'anu',
  'tub',  'mel',  'lms',  'bro',   'frei', 'wtu',  'tech',
  'itmo', 'zhej', 'man',  'kuj',   'kul',  'unsw', 'glas',
  'utw',  'unlu', 'naji', 'sou',   'hkpu', 'qut',  'humb',
  'shJi', 'stut', 'tud',  'tlavu', 'cihk', 'albt', 'indis',
  'ariz', 'kth',  'hels', 'eind',  'mil']
```

We actually obtain a very similar ranking result as the one obtained with the THE overall scores. The same group of seven Depts: *ethz*, *calt*, *mit*, *oxf*, *cmel*, *git* and *epfl*, is top-ranked. And a same group of Depts: *tlavu*, *cihk*, *indis*, *ariz*, *kth*, *'hels*, *eind*, and *mil* appears at the end of the list.

We may print out the difference between the *overall scores* based THE ranking and our *NetFlows* ranking with the following short Python script, where we make use of an ordered Python dictionary with *net flow scores*, stored in the *rdg.netFlowsRankingDict* attribute by the previous computation.

[41] The reader might try other ranking rules, like *Copeland*'s, *Kohler*'s, *Tideman*'s rule or the iterated versions of the *NetFlows* and *Copeland*'s rule. Mind that the latter *ranking-by-choosing* rules are more complex.

Listing 3.19: Comparing the robust *NetFlows* ranking with the THE ranking

```
>>> # rdg.netFlowsRankingDict: ordered dictionary with net flow
>>> # scores stored in rdg by the computeNetFlowsRanking() method
>>> # theScores = [(xScore_1,x_1), (xScore_2,x_2),... ]
>>> # is sorted in decreasing order of xscores_i
>>> print(\
...  ' NetFlows ranking   gtch  gres  gcit  gint  gind   THE ranking')

>>> for i in range(75):
...     x = nfRanking[i]
...     xScore = rdg.netFlowsRankingDict[x]['netFlow']
...     thexScore,thex = theScores[i]
...     print('%2d: %s  (%.2f) ' % (i+1,x,xScore), end=' \t')
...     for g in rdg.criteria:
...         print('%.1f ' % (t.evaluation[g][x]),end=' ')
...     print('  %s  (%.2f)' % (thex,thexScore) )

  NetFlows ranking   gtch  gres  gcit  gint  gind   THE ranking
  1: ethz (116.95)   89.2  97.3  97.1  93.6  64.1   ethz (92.88)
  2: calt (116.15)   91.5  96.0  99.8  59.1  85.9   calt (92.42)
  3: mit (112.72)    87.3  95.4  99.4  73.9  87.5   oxf (92.20)
  4: oxf (112.00)    94.0  92.0  98.8  93.6  44.3   mit (92.06)
  5: cmel (101.60)   88.1  92.3  99.4  58.9  71.1   git (89.88)
  6: git (93.40)     87.2  99.7  91.3  63.0  79.5   cmel (89.43)
  7: epfl (90.88)    86.3  91.6  94.8  97.2  42.7   icl (89.00)
  8: icl (90.62)     90.1  87.5  95.1  94.3  49.9   epfl (88.86)
  9: cou (84.60)     81.6  94.1  99.7  55.7  45.7   tum (87.70)
 10: tum (80.42)     87.6  95.1  87.9  52.9  95.1   sing (86.86)
 11: wash (76.28)    84.4  88.7  99.3  57.4  41.2   cou (86.59)
 12: sing (73.05)    89.9  91.3  83.0  95.3  50.6   ucl (86.05)
 13: hkst (71.05)    74.3  92.0  96.2  84.4  55.8   wash (85.60)
 14: ucl (66.78)     85.5  90.3  87.6  94.7  42.4   hkst (85.47)
 15: uiu (64.80)     85.0  83.1  99.2  51.4  42.2   ntu (85.46)
 16: unt (62.65)     79.9  84.4  99.6  77.6  38.4   ued (85.03)
 17: ued (58.67)     85.7  85.3  89.7  95.0  38.8   unt (84.42)
 18: ntu (57.88)     76.6  87.7  90.4  92.9  86.9   uiu (83.67)
 19: mcp (54.08)     79.7  89.3  94.6  29.8  51.7   mcp (81.53)
 20: csd (46.62)     75.2  81.6  99.8  39.7  59.8   cbu (81.25)
 21: cbu (44.27)     81.2  78.5  94.7  66.9  45.7   tsu (80.91)
 22: uta (43.27)     72.6  85.3  99.6  31.6  49.7   csd (80.45)
 23: tsu (42.42)     88.1  90.2  76.7  27.1  85.9   uwa (80.02)
 24: nyu (35.30)     71.1  77.4  99.4  78.0  39.8   nyu (79.72)
 25: uwa (28.88)     75.3  82.6  91.3  72.9  41.5   uta (79.61)
 26: csb (18.18)     65.6  70.9  94.8  72.9  74.9   kit (77.94)
 27: kit (16.32)     73.8  85.5  84.4  41.3  76.8   bju (77.04)
```

(continues on next page)

(continued from previous page)

28: utj (15.95)	92.0	91.7	48.7	25.8	49.6	csb (76.23)
29: bju (15.45)	83.0	85.3	70.1	30.7	99.4	rwth (76.06)
30: kcl (11.95)	45.5	94.6	86.3	95.1	38.3	hku (75.41)
31: chku (9.43)	64.1	69.3	94.7	75.6	49.9	pud (75.17)
32: kist (7.30)	79.4	88.2	64.2	31.6	92.8	kist (74.94)
33: rwth (5.00)	77.8	85.0	70.8	43.7	89.4	kcl (74.81)
34: pud (2.40)	76.9	84.8	70.8	58.1	56.7	chku (74.23)
35: epfr (-1.70)	81.7	60.6	78.1	85.3	62.9	epfr (73.71)
36: hku (-3.83)	77.0	73.0	77.0	96.8	39.5	dut (73.44)
37: rcu (-6.38)	64.1	53.8	99.4	63.7	46.1	tub (73.25)
38: cir (-8.20)	68.8	64.6	93.0	65.1	40.4	utj (72.92)
39: dut (-8.85)	64.1	78.3	76.3	69.8	90.1	cir (72.50)
40: ens (-8.97)	71.8	40.9	98.7	69.6	43.5	ntw (72.00)
41: ntw (-11.15)	81.5	79.8	66.6	25.5	67.6	anu (70.57)
42: anu (-11.50)	47.2	73.0	92.2	90.0	48.1	rcu (69.79)
43: tub (-12.20)	66.2	82.4	71.0	55.4	99.9	mel (69.67)
44: mel (-23.98)	56.1	70.2	83.7	83.3	50.4	lms (68.38)
45: lms (-25.43)	81.5	68.1	61.0	31.1	87.8	ens (68.35)
46: bro (-27.18)	58.5	54.9	96.8	52.3	38.6	wtu (67.86)
47: frei (-34.42)	54.2	51.6	89.5	49.7	99.9	tech (67.06)
48: wtu (-35.05)	61.8	73.5	73.7	51.9	62.2	bro (66.49)
49: tech (-37.95)	54.9	71.0	85.1	51.7	40.1	man (66.33)
50: itmo (-38.50)	58.0	32.0	98.7	39.2	68.7	zhej (65.34)
51: zhej (-43.70)	73.5	70.4	60.7	22.6	75.7	frei (65.08)
52: man (-44.83)	63.5	71.9	62.9	84.1	42.1	unsw (63.65)
53: kuj (-47.40)	75.4	72.8	49.5	28.3	51.4	kuj (62.77)
54: kul (-49.98)	35.2	55.8	92.0	46.0	88.3	sou (62.15)
55: unsw (-54.88)	60.2	58.2	70.5	87.0	44.3	shJi (61.35)
56: glas (-56.98)	35.2	52.5	91.2	85.8	39.2	itmo (60.52)
57: utw (-59.27)	38.2	52.8	87.0	69.0	60.0	kul (60.47)
58: unlu (-60.08)	35.2	44.2	87.4	99.7	54.1	glas (59.78)
59: naji (-60.52)	51.4	76.9	48.8	39.7	74.4	utw (59.40)
60: sou (-60.83)	48.2	60.7	75.5	87.4	43.2	stut (58.85)
61: hkpu (-62.05)	46.8	36.5	91.4	73.2	41.5	naji (58.61)
62: qut (-66.17)	45.5	42.6	82.8	75.2	63.0	tud (58.28)
63: humb (-68.10)	48.4	31.3	94.7	41.5	45.5	unlu (58.04)
64: shJi (-69.72)	66.9	68.3	62.4	22.8	38.5	qut (57.99)
65: stut (-69.90)	54.2	60.6	61.1	36.3	97.8	hkpu (57.69)
66: tud (-70.83)	46.6	53.6	75.9	53.7	66.5	albt (57.63)
67: tlavu (-71.50)	34.1	57.2	89.0	45.3	38.6	mil (57.47)
68: cihk (-72.20)	42.4	44.9	80.1	76.2	67.9	hels (57.40)
69: albt (-72.33)	39.2	53.3	69.9	91.9	75.4	cihk (57.33)
70: indis (-72.53)	56.9	76.1	49.3	20.1	41.5	tlavu (57.19)
71: ariz (-75.10)	28.4	61.8	84.3	59.3	42.0	indis (57.04)
72: kth (-77.10)	44.8	42.0	83.6	71.6	39.2	ariz (56.79)
73: hels (-79.55)	48.8	49.6	80.4	50.6	39.5	kth (56.36)

(continues on next page)

(continued from previous page)

```
91  74: eind (-82.85)   32.4  48.4  81.5  72.2  45.8   humb (55.34)
92  75: mil (-83.67)    46.4  64.3  69.2  44.1  38.5   eind (54.36)
```

The first inversion we observe in Listing 3.19 (Lines 20-21) concerns *Oxford University* and the *MIT*, switching positions 3 and 4. Most inversions are similarly short and concern only switching very close positions in either way. There are some slightly more important inversions concerning, for instance, the *Hong Kong University* CS Dept, ranked into position 30 in the THE ranking and here in the position 36 (Line 53). The opposite situation may also happen; the *Berlin Humboldt University* CS Dept, occupying the 74th position in the THE ranking, advances in the *NetFlows* ranking to position 63 (Line 80).

In our bipolar-valued epistemic framework, the *NetFlows* score of any CS Dept x (see Listing 3.19) corresponds to the criteria significance support for the logical statement (x is *first*-ranked). Formally

$$\mathrm{r}(x \text{ is } \textit{first}\text{-ranked}) = \sum_{y \neq x} r\big((x \succsim y) + (y \not\succsim x)\big) = \sum_{y \neq x} \big(r(x \succsim y) - r(y \succsim x)\big)$$

Using the robust outranking characteristics of digraph *rdg*, we may thus explicitly compute, for instance, *ETH Zürich*'s score, denoted *nfx* below.

```
>>> x = 'ethz'
>>> nfx = Decimal('0')
>>> for y in rdg.actions:
...     if x != y:
...         nfx += (rdg.relation[x][y] - rdg.relation[y][x])
```

```
>>> print(x, nfx)
 ethz 116.950
```

In Listing 3.19 (Line 18), we may now verify that *ETH Zürich* obtains indeed the highest *NetFlows* score, and gives, hence the **most credible** *first*-ranked CS Dept of the 75 potential candidates.

How may we now convince the reader, that our pairwise outranking based ranking result here appears more objective and trustworthy, than the classic value theory based THE ranking by overall scores?

How to judge the quality of a ranking result?

In a multiple criteria based ranking problem, inspecting pairwise marginal performance differences may give objectivity to global preferential statements. That a CS Dept x convincingly outranks Dept y may thus conveniently be checked. The *ETH Zürich* CS Dept is, for instance, first ranked before *Caltech*'s Dept in both previous rankings. Lest us check the preferential reasons.

Listing 3.20: Comparing pairwise criteria performances

```
1  >>> rdg.showPairwiseOutrankings('ethz','calt')
2   *------------  pairwise comparisons ----*
3   Valuation in range: -100.00 to +100.00
4   Comparing actions : (ethz, calt)
5   crit. wght.  g(x)  g(y)    diff      | ind   pref    r()      |
6   -------------------------------      ------------------------
7   gcit   27.50  97.10  99.80  -2.70    | 2.50  5.00   +0.00     |
8   gind   5.00  64.10  85.90  -21.80    | 2.50  5.00   -5.00     |
9   gint   7.50  93.60  59.10  +34.50    | 2.50  5.00   +7.50     |
10  gres   30.00  97.30  96.00  +1.30    | 2.50  5.00   +30.00    |
11  gtch   30.00  89.20  91.50  -2.30    | 2.50  5.00   +30.00    |
12                                          r(x >= y): +62.50
13  crit. wght.  g(y)  g(x)    diff      | ind   pref    r()      |
14  -------------------------------      ------------------------
15  gcit   27.50  99.80  97.10  +2.70    | 2.50  5.00   +27.50    |
16  gind   5.00  85.90  64.10  +21.80    | 2.50  5.00    +5.00    |
17  gint   7.50  59.10  93.60  -34.50    | 2.50  5.00    -7.50    |
18  gres   30.00  96.00  97.30  -1.30    | 2.50  5.00   +30.00    |
19  gtch   30.00  91.50  89.20  +2.30    | 2.50  5.00   +30.00    |
20                                          r(y >= x): +85.00
```

A significant positive performance difference (+34.50), concerning the *International outlook* criterion (of 7,5% significance), may be observed in favour of the *ETH Zürich* Dept (Line 9 above). Similarly, a significant positive performance difference (+21.80), concerning the *Industry income* criterion (of 5% significance), may be observed, this time, in favour of the *Caltech* Dept. The former, larger positive, performance difference, observed on a more significant criterion, gives so far a first convincing argument of 12.5% significance for putting *ETH Zürich* first, before *Caltech*. Yet, the slightly positive performance difference (+2.70) between *Caltech* and *ETH Zürich* on the *Citations* criterion (of 27.5% significance) confirms an *at least as good as* situation in favour of the *Caltech* Dept.

The inverse negative performance difference (-2.70), however, is neither *significant* ($<$ -5.00), nor insignificant ($>$ -2.50), and does hence **neither confirm nor infirm** a *not at least as good as* situation in disfavour of *ETH Zürich*. We observe here a convincing argument of 27.5% significance for putting *Caltech* first, before *ETH Zürich*.

Notice finally, that, on the *Teaching* and *Research* criteria of total significance 60%, both Depts do, with performance differences $<$ abs(2.50), one as well as the other. As these two major performance criteria necessarily support together always the highest significance with the imposed significance weight preorder: $gtch = gres > gcit > gint > gind$, both outranking situations get in fact globally confirmed at stability level $+2$ (see the advanced topic on stable outrankings with multiple criteria of ordinal significance).

We may well illustrate all such *stable outranking* situations with a browser view of the corresponding robust relation map using our *NetFlows* ranking.

```
>>> rdg.showHTMLRelationMap(tableTitle='Robust Outranking Map',
...                         rankingRule='NetFlows')
```

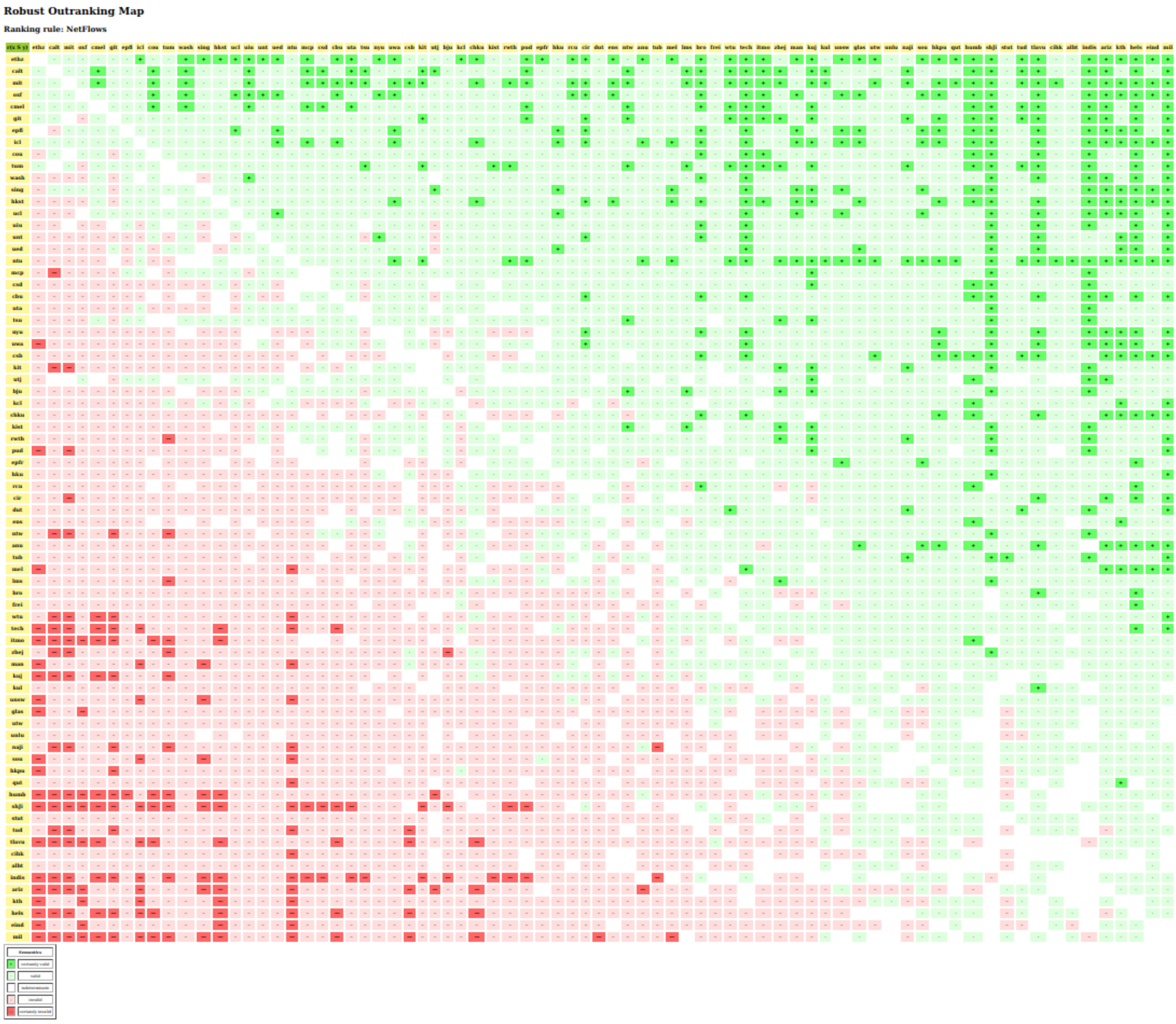


Fig. 3.8: Relation map of the robust outranking relation

In Fig. 3.8, **dark green**, resp. **light green** marked positions show *certainly*, resp. *positively* valid **outranking** situations, whereas **dark red**, resp. **light red** marked positions show *certainly*, respectively *positively* valid **outranked** situations. In the left upper corner we may verify that the five top-ranked Depts ([‘ethz’, ‘calt’, ‘oxf’, ‘mit’, ‘cmel’]) are indeed mutually outranking each other and thus are to be considered all *indifferent*. They are even robust *Condorcet* winners, i.e positively outranking all other Depts. We may by the way notice that no certainly valid outranking (dark green) and no certainly valid outranked situations (dark red) appear **below**, resp. **above** the principal diagonal; none of these are hence violated by our *netFlows* ranking.

The non reflexive **white** positions in the relation map, mark outranking or outranked situations that are **not robust** with respect to the given significance weight preorder. They are, hence, put into doubt and set to the *indeterminate* characteristic value **0**.

By measuring the **ordinal correlation** with the underlying pairwise *global* and *marginal* robust outranking situations, the **quality** of the robust *netFlows* ranking result may be formally evaluated[Page 146, 27].

Listing 3.21: Measuring the quality of the *NetFlows* ranking result

```
1 >>> corrnf = rdg.computeRankingCorrelation(nfRanking)
2 >>> rdg.showCorrelation(corrnf)
3  Correlation indexes:
4   Crisp ordinal correlation  : +0.901
5   Epistemic determination    :  0.563
6   Bipolar-valued equivalence : +0.507
```

In Listing 3.21 (Line 4), we may notice that the *NetFlows* ranking result is indeed highly ordinally correlated (+0.901, in *Kendall*'s index *tau* sense) with the pairwise global robust outranking relation. Their bipolar-valued *relational equivalence* value (+0.51, Line 6) indicates a more than 75% criteria significance support.

We may as well check how the *netFlows* ranking rule is actually balancing the five ranking criteria.

```
1 >>> rdg.showRankingConsensusQuality(nfRanking)
2  Criterion (weight): correlation
3  -------------------------------
4    gtch (0.300): +0.660
5    gres (0.300): +0.638
6    gcit (0.275): +0.370
7    gint (0.075): +0.155
8    gind (0.050): +0.101
9   Summary:
10  Weighted mean marginal correlation (a): +0.508
11  Standard deviation (b)                : +0.187
12  Ranking fairness (a)-(b)              : +0.321
```

The correlations with the marginal performance criterion rankings are nearly respecting the given significance weights preorder: $gtch$ ~ $gres > gcit > gint > gind$ (see above Lines 4-8). The mean *marginal correlation* is quite high (+0.51). Coupled with a low standard deviation (0.187), we obtain a rather fairly balanced ranking result (Lines 10-12).

We may also inspect the mutual correlation indexes observed between the marginal criterion robust outranking relations.

```
>>> rdg.showCriteriaCorrelationTable()
 Criteria ordinal correlation index
      |  gcit    gind    gint    gres    gtch
 -----|------------------------------------------
 gcit | +1.00   -0.11   +0.24   +0.13   +0.17
 gind |         +1.00   -0.18   +0.15   +0.15
 gint |                 +1.00   +0.04   -0.00
```

(continues on next page)

(continued from previous page)

```
8  gres |                         +1.00   +0.67
9  gtch |                                 +1.00
```

Slightly contradictory (-0.11) appear the *Citations* and *Industrial income* criteria (Line 5 Column 3). Due perhaps to potential confidentiality clauses, it seams not always possible to publish industrially relevant research results in highly ranked journals. However, criteria *Citations* and *International outlook* show a slightly positive correlation (+0.24, Column 4), whereas the *International outlook* criterion shows no apparent correlation with both the major *Teaching* and *Research* criteria. The latter are however highly correlated (+0.67. Line 9 Column 6).

A *Principal Component Analysis* may well illustrate the previous findings.

```
>>> rdg.export3DplotOfCriteriaCorrelation(graphType='png')
```

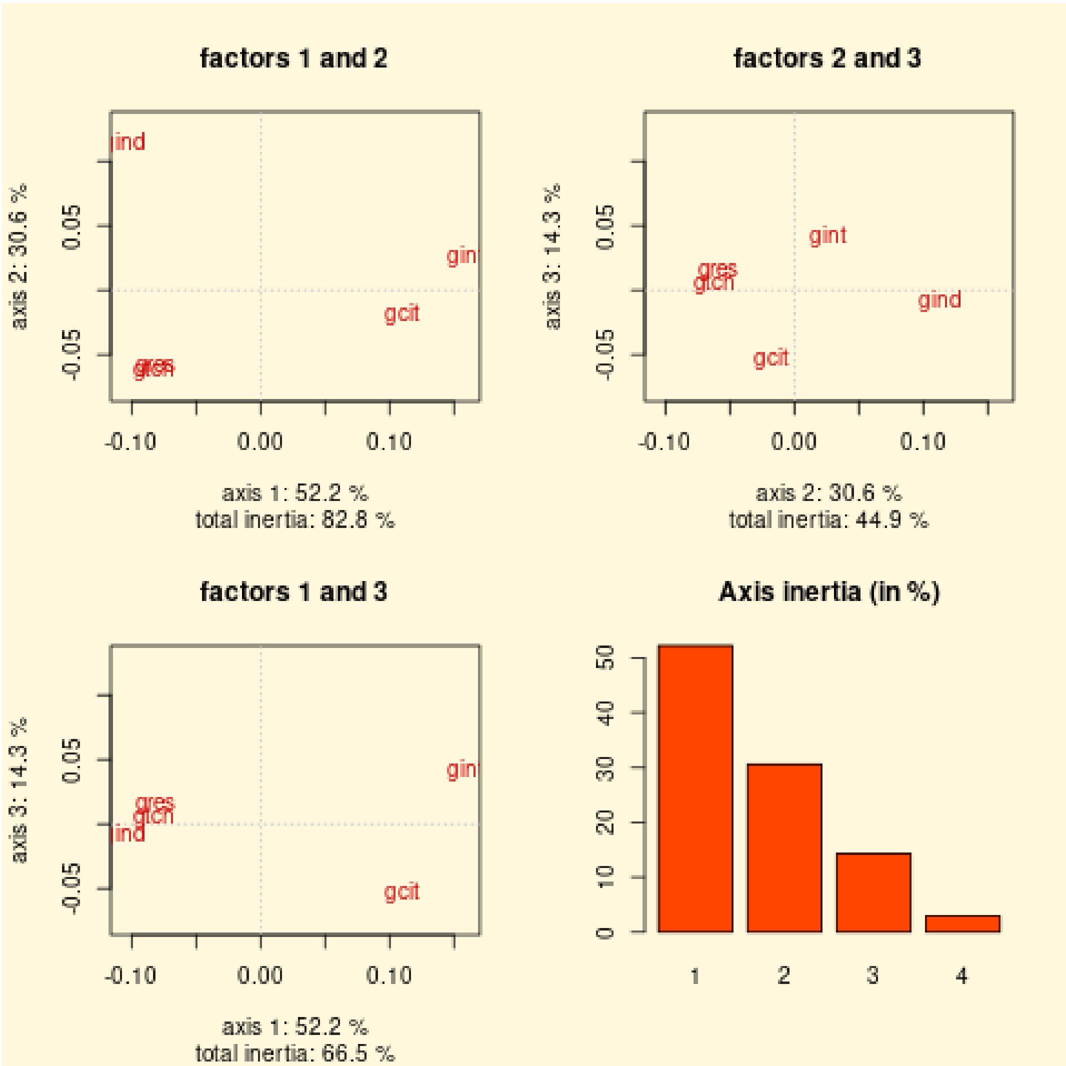


Fig. 3.9: 3D PCA plot of the pairwise criteria correlation table

In Fig. 3.9 (factors 1 and 2 plot) we may notice, first, that more than 80% of the total variance of the previous correlation table is explained by the apparent opposition between the marginal outrankings of criteria: *Teaching*, *Research* & *Industry income* on the left side, and the marginal outrankings of criteria: *Citations* & *international outlook* on the right side. Notice also in the left lower corner the nearly identical positions of the marginal outrankings of the major *Teaching* & *Research* criteria. In the factors 2 and 3 plot, about 30% of the total variance is captured by the opposition between the marginal outrankings of the *Teaching* & *Research* criteria and the marginal outrankings of the *Industrial income* criterion. Finally, in the factors 1 and 3 plot, nearly 15% of the total variance is explained by the opposition between the marginal outrankings of the *International outlook* criterion and the marginal outrankings of the *Citations* criterion.

It may, finally, be interesting to assess, similarly, the ordinal correlation of the THE overall scores based ranking with respect to our robust outranking situations.

Listing 3.22: Computing the ordinal quality of the THE ranking

```
1  >>> # theScores = [(xScore_1,x_1), (xScore_2,x_2),... ]
2  >>> # is sorted in decreasing order of xscores
3  >>> theRanking = [item[1] for item in theScores]
4  >>> corrthe = rdg.computeRankingCorrelation(theRanking)
5  >>> rdg.showCorrelation(corrthe)
6   Correlation indexes:
7    Crisp ordinal correlation  : +0.907
8    Epistemic determination    :  0.563
9    Bipolar-valued equivalence : +0.511
10 >>> rdg.showRankingConsensusQuality(theRanking)
11  Criterion (weight): correlation
12  -------------------------------
13   gtch (0.300): +0.683
14   gres (0.300): +0.670
15   gcit (0.275): +0.319
16   gint (0.075): +0.161
17   gind (0.050): +0.106
18  Summary:
19   Weighted mean marginal correlation (a): +0.511
20   Standard deviation (b)                : +0.210
21   Ranking fairness (a)-(b)              : +0.302
```

The THE ranking result is similarly correlated (+0.907, Line 7) with the pairwise global robust outranking relation. By its overall weighted scoring rule, the THE ranking induces marginal criterion correlations that are naturally compatible with the given significance weight preorder (Lines 13-17). Notice that the mean marginal correlation is of a similar value (+0.51, Line 19) as the *netFlows* ranking's. Yet, its standard deviation is higher, which leads to a slightly less fair balancing of the three major ranking criteria.

To conclude, let us emphasize, that, without any commensurability hypothesis and by taking, furthermore, into account, first, the always present more or less imprecision of any performance grading and, secondly, solely ordinal criteria significance weights, we may obtain here with our robust outranking approach a very similar ranking result with more or less a same, when not better, preference modelling quality. A convincing heatmap view of the 25 first-ranked Institutions may be generated in the default system browser with following command.

```
>>> rdg.showHTMLPerformanceHeatmap(
...       WithActionNames=True,
...       outrankingModel='this',
...       rankingRule='NetFlows',
...       ndigits=1,
...       Correlations=True,
...       fromIndex=0,toIndex=25)
```

Heatmap of Performance Tableau 'robust_the_cs_2016'

criteria	gtch	gres	gcit	gint	gind
weights	+30.00	+30.00	+27.50	+7.50	+5.00
tau(*)	+0.66	+0.64	+0.37	+0.15	+0.10
Swiss Federal Institute of Technology Zürich (ethz)	89.2	97.3	97.1	93.6	64.1
California Institute of Technology (calt)	91.5	96.0	99.8	59.1	85.9
Massachusetts Institute of Technology (mit)	87.3	95.4	99.4	73.9	87.5
University of Oxford (oxf)	94.0	92.0	98.8	93.6	44.3
Carnegie Mellon University (cmel)	88.1	92.3	99.4	58.9	71.1
Georgia Institute of Technology (git)	87.2	99.7	91.3	63.0	79.5
Swiss Federal Institute of Technology Lausanne (epfl)	86.3	91.6	94.8	97.2	42.7
Imperial College London (icl)	90.1	87.5	95.1	94.3	49.9
Cornell University (cou)	81.6	94.1	99.7	55.7	45.7
Technical University of München (tum)	87.6	95.1	87.9	52.9	95.1
University of Washington (wash)	84.4	88.7	99.3	57.4	41.2
National University of Singapore (sing)	89.9	91.3	83.0	95.3	50.6
Hong Kong University of Science and Technology (hkst)	74.3	92.0	96.2	84.4	55.8
University College London (ucl)	85.5	90.3	87.6	94.7	42.4
University of Illinois at Urbana-Champagne (uiu)	85.0	83.1	99.2	51.4	42.2
University of Toronto (unt)	79.9	84.4	99.6	77.6	38.4
University of Edinburgh (ued)	85.7	85.3	89.7	95.0	38.8
Nanyang Technological University of Singapore (ntu)	76.6	87.7	90.4	92.9	86.9
University of Maryland College Park (mcp)	79.7	89.3	94.6	29.8	51.7
University Of California at San Diego (csd)	75.2	81.6	99.8	39.7	59.8
Columbia University (cbu)	81.2	78.5	94.7	66.9	45.7
University of Texas at Austin (uta)	72.6	85.3	99.6	31.6	49.7
Tsinghua University (tsu)	88.1	90.2	76.7	27.1	85.9
New York University (nyu)	71.1	77.4	99.4	78.0	39.8
University of Waterloo (uwa)	75.3	82.6	91.3	72.9	41.5

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*
Outranking model: **this**, *Ranking rule*: **NetFlows**
Ordinal (Kendall) correlation between global ranking and global outranking relation: **+0.901**
Mean marginal correlation (a) : **+0.508**
Standard marginal correlation deviation (b) : **+0.187**
Ranking fairness (a) - (b) : **+0.321**

Fig. 3.10: Extract of a heatmap browser view on the *NetFlows* ranking result

As an exercise, the reader is invited to try out other robust outranking based ranking heuristics. Notice also that we have not challenged in this tutorial the THE provided criteria significance preorder. It would be very interesting to consider the five ranking objectives as equally important and, consequently, consider the ranking criteria to be equisignificant. Curious to see the ranking results under such settings.

Back to *Content Table* (page 1)

3.3 The best students, where do they study? A *rating* case study

In 2004, the German magazine *Der Spiegel*, with the help of *McKinsey & Company* and *AOL*, conducted an extensive online survey, assessing the apparent quality of German University students[28]. More than 80,000 students, by participating, were questioned on their ‘Abitur’ and university exams’ 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 *quality score* through a specific weighing of the collected data which depended on the subject the student is mainly studying.[29].

The eventually published results by the *Spiegel* magazine concerned nearly 50,000 students, enroled in one of fifteen popular academic subjects, like *German Studies*, *Life Sciences*, *Psychology*, *Law* or *CS*. Publishing only those subject-University combinations, where at least 18 students had correctly filled in the questionnaire, left 41 German Universities where, for at least eight out of the fifteen subjects, an average enrolment quality score could be determined[Page 178, 29].

Based on this published data[28], we would like to present and discuss in this tutorial, how to **rate** the apparent global *enrolment quality* of these 41 higher education institutions with the help of our *Digraph3* software ressources.

The performance tableau

Published data of the 2004 *Spiegel* student survey is stored, for our evaluation purpose here, in a file named studentenSpiegel04.py of `PerformanceTableau` format[32].

Listing 3.23: The 2004 Spiegel students survey data

```
>>> from perfTabs import PerformanceTableau
>>> t = PerformanceTableau('studentenSpiegel04')
>>> t
 *------- PerformanceTableau instance description ------*
  Instance class     : PerformanceTableau
```

(continues on next page)

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

[29] The methology guiding the *Spiegel* survey may be consulted in German here . A copy may be consulted in *examples* directory of the *Digraph3* ressources.

[32] The performance tableau `studentenSpiegel04.py` is also available in the `examples` directory of the Digraph3 software collection.

(continued from previous page)

```
  Instance name      : studentenSpiegel04
  # Actions          : 41 (Universities)
  # Criteria         : 15 (academic subjects)
  NA proportion (%) : 27.3
  Attributes         : ['name', 'actions', 'objectives',
                         'criteria', 'weightPreorder',
                         'evaluation']
>>> t.showHTMLPerformanceHeatmap(ndigits=1,
...                              rankingRule=None)
```

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

Color legend:

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

Fig. 3.11: Average quality of enroled students per academic subject

In Fig. 3.11, the fifteen popular academic subjects are grouped into topical '*Faculties*':

- *Humanities*; - *Law, Economics & Management*; - *Life Sciences & Medicine*; - *Natural Sciences & Mathematics*; and - *Technology*. All fifteen subjects are considered *equally significant* for our evaluation problem (see Row 2). The recorded average enrolment quality scores appear coloured along a 7-tiling scheme per subject (see last Row).

We may by the way notice that *TU Dresden* is the only Institution showing enrolment quality scores in all the fifteen academic subjects. Whereas, on the one side, *TU München* and *Kaiserslautern* are only valuated in *Sciences* and *Technology* subjects. On the other side, *Mannheim*, is only valuated in *Humanities* and *Law, Economics & Management* studies. Most of the 41 Universities are not valuated in *Engineering* studies. We are, hence, facing a large part (27.3%) of irreducible missing data (see Listing 3.23 Line 9 and the advanced topic on coping with missing data).

Details of the enrolment quality criteria (the academic subjects) may be consulted in a browser view (see Fig. 3.12 below).

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

#	Identifyer	Name	Comment	Weight	Scale			Thresholds (ax + b)		
					direction	min	max	indifference	preference	veto
1	**bio**	Life Sciences	Life Sciences & Medicine	1.00	max	45.00	65.00	0.00x + 0.10	0.00x + 0.50	
2	**chem**	Chemistry	Natural Sciences & Mathematics	1.00	max	45.00	65.00	0.00x + 0.10	0.00x + 0.50	
3	**eco**	Economics	Law, Economics & Management	1.00	max	45.00	65.00	0.00x + 0.10	0.00x + 0.50	
4	**elec**	Electrical Engineering	Technology	1.00	max	45.00	65.00	0.00x + 0.10	0.00x + 0.50	
5	**germ**	German Studies	Humanities	1.00	max	45.00	65.00	0.00x + 0.10	0.00x + 0.50	
6	**info**	Computer Science	Technology	1.00	max	45.00	65.00	0.00x + 0.10	0.00x + 0.50	
7	**law**	Law Studies	Law, Economics & Management	1.00	max	35.00	65.00	0.00x + 0.10	0.00x + 0.50	
8	**math**	Mathematics	Natural Sciences & Mathaematics	1.00	max	45.00	65.00	0.00x + 0.10	0.00x + 0.50	
9	**mec**	Mechanical Engineering	Technology	1.00	max	45.00	65.00	0.00x + 0.10	0.00x + 0.50	
10	**med**	Medicine	Life Sciences & Medicine	1.00	max	45.00	65.00	0.00x + 0.10	0.00x + 0.50	
11	**mgt**	Management	Law, Economics & Management	1.00	max	40.00	80.00	0.00x + 0.10	0.00x + 0.50	
12	**phys**	Physics	Natural Sciences & Mathematics	1.00	max	45.00	65.00	0.00x + 0.10	0.00x + 0.50	
13	**pol**	Politology	Humanities	1.00	max	50.00	70.00	0.00x + 0.10	0.00x + 0.50	
14	**psy**	Psychology	Humanities	1.00	max	50.00	70.00	0.00x + 0.10	0.00x + 0.50	
15	**soc**	Sociology	Humanities	1.00	max	45.00	65.00	0.00x + 0.10	0.00x + 0.50	

Fig. 3.12: Details of the rating criteria

The evaluation of the individual quality score for a participating student actually depends on his or her mainly enroled subject[29]. The apparent quality measurement scales thus largely differ indeed from subject to subject (see Fig. 3.12), like *Law Studies* (35.0 - 65-0) and *Politology* (50.0 - 70.0). The recorded average enrolment quality scores, hence, are in fact **incommensurable** between the subjects.

To take furthermore into account a potential and very likely *imprecision* of the individual quality scores' computation, we shall assume that, for all subjects, an average enrolment quality score difference of **0.1** is **insignificant**, wheras a difference of **0.5** is sufficient to *positively* attest a **better** enrolment quality.

The apparent *incommensurability* and very likely *imprecision* of the recorded average enrolment quality scores, renders **meaningless** any global averaging over the subjects per University of the enrolment quality. We shall therefore, similarly to the methodological approach of the *Spiegel* authors[Page 178, 29], proceed with an **order statistics** based *rating-by-ranking* approach (see tutorial on *rating with learned quantile norms* (page 108)).

Rating-by-ranking with lower-closed quantile limits

The Spiegel authors opted indeed for a simple 3-tiling of the Universities per valuated academic subject, followed by an average *Borda* scores based global ranking[Page 178, 29]. Here, our **epistemic logic** based **outranking approach**, allows us, with adequate choices of *indifference* (0.1) and *preference* (0.5) discrimination thresholds, to estimate **lower-closed 9-tiles** of the enrolment quality scores per subject and rank conjointly, with the help of the *Copeland* ranking rule[34] applied to a corresponding *bipolar-valued outranking* digraph, the 41 Universities **and** the lower limits of the estimated 9-tiles limits.

We need therefore to, first, estimate, with the help of the `PerformanceQuantiles` constructor, the lowerclosed 9-tiling of the average enrolment quality scores per academic subject.

Listing 3.24: Computing 9-tiles of the enrolment quality scores per subject

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

The *history sizes*, reported in Listing 3.24 above, indicate the number of Universities valuated in each one of the popular fifteen subjects. *German Studies*, for instance, are valuated for 39 out of 41 Universities, whereas *Electrical* and *Mechanical Engineering* are only valuated for 14, respectively 13 Institutions. None of the fifteen subjects are valuated in all the 41 Universities[30].

We may inspect the resulting 9-tiling limits in a browser view.

```
>>> pq.showHTMLLimitingQuantiles(Transposed=True,Sorted=False,
...      ndigits=1,title='9-tiled quality score limits')
```

[34] See the tutorial on *ranking with incommensurable performance criteria* (page 72).

[30] It would have been much more accurate to estimate such quantile limits from the individual qualitiy scores of all the nearly 50,000 surveyed students. But this data was not public.

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. 3.13: 9-tiling quality score limits per academic subject

In Fig. 3.13, we see confirmed again the **incommensurability** between the subjects, we noticed already in the apparent enrolment quality scoring , especially between *Law Studies* (39.1 - 51.1) and *Politology* (50.5 - 65.9). Universities valuated in *Law studies* but not in *Politology*, like the University of *Bielefeld*, would see their enrolment quality *unfairly weakened* when simply averaging the enrolment quality scores over valuated subjects.

We add, now, these 9-tiling quality score limits to the enrolment quality records of the 41 Universities and rank all these records conjointly together with the help of the `LearnedQuantilesRatingDigraph` constructor and by using the *Copeland ranking rule* (page 75).

```
>>> from sortingDigraphs import LearnedQuantilesRatingDigraph
>>> lqr = LearnedQuantilesRatingDigraph(pq,t,
...                                     rankingRule='Copeland')
```

The resulting ranking of the 41 Universities including the lower-closed 9-tiling score limits may be nicely illustrated with the help of a corresponding heatmap view (see Fig. 3.14).

```
>>> lqr.showHTMLRatingHeatmap(colorLevels=7,Correlations=True,
...                 ndigits=1,rankingRule='Copeland')
```

Ranking rule: **Copeland**; *Ranking correlation:* **0.967**

criteria	germ	chem	phys	mgt	law	bio	psy	pol	info	med	soc	math	eco	elec	mec
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
tau(*)	+0.51	+0.48	+0.47	+0.46	+0.40	+0.39	+0.38	+0.37	+0.33	+0.33	+0.29	+0.28	+0.20	+0.14	+0.12
[0.89 -	57.9	57.8	62.3	56.8	48.5	55.5	62.2	60.4	58.7	54.0	59.1	62.2	59.3	59.1	57.4
tum	NA	58.8	62.8	68.0	NA	53.6	NA	NA	58.2	60.1	NA	62.6	NA	58.2	56.9
frei	57.8	57.0	61.6	NA	50.7	55.4	64.1	60.5	58.1	54.2	57.5	60.6	53.3	NA	NA
kons	56.9	58.0	59.9	NA	48.3	55.7	59.1	65.9	59.7	NA	54.7	NA	59.0	NA	NA
leip	57.4	57.0	62.2	54.6	46.3	56.9	62.4	60.4	3.0	51.5	59.5	NA	NA	NA	NA
mu	57.2	58.1	62.0	57.5	47.3	50.4	60.9	60.1	57.1	52.6	54.0	59.6	55.8	NA	NA
hei	61.4	56.7	60.9	NA	51.1	55.2	59.8	59.5	NA	55.5	52.2	61.3	54.4	NA	NA
[0.78 -	57.3	57.1	61.8	55.7	47.2	54.8	60.8	59.6	58.0	52.3	56.2	60.7	56.8	57.5	56.4
stu	52.5	55.2	61.5	56.6	NA	57.1	NA	58.4	59.8	NA	NA	60.6	NA	60.2	57.8
berh	57.3	53.2	61.9	55.5	48.8	55.0	59.8	58.5	55.8	52.3	59.2	57.9	59.5	NA	NA
[0.67 -	56.9	56.4	61.6	55.5	46.4	53.5	59.8	58.3	57.2	51.4	55.6	60.5	55.8	57.3	55.8
aug	57.9	NA	62.3	54.3	45.6	NA	NA	54.3	58.1	NA	54.8	61.2	NA	NA	NA
mnh	52.2	NA	NA	59.7	45.0	NA	61.1	57.2	58.6	NA	55.0	NA	57.0	NA	NA
tueb	57.9	57.5	61.6	54.4	46.8	53.7	58.4	57.7	55.2	52.1	1.0	NA	60.8	NA	NA
mnst	55.4	55.0	55.2	56.3	46.4	51.2	62.2	56.7	57.7	52.8	56.3	56.9	54.1	NA	NA
jena	56.5	57.8	61.6	56.2	45.3	52.7	58.5	55.8	57.2	51.1	52.8	NA	NA	NA	NA
reg	54.8	55.8	60.5	55.5	46.1	54.6	62.1	55.4	NA	50.9	NA	59.2	52.3	NA	NA
saar	57.9	56.2	61.2	52.2	48.1	NA	56.5	NA	57.6	49.6	NA	NA	NA	NA	NA
[0.56 -	55.1	55.6	60.7	54.6	46.1	53.0	59.7	57.6	56.2	51.0	55.0	59.4	53.9	56.1	55.1
wrzb	56.9	56.6	60.2	52.8	46.4	53.0	59.8	56.0	55.9	52.2	NA	NA	53.3	NA	NA
dres	55.2	55.8	59.9	54.8	44.0	55.3	60.6	55.9	56.2	49.2	56.2	57.8	56.7	56.1	54.8
ksl	NA	54.6	59.7	55.9	NA	50.8	NA	NA	56.2	NA	NA	62.2	NA	57.5	56.3
marb	53.6	57.8	62.8	55.5	40.3	53.2	57.6	54.7	55.6	51.1	59.8	NA	NA	NA	NA
berf	54.7	57.4	61.6	52.2	45.7	51.6	59.8	61.4	54.9	49.0	55.5	NA	50.5	NA	NA
chem	54.3	NA	NA	52.7	NA	NA	60.8	57.1	57.7	NA	53.3	NA	NA	57.5	53.6
koel	51.7	54.0	58.7	54.6	46.1	51.8	58.9	57.6	NA	50.7	56.1	56.5	56.1	NA	NA
erl	57.9	54.0	60.3	55.6	42.9	51.8	58.7	55.1	54.6	49.3	55.4	60.5	NA	55.9	55.1
tri	54.1	NA	NA	52.8	46.3	NA	58.3	58.0	52.3	NA	54.9	60.7	52.8	NA	NA
[0.44 -	54.1	54.4	60.0	53.5	45.4	52.3	58.6	56.7	55.7	50.2	54.5	57.9	53.5	55.9	54.7
goet	58.7	53.9	60.4	52.6	44.8	50.5	59.8	56.3	NA	48.9	53.5	63.1	53.6	NA	NA
main	54.2	56.3	60.8	53.1	46.5	50.0	56.9	57.9	NA	49.2	55.7	54.7	50.7	NA	NA
bon	54.1	53.1	59.9	NA	47.2	50.1	60.3	57.3	53.7	3.0	56.0	59.4	53.6	NA	NA
brem	56.9	NA	59.7	55.4	40.9	53.3	52.5	55.5	54.1	NA	54.5	NA	NA	50.1	NA
[0.33 -	53.5	54.0	59.7	52.8	44.9	51.4	58.3	55.8	54.9	49.6	53.4	57.0	53.3	54.4	54.4
fran	51.7	55.5	62.1	52.0	41.9	51.3	58.0	53.1	52.4	51.2	51.5	57.0	53.5	NA	NA
ham	57.0	54.2	56.4	49.8	44.1	52.7	57.3	60.2	54.7	49.2	53.6	54.9	52.1	NA	NA
kiel	51.9	52.8	59.7	52.8	45.1	52.7	58.4	52.2	54.9	49.6	NA	NA	50.5	54.2	NA
aach	53.3	53.6	59.1	49.6	NA	52.2	62.7	50.8	54.6	49.5	51.0	58.6	NA	57.2	54.4
bertu	51.4	52.0	58.9	54.0	NA	NA	57.7	NA	55.4	NA	59.1	56.8	49.6	56.1	54.3
brau	53.5	50.1	59.8	53.4	NA	53.1	NA	54.0	52.6	NA	51.5	54.7	NA	54.5	55.2
darm	1.0	2.0	62.5	NA	NA	1.0	58.6	59.7	3.0	NA	52.0	59.4	NA	NA	56.1
[0.22 -	52.4	53.5	58.9	52.2	43.0	50.5	57.7	54.9	54.6	49.2	52.0	56.6	52.2	54.2	54.2
gie	53.0	NA	57.6	51.2	41.9	50.4	58.0	59.0	NA	50.0	NA	NA	NA	NA	NA
dsd	53.5	53.5	NA	50.5	44.9	47.3	57.5	NA	NA	50.5	48.8	NA	NA	NA	NA
bie	51.4	54.2	53.9	50.7	41.9	49.7	54.4	NA	55.8	NA	55.6	56.3	NA	NA	NA
boc	53.9	53.3	56.8	NA	39.1	48.0	55.2	NA	NA	49.8	NA	57.6	NA	54.2	54.4
han	50.4	53.6	57.5	NA	41.2	53.8	NA	52.8	58.8	NA	49.9	56.6	NA	53.5	53.6
[0.11 -	51.5	52.8	56.9	50.7	41.6	49.9	56.8	53.0	52.5	49.0	50.5	54.9	50.6	53.6	53.6
duis	50.6	52.8	54.6	47.5	NA	NA	NA	52.5	56.8	48.0	47.9	51.6	NA	53.6	51.9
[0.00 -	1.0	2.0	53.9	47.5	39.1	1.0	52.5	50.8	3.0	3.0	1.0	51.6	49.6	50.1	51.9

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.

Fig. 3.14: Heatmap view of the 9-tiles rating-by-ranking result

The *ordinal correlation* (+0.967)[35] of the *Copeland ranking* with the underlying bipolar-valued outranking digraph is very high (see Fig. 3.14 Row 1). Most correlated subjects with this *rating-by-ranking* result appear to be *German Studies* (+0.51), *Chemistry* (+0.48), *Management* (+0.47) and *Physics* (+0.46). Both *Electrical* (+0.07) and *Mechanical Engineering* (+0.05) are the less correlated subjects (see Row 3).

From the actual ranking position of the lower 9-tiling limits, we may now immediately deduce the 9-tile enrolment quality equivalence classes. No University reaches the highest 9-tile ([0.89 – [). In the lowest 9-tile ([0.00 – 0.11]) we find the University *Duisburg*. The complete rating result may be easily printed out as follows.

Listing 3.25: Rating the Universities into enrolment quality 9-tiles

```
1  >>> lqr.showQuantilesRating()
2   *-------- Quantiles rating result ---------
3    [0.89 - 1.00] []
4    [0.78 - 0.89[ ['tum', 'frei', 'kons', 'leip', 'mu', 'hei']
5    [0.67 - 0.78[ ['stu', 'berh']
6    [0.56 - 0.67[ ['aug', 'mnh', 'tueb', 'mnst', 'jena',
7                   'reg', 'saar']
8    [0.44 - 0.56[ ['wrzb', 'dres', 'ksl', 'marb', 'berf',
9                   'chem', 'koel', 'erl', 'tri']
10   [0.33 - 0.44[ ['goet', 'main', 'bon', 'brem']
11   [0.22 - 0.33[ ['fran', 'ham', 'kiel', 'aach',
12                  'bertu', 'brau', 'darm']
13   [0.11 - 0.22[ ['gie', 'dsd', 'bie', 'boc', 'han']
14   [0.00 - 0.11[ ['duis']
```

Following Universities: *TU München*, *Freiburg*, *Konstanz*, *Leipzig*, *München* as well as *Heidelberg*, appear best rated in the eigth 9-tile ([0.78 – 0.89[, see Listing 3.25 Line 4). Lowest-rated in the first 9-tile, as mentioned before, appears University *Duisburg* (Line 14). Midfield, the fifth 9-tile ([0.44 – 0.56[), consists of the Universities *Würzburg*, *TU Dresden*, *Kaiserslautern*, *Marburg*, *FU Berlin*, *Chemnitz*, *Köln* , *Erlangen-Nürnberg* and *Trier* (Lines 8-9).

A corresponding *graphviz* drawing may well illustrate all these enrolment quality equivalence classes.

```
>>> lqr.exportRatingByRankingGraphViz(fileName='ratingResult',
...                            graphSize='12,12')
 *---- exporting a dot file for GraphViz tools ---------*
  Exporting to ratingResult.dot
  dot -Grankdir=TB -Tpdf dot -o ratingResult.png
```

[35] See the advanced topic on the ordinal correlation of bipolar-valued digraphs.

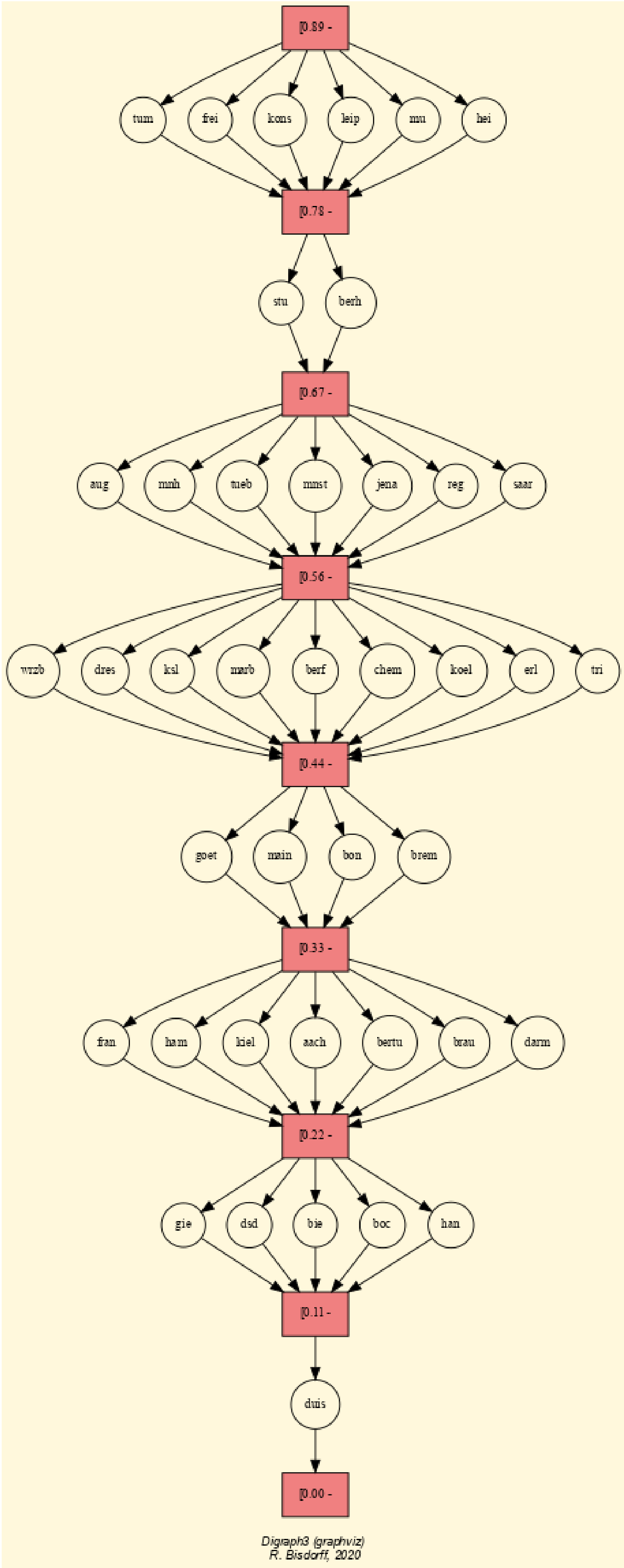


Fig. 3.15: Drawing of the 9-tiles rating-by-ranking result

We have noticed in the tutorial on *ranking with multiple criteria* (page 72), that there is not a single optimal rule for ranking from a given outranking digraph. The *Copeland* rule, for instance, has the advantage of being *Condorcet* consistent, i.e. when the outranking digraph models in fact a linear ranking, this ranking will necessarily be the result of the *Copeland* rule. When this is not the case, and especially when the outranking digraph shows many circuits, all potential ranking rules may give very divergent ranking results, and hence also substantially divergent rating-by-ranking results.

How *confident*, hence, is our precise *Copeland rating-by-ranking* result? To investigate this question, let us now inspect the **outranking digraph** on which we actually apply the *Copeland* ranking rule.

Inspecting the bipolar-valued outranking digraph

We say that University x **outranks** (resp. **is outranked by**) University y in enrolment quality when there exists a **majority** (resp. only a **minority**) of valuated subjects showing an **at least as good as** average enrolment quality score.

To compute these outranking situations, we use the `BipolarOutrankingDigraph` constructor.

Listing 3.26: Inspecting the bipolar-valued 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_studentenSpiegel04
7    # Actions            : 41 (Universities)
8    # Criteria           : 15 (subjects)
9    Size                 : 828 (outranking situations)
10   Determinateness (%)  : 63.67
11   Valuation domain     : [-1.00;1.00]
12 >>> dg.computeTransitivityDegree(Comments=True)
13  Transitivity degree of digraph <rel_studentenSpiegel04>:
14   #triples x>y>z: 57837, #closed: 30714, #open: 27123
15   (#closed/#triples) =  0.531
16 >>> dg.computeSymmetryDegree(Comments=True)
17  Symmetry degree of digraph <rel_studentenSpiegel04>:
18   #arcs x>y: 793, #symmetric: 35, #asymmetric: 758
19   #symmetric/#arcs =  0.044
```

The bipolar-valued outranking digraph *dg* (see Listing 3.23 Line 2), obtained with the given performance tableau *t*, shows 828 positively validated pairwise outranking situations (Line 9). Unfortunately, the transitivity of digraph *dg* is far from being satisfied: nearly half of the transitive closure is missing (Line 15). Despite the rather large *preference discrimination* threshold (0.5) we have assumed (see Fig. 3.12), there does not occur many indifference situations (Line 19).

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

Listing 3.27: Enumerating chordless outranking circuits

```
>>> dg.computeChordlessCircuits()
>>> dg.showChordlessCircuits()
 *---- Chordless circuits ----*
  93 circuits.
   1:  ['aach', 'bie', 'darm', 'brau'] , credibility : 0.067
   2:  ['aach', 'bertu', 'brau'] , credibility : 0.200
   3:  ['aach', 'bertu', 'brem'] , credibility : 0.067
   4:  ['aach', 'bertu', 'ham'] , credibility : 0.200
   5:  ['aug', 'tri', 'marb'] , credibility : 0.067
   6:  ['aug', 'jena', 'marb'] , credibility : 0.067
   7:  ['aug', 'jena', 'koel'] , credibility : 0.067
  ...
  ...
  29:  ['berh', 'kons', 'mu'] , credibility : 0.133
  ...
  ...
  88:  ['main', 'mnh', 'marb'] , credibility : 0.067
  89:  ['marb', 'saar', 'wrzb'] , credibility : 0.067
  90:  ['marb', 'saar', 'reg'] , credibility : 0.067
  91:  ['marb', 'saar', 'mnst'] , credibility : 0.133
  92:  ['marb', 'saar', 'tri'] , credibility : 0.067
  93:  ['mnh', 'mu', 'stu'] , credibility : 0.133
```

Here we observe indeed 93 such outranking circuits, like: *Berlin Humboldt* > *Konstanz* > *München* > *Berlin Humboldt* supported by a $(0.133 + 1.0)/2 = 56.7\%$ majority of subjects[31] (see Listing 3.27 circuit 29 above). In the *Copeland* ranking result shown in Fig. 3.14, these Universities appear positioned respectively at ranks 10, 4 and 6. In the *NetFlows* ranking result they would appear respectively at ranks 10, 6 and 5, thus inverting the positions of *Konstanz* and *München*. The occurrence in digraph *dg* of so many outranking circuits makes thus *doubtful* any *forced* linear ranking, independently of the specific ranking rule we might have applied.

To effectively check the quality of our *Copeland rating-by-ranking* result, we shall now compute a direct **sorting into 9-tiles** of the enrolment quality scores, without using any outranking digraph based ranking rule.

[31] Converted by a +1.0 shift and a 0.5 * 100 scale transform from a bipolar-valued credibility of +0.07 in [-1.0, +1.0] to a majority (in %) support.

Rating by quantiles sorting

In our case here, the Universities represent the decision actions: *where to study*. We say now that University x is sorted into the lower-closed 9-tile q when the performance record of x **positively outranks the lower limit** record of 9-tile q and x **does not positively outrank the upper limit** record of 9-tile q.

Listing 3.28: Lower-closed 9-tiles sorting of the 41 Universities

```
>>> lqr.showActionsSortingResult()
 Quantiles sorting result per decision action
 [0.33 - 0.44[: aach with credibility: 0.13 = min(0.13,0.27)
 [0.56 - 0.89[: aug with credibility: 0.13 = min(0.13,0.27)
 [0.44 - 0.67[: berf with credibility: 0.13 = min(0.13,0.20)
 [0.78 - 0.89[: berh with credibility: 0.13 = min(0.13,0.33)
 [0.22 - 0.44[: bertu with credibility: 0.20 = min(0.33,0.20)
 [0.11 - 0.22[: bie with credibility: 0.20 = min(0.33,0.20)
 [0.22 - 0.33[: boc with credibility: 0.07 = min(0.07,0.07)
 [0.44 - 0.56[: bon with credibility: 0.13 = min(0.20,0.13)
 [0.33 - 0.44[: brau with credibility: 0.07 = min(0.07,0.27)
 [0.33 - 0.44[: brem with credibility: 0.07 = min(0.07,0.07)
 [0.44 - 0.56[: chem with credibility: 0.07 = min(0.13,0.07)
 [0.22 - 0.56[: darm with credibility: 0.13 = min(0.13,0.13)
 [0.56 - 0.67[: dres with credibility: 0.27 = min(0.27,0.47)
 [0.22 - 0.33[: dsd with credibility: 0.07 = min(0.07,0.07)
 [0.00 - 0.11[: duis with credibility: 0.33 = min(0.73,0.33)
 [0.44 - 0.56[: erl with credibility: 0.13 = min(0.27,0.13)
 [0.22 - 0.44[: fran with credibility: 0.13 = min(0.13,0.33)
 [0.78 - <[: frei with credibility: 0.53 = min(0.53,1.00)
 [0.22 - 0.33[: gie with credibility: 0.13 = min(0.13,0.20)
 [0.33 - 0.44[: goet with credibility: 0.07 = min(0.47,0.07)
 [0.22 - 0.33[: ham with credibility: 0.07 = min(0.33,0.07)
 [0.11 - 0.22[: han with credibility: 0.20 = min(0.33,0.20)
 [0.78 - 0.89[: hei with credibility: 0.13 = min(0.13,0.27)
 [0.56 - 0.67[: jena with credibility: 0.07 = min(0.13,0.07)
 [0.33 - 0.44[: kiel with credibility: 0.20 = min(0.20,0.47)
 [0.44 - 0.56[: koel with credibility: 0.07 = min(0.27,0.07)
 [0.78 - <[: kons with credibility: 0.20 = min(0.20,1.00)
 [0.56 - 0.89[: ksl with credibility: 0.13 = min(0.13,0.40)
 [0.78 - 0.89[: leip with credibility: 0.07 = min(0.20,0.07)
 [0.44 - 0.56[: main with credibility: 0.07 = min(0.07,0.13)
 [0.56 - 0.67[: marb with credibility: 0.07 = min(0.07,0.07)
 [0.56 - 0.89[: mnh with credibility: 0.20 = min(0.20,0.27)
 [0.56 - 0.67[: mnst with credibility: 0.07 = min(0.20,0.07)
 [0.78 - 0.89[: mu with credibility: 0.13 = min(0.13,0.47)
 [0.56 - 0.67[: reg with credibility: 0.20 = min(0.20,0.27)
 [0.56 - 0.78[: saar with credibility: 0.13 = min(0.13,0.20)
```

(continues on next page)

(continued from previous page)

```
39  [0.78 - 0.89[: stu with credibility: 0.07 = min(0.13,0.07)
40  [0.44 - 0.56[: tri with credibility: 0.07 = min(0.13,0.07)
41  [0.67 - 0.78[: tueb with credibility: 0.13 = min(0.13,0.20)
42  [0.89 - <[: tum with credibility: 0.13 = min(0.13,1.00)
43  [0.56 - 0.67[: wrzb with credibility: 0.07 = min(0.20,0.07)
```

In the 9-tiles sorting result, shown in Listing 3.28, we notice for instance in Lines 3-4 that the *RWTH Aachen* is precisely rated into the 4th 9-tile ($[0.33 - 0.44[$), whereas the University *Augsburg* is less precisely rated conjointly into the *6th*, the *7th* and the *8th* 9-tile ($[0.56 - 0.89[$). In Line 42, *TU München* appears best rated into the unique highest 9-tile ($[0.89 - < [$). All three rating results are supported by a $(0.07 + 1.0)/2 = 53.5\%$ majority of valuated subjects[Page 188, 31]. With the support of a 76.5% majority of valuated subjects (Line 20), the apparent most confident rating result is the one of University *Freiburg* (see also Fig. 3.11 and Fig. 3.14).

We shall now lexicographically sort these individual rating results per University, by *average* rated 9-tile limits and *highest-rated* upper 9-tile limit, into ordered, but not necessarily disjoint, enrolment quality quantiles.

```
>>> lqr.showHTMLQuantilesSorting(strategy='average')
```

quantile limits	Ordering by average quantile class limits
[0.89-<[	['tum']
[0.78-<[	['frei', 'kons']
[0.78-0.89[	['berh', 'hei', 'leip', 'mu', 'stu']
[0.56-0.89[	['aug', 'ksl', 'mnh']
[0.67-0.78[	['tueb']
[0.56-0.78[	['saar']
[0.56-0.67[	['dres', 'jena', 'marb', 'mnst', 'reg', 'wrzb']
[0.44-0.67[	['berf']
[0.44-0.56[	['bon', 'chem', 'erl', 'koel', 'main', 'tri']
[0.22-0.56[	['darm']
[0.33-0.44[	['aach', 'brau', 'brem', 'goet', 'kiel']
[0.22-0.44[	['bertu', 'fran']
[0.22-0.33[	['boc', 'dsd', 'gie', 'ham']
[0.11-0.22[	['bie', 'han']
[0.00-0.11[	['duis']

Fig. 3.16: The ranked 9-tiles rating-by-sorting result

In Fig. 3.16 we may notice that the Universities: *Augsburg*, *Kaiserslautern*, *Mannheim* and *Tübingen* for instance, show in fact the same average rated 9-tiles score of 0.725; yet, the rated upper 9-tile limit of *Tuebingen* is only 0.78, whereas the one of the other Universities reaches 0.89. Hence, *Tuebingen* is ranked below *Augsburg*, *Kaiserslautern* and *Mannheim* .

With a special *graphviz* drawing of the `LearnedQuantilesRatingDigraph` instance *lqr*, we may, without requiring any specific ordering strategy, as well illustrate our 9-tiles *rating-by-sorting* result.

```
>>> lqr.exportRatingBySortingGraphViz(\
...           'nineTilingDrawing',graphSize='12,12')
 *---- exporting a dot file for GraphViz tools ---------*
  Exporting to nineTilingDrawing.dot
  dot -Grankdir=TB -Tpng nineTilingDrawing.dot -o nineTilingDrawing.png
```

Fig. 3.17: Graphviz drawing of the 9-tiles sorting digraph

In Fig. 3.17 we actually see the *skeleton* (transitive closure removed) of a **partial order**, where an oriented arc is drawn between Universities x and y when their 9-tiles sorting results are **disjoint** and the one of x is **higher rated** than the one of y. The rating for *TU München* (see Listing 3.28 Lines 45), for instance, is disjoint and higher rated than the one of the Universities *Freiburg* and *Konstanz* (Lines 23, 32). And, both the ratings of *Feiburg* and *Konstanz* are, however, not disjoint from the one, for instance, of the Universty of *Stuttgart* (Line 42).

The partial ranking, shown in Fig. 3.17, is in fact **independent** of any ordering strategy: - *average*, - *optimistic* or - *pessimistic*, of overlapping 9-tiles sorting results, and confirms that the same Universities as with the previous *rating-by-ranking* approach, namely *TU München*, *Freiburg*, *Konstanz*, *Stuttgart*, *Berlin Humboldt*, *Heidelberg* and *Leipzig* appear top-rated. Similarly, the Universities of *Duisburg*, *Bielefeld*, *Hanover*, *Bochum*, *Giessen*, *Düsseldorf* and *Hamburg* give the lowest-rated group. The midfield here is again consisting of more or less the same Universities as the one observed in the previous *rating-by-ranking* approach (see Fig. 3.15).

To conclude

In the end, both the *Copeland rating-by-ranking*, as well as the *rating-by-sorting* approach give luckily, in our case study here, very similar results. The first approach, with its *forced* linear ranking, determines on the one hand, *precise* enrolment quality equivalence classes; a result, depending potentially a lot on the actually applied ranking rule. The *rating-by-sorting* approach, on the other hand, only determines for each University a less precise but *prudent* rating of its individual enrolment quality, furthermore supported by a known majority of performance criteria significance; a somehow *fairer* and *robuster* result, but, much less evident for easily comparing the apparent enrolment quality among Universities. Contradictorily, or sparsely valuated Universities, for instance, will appear trivially rated into a large midfield of adjacent 9-tiles.

Let us conclude by saying that we prefer this latter *rating-by-sorting* approach; perhaps impreciser, due the case given, to missing and contradictory performance data; yet, well grounded in a powerful bipolar-valued logical and espistemic framework (see the advanced topics of the Digraph3 documentation).

Back to *Content Table* (page 1)

3.4 Exercises

We propose hereafter some decision problems which may serve as exercises and exam questions in an *Algorithmic Decision Theory* Course. They cover *selection*, *ranking* and *rating* decision problems. The exercises are marked as follows: § (warming up), §§ (home work), §§§ (research work).

Solutions should be supported both by computational Python code using the **Digraph3** programming resources as well as by methodological and algorithmic arguments from the Algorithmic Decision Theory Lectures.

Who will receive the best student award? (§)

Data

Below in Table 3.3 you see the actual grades obtained by four students : *Ariana* (A), *Bruce* (B), *Clare* (C) and *Daniel* (D) in five courses: *C1*, *C2*, *C3*, *C4* and *C5* weighted by their respective ECTS points.

Table 3.3: Grades obtained by the students

Course ECTS	C1 2	C2 3	C3 4	C4 2	C5 4
Ariana (A)	11	13	9	15	11
Bruce (B)	12	9	13	10	13
Clare (C)	8	11	14	12	14
Daniel (D)	15	10	12	8	13

The grades shown in Table 3.3 are given on an ordinal performance scale from 0 pts (weakest) to 20 pts (highest). Assume that the grading admits a *preference* threshold of 1 points. No *considerable* performance differences are given. The more **ECTS** points, the more importance a course takes in the curriculum of the students. An award is to be granted to the *best* amongst these four students.

Questions

1. Edit a *PerformanceTableau* (page 47) instance with the data shown above.
2. Who would you nominate ?
3. Explain and motivate your selection algorithm.
4. Assume that the grading may actually admit an *indifference* threshold of 1 point and a *preference* threshold of 2 points. How stable is your result with respect to the actual preference discrimination power of the grading scale?

How to fairly rank movies (§)

Data

- File graffiti03.py contains a performance tableau about the rating of movies to be seen in the city of Luxembourg, February 2003. Its content is shown in Fig. 3.18 below.

```
>>> from perfTabs import PerformanceTableau
>>> t = PerformanceTableau('graffiti03')
>>> t.showHTMLPerformanceHeatmap(WithActionNames=True,
...                              pageTitle='Graffiti Star wars',
...                              rankingRule=None,colorLevels=5,
...                              ndigits=0)
```

Graffiti Star wars

movies (id) \ critics	jh	vt	ap	as	cf	cn	cs	dr	jt	mk	mr	rr	td
weights	+2.00	+2.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00	+1.00
Ah si j'étais riche (ah)	1	NA	NA	NA	NA	-1	1	NA	1	2	NA	1	3
A walk to remember (aw)	NA	NA	-1	NA	1	NA	2	NA	-1	1	NA	1	NA
Bend it like Beckham (bb)	2	1	2	1	2	2	3	2	2	3	2	3	1
Demonlover (dl)	1	-1	-1	-1	-1	NA	-1	1	1	NA	1	1	1
Gangs of New York (gny)	3	3	2	4	2	4	2	3	4	2	4	3	2
Ghost Ship (gs)	NA	NA	1	NA	-1	NA	1	1	1	NA	-1	-1	-1
El Hija de la Novia (hn)	2	1	3	NA	3	2	2	NA	2	3	2	2	NA
Lantana (la)	3	3	3	2	3	3	2	2	3	3	4	3	NA
Lord of the Rings - The Two Towers (lor)	3	2	2	3	3	NA	3	4	4	1	2	2	2
The Magdalene Sisters (ma)	3	3	NA	NA	NA	3	2	3	3	3	2	2	3
Mr. Deeds (md)	NA	NA	1	1	-1	NA	NA	-1	1	-1	NA	-1	1
Mon Idole (mi)	1	1	NA	-1	NA	1	-1	NA	2	NA	NA	-1	2
the Slaton Sea (sa)	2	NA	NA	NA	NA	NA	2	NA	1	3	1	NA	NA
the santa Clause 2 (sc)	NA	NA	1	NA	1	NA	-1	NA	1	1	NA	1	NA
Sweet home Alabama (sha)	-1	NA	2	-1	1	1	2	2	2	1	1	1	NA
Sweet Sixteen (ss)	3	3	3	3	3	4	2	3	3	3	3	1	3
24 heures de la vie d'une femme (vf)	1	NA	NA	NA	NA	1	NA	NA	1	NA	NA	1	1

Color legend:

quantile	20.00%	40.00%	60.00%	80.00%	100.00%

Fig. 3.18: Graffiti magazine's movie ratings from February 2003

The critic's opinions are expressed on a 7-graded scale: -2 (two zeros, *I hate*), -1 (one zero, *I don't like*), 1 (one star, *maybe*), 2 (two stars, *good*), 3 (three stars, *excellent*), 4 (four stars, *not to be missed*), and 5 (five stares, *a master piece*). Notice the many missing data (NA) when a critic had not seen the respective movie. Mind also that the ratings of two movie critics (*jh* and *vt*) are given a higher significance weight.

Questions

1. The Graffiti magazine suggest a best rated movie with the help of an average number of stars, ignoring the missing data and any significance weights of the critics. By taking into account missing data and varying significance weights, how may one find the best rated movie without computing any average rating scores ?
2. How would one rank these movies so as to at best respect the weighted rating opinions of each movie critic ?
3. In what ranking position would appear a movie not seen by any movie critic ? Confirm computationally the answer by adding such a fictive, *not at all evaluated*, movie to the given performance tableau instance.
4. How robust are the preceeding results when the significance weights of the movie critics are considered to be only ordinal grades ?

What is your best choice recommendation? (§)

Data[46]

A person, who wants to by a TV set, retains after a first selection, eight potential TV models. To make up her choice these eight models were evaluated with respect to three decision objectives of *equal importance*: - **Costs** of the set (to be minimized); - **Picture and Sound** quality of the TV (to be maximized): - **Maintenace contract** quality of the provider (to be maximized).

The **Costs** objective is assessed by the price of the TV set (criterion *Pr* to be minimized). *Picture* quality (criterion *Pq*), *Sound* quality (criterion *Sq*) and *Maintenace contract* quality (criterion *Mq*) are each assessed on a four-level qualitative performance scale: -1 (*not good*), 0 (*average*), 1 (*good*) and 2 (*very good*).

The actual evaluation data are gathered in Table 3.4 below.

Table 3.4: Performance evaluations of the potential TV sets

Criteria Significance	Pr (€) 2	Pq 1	Sq 1	Mq 2
Model T1	-1300	2	2	0
Model T2	-1200	2	2	1
Model T3	-1150	2	1	1
Model T4	-1000	1	1	-1
Model T5	-950	1	1	0
Model T6	-950	0	1	-1
Model T7	-900	1	0	-1
Model T8	-900	0	0	0

The *Price* criterion *Pr* supports furthermore an *indifference* threshold of 25.00 € and a *preference* threshold of 75.00 €. No considerable performance differences (*veto* thresholds) are to be considered.

Questions

1. Edit a *PerformanceTableau* (page 47) instance with the data shown above and illustrate its content by best showing *objectives*, *criteria*, *decision alternatives* and *performance table*. If needed, write adequate python code.
2. What is the best TV set to recommend?
3. Illustrate your best choice recommendation with an adequate graphviz drawing.
4. Explain and motivate your selection algorithm.
5. Assume that the qualitative criteria: *Picture* quality (*Pq*), *Sound* quality (*Sq*), and *Maintenace contract* quality (*Mq*), are all three considered to be *equi-significant* and that the significance of the Price criterion (*Pr*) equals the significance of these three

[46] The data is taken from Ph. Vincke, *Multicriteria Decision-Aid*, John Wiley & Sons Ltd, Chichester UK 1992, p.33-35.

quality criteria taken together. How stable is your best choice recommendation with respect to changing these criteria significance weights?

What is the best public policy? (§§)

Data files

- File perfTab_1.py contains a *3 Objectives performance tableau* (page 39) with 100 performance records concerning public policies evaluated with respect to an economic, a societal and an environmental public decision objective.
- File historicalData_1.py contains a performance tableau of the same kind with 2000 historical performance records.

Questions

1. Illustrate the content of the given *perfTab_1.py* performance tableau by best showing *objectives*, *criteria*, *decision alternatives* and *performance table*. If needed, write adequate python code.
2. Construct the corresponding bipolar-valued outranking digraph. How *confident* and/or *robust* are the apparent outranking situations?
3. What are apparently the 5 best-ranked decision alternatives in your decision problem from the different decision objectives point of views and from a global fair compromise view? Justify your ranking approach from a methodological point of view.
4. How would you rate your 100 public policies into relative deciles classes ?
5. Using the given historical records in historicalData_1.py, how would you rate your 100 public policies into absolute deciles classes ? Explain the differencea you may observe between the absolute and the previous relative rating result.
6. Select among your 100 potential policies a shortlist of up to 15 potential first policies, all reaching an absolute performance quantile of at least 66.67%.
7. Based on the previous best policies shortlist (see Question 6), what is your eventual best-choice recommendation? Is it perhaps an unopposed best choice by all three objectives?

A fair diploma validation decision (§§§)

Data

Use the `RandomAcademicPerformanceTableau` constructor from the **Digraph3** Python resources for generating realistic random students performance tableaux concerning a curriculum of nine ECTS weighted Courses. Assume that all the gradings are done on an integer scale from 0 (weakest) to 20 (best). It is known that all grading procedures are inevitably imprecise; therefore we will assume an indifference threshold of 1 point and a preference theshold of 2 points. Thurthermore, a performance difference of more than

12 points is considerable and will trigger a veto situation. To validate eventually their curriculum, the students are required to obtain more or less 10 points in each course.

Questions

1. Design and implement a fair diploma validation decision rule based on the grades obtained in the nine Courses.
2. Run simulation tests with random students performance tableaux for validating your design and implementation.

Back to *Content Table* (page 1)

---

4 Moving on to undirected graphs

4.1 Working with the `graphs` module

Structure of a `Graph` object

In the **`graphs`** module, the root `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

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

The saved **Graph** instance named ‘tutorialGraph.py’ is encoded in python3 as follows.

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

The stored graph can be recalled and plotted with the generic `exportGraphViz()` Page 7, 1 method as follows.

```
>>> g = Graph('tutorialGraph')
>>> g.exportGraphViz()
*---- exporting a dot file for GraphViz tools ---------*
```

(continues on next page)

(continued from previous page)

```
Exporting to tutorialGraph.dot
fdp -Tpng tutorialGraph.dot -o tutorialGraph.png
```

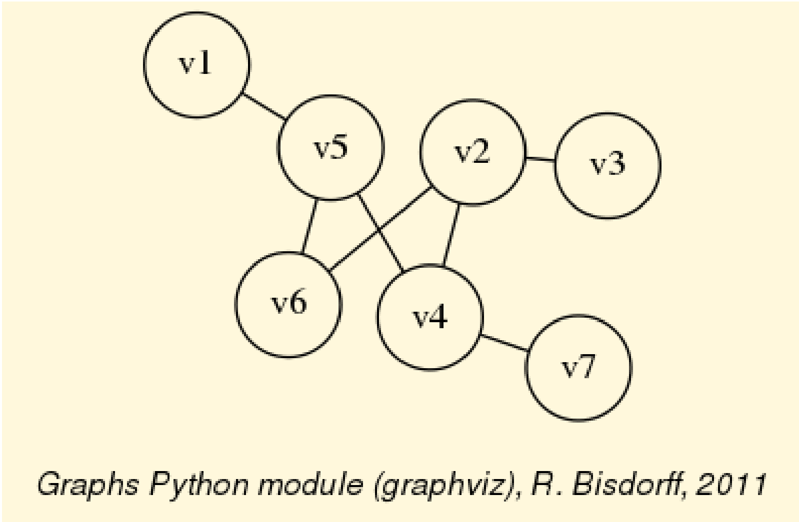


Fig. 4.1: Tutorial graph instance

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 `graph2Digraph()`, and vice versa with the `digraph2Graph()` method. Thus, all resources of the `Digraph` class, suitable for symmetric digraphs, become readily available, and vice versa.

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

q-coloring of a graph

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

```
>>> from graphs import Q_Coloring
>>> qc = Q_Coloring(g)
 Running a Gibbs Sampler for 42 step !
 The q-coloring with 3 colors is feasible !!
>>> qc.showConfiguration()
 v5 lightblue
 v3 gold
 v7 gold
 v2 lightblue
 v4 lightcoral
```

(continues on next page)

(continued from previous page)

```
v1 gold
v6 lightcoral
>>> qc.exportGraphViz('tutorial-3-coloring')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to tutorial-3-coloring.dot
fdp -Tpng tutorial-3-coloring.dot -o tutorial-3-coloring.png
```

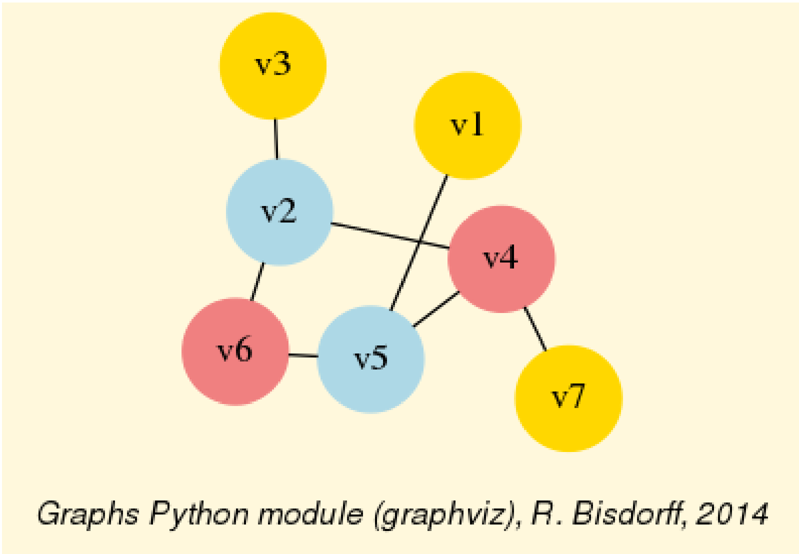


Fig. 4.2: 3-Coloring of the tutorial graph

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

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

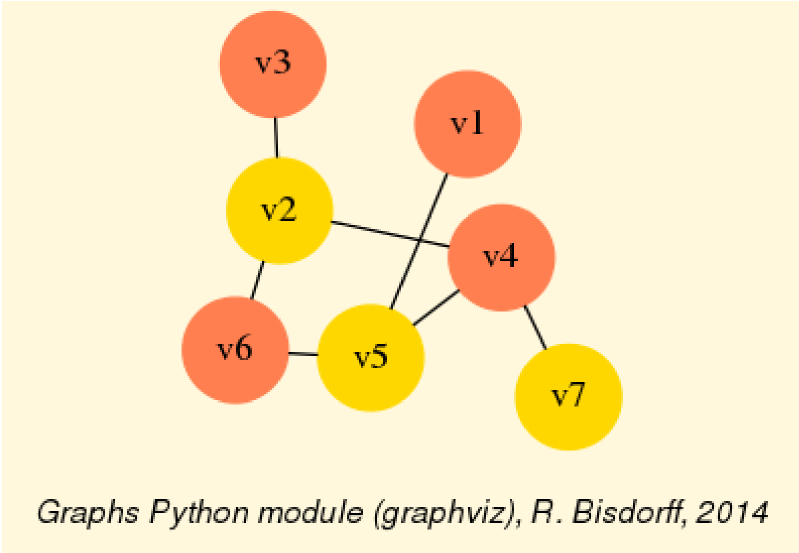


Fig. 4.3: 2-coloring of the tutorial graph

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 `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[Page 18, 14]), corresponds to a maximal **clique**, i.e. a maximal complete subgraph in g. Maximal cliques may be directly enumerated with the `showCliques()` method.

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

Line graphs and maximal matchings

The module also provides a `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.

```
>>> from graphs import CycleGraph, LineGraph
>>> g = CycleGraph(order=5)
>>> g
 *------- Graph instance description ------*
 Instance class   : CycleGraph
 Instance name    : cycleGraph
 Graph Order      : 5
 Graph Size       : 5
 Valuation domain : [-1.00; 1.00]
 Attributes       : ['name', 'order', 'vertices', 'valuationDomain',
                     'edges', 'size', 'gamma']
>>> lg = LineGraph(g)
>>> lg
 *------- Graph instance description ------*
 Instance class   : LineGraph
 Instance name    : line-cycleGraph
 Graph Order      : 5
 Graph Size       : 5
 Valuation domain : [-1.00; 1.00]
```

(continues on next page)

(continued from previous page)

```
 Attributes       : ['name', 'graph', 'valuationDomain', 'vertices',
                     'order', 'edges', 'size', 'gamma']
>>> lg.showShort()
 *---- short description of the graph ----*
 Name             : 'line-cycleGraph'
 Vertices         :  [frozenset({'v1', 'v2'}), frozenset({'v1', 'v5'}), frozenset({'v2', 'v3'}),
                      frozenset({'v3', 'v4'}), frozenset({'v4', 'v5'})]
 Valuation domain :  {'min': Decimal('-1'), 'med': Decimal('0'), 'max': Decimal('1')}
 Gamma function   :
 frozenset({'v1', 'v2'}) -> [frozenset({'v2', 'v3'}), frozenset({'v1', 'v5'})]
 frozenset({'v1', 'v5'}) -> [frozenset({'v1', 'v2'}), frozenset({'v4', 'v5'})]
 frozenset({'v2', 'v3'}) -> [frozenset({'v1', 'v2'}), frozenset({'v3', 'v4'})]
 frozenset({'v3', 'v4'}) -> [frozenset({'v2', 'v3'}), frozenset({'v4', 'v5'})]
 frozenset({'v4', 'v5'}) -> [frozenset({'v4', 'v3'}), frozenset({'v1', 'v5'})]
 degrees      :  [0, 1, 2, 3, 4]
 distribution :  [0, 0, 5, 0, 0]
 nbh depths   :  [0, 1, 2, 3, 4, 'inf.']
 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.

```
>>> c8 = CycleGraph(order=8)
>>> lc8 = LineGraph(c8)
>>> lc8.showMIS()
 *---  Maximal Independent Sets ---*
 [frozenset({'v3', 'v4'}), frozenset({'v5', 'v6'}), frozenset({'v1', 'v8'})]
 [frozenset({'v2', 'v3'}), frozenset({'v5', 'v6'}), frozenset({'v1', 'v8'})]
 [frozenset({'v8', 'v7'}), frozenset({'v2', 'v3'}), frozenset({'v5', 'v6'})]
 [frozenset({'v8', 'v7'}), frozenset({'v2', 'v3'}), frozenset({'v4', 'v5'})]
 [frozenset({'v7', 'v6'}), frozenset({'v3', 'v4'}), frozenset({'v1', 'v8'
```

(continues on next page)

```
↪'})]
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.

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

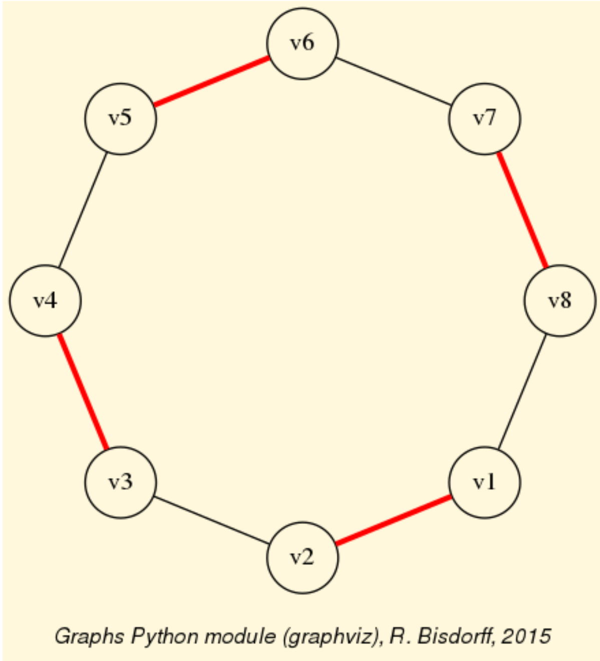


Fig. 4.4: A perfect maximum matching of the 8-cycle graph

Grids and the Ising model

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

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

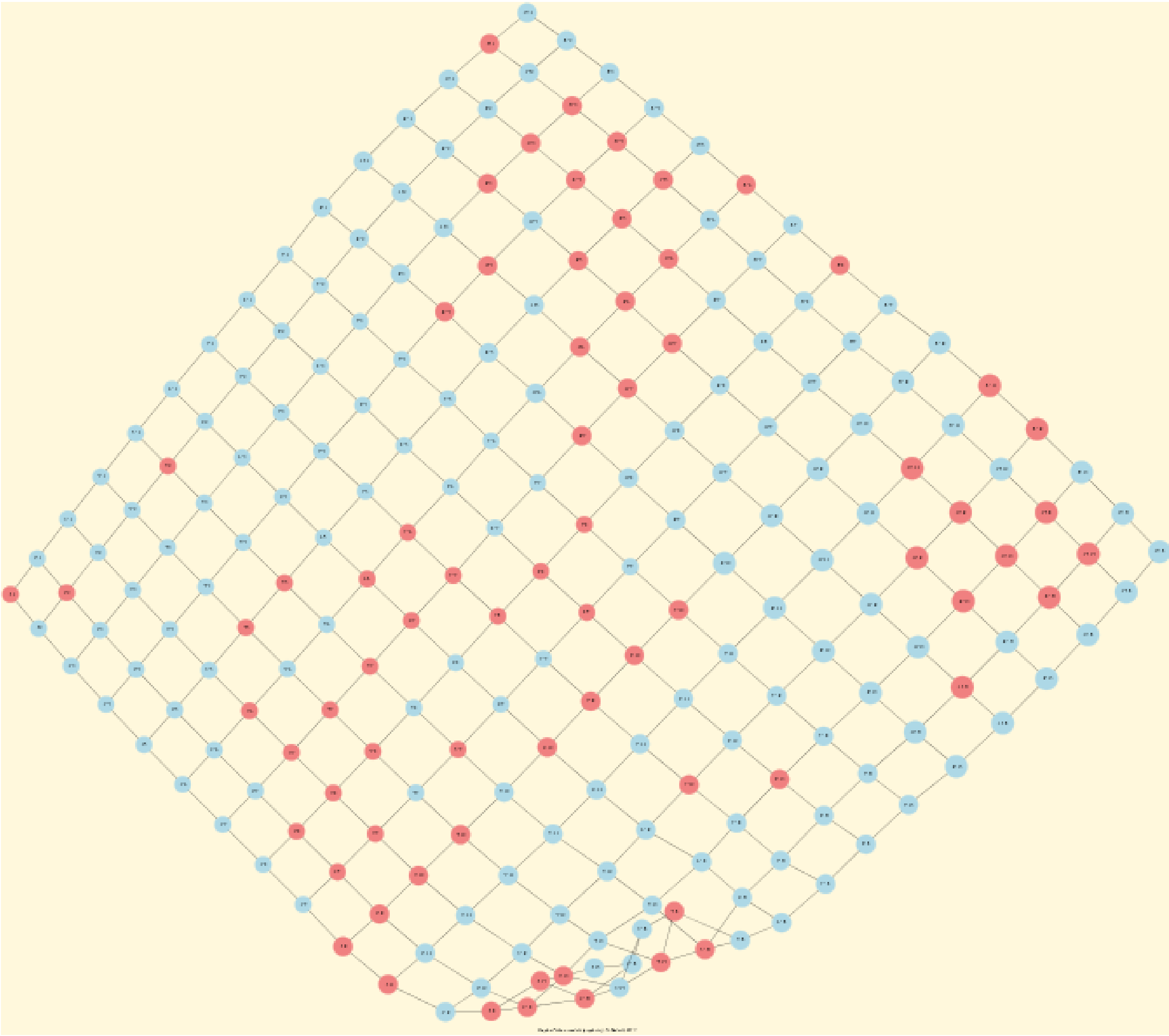


Fig. 4.5: Ising model of the 15x15 grid graph

Simulating Metropolis random walks

Finally, we provide the `MetropolisChain` class, a specialization of the `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']
```

(continues on next page)

(continued from previous page)

```
v2 -> ['v1', 'v4']
v3 -> ['v5', 'v1']
v4 -> ['v2', 'v5', 'v1']
v5 -> ['v3', 'v4']
```

```
>>> probs = {}  # initialize a potential stationary probability vector
>>> n = g.order # for instance: probs[v_i] = n-i/Sum(1:n) for i in 1:n
>>> i = 0
>>> verticesList = [x for x in g.vertices]
>>> verticesList.sort()
>>> for v in verticesList:
...     probs[v] = (n - i)/(n*(n+1)/2)
...     i += 1
```

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.

```
>>> met = MetropolisChain(g,probs)
>>> frequency = met.checkSampling(verticesList[0],nSim=30000)
>>> for v in verticesList:
...     print(v,probs[v],frequency[v])

v1 0.3333 0.3343
v2 0.2666 0.2680
v3 0.2    0.2030
v4 0.1333 0.1311
v5 0.0666 0.0635
```

In this example, the stationary transition probability distribution, shown by the `showTransitionMatrix()` method above (see below), is quite adequately simulated.

```
>>> met.showTransitionMatrix()
* ---- Transition Matrix -----
  Pij  | 'v1'    'v2'    'v3'    'v4'    'v5'
  -----|-------------------------------------
  'v1' |  0.23   0.33    0.30    0.13    0.00
  'v2' |  0.42   0.42    0.00    0.17    0.00
  'v3' |  0.50   0.00    0.33    0.00    0.17
  'v4' |  0.33   0.33    0.00    0.08    0.25
  'v5' |  0.00   0.00    0.50    0.50    0.00
```

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 1)

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

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 `CirculantDigraph` class instance of order 12 and symmetric circulants 1 and -1.

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

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

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

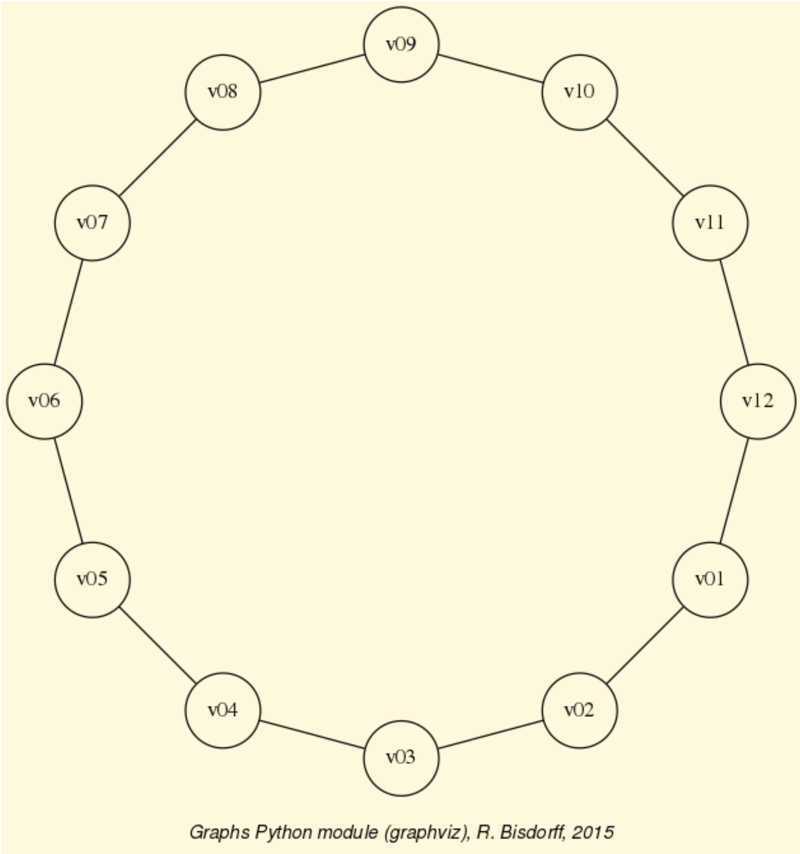


Fig. 4.6: The 12-cycle graph

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 `showMIS()` method.

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

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

[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`

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

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

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

(continues on next page)

`installPerrinUser` installs it instead without sudo into the user's private `<$Home/.bin>` directory.

(continued from previous page)

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

Computing the automorphism group

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

```
>>> c12.automorphismGenerators()

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

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

Computing the isomorphic MISs

The command `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    ...
4    *---- Global result ----
5    Number of MIS:  29
```

(continues on next page)

[2] Dependency: The `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`.

(continued from previous page)

```
6   Number of orbits :  4
7   Labelled representatives and cardinality:
8   1: ['2','4','6','8','10','12'], 2
9   2: ['2','5','8','11'], 3
10  3: ['2','4','6','9','11'], 12
11  4: ['1','4','7','9','11'], 12
12  Symmetry vector
13  stabilizer size: [1, 2, 3, ..., 8, 9, ..., 12, 13, ...]
14  frequency      : [0, 2, 0, ..., 1, 0, ...,  1,  0, ...]
```

The corresponding group stabilizers' sizes and frequencies – orbit 1 with 6 symmetry axes, orbit 2 with 4 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. 4.7 below.

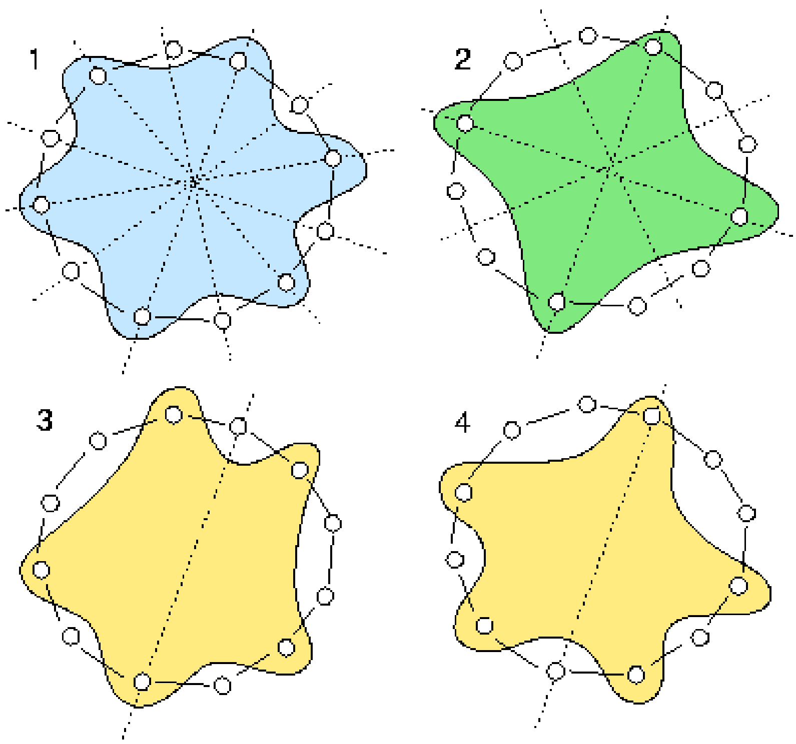


Fig. 4.7: The symmetry axes of the four non isomorphic MISs of the 12-cycle graph

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 1)

4.3 About split, interval and permutation graphs

A multiply *perfect* graph

A graph g is called:

- *Berge* or **perfect** when g and its dual $-g$ both don't contain any chordless odd cycles of length greater than 3 ([BER-1963], [CHU-2006]),
- **Triangulated** when g does not contain any chordless cycle of length 4.

Following Martin Golumbic (see [GOL-2004] p. 149), we call a given graph g :

- **Comparability graph** when g is *transitively orientable*;
- **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.

All these four kinds of graphs are in fact *perfect* graphs. To illustrate these graph classes, we generate from 8 intervals, randomly chosen in the default integer range [0,10], a `RandomIntervalIntersectionsGraph` instance g (see Listing 4.1 Line 2 below).

Listing 4.1: 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 g, which is **conjointly** a *triangulated*, a *comparability*, a *split* and a *permutation* graph (see Listing 4.2 Lines 6,10,14).

Listing 4.2: 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
```

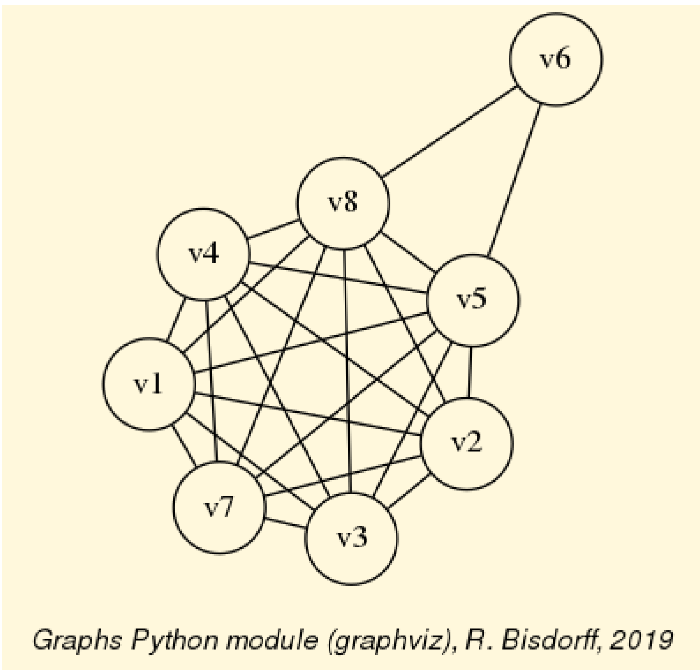


Fig. 4.8: A conjointly triangulated, comparability, interval, permutation and split graph

In Fig. 4.8 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).

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 `Graph` instance data:

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

(continues on next page)

[18] A Digraph3 *graphs.Graph* encoded file is available in the `examples` directory of the Digraph3 software collection.

(continued from previous page)

```
frozenset(['E','I']) : 1,
}
```

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.

```
>>> from graphs import Graph
>>> g = Graph('berge')
>>> g.showShort()
 *---- short description of the graph ----*
 Name              : 'berge'
 Vertices          :  ['A', 'B', 'C', 'D', 'E', 'I']
 Valuation domain :  {'min': -1, 'med': 0, 'max': 1}
 Gamma function    :
 A -> ['D', 'B', 'E']
 B -> ['E', 'I', 'A']
 C -> ['E', 'D', 'I']
 D -> ['C', 'I', 'A']
 E -> ['C', 'B', 'I', 'A']
 I -> ['C', 'E', 'B', 'D']
>>> g.exportGraphViz('berge1')
 *---- exporting a dot file for GraphViz tools ---------*
 Exporting to berge1.dot
 fdp -Tpng berge1.dot -o berge1.png
```

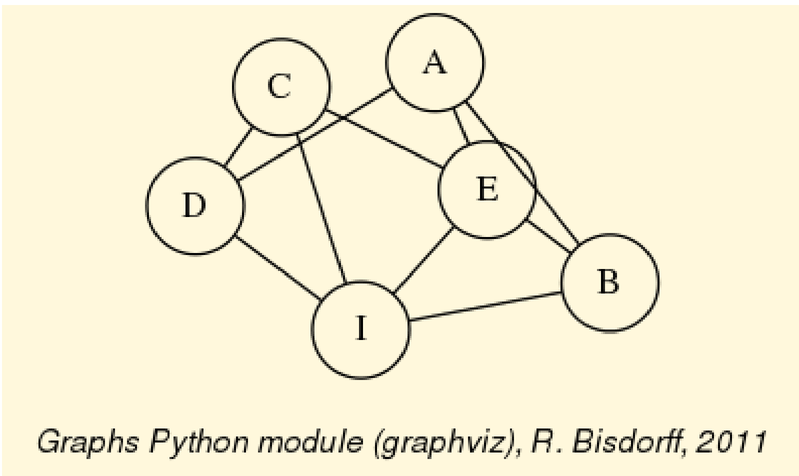


Fig. 4.9: Graph representation of the testimonies of the professors

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.

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

We see three intersection cycles of length 4, which is impossible to occur on the linear time line. Obviously one professor lied!

And it is D ; if we put to doubt 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
```

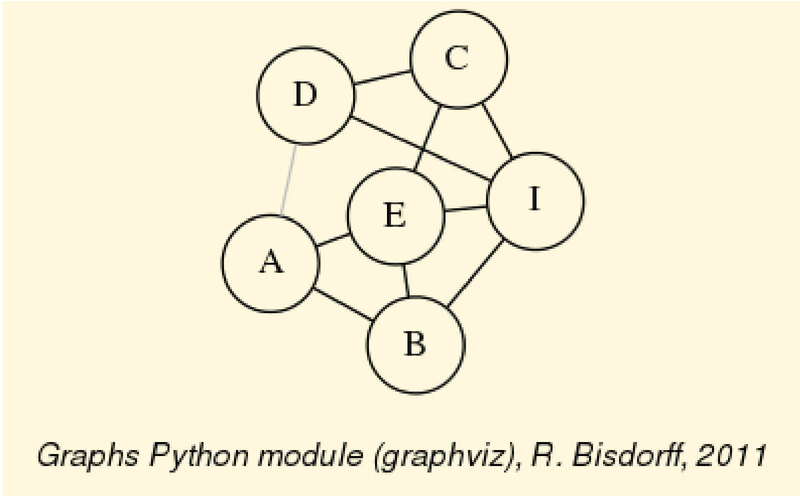


Fig. 4.10: The triangulated testimonies graph

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.

```
>>> from graphs import PermutationGraph
>>> g = PermutationGraph(permutation = [4, 3, 6, 1, 5, 2])
>>> g
 *------- Graph instance description ------*
 Instance class    : PermutationGraph
 Instance name     : permutationGraph
 Graph Order       : 6
 Permutation       : [4, 3, 6, 1, 5, 2]
 Graph Size        : 9
 Valuation domain : [-1.00; 1.00]
 Attributes        : ['name', 'vertices', 'order', 'permutation',
                      'valuationDomain', 'edges', 'size', 'gamma']
>>> g.isPerfectGraph()
 True
>>> g.exportGraphViz()
 *---- exporting a dot file for GraphViz tools ---------*
 Exporting to permutationGraph.dot
 fdp -Tpng permutationGraph.dot -o permutationGraph.png
```

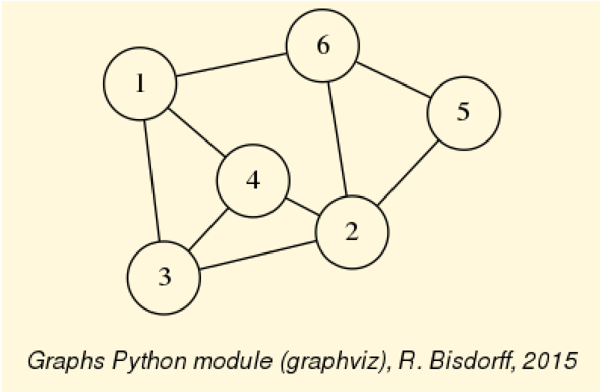


Fig. 4.11: The default permutation graph

By using color sorting queues, the minimal vertex coloring for a permutation graph is computable in $O(nlog(n))$ (see [GOL-2004]).

```
>>> g.computeMinimalVertexColoring(Comments=True)
 vertex 1: lightcoral
 vertex 2: lightcoral
 vertex 3: lightblue
 vertex 4: gold
 vertex 5: lightblue
 vertex 6: gold
>>> g.exportGraphViz(fileName='coloredPermutationGraph',
...                     WithVertexColoring=True)
 *---- exporting a dot file for GraphViz tools ---------*
  Exporting to coloredPermutationGraph.dot
  fdp -Tpng coloredPermutationGraph.dot -o coloredPermutationGraph.png
```

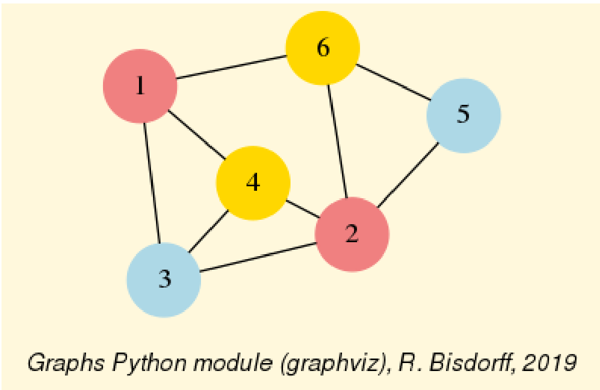


Fig. 4.12: Minimal vertex coloring of the permutation graph

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, 1, 5, 2], may as well be drawn with the graphviz *neato* layout and explicitly positioned horizontal lists of vertices (see Fig. 4.13).

```
>>> g.exportPermutationGraphViz(WithEdgeColoring=True)
*---- exporting a dot file for GraphViz tools ---------*
Exporting to perm_permutationGraph.dot
neato -n -Tpng perm_permutationGraph.dot -o perm_permutationGraph.png
```

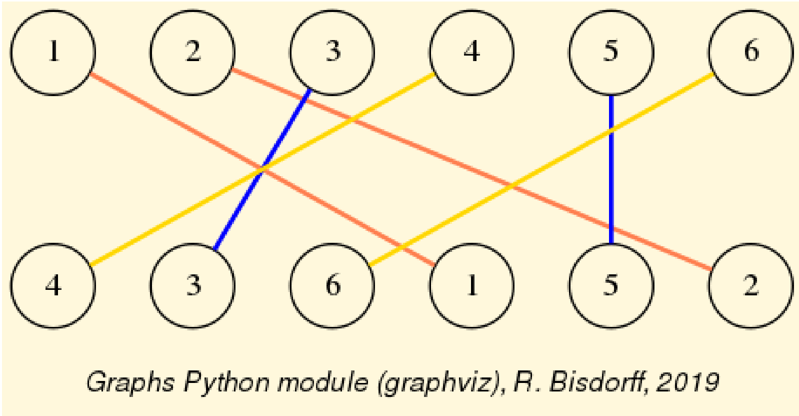


Fig. 4.13: Colored matching diagram of the permutation [4, 3, 6, 1, 5, 2]

As mentioned before, a permutation graph and its dual are **transitively orientable**. The `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.

```
>>> dg = g.transitiveOrientation()
>>> dg
*------- Digraph instance description ------*
Instance class   : TransitiveDigraph
Instance name    : oriented_permutationGraph
Digraph Order      : 6
Digraph Size       : 9
Valuation domain : [-1.00; 1.00]
Determinateness  : 100.000
Attributes       : ['name', 'order', 'actions', 'valuationdomain',
                    'relation', 'gamma', 'notGamma', 'size']
>>> print('Transitivity degree: %.3f' % dg.computeTransitivityDegree() )
Transitivity degree: 1.000
>>> dg.exportGraphViz()
*---- exporting a dot file for GraphViz tools ---------*
Exporting to oriented_permutationGraph.dot
0 { rank = same; 1; 2; }
1 { rank = same; 5; 3; }
2 { rank = same; 4; 6; }
dot -Grankdir=TB -Tpng oriented_permutationGraph.dot -o oriented_
↪permutationGraph.png
```

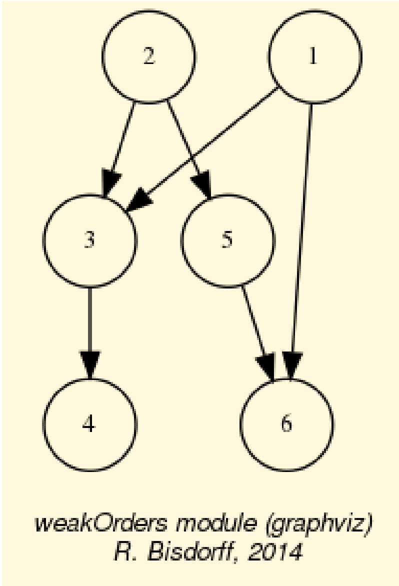


Fig. 4.14: Hasse diagram of the transitive orientation of the permutation graph

The dual of a permutation graph is *again* a permutation graph and as such also transitively orientable.

```
>>> dgd = (-g).transitiveOrientation()
>>> print('Dual transitivity degree: %.3f' %\
...            dgd.computeTransitivityDegree() )

 Dual transitivity degree: 1.000
```

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.

```
>>> from graphs import *
>>> g = RandomGraph(order=8,edgeProbability=0.4,seed=4335)
>>> g
 *------- Graph instance description ------*
 Instance class    : RandomGraph
 Instance name     : randomGraph
 Seed              : 4335
 Edge probability : 0.4
 Graph Order       : 8
 Graph Size        : 10
```

(continues on next page)

(continued from previous page)

```
Valuation domain : [-1.00; 1.00]
Attributes       : ['name', 'order', 'vertices', 'valuationDomain',
                    'seed', 'edges', 'size',
                    'gamma', 'edgeProbability']
>>> g.isPerfectGraph()
True
>>> g.exportGraphViz()
```

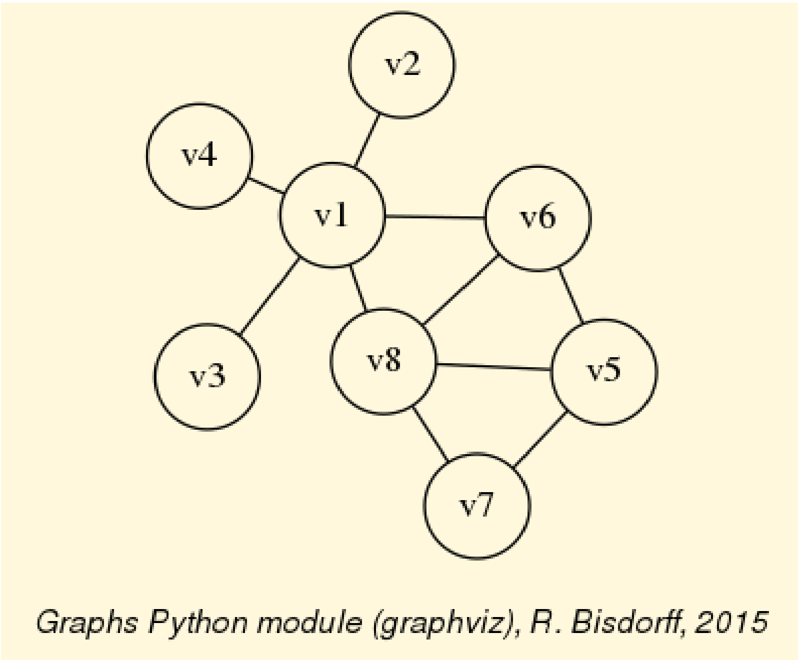


Fig. 4.15: Random graph of order 8 generated with edge probability 0.4

If the random perfect graph instance *g* (see Fig. 4.15) is indeed a permutation graph, *g* and its dual *-g* should be *transitively orientable*, i.e. **comparability graphs** (see [GOL-2004]). With the `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).

```
>>> if g.isComparabilityGraph():
...     print(g.edgeOrientations)

{('v1', 'v1'): 0, ('v1', 'v2'): 1, ('v2', 'v1'): -1, ('v1', 'v3'): 1,
 ('v3', 'v1'): -1, ('v1', 'v4'): 1, ('v4', 'v1'): -1, ('v1', 'v5'): 0,
 ('v5', 'v1'): 0, ('v1', 'v6'): 1, ('v6', 'v1'): -1, ('v1', 'v7'): 0,
 ('v7', 'v1'): 0, ('v1', 'v8'): 1, ('v8', 'v1'): -1, ('v2', 'v2'): 0,
 ('v2', 'v3'): 0, ('v3', 'v2'): 0, ('v2', 'v4'): 0, ('v4', 'v2'): 0,
 ('v2', 'v5'): 0, ('v5', 'v2'): 0, ('v2', 'v6'): 0, ('v6', 'v2'): 0,
 ('v2', 'v7'): 0, ('v7', 'v2'): 0, ('v2', 'v8'): 0, ('v8', 'v2'): 0,
 ('v3', 'v3'): 0, ('v3', 'v4'): 0, ('v4', 'v3'): 0, ('v3', 'v5'): 0,
```

(continues on next page)

(continued from previous page)

```
 ('v5', 'v3'): 0, ('v3', 'v6'): 0, ('v6', 'v3'): 0, ('v3', 'v7'): 0,
 ('v7', 'v3'): 0, ('v3', 'v8'): 0, ('v8', 'v3'): 0, ('v4', 'v4'): 0,
 ('v4', 'v5'): 0, ('v5', 'v4'): 0, ('v4', 'v6'): 0, ('v6', 'v4'): 0,
 ('v4', 'v7'): 0, ('v7', 'v4'): 0, ('v4', 'v8'): 0, ('v8', 'v4'): 0,
 ('v5', 'v5'): 0, ('v5', 'v6'): 1, ('v6', 'v5'): -1, ('v5', 'v7'): 1,
 ('v7', 'v5'): -1, ('v5', 'v8'): 1, ('v8', 'v5'): -1, ('v6', 'v6'): 0,
 ('v6', 'v7'): 0, ('v7', 'v6'): 0, ('v6', 'v8'): 1, ('v8', 'v6'): -1,
 ('v7', 'v7'): 0, ('v7', 'v8'): 1, ('v8', 'v7'): -1, ('v8', 'v8'): 0}
```

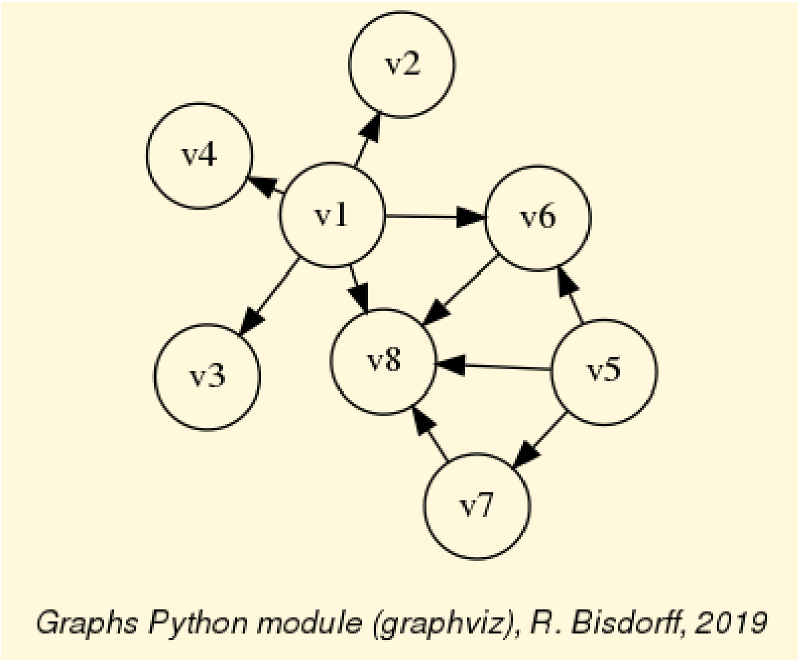


Fig. 4.16: Transitive neighbourhoods of the graph g

The resulting orientation of the edges of g (see Fig. 4.16) is indeed transitive. The same procedure applied to the dual graph gd = -g gives a transitive orientation to the edges of -g.

```
>>> gd = -g
>>> if gd.isComparabilityGraph():
...     print(gd.edgeOrientations)

{('v1', 'v1'): 0, ('v1', 'v2'): 0, ('v2', 'v1'): 0, ('v1', 'v3'): 0,
 ('v3', 'v1'): 0, ('v1', 'v4'): 0, ('v4', 'v1'): 0, ('v1', 'v5'): 1,
 ('v5', 'v1'): -1, ('v1', 'v6'): 0, ('v6', 'v1'): 0, ('v1', 'v7'): 1,
 ('v7', 'v1'): -1, ('v1', 'v8'): 0, ('v8', 'v1'): 0, ('v2', 'v2'): 0,
 ('v2', 'v3'): -2, ('v3', 'v2'): 2, ('v2', 'v4'): -3, ('v4', 'v2'): 3,
 ('v2', 'v5'): 1, ('v5', 'v2'): -1, ('v2', 'v6'): 1, ('v6', 'v2'): -1,
 ('v2', 'v7'): 1, ('v7', 'v2'): -1, ('v2', 'v8'): 1, ('v8', 'v2'): -1,
 ('v3', 'v3'): 0, ('v3', 'v4'): -3, ('v4', 'v3'): 3, ('v3', 'v5'): 1,
 ('v5', 'v3'): -1, ('v3', 'v6'): 1, ('v6', 'v3'): -1, ('v3', 'v7'): 1,
```

(continues on next page)

(continued from previous page)

```
('v7', 'v3'): -1, ('v3', 'v8'): 1, ('v8', 'v3'): -1, ('v4', 'v4'): 0,
('v4', 'v5'): 1, ('v5', 'v4'): -1, ('v4', 'v6'): 1, ('v6', 'v4'): -1,
('v4', 'v7'): 1, ('v7', 'v4'): -1, ('v4', 'v8'): 1, ('v8', 'v4'): -1,
('v5', 'v5'): 0, ('v5', 'v6'): 0, ('v6', 'v5'): 0, ('v5', 'v7'): 0,
('v7', 'v5'): 0, ('v5', 'v8'): 0, ('v8', 'v5'): 0, ('v6', 'v6'): 0,
('v6', 'v7'): 1, ('v7', 'v6'): -1, ('v6', 'v8'): 0, ('v8', 'v6'): 0,
('v7', 'v7'): 0, ('v7', 'v8'): 0, ('v8', 'v7'): 0, ('v8', 'v8'): 0}
```

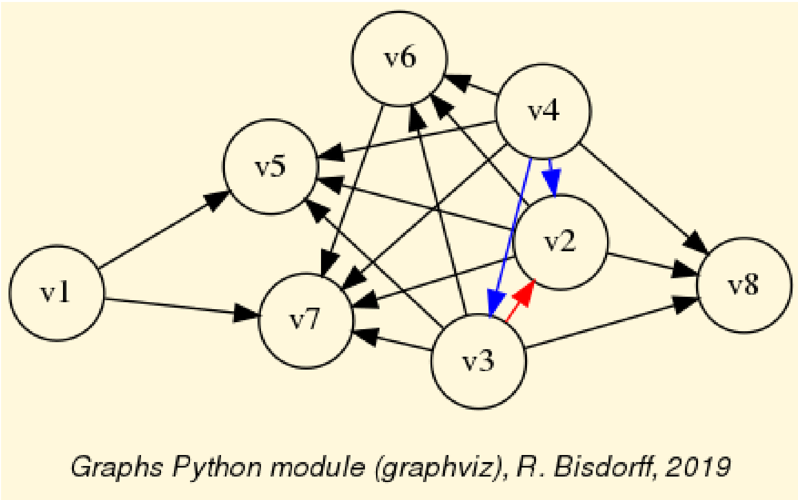


Fig. 4.17: Transitive neighbourhoods of the dual graph -*g*

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. 4.17).

Let us recheck these facts by explicitly constructing transitively oriented digraph instances with the `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 4) 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 `omax()` operator), hence, the partially determined orientations requested above.

Listing 4.3: 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 tutorial on *ranking with incommensurable criteria* (page 72) and the `computeCopelandRanking()` method) a linear ordering of the vertices (see Listing 4.3 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 
5 >>> print(permutation)
6  [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:

```
s1: ['v1', 'v4', 'v3', 'v2', 'v5', 'v6', 'v7', 'v8']
s2: ['v4', 'v3', 'v2', 'v8', 'v6', 'v1', 'v7', 'v5']
(s1 -> s2): [2, 3, 4, 8, 6, 1, 7, 5]
(s2 -> s1): [6, 1, 2, 3, 8, 5, 7, 4]
```

And:

```
s3: ['v5', 'v7', 'v1', 'v6', 'v8', 'v4', 'v3', 'v2']
s4: ['v8', 'v7', 'v6', 'v5', 'v4', 'v3', 'v2', 'v1']
(s3 -> s4): [5, 2, 4, 1, 6, 7, 8, 3]
(s4 -> s3) = [4, 2, 8, 3, 1, 5, 6, 7]
```

The `computePermutation()` method does directly operate all these steps: - computing transitive orientations, - ranking their epistemic fusion and, - delivering a corresponding permutation.

```
>>> g.computePermutation(Comments=True)
 ['v1', 'v2', 'v3', 'v4', 'v5', 'v6', 'v7', 'v8']
 ['v2', 'v3', 'v4', 'v8', 'v6', 'v1', 'v7', 'v5']
 [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.

```
>>> gtesta = PermutationGraph(permutation=[2, 3, 4, 8, 6, 1, 7, 5])
>>> gtestb = PermutationGraph(permutation=[4, 2, 8, 3, 1, 5, 6, 7])
>>> gtesta.exportGraphViz('gtesta')
>>> gtestb.exportGraphViz('gtestb')
```

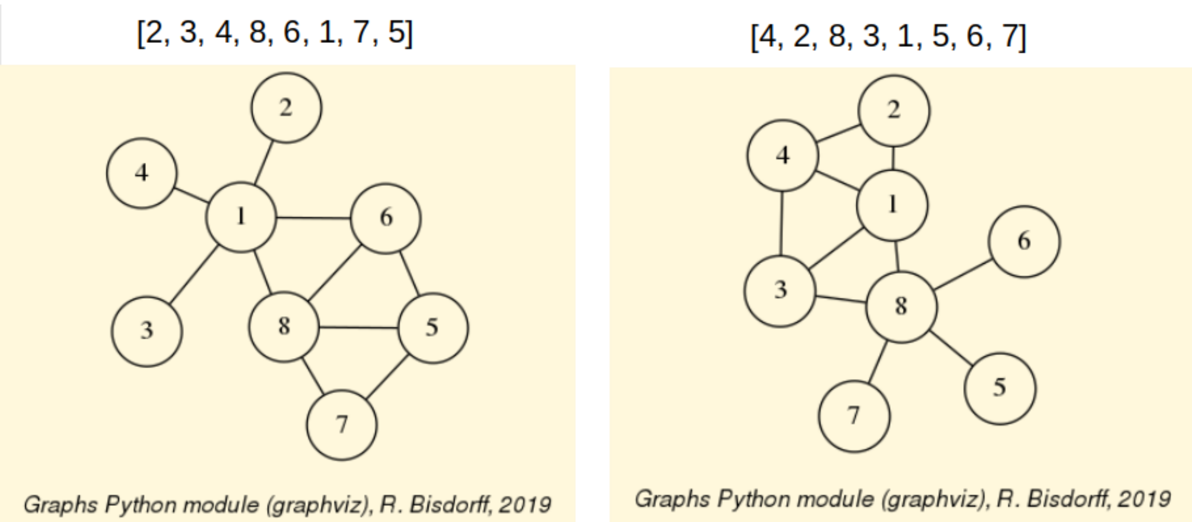


Fig. 4.18: Isomorphic permutation graphs

And, we recover indeed two *isomorphic copies* of the original random graph (compare Fig. 4.18 with Fig. 4.15).

Back to *Content Table* (page 1)

4.4 On computing fair intergroup pairings

The fair intergroup pairing problem

> **Fairness**: *impartial and just treatment or behaviour without favouritism or discrimination*
> – Oxford Languages

A set of persons consists of two groups –group A and group B– of equal size k. For a social happening, it is requested to build k pairs of persons from each group.

In order to guide the matching decisions, each person of group A communicates her pairing preferences with a linear ranking of the persons in group B and each person of group B communicates her pairing preferences with a linear ranking of the persons in group A.

The set of all potential matching decisions corresponds to the set of maximal matchings of the complete bipartite graph formed by the two groups A and B. Its cardinality is factorial k.

How to choose now in this possibly huge set the one maximal matching that makes a fair balance of the given individual pairing preferences? To help make this decision we will compute for all maximal matchings a fitness score consisting of their average ordinal correlation index with the given marginal pairing preferences. Eventually we will choose a maximal matching that results in the highest possible fitness score.

Let us consider for instance a set of four persons divided into group A, $\{a1, a2\}$, and group B, $\{b1, b2\}$. Person *a1* prefers as partner Person *b2*, and Person *a2* prefers as partner Person *b1*. Reciprocally, Person *b1* prefers Person *a2* over *a1* and Person *b2* finally prefers *a1* over *a2*. There exist only two possible maximal matchings,

(1) *a1* with *b1* and *a2* with *b2*, or

(2) *a1* with *b2* and *a2* with *b1*.

Making the best matching decision in this setting here is trivial. Choosing matching (1) will result in an ordinal correlation index of -1 for all four persons, whereas matching (2) is in total ordinal concordance with everybody's preferences and will result in an average ordinal correlation index of +1.0.

Can we generalise this approach to larger groups and partially determined ordinal correlation scores?

Reciprocal linear voting profiles

Let us consider two groups of size $k = 5$. Individual pairing preferences of the persons in group A and group B may be randomly generated with *reciprocal* `RandomLinearVotingProfile` instances called *lvA1* and *lvB1* (see below).

```
>>> from votingProfiles import RandomLinearVotingProfile
>>> k = 5
>>> lvA1 = RandomLinearVotingProfile(
...          numberOfVoters=k,numberOfCandidates=k,
...          votersIdPrefix='a',
...          candidatesIdPrefix='b',seed=1)
>>> lvA1.save('lvA1')
>>> lvB1 = RandomLinearVotingProfile(
...          numberOfVoters=k,numberOfCandidates=k,
...          votersIdPrefix='b',
...          candidatesIdPrefix='a',seed=2)
>>> lvB1.save('lvB1')
```

We may inspect the resulting stored pairing preferences for each person in group A and each person in group B with the `showLinearBallots()` method[49].

```
>>> from votingProfiles import LinearVotingProfile
>>> lvA1 = LinearVotingProfile('lvA1')
>>> lvA1.showLinearBallots()
 voters            marginal
(weight)      candidates rankings
 a1(1):       ['b3', 'b4', 'b5', 'b1', 'b2']
 a2(1):       ['b3', 'b5', 'b4', 'b2', 'b1']
 a3(1):       ['b4', 'b2', 'b1', 'b3', 'b5']
 a4(1):       ['b2', 'b4', 'b1', 'b5', 'b3']
 a5(1):       ['b4', 'b2', 'b3', 'b1', 'b5']
>>> lvB1 = LinearProfile('lvB1')
>>> lvB1.showLinearBallots()
 voters            marginal
(weight)      candidates rankings
 b1(1):       ['a3', 'a2', 'a4', 'a5', 'a1']
 b2(1):       ['a5', 'a3', 'a1', 'a4', 'a2']
```

(continues on next page)

[49] The stored versions *lvAx.py*, *lvBx.py*, *apA1.py* and *apB1.py* of the examples of reciprocal randdom voting profiles discussed in the intergroup pairing tutorial may be found in the *examples* directory of the *Digraph3* resources.

(continued from previous page)

```
b3(1):        ['a3', 'a4', 'a1', 'a5', 'a2']
b4(1):        ['a3', 'a4', 'a1', 'a2', 'a5']
b5(1):        ['a3', 'a4', 'a1', 'a2', 'a5']
```

With these given individual pairing preferences, there does no more exist a quick trivial matching solution to our pairing problem. Persons *a1* and *a2* prefer indeed to be matched to the same Person *b3*. Worse, Persons *b1*, *b3*, *b4* and *b5* all four want also to be preferably matched to a same Person *a3*, but Person *a3* apparently prefers as partner only Person *b4*.

How to find now a maximal matching that will fairly balance the individual pairing preferences of both groups? To solve this decision problem, we first must generate the potential decision actions, i.e. all potential maximal matchings between group *A* and group *B*.

Generating the set of potential maximal matchings

The maximal matchings correspond in fact to the maximal independent sets of edges of the complete bipartite graph linking group *A* to group *B*. To compute this set we will use the `CompleteBipartiteGraph` class from the `graphs` module (see Lines 3-4 below).

```
1  >>> groupA = [p for p in lvA1.voters]
2  >>> groupB = [p for p in lvB1.voters]
3  >>> from graphs import CompleteBipartiteGraph
4  >>> bpg = CompleteBipartiteGraph(groupA,groupB)
5  >>> bpg
6   *------- Graph instance description ------*
7    Instance class    : Graph
8    Instance name     : bipartitegraph
9    Graph Order       : 10
10   Graph Size        : 25
11   Valuation domain : [-1.00; 1.00]
12   Attributes        : ['name', 'vertices',
13                        'verticesKeysA', 'verticesKeysB',
14                        'order', 'valuationDomain',
15                        'edges', 'size', 'gamma']
```

Now, the maximal matchings of the bipartte graph *bpg* correspond to the MISs of its line graph *lbpg*. Therefore we use the `LineGraph` class from the `graphs` module.

```
>>> from graphs import LineGraph
>>> lbpg = LineGraph(bpg)
>>> lbpg
 *------- Graph instance description ------*
  Instance class    : LineGraph
  Instance name     : line-bipartite_completeGraph_graph
```

(continues on next page)

(continued from previous page)

```
7    Graph Order      : 25
8    Graph Size       : 100
9   >>> lbpg.computeMIS()
10  >>> lbpg.showMIS()
11   *---  Maximal Independent Sets ---*
12    number of solutions:  120
13    cardinality distribution
14    card.:  [0, 1, 2, 3, 4,  5,  6, 7, 8, 9, 10, ....]
15    freq.:  [0, 0, 0, 0, 0, 120, 0, 0, 0, 0,  0, ....]
16    stability number :  5
17    execution time: 0.01483 sec.
18    Results in self.misset
```

The set of maximal matchings between persons of groups A and B has cardinality *factorial* $5! = 120$ (see Line 15 above) and is stored in attribute *lbpg.misset*. We may for instance print the pairing corresponding to the first maximal matching.

```
>>> for m in lbpg.misset[0]:
...     pair = list(m)
...     pair.sort()
...     print(pair)
 ['a1', 'b4']
 ['a2', 'b3']
 ['a3', 'b5']
 ['a4', 'b2']
 ['a5', 'b1']
```

Each maximal matching delivers thus for each person a partially determined ranking. For Person *a1*, for instance, this matching ranks *b4* before all the other persons from group B and for Person *b4*, for instance, this matching ranks *a1* before all other persons from group A.

How to judge now the global pairing fitness of this matching?

Measuring the fitness of a matching from a personal perspective

Below we may reinspect the actual pairing preferences of each person.

```
>>> lvA1.showLinearBallots()
  voters            marginal
 (weight)     candidates rankings
  a1(1):       ['b3', 'b4', 'b5', 'b1', 'b2']
  a2(1):       ['b3', 'b5', 'b4', 'b2', 'b1']
  a3(1):       ['b4', 'b2', 'b1', 'b3', 'b5']
  a4(1):       ['b2', 'b4', 'b1', 'b5', 'b3']
  a5(1):       ['b4', 'b2', 'b3', 'b1', 'b5']
```

(continues on next page)

(continued from previous page)

```
>>> lvB1.showLinearBallots()
  voters             marginal
 (weight)      candidates rankings
  b1(1):       ['a3', 'a2', 'a4', 'a5', 'a1']
  b2(1):       ['a5', 'a3', 'a1', 'a4', 'a2']
  b3(1):       ['a3', 'a4', 'a1', 'a5', 'a2']
  b4(1):       ['a3', 'a4', 'a1', 'a2', 'a5']
  b5(1):       ['a3', 'a4', 'a1', 'a2', 'a5']
```

In the first matching shown in the previous Listing, Person *a1* is for instance matched with Person *b4*, which was her second choice. Whereas for Person *b4* the match with Person *a1* is only her third choice.

For a given person, we may hence compute the ordinal correlation –the relative number of correctly ranked persons minus the relative number of incorrectly ranked persons– between the partial ranking defined by the given matching and the individual pairing preferences, just ignoring the indeterminate comparisons.

For Person *a1*, for instance, the matching ranks *b4* before all the other persons from group *B* whereas *a1*'s individual preferences rank *b4* second behind *b3*. We observe hence 3 correctly ranked persons –*b5*, *b1* and *b2*– minus 1 incorrectly ranked person –*b3*– out of four determined comparisons. The resulting ordinal correlation index amounts to (3-1)/4 = +0.5.

For Person *b4*, similarly, we count 2 correctly ranked persons –*a2* and *a5*– and 2 incorrectly ranked persons –*a3* and *a4*– out of the four determined comparisons. The resulting ordinal correlation amounts hence to (2-2)/4 = 0.0

For a given maximal matching we obtain thus 10 ordinal correlation indexes, one for each person in both groups. And, we may now score the global fitness of a given matching by computing the average over all the individual ordinal correlation indexes observed in group *A* and group *B*.

Computing the fairest intergroup pairing

The `pairings` module provides the `FairestInterGroupPairing` class for solving, following this way, a given pairing problem of tiny order 5 (see below).

```
>>> from pairings import FairestInterGroupPairing
>>> fp = FairestInterGroupPairing(lvA1,lvB1)
>>> fp
 *------- FairPairing instance description ------*
  Instance class      : FairestInterGroupPairing
  Instance name       : pairingProblem
  Groups A and B size : 5
  Attributes          : ['name', 'order', 'vpA', 'vpB',
                         'pairings', 'matching',
```

(continues on next page)

(continued from previous page)

```
                              'vertices', 'valuationDomain',
                              'edges', 'gamma', 'runTimes']
```

The class takes as input two reciprocal `VotingProfile` objects describing the individual pairing preferences of the two groups A and B of persons. The class constructor delivers the attributes shown above. *vpA* and *vpB* contain the pairing preferences. The *pairings* attribute gathers all maximal matchings –the potential decision actions– ordered by decreasing average ordinal correlation with the individual pairing preferences, whereas the *matching* attribute delivers directly the first-ranked maximal matching –*pairings[0][0]*– and may be consulted as shown in the Listing below. The resulting *fp* object models in fact a `BipartiteGraph` object where the *vertices* correspond to the set of persons in both groups and the bipartite *edges* model the fairest maximal matching. The `showFairestPairing()` method prints out the fairest matching.

```
>>> fp.showFairestPairing(rank=1,
...    WithIndividualCorrelations=True)
 *-----------------------------*
 Fairest pairing
  ['a1', 'b3']
  ['a2', 'b5']
  ['a3', 'b1']
  ['a4', 'b4']
  ['a5', 'b2']
 groupA correlations:
  'a1': +1.000
  'a2': +0.500
  'a3':  0.000
  'a4': +0.500
  'a5': +0.500
 group A average correlations (a) : 0.500
 group A standard deviation       : 0.354
 ----
 groupB Correlations:
  'b1': +1.000
  'b2': +1.000
  'b3':  0.000
  'b4': +0.500
  'b5': -0.500
 group B average correlations (b) : 0.400
 group B standard deviation       : 0.652
 ----
 Average correlation    : 0.450
 Standard Deviation     : 0.497
 Unfairness |(a) - (b)| : 0.100
```

Three persons –*a1*, *b1* and *b2*– get as partner their first choice (+1.0). Four persons –*a2*, *a4*, *a5* and *b4*– get their second choice (+0.5). Two persons –*a3* and *b3*– get their third choice (0.0). Person *b5* gets only her fourth choice. Both group get very similar average ordinal correlation results – +0.500 versus +0.400– resulting in a low unfairness score (see last Line above)

In this problem we may observe a 2nd-ranked pairing, of same average correlation score +0.450, but with both a larger standard deviation (0.55 versus 0.45) and a larger unfairness score (0.300 versus 0.100).

```
>>> fp.showFairestPairing(rank=2,
...       WithIndividualCorrelations=True)
 *-----------------------------*
 2nd-ranked pairing
 ['a1', 'b3']
 ['a2', 'b5']
 ['a3', 'b4']
 ['a4', 'b1']
 ['a5', 'b2']
 group A correlations:
  'a1': +1.000
  'a2': +0.500
  'a3': +1.000
  'a4': +0.000
  'a5': +0.500
 group A average correlations (a) : 0.600
 group A standard deviation       : 0.418
 ----
 group B correlations:
  'b1': +0.000
  'b2': +1.000
  'b3': +0.000
  'b4': +1.000
  'b5': -0.500
 group B average correlations (b) : 0.300
 group B standard deviation       : 0.671
 ---
 Average correlation    : 0.450
 Standard Deviation     : 0.550
 Unfairness |(a) - (b)| : 0.300
```

In this second-fairest pairing solution, four persons –*a1*, *a3*, *b2* and *b4*– get their first choice. Only two persons –*a2* and *a5*– get their second choice, but three persons –*a4*, *b1* and *b3*– now only get their third choice. Person *b5* gets unchanged her fourth choice. Despite a same average correlation (+0.45), the distribution of the individual correlations appears less balanced than in the previous solution, as confirmed by the higher standard deviation. In the latter pairing, group A shows indeed an average correlation of +3.000/5 = +0.600, whereas group B obtains only an average correlation of 1.500/5 = +0.300.

In the previous pairing, group A gets a lesser average correlation of +0.500. And, group B obtains here a higher average correlation of 2.000/5 = +0.400. Which makes the first-ranked pairing with same average ordinal correlation yet lower standard deviation, an effectively fairer matching decision.

One may visualise a pairing result with the `exportPairingGraphViz()` method (see Fig. 4.19 below).

```
>>> fp.exportPairingGraphViz(fileName='fairPairing',
...                          matching=fp.matching)
 dot -Tpng fairPairing.dot -o fairPairing.png
```

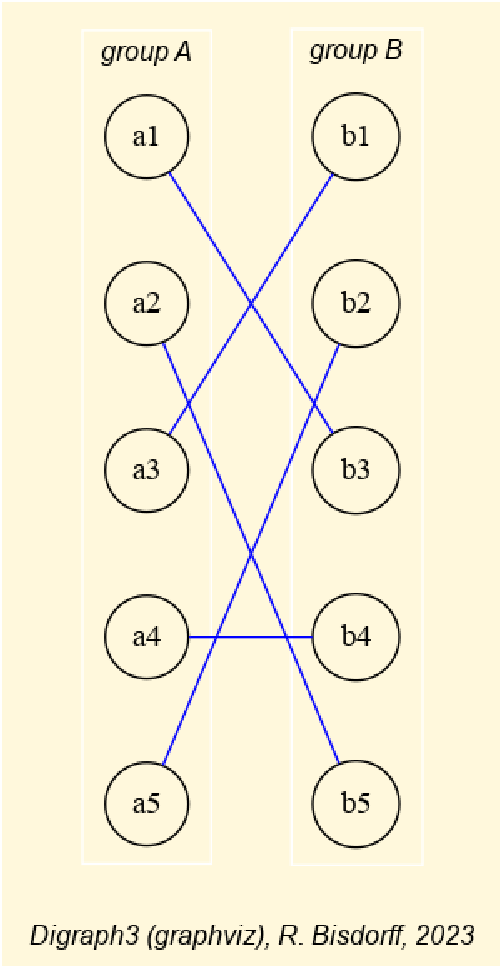


Fig. 4.19: Fairest intergroup pairing decision

A matching corresponds in fact to a certain permutation of the positional indexes of the persons in group B. We may compute this permutation and construct the corresponding permutation graph.

```
>>> permutation = fp.computePermutation(fp.matching)
>>> from graphs import PermutationGraph
>>> pg = PermutationGraph(permutation)
>>> pg
 *------- Graph instance description ------*
  Instance class    : PermutationGraph
```

(continues on next page)

(continued from previous page)

```
  Instance name     : matching-permutation
  Graph Order       : 5
  Permutation       : [3, 5, 1, 4, 2]
  Graph Size        : 6
  Valuation domain  : [-1.00; 1.00]
  Attributes        : ['name', 'vertices', 'order',
                       'permutation', 'valuationDomain',
                       'edges', 'size', 'gamma']
>>> pg.exportPermutationGraphViz(fileName='fairPairingPermutation')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to farPairingPermutation.dot
neato -n -Tpng fairPairingPermutation.dot -o fairPairingPermutation.png
```

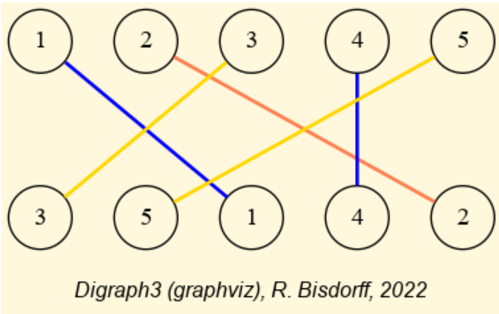


Fig. 4.20: Fairest pairing's coloured matching diagram

In Fig. 4.20 is shown the coloured matching diagram of the index permutation [3, 5, 1, 4, 2] modelled by the fairest pairing decision.

Mind that our `FairestInterGroupPairing` class does not provide an efficient algorithm for computing fair pairings; far from it. Our class constructor's complexity is in $O(k!)$, which makes the class totally unfit for solving any real pairing problem even of small size. The class has solely the didactic purpose of giving a first insight into this important and practically relevant decision problem. For efficiently solving this kind of pairing decision problems it is usual professional practice to concentrate the set of potential pairing decisions on *stable* matchings[45] .

Fair versus stable pairings

In classical economics, where the homo economicus is supposed to ignore any idea of fairness and behave solely in exact accordance with his rational self-interest, a pairing is only considered suitable when there appear no matching *instabilities*. A matching is indeed called *stable* when there does not exist in the matching a couple of pairs such that it may be interesting for both a paired person from group A and a paired person from group B to abandon their given partners and form together a new pair. Let us consider for instance the following situation,

[45] See https://en.wikipedia.org/wiki/Gale%E2%80%93Shapley_algorithm

Person *a3* is paired with Person *b1*.
Person *b4* is paired with Person *a4*.
Person *a3* would rather be with Person *b4*
Person *b4* would rather be with Person *a3*

Computing such a *stable* matching may be done with the famous *Gale-Shapley* algorithm ([43, Page 236, 45]), available via the `FairestGaleShapleyMatching` class (see below Line 1).

```
1  >>> from pairings import FairestGaleShapleyMatching
2  >>> fgs = FairestGaleShapleyMatching(lvA1,lvB1)
3  >>> fgs.showPairing(fgs.matching)
4   *-----------*
5      Pairing
6    ['a1', 'b3']
7    ['a2', 'b5']
8    ['a3', 'b4']
9    ['a4', 'b1']
10   ['a5', 'b2']
```

We have already seen this *Gale-Shapley* pairing solution. It is in fact the 2nd-ranked fairest pairing, discussed in the previous section. Now, is the fact of being *stable* any essential characteristic of a fair pairing solution?

In a Monte Carlo simulation of solving 1000 random pairing problems of order 5, we obtain the following distribution of the actual fairness ranking indexes of the fairest stable matching.

[43] [GAL-1962]

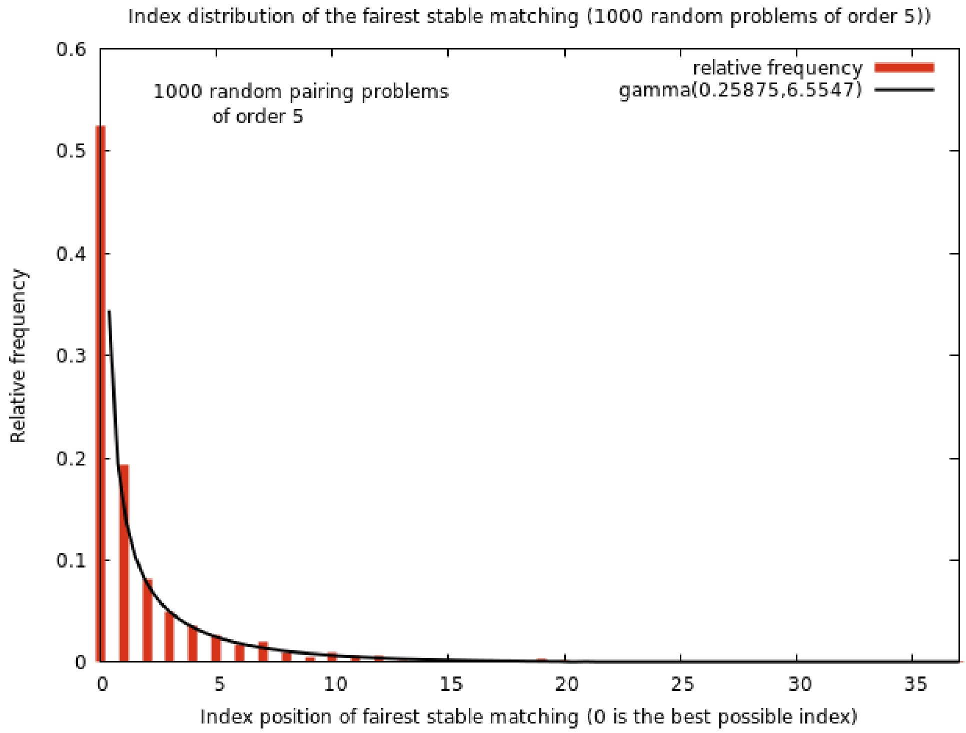


Fig. 4.21: Distribution of the fairness rank of the fairest stable matching

In Fig. 4.21 we may notice that only in a bit more than 50% of the cases, the overall fairest matching –of index 0 in the *fp.pairings* list– is indeed stable.

And the overall fairest matching in our example above is, for instance, *not* a stable matching (see Lines 2-3 below).

```
1  >>> fp.isStableMatching(fp.matching,Comments=True)
2   *-----------------------------*
3   ['a1', 'b3']
4   ['a2', 'b5']
5   ['a3', 'b1']
6   ['a4', 'b4']
7   ['a5', 'b2']
8     is unstable!
9   a3 b4 <-- b1: rank improvement 0 --> 2
10  b4 a3 <-- a4: rank improvement 0 --> 1
```

If we resolve its unstable pairs –[*a3*, *b1*] –> [*a3*, *b4*] , and [*a4*, *b4*] –> [*a4*, *b1*]– we recover the previous *Gale-Shapley* solution, i.e the 2nd-fairest pairing solution (see above).

Unfairness of the Gale-Shapley solution

The *Gale-Shapley* algorithm is actually based on an asymmetric handling of the two groups of persons by distinguishing a matches proposing group. In our implementation here[44], it is group A. Now, the proposing group gets by the *Gale-Shapley* algorithm the

[44] Our implementation is based on John Lekberg's blog. See https://johnlekberg.com/blog/2020-08-22-stable-matching.html

best possible average group correlation, but of costs of the non-proposing group who gets the worst possible average group correlation in any stable matching[Page 236, 45]. We may check as follows this unfair result on the previous *Gale-Shapley* solution.

```
>>> fgs.showMatchingFairness(fgs.matching,
...         WithIndividualCorrelations=True)
 *------------------------------*
 ['a1', 'b3']
 ['a2', 'b5']
 ['a3', 'b4']
 ['a4', 'b1']
 ['a5', 'b2']
 -----
 group A correlations:
  'a1': +1.000
  'a2': +0.500
  'a3': +1.000
  'a4': +0.000
  'a5': +0.500
 group A average correlations (a) : 0.600
 group A standard deviation       : 0.418
 -----
 group B correlations:
  'b1': +0.000
  'b2': +1.000
  'b3': +0.000
  'b4': +1.000
  'b5': -0.500
 group B average correlations (b) : 0.300
 group B standard deviation       : 0.671
 -----
 Average correlation    : 0.450
 Standard Deviation     : 0.550
 Unfairness |(a) - (b)| : 0.300
```

Four persons out of five from group A are matched to their first or second choices. When reversing the order of the given linear voting profiles *lvA1* and *lvB1*, we obtain a second *Gale-Shapley* solution *gs2* favouring this time the persons in group B.

```
>>> gs2 = fgs.computeGaleShapleyMatching(Reverse=True)
>>> fgs.showMatchingFairness(gs2,
...      WithIndividualCorrelations=True)
 *------------------------------*
 ['a1', 'b3']
 ['a2', 'b1']
 ['a3', 'b4']
 ['a4', 'b5']
 ['a5', 'b2']
```

(continues on next page)

(continued from previous page)

```
-----
group A correlations:
 'a1': +1.000
 'a2': -1.000
 'a3': +1.000
 'a4': -0.500
 'a5': +0.500
group A average correlations (a) : 0.200
group A standard deviation       : 0.908
-----
group B correlations:
 'b1': +0.500
 'b2': +1.000
 'b3': +0.000
 'b4': +1.000
 'b5': +0.500
group B average correlations (b) : 0.600
group B standard deviation       : 0.418
-----
Average correlation    : 0.400
Standard Deviation     : 0.699
Unfairness |(a) - (b)| : 0.400
```

In this reversed *Gale-Shapley* pairing solution, it is indeed the group B that appears now better served. Yet, it is necessary to notice now, besides the even more unbalanced group average correlations, the lower global average correlation (+0.400 compared to +0.450) coupled with both an even higher standard deviation (0.699 compared to 0.550) and a higher unfairness score (0.400 versus 0.300).

It may however also happen that both *Gale-Shapley* matchings, as well as the overall fairest one, are a same unique fairest pairing solution. This is for instance the case when considering the following example of reciprocal *lvA2* and *lvB2* profiles[Page 229, 49] .

```
>>> lvA2 = LinearVotingProfiles('lvA2')
>>> lvA2.showLinearBallots()
 voters            marginal
 (weight)     candidates rankings
 a1(1):       ['b1', 'b5', 'b2', 'b4', 'b3']
 a2(1):       ['b4', 'b3', 'b5', 'b2', 'b1']
 a3(1):       ['b3', 'b5', 'b1', 'b2', 'b4']
 a4(1):       ['b4', 'b2', 'b5', 'b3', 'b1']
 a5(1):       ['b5', 'b2', 'b3', 'b4', 'b1']
 # voters:  5
>>> lvB2 = LinearVotingProfile('lvB2')
>>> lvB2.showLinearBallots()
 voters            marginal
 (weight)     candidates rankings
```

(continues on next page)

(continued from previous page)

```
15 b1(1):        ['a1', 'a2', 'a5', 'a3', 'a4']
16 b2(1):        ['a2', 'a5', 'a3', 'a4', 'a1']
17 b3(1):        ['a3', 'a4', 'a1', 'a5', 'a2']
18 b4(1):        ['a4', 'a1', 'a2', 'a3', 'a5']
19 b5(1):        ['a2', 'a1', 'a5', 'a3', 'a4']
20 # voters:  5
21 >>> fp = FairestInterGroupPairing(lvA2,lvB2,StableMatchings=True)
22 >>> fp.showMatchingFairness()
23 *------------------------------*
24 ['a1', 'b1']
25 ['a2', 'b5']
26 ['a3', 'b3']
27 ['a4', 'b4']
28 ['a5', 'b2']
29 group A average correlations (a) : 0.700
30 group A standard deviation       : 0.447
31 group B average correlations (b) : 0.900
32 group B standard deviation       : 0.224
33 Average correlation    : 0.800
34 Standard Deviation     : 0.350
35 Unfairness |(a) - (b)| : 0.200
36 >>> print('Index of stable matchings:'. fp.stableIndex)
37 Index of stable matchings: [0]
```

In this case, the individual pairing preferences lead easily to the overall fairest pairing (see above). Indeed, three couples out of 5, namely [*a1*, *b1*], [*a3*, *b3*] and [*a4*, *b4*], do share their mutual first choices. For the remaining couples – [*a2*, *b5*] and [*a5*, *b2*]– the fairest matching gives them their third and first, respectively their first and second choice. Furthermore, their exists only one stable matching and it is actually the overall fairest one. When setting the *StableMatchings* flag of the `FairestInterGroupPairing` class to *True*, we get the *stableIndex* list with the actual index numbers of all stable maximal matchings (see Lines 19 and 34-35).

But the contrary may also happen. Below we show individual pairing preferences –stored in files *lvA3.py* and *lvB3.py*– for which the *Gale-Shapley* algorithm is not delivering a satisfactory pairing solutionPage 229, 49.

```
>>> from votingProfiles import LinearVotingProfile
>>> lvA3 = LinearVotingProfile('lvA3')
>>> lvA3.showLinearBallots()
voters              marginal
(weight)       candidates rankings
 a1(1):   ['b5', 'b6', 'b4', 'b3', 'b1', 'b2']
 a2(1):   ['b6', 'b1', 'b4', 'b5', 'b3', 'b2']
 a3(1):   ['b6', 'b3', 'b4', 'b1', 'b5', 'b2']
 a4(1):   ['b3', 'b4', 'b2', 'b6', 'b5', 'b1']
 a5(1):   ['b3', 'b4', 'b5', 'b1', 'b6', 'b2']
```

(continues on next page)

(continued from previous page)

```
 a6(1):  ['b3', 'b5', 'b1', 'b6', 'b4', 'b2']
  # voters:  6
>>> lvB3 = LinearVotingProfile('lvB3')
>>> lvB3.showLinearBallots()
 voters            marginal
 (weight)      candidates rankings
 b1(1):  ['a3', 'a4', 'a6', 'a1', 'a5', 'a2']
 b2(1):  ['a6', 'a4', 'a1', 'a3', 'a5', 'a2']
 b3(1):  ['a3', 'a2', 'a4', 'a1', 'a6', 'a5']
 b4(1):  ['a4', 'a2', 'a5', 'a6', 'a1', 'a3']
 b5(1):  ['a4', 'a2', 'a3', 'a6', 'a1', 'a5']
 b6(1):  ['a4', 'a3', 'a1', 'a5', 'a6', 'a2']
  # voters:  6
```

The individual pairing preferences are very contradictory. For instance, Person's *a2* first choice is *b6* whereas Person *b6* dislikes Person *a2* most. Similar situation is given with Persons *a5* and *b3*.

In this pairing problem there does exist only one matching which is actually stable and it is a very unfair pairing. Its fairness index is 140 (see Line 3-4 below).

```
1  >>> fp = FairestInterGroupPairing(lvA3,lvB3,
2  ...                    StableMatchings=True)
3  >>> fp.stableIndex
4   [140]
5  >>> g1 = fp.computeGaleShapleyMatching()
6  >>> fp.showMatchingFairness(g1,
7  ...               WithIndividualCorrelations=True)
8  *------------------------------*
9   ['a1', 'b1']
10  ['a2', 'b4']
11  ['a3', 'b6']
12  ['a4', 'b3']
13  ['a5', 'b2']
14  ['a6', 'b5']
15 ------
16 group A correlations:
17  'a1': -0.600
18  'a2': +0.200
19  'a3': +1.000
20  'a4': +1.000
21  'a5': -1.000
22  'a6': +0.600
23 group A average correlation (a) : 0.200
24 group A standard deviation      : 0.839
25 -----
26 group B correlations:
```

(continues on next page)

(continued from previous page)

```
27  'b1': -0.200
28  'b2': -0.600
29  'b3': +0.200
30  'b4': +0.600
31  'b5': -0.200
32  'b6': +0.600
33 group B average correlation (b) : 0.067
34 group B standard deviation      : 0.484
35 -----
36 Average correlation    : 0.133
37 Standard Deviation     : 0.657
38 Unfairness |(a) - (b)| : 0.133
```

Indeed, both group correlations are very weak and show furthermore high standard deviations. Five out of the twelve persons obtain a negative correlation with their respective pairing preferences. Only two persons from group A –*a3* and *a4*– get their first choice, whereas Person *a5* is matched with her least preferred partner (see Lines 19-21). In group B, no apparent attention is put on choosing interesting partners (see Lines 27-32).

The fairest matching looks definitely more convincing.

```
>>> fp.showMatchingFairness(fp.matching,
...     WithIndividualCorrelations=True)
*-----------------------------*
 ['a1', 'b6']
 ['a2', 'b5']
 ['a3', 'b3']
 ['a4', 'b2']
 ['a5', 'b4']
 ['a6', 'b1']
 -----
 group A correlations:
  'a1': +0.600
  'a2': -0.200
  'a3': +0.600
  'a4': +0.200
  'a5': +0.600
  'a6': +0.200
 group A average correlation (a) : 0.333
 group A standard deviation      : 0.327
 -----
 group B correlations:
  'b1': +0.200
  'b2': +0.600
  'b3': +1.000
  'b4': +0.200
  'b5': +0.600
```

(continues on next page)

(continued from previous page)

```
27  'b6': +0.200
28 group B average correlation (b) : 0.467
29 group B standard deviation      : 0.327
30 -----
31 Average correlation    : 0.400
32 Standard Deviation     : 0.319
33 Unfairness |(a) - (b)| : 0.133
```

Despite the very contradictory individual pairing preferences and a same unfairness score, only one person, namely *a2*, obtains here a choice in negative correlation with her preferences (see Line 13). The group correlations and standard deviations are furthermore very similar (lines 18 and 28).

The fairest solution is however far from being stable. With three couples of pairs that are potentially unstable, the first and stable unique *Gale-Shapley* matching is with its fairness index 140 indeed far behind many fairer pairing solutions (see below).

```
>>> fp.isStableMatching(fp.matching,Comments=True)
Unstable match:  Pair(groupA='a4', groupB='b2')
                 Pair(groupA='a5', groupB='b4')
  a4 b2 <-- b4
  b4 a5 <-- a4
Unstable match:  Pair(groupA='a2', groupB='b5')
                 Pair(groupA='a5', groupB='b4')
  a2 b5 <-- b4
  b4 a5 <-- a2
Unstable match:  Pair(groupA='a3', groupB='b3')
                 Pair(groupA='a1', groupB='b6')
  a3 b3 <-- b6
  b6 a1 <-- a3
```

How likely is it to obtain such an unfair *Gale-Shapley* matching? With our Monte Carlo simulation of 1000 random pairing problems of order 5, we may empirically check the likely fairness index of the fairest of both *Gale-Shapley* solutions.

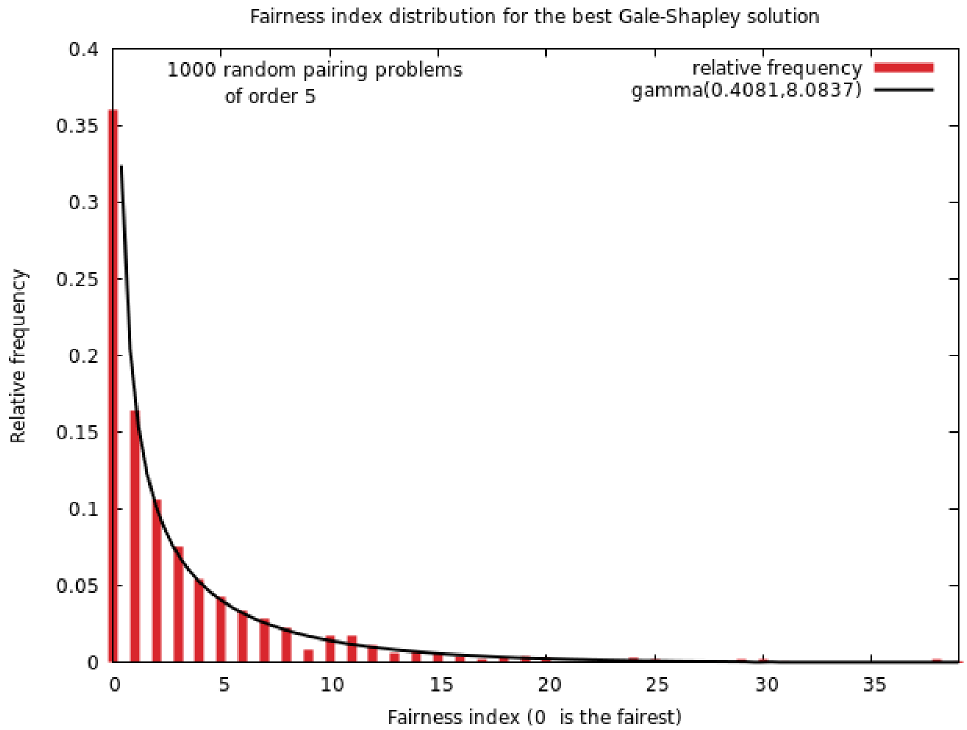


Fig. 4.22: Distribution of the fairness index of the fairest *Gale-Shapley* matching

In Fig. 4.22, we see that the fairest of both *Gale-Shapley* solutions will correspond to the overall fairest pairing (index $= 0$) in about *36%* out of the 1000 random cases. Yet, it is indeed the complexity in $O(k^2)$ of the *Gale-Shapley* algorithm that makes it an interesting alternative to our brute force approach in complexity $O(k!)$.

It is worthwhile noticing furthermore that the number of stable matchings is in general very small compared to the size of the huge set of potential maximal matchings as shown in Fig. 4.23.

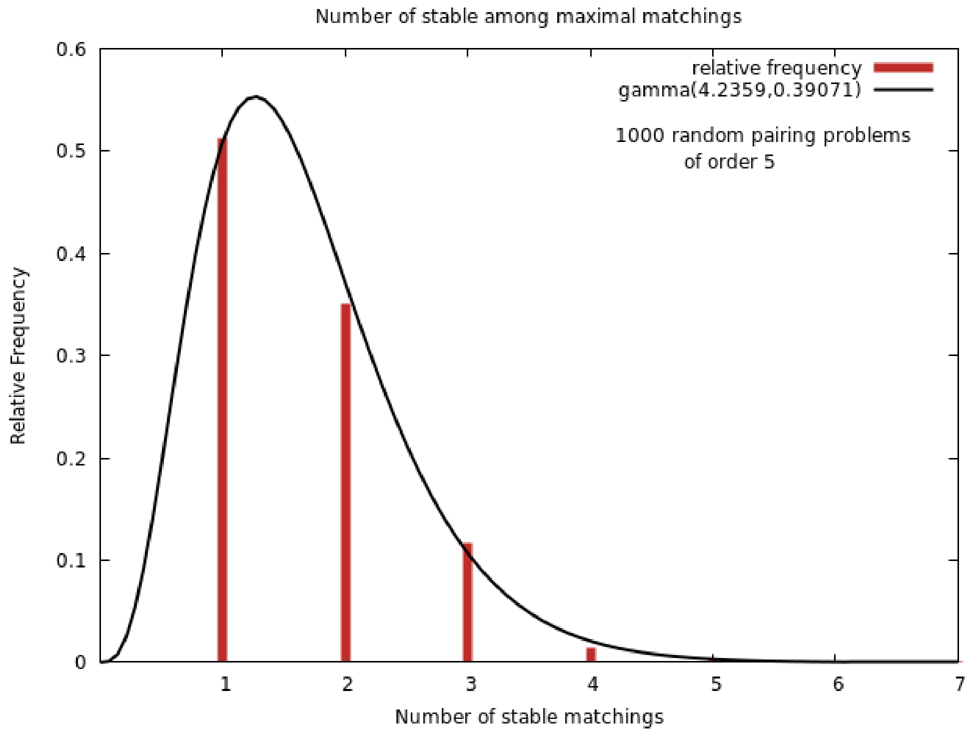


Fig. 4.23: Distribution of the number of stable matchings

In the simulation of 1000 random pairing problems of order 5, we observe indeed never more than seven stable matchings and the expected number of stable matchings is between one and two out of 120. It could therefore be opportune to limit our potential set of maximal matchings –the decisions actions– to solely stable matchings, as is currently the usual professional solving approach in pairing problems of this kind. Even if we would very likely miss the overall fairest pairing solution.

Dropping the stability requirement

Dropping however the *stability* requirement opens a second way of reducing the actual complexity of the fair pairing problem. This way goes by trying to enhance the fairness of a *Gale-Shapley* matching via a *hill-climbing* heuristic where we swap partners in couples of pairs that mostly increase the average ordinal correlation and decrease the gap between the groups' correlations.

With this strategy we may hence expect to likely reach one of the fairest possible matching solutions. In a Monte Carlo simulation of 1000 random pairing problems of order 6 we may indeed notice in Fig. 4.24 that we reach in a very limited number of swaps –less than $2 \times k$– a fairness index less than [3] in nearly 95% of the cases. The weakest fairness index found is 16.

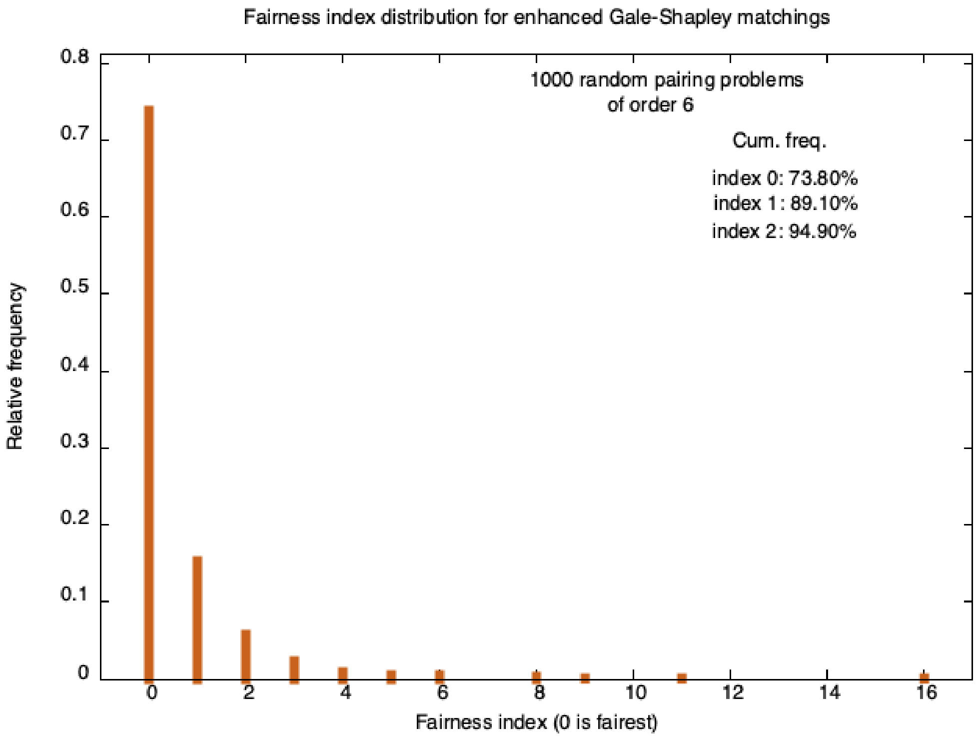


Fig. 4.24: Distribution of the fairness index of enhanced Gale-Shapley solutions

In the following example of a pairing problem of order 6, we observe only one unique stable matching with fairness index [12], in fact a very unfair *Gale-Shapley* matching completely ignoring the individual pairing preferences of the persons in group B (see Line 15 below).

```
1  >>> gs = FairestGaleShapleyMatching(lvA,lvB,
2  ...                         Comments=True)
3   Fairest Gale-Shapley matching
4   -----------------------------
5    ['a1', 'b3']
6    ['a2', 'b5']
7    ['a3', 'b4']
8    ['a4', 'b1']
9    ['a5', 'b6']
10   ['a6', 'b2']
11   -----
12   group A average correlation (a) : 0.867
13   group A standard deviation      : 0.327
14   -----
15   group B average correlation (b) : 0.000
16   group B standard deviation      : 0.704
17   -----
18   Average correlation    : 0.433
19   Standard Deviation     : 0.692
20   Unfairness |(a) - (b)| : 0.867
```

Taking this *Gale-Shapley* solution –*gs.matching*– as initial starting point, we try to swapp partners in couple of pairs in order to improve the average ordinal correlation with all the individual pairing preferences and to reduce the gap between both groups. The `pairings` module provides the `FairnessEnhancedInterGroupMatching` class for this purpose.

```
1 >>> from pairings import \
2 ...       FairnessEnhancedInterGroupMatching
3 >>> egs = FairnessEnhancedInterGroupMatching(
4 ...          lvA,lvB,initialMatching=gs.matching)
5 >>> egs.iterations
6  4
7 >>> egs.showMatchingFairness(egs.matching)
8  Fairness enhanced matching
9  --------------------------
10   ['a1', 'b3']
11   ['a2', 'b2']
12   ['a3', 'b4']
13   ['a4', 'b6']
14   ['a5', 'b5']
15   ['a6', 'b1']
16   -----
17   group A average correlation (a) : 0.533
18   group A standard deviation      : 0.468
19   -----
20   group B average correlation (b) : 0.533
21   group B standard deviation      : 0.641
22   -----
23   Average correlation    : 0.533
24   Standard Deviation     : 0.535
25   Unfairness |(a) - (b)| : 0.000
26 >>> fp = FairestInterGroupPairing(lvA,lvB)
27 >>> fp.computeMatchingFairnessIndex(egs.matching)
28  0
```

With a slightly enhanced overall correlation (+0.533 versus +0.433), both groups obtain after four swapping iterations the same group correlation of +0.533 (Unfairness score = 0.0, see Lines 17, 20 and 25 above). And, furthermore, the fairness enhancing procedure attains the fairest possible pairing solution (see last Line).

Our *hill-climbing* fairness enhancing algorithm seams hence to be quite efficient. Considering that its complexity is about $O(k^3)$, we are effectively able to solve pairing problems of realistic orders.

Do we really need to start the fairness enhancing strategy from a previously computed *Gale-Shapley* solution? No, we may start from any initial matching. This opens the way for taking into account more realistic versions of the individual pairing preferences than complete reciprocal linear voting profiles.

Relaxing the requirement for complete linear voting profiles

Partial individual pairing preferences

In the classical approach to the pairing decision problem, it is indeed required that each person communicates a complete linearly ordered list of the potential partners. It seams more adequate to ask for only partially ordered lists of potential partners. With the *PartialLinearBallots* flag and the *lengthProbability* parameter the `RandomLinearVotingProfile` class provides a random generator for such a kind of individual pairing preferences (see Lines 5-6 below).

```
1  >>> from votingProfiles import RandomLinearVotingProfile
2  >>> vpA = RandomLinearVotingProfile(
3  ...            numberOfVoters=7,numberOfCandidates=7,
4  ...            votersIdPrefix='a',candidatesIdPrefix='b',
5  ...            PartialLinearBallots=True,
6  ...            lengthProbability=0.5,
7  ...            seed=1)
8  >>> vpA.showLinearBallots()
9   voters          marginal
10 (weight)     candidates rankings
11  a1(1):      ['b4', 'b7', 'b6', 'b3', 'b1']
12  a2(1):      ['b7', 'b5', 'b2', 'b6']
13  a3(1):      ['b1']
14  a4(1):      ['b2', 'b3', 'b5']
15  a5(1):      ['b2', 'b1', 'b4']
16  a6(1):      ['b6', 'b7', 'b2', 'b3']
17  a7(1):      ['b7', 'b6', 'b1', 'b3', 'b5']
18 # voters:  7
```

With length probability of 0.5, we obtain here for the seven persons in group A the partial lists shown above. Person *a3*, for instance, only likes to be paired with Person *b1*, whereas Person *a4* indicates three preferred partners in decreasing order of preference (see Lines 13-14 above).

We may generate similar reciprocal partial linear voting profiles for the seven persons in group B.

```
>>> vpB = RandomLinearVotingProfile(
...           numberOfVoters=7,numberOfCandidates=7,
...           votersIdPrefix='b',
...           candidatesIdPrefix='a',
...           PartialLinearBallots=True,
...           lengthProbability=0.5,
...           seed=2)
>>> vpB.showLinearBallots()
 voters          marginal
(weight)     candidates rankings
 b1(1):      ['a3', 'a4']
```

(continues on next page)

(continued from previous page)

```
12  b2(1):      ['a3', 'a4']
13  b3(1):      ['a2', 'a6', 'a3', 'a1']
14  b4(1):      ['a2', 'a6', 'a4']
15  b5(1):      ['a2', 'a1', 'a5']
16  b6(1):      ['a2', 'a7']
17  b7(1):      ['a7', 'a2', 'a1', 'a4']
18 # voters:  7
```

This time, Persons *b1* and *b2* indicate only two preferred pairing partners, namely both times Person *a3* before Person *a4* (see Lines 11-12 above).

Yet, it may be even more effective to only ask for reciprocal **approvals** and **disapprovals** of potential pairing partners.

Reciprocal bipolar approval voting profiles

Such random *bipolar approval* voting profiles may be generated with the `RandomBipolarApprovalVotingProfile` class (see below).

```
>>> from votingProfiles import \
...      RandomBipolarApprovalVotingProfile
>>> k = 5
>>> apA1 = RandomBipolarApprovalVotingProfile(
...              numberOfVoters=k,
...              numberOfCandidates=k,
...              votersIdPrefix='a',
...              candidatesIdPrefix='b',
...              approvalProbability=0.5,
...              disapprovalProbability=0.5,
...              seed=None)
>>> apA1.save('apA1')
>>> apA1.showBipolarApprovals()
 Bipolar approval ballots
 ------------------------
 a1 :
 Approvals    : ['b1', 'b5']
 Disapprovals: ['b2']
 a2 :
 Approvals    : ['b2']
 Disapprovals: ['b1', 'b3', 'b4']
 a3 :
 Approvals    : []
 Disapprovals: ['b3', 'b5']
 a4 :
 Approvals    : ['b1', 'b5']
 Disapprovals: ['b2', 'b3', 'b4']
 a5 :
 Approvals    : ['b2', 'b3']
```

(continues on next page)

(continued from previous page)

```
30 Disapprovals: ['b1', 'b5']
31 Bipolar approval ballots
```

The *approvalProbability* and *disapprovalProbability* parameters determine the expected number of approved, respectively disapproved, potential pairing partners (see Lines 9-10). Person *a1*, for instance, approves two persons –*b1* and *b5*– and disapproves only Person *b2* (see Lines 17-18). Whereas Person *a3* does not approve anybody from group B, yet, disapproves *b3* and *b5*.

We may generate a similar random reciprocal bipolar approval voting profile for the persons in group B.

```
1 >>> apB1 = RandomBipolarApprovalVotingProfile(
2 ...             numberOfVoters=k,
3 ...             numberOfCandidates=k,
4 ...             votersIdPrefix='b',
5 ...             candidatesIdPrefix='a',
6 ...             approvalProbability=0.5,
7 ...             disapprovalProbability=0.5,
8 ...             seed=None)
9 >>> apB1.save('apB1')
10 >>> apB1.showBipolarApprovals()
11  Bipolar approval ballots
12  ------------------------
13  b1 :
14  Approvals    : ['a2', 'a3']
15  Disapprovals: ['a1', 'a4', 'a5']
16  b2 :
17  Approvals    : ['a1', 'a2']
18  Disapprovals: ['a4']
19  b3 :
20  Approvals    : ['a5']
21  Disapprovals: ['a2', 'a3']
22  b4 :
23  Approvals    : ['a2']
24  Disapprovals: ['a3', 'a5']
25  b5 :
26  Approvals    : ['a4']
27  Disapprovals: ['a1']
```

This time, Person *b1* approves two persons –*a2* and *a3*– and disapproves three persons –*a1*, *a4*, and *a5*– (see Lines 14-15 above).

Using Copeland scores for guiding the fairness enhancement

The partial linear voting profiles as well as the bipolar approval profiles determine for each person in both groups only a partial order on their potential pairing partners. In order to enhance the fairness of any given maximal matching, we must therefore replace the rank information of the complete linear voting profiles, as used in the *Gale-Shapley* algorithm, with the *Copeland* ranking scores obtained from the partial pairwise comparisons of potential partners. For this purpose we reuse again the `FairnessEnhancedInterGroupMatching` class , but without providing any initial matching (see below[Page 229, 49]).

```
1  >>> from pairings import \
2  ...         FairnessEnhancedInterGroupMatching
3  >>> from votingProfiles import BipolarApprovalVotingProfile
4  >>> apA1 = BipolarApprovalVotingProfile('apA1')
5  >>> apB1 = BipolarApprovalVotingProfile('apB1')
6  >>> fem = FairnessEnhancedInterGroupMatching(
7  ...            apA1,apB1,initialMatching=None,
8  ...            maxIterations=2*k,
9  ...            Comments=False)
10 >>> fem
11  *------- InterGroupPairing instance description ------*
12  Instance class      : FairnessEnhancedInterGroupMatching
13  Instance name       : fairness-enhanced-matching
14  Group sizes         : 5
15  Graph Order         : 10
16  Graph size          : 5
17  Partners swappings  : 5
18  Attributes          : ['runTimes', 'vpA', 'vpB',
19                'verticesKeysA', 'verticesKeysB', 'name',
20                'order', 'maxIterations', 'copelandScores',
21                'initialMatching', 'matching', 'iterations', 'history',
22                'maxCorr', 'stDev', 'groupAScores', 'groupBScores',
23                'vertices', 'valuationDomain', 'edges', 'size', 'gamma']
```

When no initial matching is given –*initialMatching = None*, which is the default setting– two initial matchings –the left matching (*ai*, *bi*) and the right matching (*ai*, *b-i*) for i = 1, . . . k– are used for starting the fairness enhancing procedure (see Line 7). The best solution of both is eventually retained. When the *initialMatching* parameter is set to *'random'*, a random shuffling –with given seed– of the persons in group B preceeds the construction of the right and left initial matchings. By default, the computation is limited to $2 \times k$ swappings of partners in order to master the potential occurrence of cycling situations. This limit may be adjusted if necessary with the *maxIterations* parameter (see Line 8). Such cycling swappings are furthermore controlled by the *history* attribute (see Line 21). The fairness enhanced *fem.matching* solution determines in fact a `BipartiteGraph` object (see last Line 23).

The actual pairing result obtained with the given bipolar approval ballots above is shown with the `showMatchingFairness()` method (see the Listing below). The *WithIndividu-*

alCorrelations flag allows to print out the inidividual pairing preference correlations for all persons in both groups (see Line 2).

```
1  >>> fem.showMatchingFairness(
2  ...                WithIndividualCorrelations=True)
3   *-----------------------------*
4   ['a1', 'b4']
5   ['a2', 'b2']
6   ['a3', 'b1']
7   ['a4', 'b5']
8   ['a5', 'b3']
9   -----
10  group A correlations:
11   'a1': -0.333
12   'a2': +1.000
13   'a3': +1.000
14   'a4': +1.000
15   'a5': +1.000
16  group A average correlation (a) : 0.733
17  group A standard deviation      : 0.596
18  -----
19  group B correlations:
20   'b1': +1.000
21   'b2': +1.000
22   'b3': +1.000
23   'b4': +0.333
24   'b5': +1.000
25  group B average correlation (b) : 0.867
26  group B standard deviation      : 0.298
27  -----
28  Average correlation    : 0.800
29  Standard Deviation     : 0.450
30  Unfairness |(a) - (b)| : 0.133
```

In group A and group B, all persons except *a1* and *b4* get an approved partner (see Lines 11 and 23). Yet, Persons *a1* and *b4* do not actually disapprove their respective match. Hence, the resulting overall ordinal correlation is very high (+0.800, see Line 28) and both groups show quite similar marginal correlation values (+0.733 versus +0.867, see Lines 16 and 25). The fairness enhanced matching we obtain in this case corresponds actually to the very fairest among all potential maximal matchings (see Lines 2-3 below).

```
1  >>> from pairings import FairestInterGroupPairing
2  >>> fp = FairestInterGroupPairing(apA1,apB1)
3  >>> fp.computeMatchingFairnessIndex(fem.matching)
4   0
```

Mind however that our fairness enhancing algorithm does not guarantee to end always in the very fairest potential maximal matching. In Fig. 4.25 is shown the result of a Monte Carlo simulation of 1000 random intergroup pairing problems of order 6 envolving bipolar approval voting profiles with approval, resp. disapproval probalities of 50%, resp. 20%. The failure rate to obtain the fairest pairing solution amounts to 12.4% with an average failure –optimal minus fairness enhanced average ordinal correlation– of -0.056 and a maximum failure of -0.292.

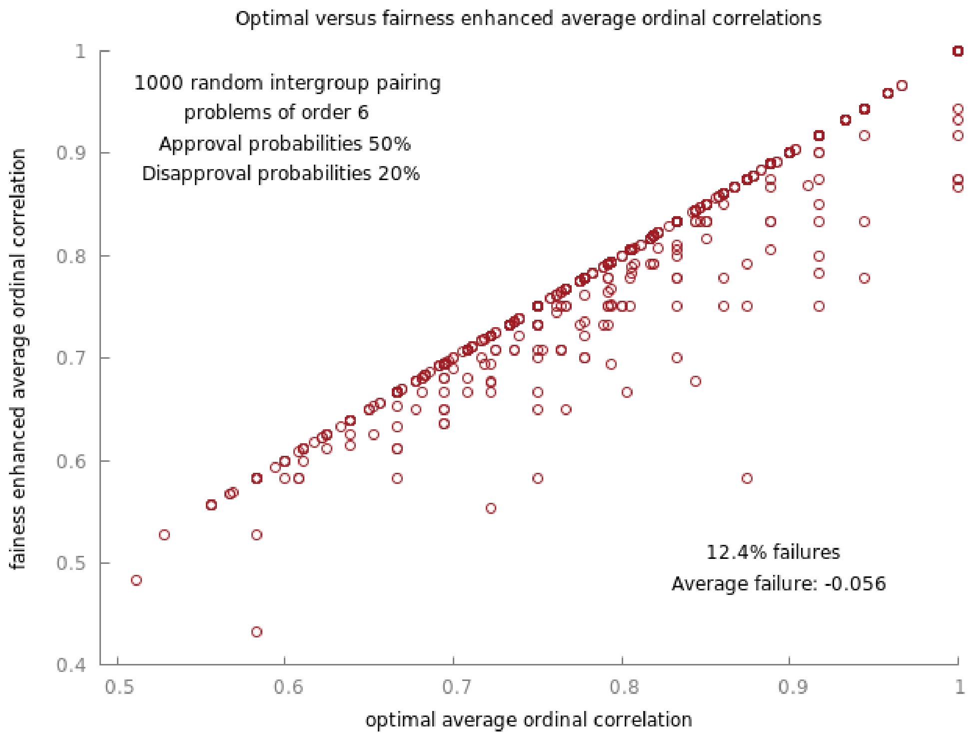


Fig. 4.25: Optimal versus fairness enhanced ordinal correlations

The proportion of failures depends evidently on the difficulty and the order of the pairing problem. We may however enhance the success rate of the fairness enhancing heuristic by choosing, like a Gale-Shapley stable in the case of linear voting profiles, a best determined *Copeland* ranking scores based initial matching.

Starting the fairness enhancement from a best determined Copeland matching

The partner swapping strategy relies on the *Copeland* ranking scores of a potential pairing candidate for all persons in bothe groups. These scores are precomputed and stored in the *copelandScores* attribute of the `FairnessEnhancedInterGroupMatching` object. When we add, for a pair $\{ai, bj\}$ both the *Copeland* ranking score of partner *bj* from the perspective of Person *ai* to the corresponding *Copeland* ranking score of partner *ai* from the perspective of Person *bj* to two times the observed minimal *Copeland* ranking score, we obtain a weakly determined complete bipartite graph object.

```
>>> from pairings import BestCopelandInterGroupMatching
>>> bcop = BestCopelandInterGroupMatching(apA1,apB1)
>>> bcop.showEdgesCharacteristicValues()
         |   'b1'    'b2'    'b3'    'b4'    'b5'
```

(continues on next page)

(continued from previous page)

```
5   --------|-----------------------------------------
6       'a1' |  +0.56   +0.44   +0.50   +0.50   +0.44
7       'a2' |  +0.56   +0.94   +0.19   +0.62   +0.62
8       'a3' |  +0.81   +0.56   +0.12   +0.44   +0.31
9       'a4' |  +0.56   +0.12   +0.44   +0.44   +0.94
10      'a5' |  +0.19   +0.62   +0.94   +0.31   +0.31
11   Valuation domain: [-1.00;1.00]
12  >>> bcop.showPairing()
13   *-----------------------------*
14   ['a1', 'b4']
15   ['a2', 'b2']
16   ['a3', 'b1']
17   ['a4', 'b5']
18   ['a5', 'b3']
```

By following a kind of ranked pairs rule, we may construct in this graph a best determined bipartite maximal matching. The matches [*a2*, *b2*], [*a4*, *b5*] and [*a5*, *b3*] show the highest Copeland scores (+0.94, see Lines 7,9-10), followed by [*a3*, *b1*] (+0.81 Line 6). For Person *a1*, the best eventually available partner is *b4* (+050, line 6).

We are lucky here with the given example of reciprocal bipolar approval voting profiles *apA1* and *apB1* as we recover immediately the fairest enhanced matching obtained previously. The best determined *Copeland* matching is hence very opportune to take as initial start for the fairness enhancing procedure as it may similarly drastically reduce the potential number of fairness enhancing partner swappings (see Lines 3 and last below).

```
1   >>> fecop = FairnessEnhancedInterGroupMatching(
2   ...                     apA1,apB1,
3   ...                     initialMatching='bestCopeland',
4   ...                     Comments=False)
5   >>> fecop.showPairing()
6    *-----------------------------*
7    ['a1', 'b4']
8    ['a2', 'b2']
9    ['a3', 'b1']
10   ['a4', 'b5']
11   ['a5', 'b3']
12  >>> fecop.Iterations
13   0
```

A Monte Carlo simulation with 1000 intergroup pairing problems of order 6 with approval and disapproval probabilities of 30% shows actually that both starting points –*initalMatching* = *None* and *initialMatching* = 'bestCopeland'– of the fairness enhancing heuristic may diverge positively and negatively in their respective best solutions.

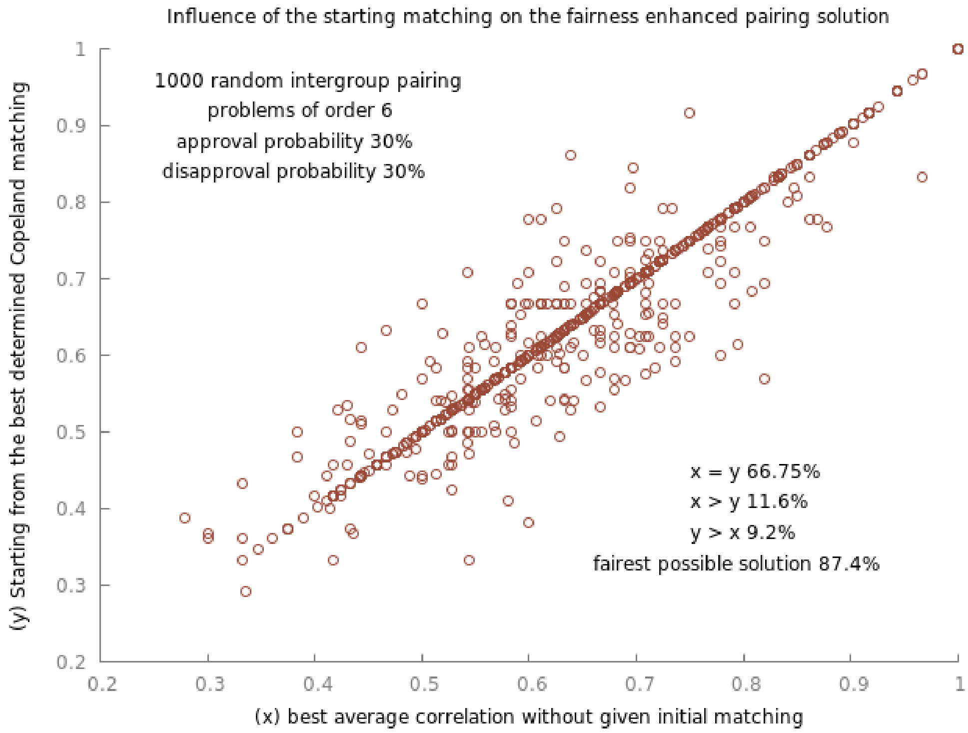


Fig. 4.26: Influence of the starting point on the fainess enhanced pairing solution

Discuss Fig. 4.26 fem 78.18% success rate fecop 75.78% success rate

If we run the fairness enhancing heuristic from both the left and right initial matchings as well as from the best determined Copeland matching and retain in fact the respective fairest solution of these three, we obtain, as shown in Fig. 4.27, a success rate of 87.39% for reaching the fairest possible pairing solution with an average failure of -0.036 and a maximum failure of -0.150.

Fig. 4.27: Optimal versus best fairness enhanced pairing solution

For intergroup pairing problems of higher order, it appears however that the best determined *Copeland* matching gives in general a more efficient initial starting point for the fairness enhancing heuristic than both the left and right initial ones. In a Monte Carlo simulation with 1000 random bipolar approval pairing problems of order 50 and approval-disapproval probabilities of 20%, we obtain the results shown below.

Variables	Mean	Median	S.D.	Min	Max
Correlation	+0.886	+0.888	0.018	+0.850	+0.923
Unfairness	0.053	0.044	0.037	0.000	0.144
Run time (sec.)	1.901	1.895	0.029	1.868	2.142

The median overall average correlation with the individual pairing preferences amounts to +0.886 with a maximum at +0.923. The *Unfairness* statistic indicates the absolute difference between the average correlations obtained in group A versus group B.

In order to study the potential difference in quality and fairness of the pairing solutions obtained by starting the fairness enhancing procedure from both the left and right inital matching, from the best determined *Copeland* matching as well as from the fairest *Gale-Shapley* we ran a Monte Carlo simulation with 1000 random intergroup pairing problems of order 20 and where the individual pairing preferences were given with complete linear voting profiles (see Fig. 4.28).

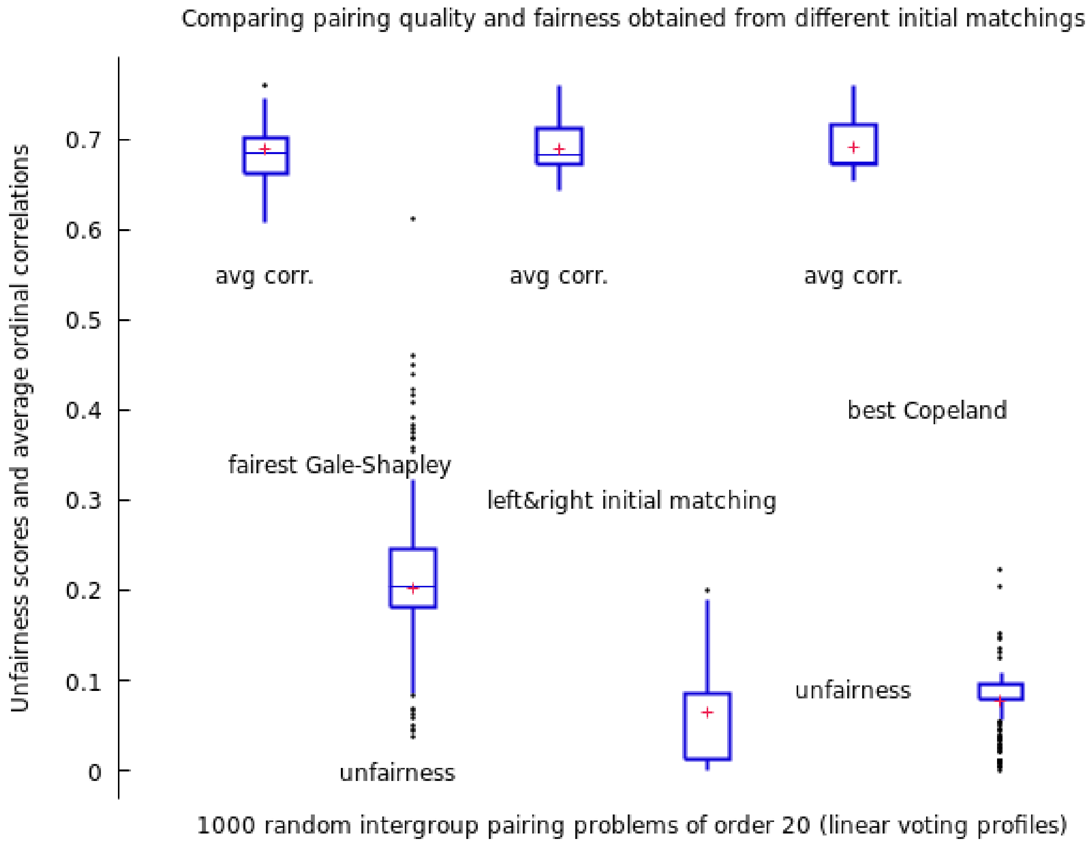


Fig. 4.28: Comparing pairing results from different fairnesss enhancing start points

If the average ordinal correlations obtained with the three starting matchings are quite similar –means within +0.690 and +0.693– the differences between the average correlations of group A and group B show a potential advantage for the left&right initial matchings (mean unfairness: 0.065) versus the best *Copeland* (mean unfairness: 0.078) and, even more versus the fairest *Gale-Shapley* matching (mean unfairness: 0.203, see Fig. 4.28). The essential unfairness of stable *Gale-Shapley* matchings may in fact not being corrected with our fairness enhancing procedure.

Back to *Content Table* (page 1)

4.5 On computing fair intragroup pairings

- *Fairness enhancing of a given pairing decision* (page 263)

The fair intragroup pairing problem

A very similar decision problem to the intergroup pairing one appears when, instead of pairing two different set of persons, we are asked to pair an even-sized set of persons by fairly balancing again the individual pairing preferences of each person.

Let us consider a set of four persons {*p1*, *p2*, *p3*, *p4*} to be paired. We may propose three potential pairing decisions :

(1) *p1* with *p2* and *p3* with *p4*,
(2) *p1* with *p3* and *p2* with *p4*, and
(3) *p1* with *p4* and *p2* with *p3*.

The individual pairing preferences, expressed under the format of bipolar approval ballots, are shown below:

```
1  Bipolar approval ballots
2  ------------------------
3  p1 :
4  Approvals    : ['p3', 'p4']
5  Disapprovals: ['p2']
6  p2 :
7  Approvals    : ['p1']
8  Disapprovals: ['p3']
9  p3 :
10 Approvals    : ['p1', 'p2', 'p4']
11 Disapprovals: []
12 p4 :
13 Approvals    : ['p2']
14 Disapprovals: ['p1', 'p3']
```

Person *p1*, for instance, approves as potential partner both Persons *p3* and *p4*, but disapproves Person *p2* (see Lines 3-5). Person *p3* approves all potential partners, i.e. disapproves none of them (see Lines 9-11).

Out of the three potential pairing decision, which is the one that most fairly balances the given individual pairing preferences shown above? If we take decision (1), Person *p1* will be paired with a disapproved partner. If we take decision (3), Person *p2* will be paired with a disapproved partner. Only pairing decision (2) allocates no disapproved partner to all the persons.

We will generalise this approach to larger groups of persons in a similar way as we do in the intergroup pairing case.

Generating random intragroup bipolar approval voting profiles

Let us consider a group of six persons. Individual intragroup pairing preferences may be randomly generated with the `RandomBipolarApprovalVotingProfile` class by setting the *IntraGroup* parameter to *True* (see Line 6 below)

```
1 >>> from votingProfiles import\
2 ...                    RandomBipolarApprovalVotingProfile
3 >>> vpG = RandomBipolarApprovalVotingProfile(
4 ...                         numberOfVoters=6,
5 ...                         votersIdPrefix='p',
6 ...                         IntraGroup=True,
7 ...                         approvalProbability=0.5,
8 ...                         disapprovalProbability=0.2,
9 ...                         seed=1)
10 >>> vpG.showBipolarApprovals()
11  Bipolar approval ballots
12  ------------------------
13  p1 :
14  Approvals    : ['p4', 'p5']
15  Disapprovals: []
16  p2 :
17  Approvals    : ['p1']
18  Disapprovals: ['p5']
19  p3 :
20  Approvals    : []
21  Disapprovals: ['p2']
22  p4 :
23  Approvals    : ['p1', 'p2', 'p3']
24  Disapprovals: ['p5']
25  p5 :
26  Approvals    : ['p1', 'p2', 'p3', 'p6']
27  Disapprovals: ['p4']
28  p6 :
29  Approvals    : ['p1', 'p2', 'p3', 'p4']
30  Disapprovals: []
```

With an approval probability of 50% and a disapproval probability of 20% we obtain the bipolar approvals shown above. Person *p1* approves *p4* and *p5* and disapproves nobody, whereas Person *p2* approves *p1* and disapproves *p5* (see Lines 14-15 and 17-18). To solve this intragroup pairing problem, we need to generate the set of potential matching decisions.

The set of potential ìntragroup pairing decisions

In the intergroup pairing problem, the potential pairing decisions are given by the maximal independent sets of the line graph of the bipartite graph formed between two even-sized groups of persons. Here the set of potential pairing decisions is given by the maximal independents sets –the perfect matchings[48]– of the line graph of the complete graph obtained from the given set of six persons (see below).

```
1  >>> persons = [p for p in vpG.voters]
2  >>> persons
3   ['p1', 'p2', 'p3', 'p4', 'p5', 'p6']
4  >>> from graphs import CompleteGraph, LineGraph
5  >>> cg = CompleteGraph(verticesKeys=persons)
6  >>> lcg = LineGraph(cg)
7  >>> lcg.computeMIS()
8  ... # result is stored into lcg.misset
9  >>> len(lcg.misset)
10  15
11 >>> lcg.misset[0]
12  frozenset({frozenset({'p5', 'p2'}),
13             frozenset({'p1', 'p6'}),
14             frozenset({'p3', 'p4'})})
```

In the intragroup case we observe 15 potential pairing decisions (see Line 10). For a set of persons of size $2 \times k$, the number of potential intragroup pairing decisions is actually given by the *double factorial of odd numbers*[47] .

$$1 \times 3 \times 5 \times ... \times (2 \times k - 1) \;=\; (2 \times k - 1)!!$$

For the first pair we have indeed $(2 \times k) - 1$ partner choices, for the second pair we have $(2 \times k) - 3$ partner choices, etc. This double factorial of odd numbers is far larger than the simple $k!$ number of potential pairing decisions in a corresponding intergroup pairing problem of order k.

In order to find now the fairest pairing among this potentially huge set of intragroup pairing decisions, we will reuse the same strategy as for the intergroup case. For each potential pairing solution, we are computing the average ordinal correlation between each potential pairing solution and the individual pairing preferences. The fairest pairing decision is eventually determined by the highest average coupled with the lowest standard deviation of the individual ordinal correlation indexes.

[48] A perfect matching is a saturated matching, i.e. a maximal matching which leaves no vertice unconnected.

[47] Integer sequence http://oeis.org/A001147

Computing the fairest intragroup pairing

For a pairing problem of tiny order ($k = 6$) we may use the `FairestIntraGroupPairing` class for computing in a brute force approach the fairest possible pairing solution :

```
1  >>> from pairings import FairestIntraGroupPairing
2  >>> fp = FairestIntraGroupPairing(vpG)
3  >>> fp.nbrOfMatchings
4   15
5  >>> fp.showMatchingFairness()
6   Matched pairs
7   {'p1', 'p4'}, {'p3', 'p5'}, {'p6', 'p2'}
8   ----
9   Individual correlations:
10   'p1': +1.000, 'p2': +0.000, 'p3': +1.000
11   'p4': +1.000, 'p5': +1.000, 'p6': +1.000
12  -----
13  Average correlation : +0.833
14  Unfairness (stdev)    : 0.408
```

As expected, we observe with a problem of order 6 a set of 1 x 3 x 5 = 15 potential pairings (see Line 4) and the fairest pairing solution –highest correlation (+0.833) with given individual pairing preferences– is shown in Line 7 above. All persons, except *p2* are paired with an approved partner and nobody is paired with a disapproved partner (see Lines 10-11).

In the intergroup pairing case, an indicator of the actual fairness of a pairing solution is given by the absolute difference between both group correlation values. In the intragroup case here, an indicator of the fairness is given by the standard deviation of the individual correlations (see Line 14). The lower this standard deviation with a same overall correlation result, the fairer appears to be in fact the pairing solution[50] .

The *fp* object models in fact a generic `Graph` object whose edges correspond to the fairest possible pairing solution (see Lines 11-12). We may hence produce in Fig. 4.29 a drawing of the fairest pairing solution by using the standard `exportGraphViz()` method for undirected graphs.

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

[50] The inter- and intragroup pairing solvers solely maximise the overall correlation with the individual pairing preferences. It may happen that a slightly lesser overall correlation result comes with a considerable lower standard deviation. Is this pairing solution than fairer than the one with a higher overall correlation? Asked more generally: is a society with highest global welfare but uneven wealth distribution a fairer society than the one showing less global welfare but with a considerable less uneven wealth distribution?

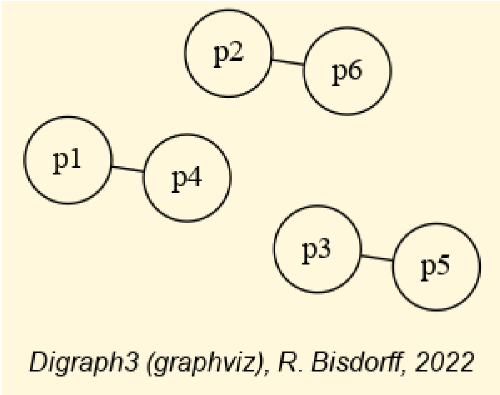


Fig. 4.29: Fairest intragroup pairing solution

Unfortunately, this brute force approach to find the fairest possible pairing solution fails in view of the explosive character of the double factorial of odd numbers. For a group of 20 persons, we observe indeed already more than 650 millions of potential pairing decisions. Similar to the intergroup pairing case, we may use instead a kind of hill climbing heuristic for computing a fair intragroup pairing solution.

Fairness enhancing of a given pairing decision

The `FairnessEnhancedIntraGroupMatching` class delivers such a solution. When no initial matching is given (see Line 3 below), our hill climbing strategy will start, similar to the intergroup pairing case, from two initial maximal matchings. The *left* one matches Person *pi* with Person *pi+1* for i in range 1 to 5 by step 3 (see Line 5-6) and the right one matches Person *pi* with Person *p-i* for i in range 1 to 3 (see Line 8-9).

```
1  >>> from pairings import FairnessEnhancedIntraGroupMatching
2  >>> fem = FairnessEnhancedIntraGroupMatching(vpG,
3  ...          initialMatching=None,Comments=True)
4   ===>>> Enhancing left initial matching
5   Initial left matching
6   [['p1', 'p2'], ['p3', 'p4'], ['p5', 'p6']]
7   Fairness enhanced left matching
8   [['p1', 'p4'], ['p3', 'p5'], ['p2', 'p6']] , correlation: 0.833
9   ===>>> Enhancing right initial matching
10  Initial right matching
11  [['p1', 'p6'], ['p3', 'p4'], ['p5', 'p2']]
12  Fairness enhanced right matching
13  [['p1', 'p4'], ['p3', 'p5'], ['p6', 'p2']] , correlation: 0.833
14  ===>>> Best fairness enhanced matching
15  Matched pairs
16  {'p1', 'p4'}, {'p2', 'p6'}, {'p3', 'p5'}
17  Average correlation: +0.833
```

The correlation enhancing search is similar to the one used for the intergroup heuristic. For each couple of pairs [{*pi*, *pj*}, {*pr*, *ps*}] in the respective initial matchings we have in the intragroup case in fact **two** partners swapping opportunities: (1) *pj* <-> *ps* or, (2) *pj* <-> *pr*. For both ways, we assess the expected individual correlation gains with the differences of the *Copeland* scores induced by the potential swappings. And we eventually proceed with a swapping of highest expected average correlation gain among all couple of pairs.

In the case of the previous bipolar approval intragroup voting profile *vpG*, both starting points for the hill climbing heuristic give the same solution, in fact the fairest possible pairing solution we have already obtained with the brute force algorithm in the preceding Section (see above).

To illustrate why starting from two initial matchings may be useful, we solve below a random intragroup pairing problem of order 20 where we assume an approval probability of 30% and a disapproval probability of 20% (see Line 3 below).

```
1  >>> vpG1 = RandomBipolarApprovalVotingProfile(
2  ...             numberOfVoters=20,votersIdPrefix='p',
3  ...             IntraGroup=True,approvalProbability=0.3,
4  ...             disapprovalProbability=0.2,seed=1)
5  >>> fem1 = FairnessEnhancedIntraGroupMatching(vpG1,
6  ...             initialMatching=None,Comments=True)
7   ===>>> Enhancing left initial matching
8   Initial left matching
9   [['p01', 'p02'], ['p03', 'p04'], ['p05', 'p06'], ['p07', 'p08'], ['p09',
   ↪ 'p10'],
10   ['p11', 'p12'], ['p13', 'p14'], ['p15', 'p16'], ['p17', 'p18'], ['p19',
   ↪ 'p20']]
11  Fairness enhanced left matching
12  [['p01', 'p02'], ['p03', 'p04'], ['p05', 'p15'], ['p06', 'p11'], ['p09',
   ↪ 'p17'],
13   ['p07', 'p12'], ['p13', 'p14'], ['p08', 'p16'], ['p20', 'p18'], ['p19',
   ↪ 'p10']],
14   correlation: +0.785
15  ===>>> Enhancing right initial matching
16  Initialright matching
17  [['p01', 'p20'], ['p03', 'p18'], ['p05', 'p16'], ['p07', 'p14'], ['p09',
   ↪ 'p12'],
18   ['p11', 'p10'], ['p13', 'p08'], ['p15', 'p06'], ['p17', 'p04'], ['p19',
   ↪ 'p02']]
19  Fairness enhanced right matching
20  [['p01', 'p19'], ['p03', 'p02'], ['p05', 'p15'], ['p07', 'p18'], ['p09',
   ↪ 'p17'],
21   ['p14', 'p13'], ['p10', 'p04'], ['p08', 'p12'], ['p20', 'p16'], ['p06',
   ↪ 'p11']],
22   correlation: +0.851
23  ===>>> Best fairness enhanced matching
24  Matched pairs
25  {'p01', 'p19'}, {'p03', 'p02'}, {'p05', 'p15'}, {'p06', 'p11'},
```

(continues on next page)

(continued from previous page)

```
26 {'p07', 'p18'}, {'p08', 'p12'}, {'p09', 'p17'}, {'p10', 'p04'},
27 {'p14', 'p13'}, {'p20', 'p16'}
28 Average correlation: +0.851
```

The hill climbing from the left initial matching attains an average ordinal correlation of +0.785, whereas the one starting from the right initial matching improves this result to an average ordinal correlation of +0.851 (see Lines 14 and 22).

We may below inspect with the `showMatchingFairness()` method the individual ordinal correlation indexes obtained this way.

```
1  >>> fem1.showMatchingFairness(WithIndividualCorrelations=True)
2  Matched pairs
3  {'p01', 'p19'}, {'p03', 'p02'}, {'p05', 'p15'},
4  {'p06', 'p11'}, {'p07', 'p18'}, {'p08', 'p12'},
5  {'p09', 'p17'}, {'p10', 'p04'}, {'p14', 'p13'},
6  {'p20', 'p16'}
7  ----
8  Individual correlations:
9   'p01': +1.000, 'p02': +1.000, 'p03': +1.000, 'p04': -0.143, 'p05': +1.
   ↪000,
10  'p06': +1.000, 'p07': +0.500, 'p08': -0.333, 'p09': +1.000, 'p10': +1.
   ↪000,
11  'p11': +1.000, 'p12': +1.000, 'p13': +1.000, 'p14': +1.000, 'p15': +1.
   ↪000,
12  'p16': +1.000, 'p17': +1.000, 'p18': +1.000, 'p19': +1.000, 'p20': +1.
   ↪000
13 -----
14 Average correlation : +0.851
15 Standard Deviation  :  0.390
```

Only three persons –*p04*, *p07* and *p08*– are not matched with a mutually approved partner (see Lines 9-10 above). Yet, they are all three actually matched with a partner they neither approve nor disapprove but who in return approves them as partner(see Lines 10, 19 and 27 below).

```
1  >>> vpG1.showBipolarApprovals()
2  Bipolar approval ballots
3  ------------------------
4  ...
5  ...
6  p04 :
7  Approvals    : ['p03', 'p12', 'p14', 'p19']
8  Disapprovals: ['p15', 'p18', 'p20']
9  p10 :
10 Approvals    : ['p04', 'p17', 'p20']
11 Disapprovals: ['p01', 'p02', 'p05', 'p06', 'p07', 'p08',
```

(continues on next page)

(continued from previous page)

```
                 'p09', 'p11', 'p12', 'p16', 'p18']
...
...
p07 :
Approvals    : ['p11']
Disapprovals: ['p01', 'p14', 'p19']
p12 :
Approvals    : ['p06', 'p07', 'p08', 'p10', 'p16', 'p19']
Disapprovals: ['p11', 'p14']
...
...
p08 :
Approvals    : ['p02', 'p05', 'p06', 'p14', 'p16', 'p19']
Disapprovals: ['p01', 'p13', 'p15']
p05 :
Approvals    : ['p01', 'p04', 'p06', 'p07', 'p08', 'p11', 'p15', 'p16',
↪'p18']
Disapprovals: ['p13', 'p19']
...
...
```

As the size of the potential maximal matchings with a pairing problem of order 20 exceeds 650 million instances, computing the overall fairest pairing solution becomes intractable and we are unable to check if we reached or not this optimal pairing solution. A Monte Carlo simulation with 1000 random intragroup pairing problem of order 8, applying an approval probability of 50% and a disapproval probability of 20%, shows however in Fig. 4.30 the apparent operational efficiency of our hill climbing heuristic, at least for small orders.

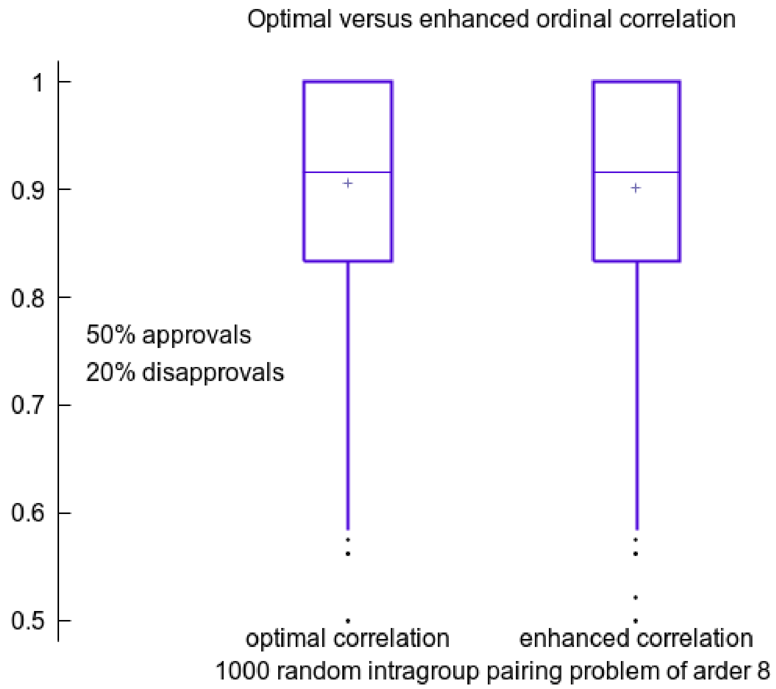


Fig. 4.30: Quality of fairness enhanced intragroup pairing solutions of order 8

Only 43 failures to reach the optimal average correlation were counted among the 1000 computations (4.3%) with a maximal difference in between of +0.250.

A similar simulation with more constrained random intragroup pairing problems of order 10, applying an approval and disapproval probability of only 30%, gives a failure rate of 19.1% to attain the optimal fairest pairing solution (see Fig. 4.31).

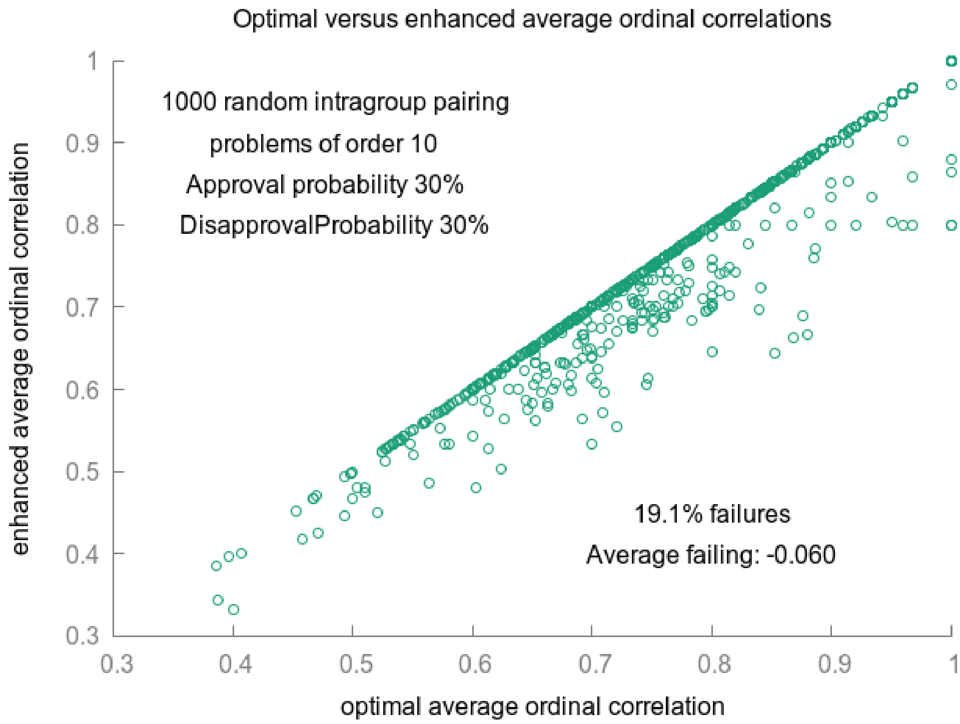


Fig. 4.31: Quality of fairness enhanced intragroup pairing solutions of order 10

Choosing, as in the intergroup pairing case, a more efficient initial matching for the fairness enhancing procedure becomes essential. For this purpose we may rely again on the best determined *Copeland* matching obtained with the pairwise *Copeland* scores computed on the complete intragroup graph. When we add indeed, for a pair $\{pi, pj\}$ both the *Copeland* ranking score of partner pj from the perspective of Person pi to the corresponding *Copeland* ranking score of partner pi from the perspective of Person pj we may obtain a complete positively valued graph object. In this graph we can, with a greedy ranked pairs rule, construct a best determined perfect matching which we may use as efficient initial start for the fairness enhancing heuristic (see below).

```
>>> from pairings import BestCopelandIntraGroupMatching
>>> cop = BestCopelandIntraGroupMatching(vpG1)
>>> cop.showPairing(cop.matching)
 Matched pairs
 {'p02', 'p15'}, {'p04', 'p03'}, {'p08', 'p05'}, {'p09', 'p20'}
 {'p11', 'p06'}, {'p12', 'p16'}, {'p14', 'p13'}, {'p17', 'p10'}
 {'p18', 'p07'}, {'p19', 'p01'}
>>> fem2 = FairnessEnhancedIntraGroupMatching(vpG1,
...                      initialMatching=cop.matching,Comments=True)
 *---- Initial matching ----*
 [['p02', 'p15'], ['p04', 'p03'], ['p08', 'p05'], ['p09', 'p20'],
  ['p11', 'p06'], ['p12', 'p16'], ['p14', 'p13'], ['p17', 'p10'],
  ['p18', 'p07'], ['p19', 'p01']]
 Enhancing iteration :  1
 Enhancing iteration :  2
```

(continues on next page)

(continued from previous page)

```
===>>> Best fairness enhanced matching
Matched pairs
{'p02', 'p04'}, {'p08', 'p05'}, {'p09', 'p20'},
{'p11', 'p06'}, {'p12', 'p16'}, {'p14', 'p13'},
{'p15', 'p03'}, {'p17', 'p10'}, {'p18', 'p07'},
{'p19', 'p01'}
Average correlation: +0.872
Total run time: 0.193 sec.
```

With the best determined *Copeland* matching we actually reach in two partner swappings a fairer pairing solution (+0.872) than the fairest one obtained with the default left and right initial matchings (+0.851). This is however not always the case. In order to check this issue, we ran a Monte Carlo experiment with 1000 random intragroup pairing problems of order 30 where approval and disapproval probabilities were set to 20%. Summary statistics of the results are shown in the Table below.

Variables	Mean	Median	S.D.	Min	Max
Correlation	+0.823	+0.825	0.044	+0.682	+0.948
Std deviation	0.361	0.362	0.051	0.186	0.575
Iterations	23.69	23.000	3.818	14.00	36.00
Run time	3.990	3.910	0.636	2.340	6.930

These statistics were obtained by trying both the left and right initial matchings as well as the best determined *Copeland* matching as starting point for the fairness enhancing procedure and keeping eventually the best average correlation result. The overall ordinal correlation hence obtained is convincingly high with a mean of +0.823, coupled with a reasonable mean standard deviation of 0.361 over the 30 personal correlations. Run times depend essentially on the number of enhancing iterations. On average, about 24 partner swappings were sufficient for computing all three variants in less than 4 seconds. In slightly more than two third only of the random pairing problems (69.4%), starting the fairness enhancing procedure from the best determined *Copeland* matching leads indeed to the best overall ordinal correlation with the individual pairing preferences.

When enhancing thus the fairness solely by starting from the best determined *Copeland* matching, we may solve with the `FairnessEnhancedIntraGroupMatching` solver in on average about 30 seconds an intragroup pairing problem of order 100 with random bipolar approval voting profiles and approval and disapproval probabilities of 10%. The average overall ordinal correlation we may obtain is about +0.800.

Mind however that the higher the order of the pairing problem, the more likely gets the fact that we actually may miss the overall fairest pairing solution. Eventually, a good expertise in metaheuristics is needed in order to effectively solve big intragroup pairing problems (*Avis aux amateurs*).

Back to *Content Table* (page 1)

4.6 On tree graphs and graph forests

Generating random tree graphs

Using the `RandomTree` class, we may, for instance, generate a random tree graph with 9 vertices.

```
>>> from graphs import RandomTree
>>> t = RandomTree(order=9,seed=100)
>>> t
 *------- Graph instance description ------*
 Instance class   : RandomTree
 Instance name    : randomTree
 Graph Order      : 9
 Graph Size       : 8
 Valuation domain : [-1.00; 1.00]
 Attributes       : ['name', 'order', 'vertices', 'valuationDomain',
                     'edges', 'prueferCode', 'size', 'gamma']
 *---- RandomTree specific data ----*
 Prüfer code  : ['v3', 'v8', 'v8', 'v3', 'v7', 'v6', 'v7']
>>> t.exportGraphViz('tutRandomTree')
 *---- exporting a dot file for GraphViz tools ---------*
 Exporting to tutRandomTree.dot
 neato -Tpng tutRandomTree.dot -o tutRandomTree.png
```

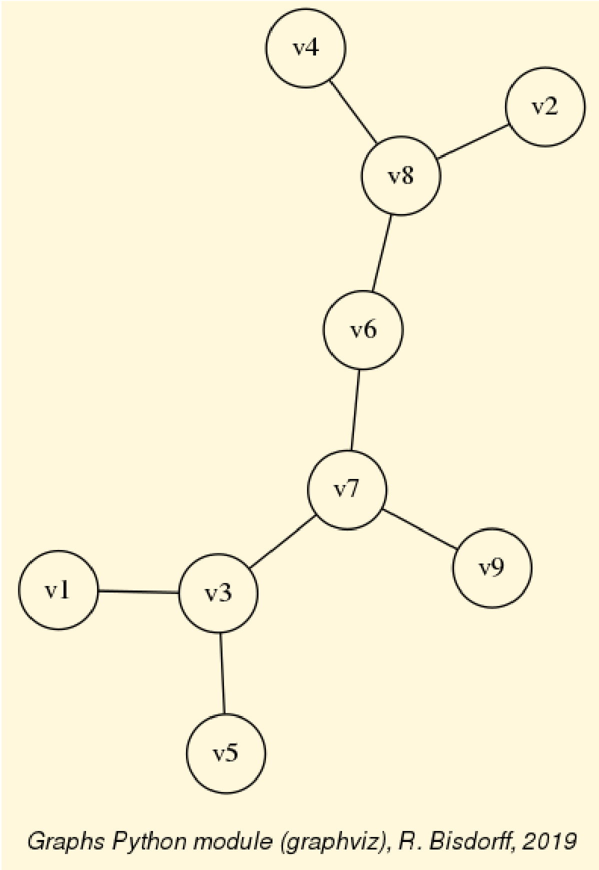


Fig. 4.32: Random Tree instance of order 9

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]).

```
>>> verticesList = ['v1','v2','v3','v4','v5','v6','v7']
>>> n = len(verticesList)
>>> import random
```

(continues on next page)

(continued from previous page)

```
>>> random.seed(101)
>>> code = []
>>> for k in range(n-2):
...     code.append( random.choice(verticesList) )

>>> print(code)
 ['v5', 'v7', 'v2', 'v5', 'v3']
>>> t = RandomTree(prueferCode=['v5', 'v7', 'v2', 'v5', 'v3'])
>>> t
 *------- Graph instance description ------*
 Instance class    : RandomTree
 Instance name     : randomTree
 Graph Order       : 7
 Graph Size        : 6
 Valuation domain : [-1.00; 1.00]
 Attributes        : ['name', 'order', 'vertices', 'valuationDomain',
                     'edges', 'prueferCode', 'size', 'gamma']
 *---- RandomTree specific data ----*
 Prüfer code   : ['v5', 'v7', 'v2', 'v5', 'v3']
>>> t.exportGraphViz('tutPruefTree')
 *---- exporting a dot file for GraphViz tools ---------*
 Exporting to tutPruefTree.dot
 neato -Tpng tutPruefTree.dot -o tutPruefTree.png
```

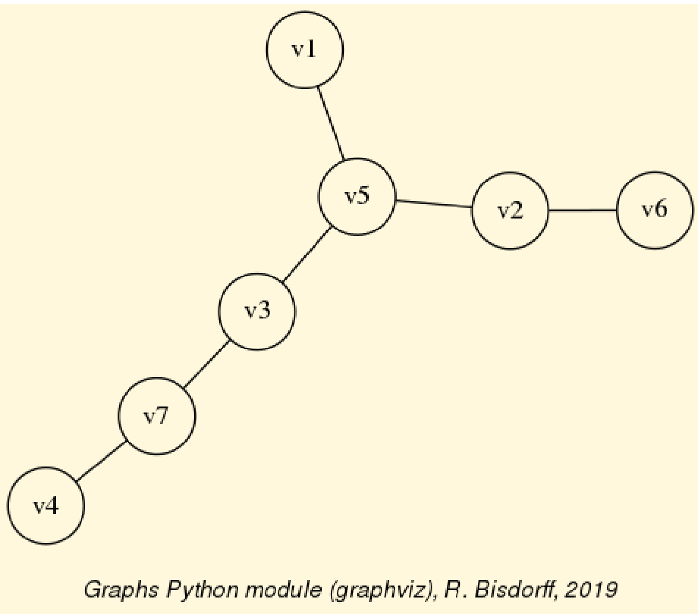


Fig. 4.33: Tree instance from a random Prüfer code

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 ?

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=8,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
8  The graph is a tree ? True
```

The random graph of order 8 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. 4.34 (see also the `isTree()` method for an implemented alternative test).

```
>>> g.exportGraphViz('test62')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to test62.dot
fdp -Tpng test62.dot -o test62.png
```

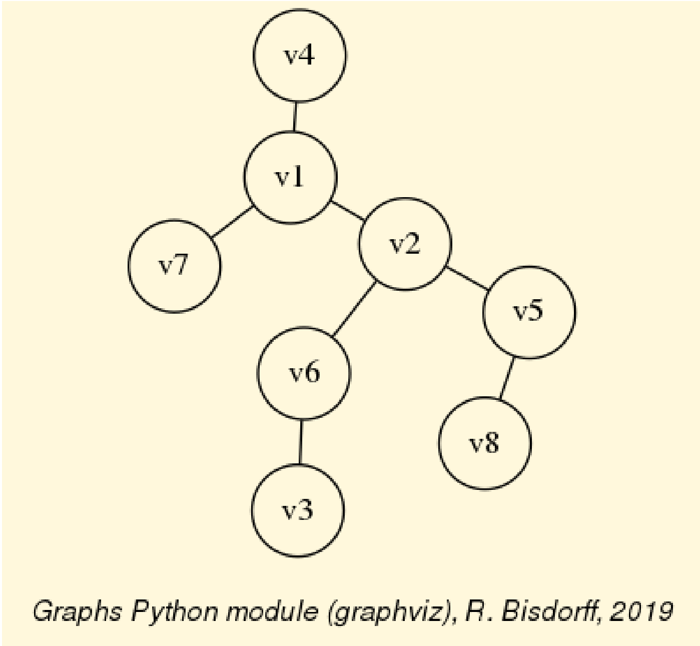


Fig. 4.34: Recognizing a tree instance

Yet, we still have to recover its corresponding *Prüfer* code. Therefore, we may use the `tree2Pruefer()` method.

```
>>> from graphs import TreeGraph
>>> g.__class__ = TreeGraph
>>> g.tree2Pruefer()
['v6', 'v1', 'v2', 'v1', 'v2', 'v5']
```

In Fig. 4.34 we also notice that vertex *v2* is actually situated in the **centre** of the tree with a neighborhood depth of 2. We may draw a correspondingly rooted and oriented tree graph.

```
>>> g.computeGraphCentres()
 {'v2': 2}
>>> g.exportOrientedTreeGraphViz(fileName='rootedTree',
...                              root='v2')
```

—- *exporting a dot file for GraphViz tools* ——— Exporting to rootedTree.dot dot -Grankdir=TB -Tpng rootedTree.dot -o rootedTree.png

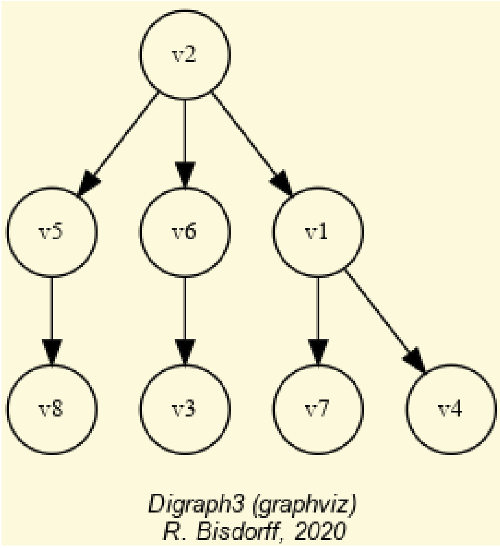


Fig. 4.35: Drawing an oriented tree rooted at its centre

Let us now turn our attention toward a major application of tree graphs, namely *spanning trees* and *forests* related to graph traversals.

Spanning trees and forests

With the `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].

```
>>> from graphs import *
>>> g = RandomGraph(order=9,edgeProbability=0.4,seed=100)
>>> spt = RandomSpanningTree(g)
>>> spt
 *------- Graph instance description ------*
 Instance class   : RandomSpanningTree
 Instance name    : randomGraph_randomSpanningTree
 Graph Order      : 9
 Graph Size       : 8
 Valuation domain : [-1.00; 1.00]
 Attributes       : ['name','vertices','order','valuationDomain',
                     'edges','size','gamma','dfs','date',
                     'dfsx','prueferCode']
 *---- RandomTree specific data ----*
```

(continues on next page)

[4] *Wilson*'s algorithm uses *loop-erased random walks*. See https://en.wikipedia.org/wiki/Loop-erased_random_walk .

(continued from previous page)

```
Prüfer code  : ['v7', 'v9', 'v5', 'v1', 'v8', 'v4', 'v9']
>>> spt.exportGraphViz(fileName='randomSpanningTree',
...                    WithSpanningTree=True)
*---- exporting a dot file for GraphViz tools ---------*
Exporting to randomSpanningTree.dot
[['v1', 'v5', 'v6', 'v5', 'v1', 'v8', 'v9', 'v3', 'v9', 'v4',
  'v7', 'v2', 'v7', 'v4', 'v9', 'v8', 'v1']]
neato -Tpng randomSpanningTree.dot -o randomSpanningTree.png
```

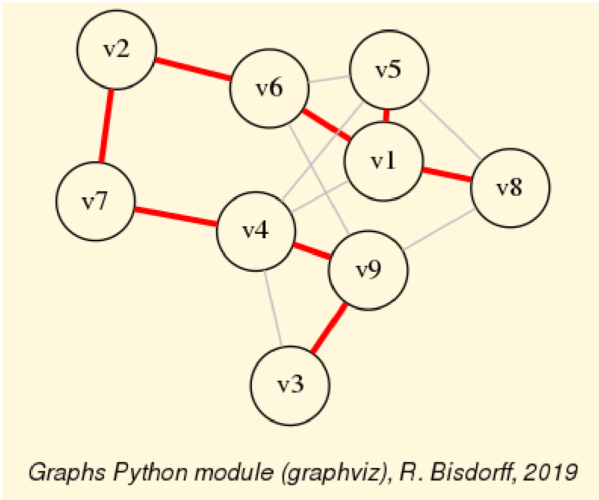


Fig. 4.36: Random spanning tree

More general, and in case of a not connected graph, we may generate with the `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.

```
>>> g = RandomGraph(order=15,edgeProbability=0.1,seed=140)
>>> g.computeComponents()
[{'v12', 'v01', 'v13'}, {'v02', 'v06'},
 {'v08', 'v03', 'v07'}, {'v15', 'v11', 'v10', 'v04', 'v05'},
 {'v09', 'v14'}]
>>> spf = RandomSpanningForest(g,seed=100)
>>> spf.exportGraphViz(fileName='spanningForest',WithSpanningTree=True)
*---- exporting a dot file for GraphViz tools ---------*
Exporting to spanningForest.dot
[['v03', 'v07', 'v08', 'v07', 'v03'],
 ['v13', 'v12', 'v13', 'v01', 'v13'],
 ['v02', 'v06', 'v02'],
 ['v15', 'v11', 'v04', 'v11', 'v15', 'v10', 'v05', 'v10', 'v15'],
 ['v09', 'v14', 'v09']]
neato -Tpng spanningForest.dot -o spanningForest.png
```

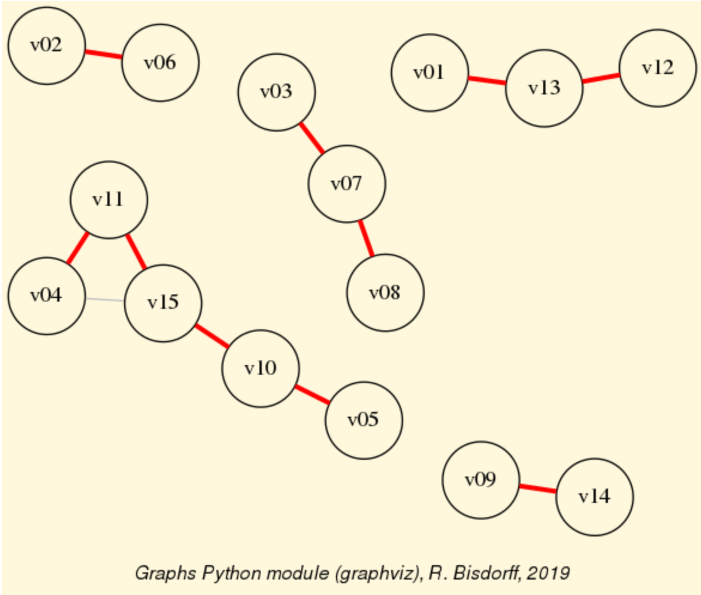


Fig. 4.37: Random spanning forest instance

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].

```
>>> from graphs import *
>>> g = RandomValuationGraph(seed=2)
>>> print(g)
 *------- Graph instance description ------*
 Instance class   : RandomValuationGraph
 Instance name    : randomGraph
 Graph Order      : 5
 Graph Size       : 7
 Valuation domain : [-1.00; 1.00]
 Attributes       : ['name', 'order', 'vertices', 'valuationDomain',
                     '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 `showRelationTable()` method (see Line 2 below) for printing its **symmetric adjacency matrix**.

[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 .

```
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 `BestDeterminedSpanningForest` class constructor.

```
>>> mt = BestDeterminedSpanningForest(g)
>>> print(mt)
 *------- Graph instance description ------*
 Instance class    : BestDeterminedSpanningForest
 Instance name     : randomGraph_randomSpanningForest
 Graph Order       : 5
 Graph Size        : 4
 Valuation domain : [-1.00; 1.00]
 Attributes        : ['name','vertices','order','valuationDomain',
                      'edges','size','gamma','dfs',
                      'date', 'averageTreeDetermination']
 *---- best determined spanning tree specific data ----*
 Depth first search path(s) :
 [['v1', 'v2', 'v4', 'v2', 'v5', 'v2', 'v1', 'v3', 'v1']]
 Average determination(s) : [Decimal('0.655')]
```

The given graph is connected and, hence, admits a single spanning tree (see Fig. 4.38) 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).

```
>>> mt.exportGraphViz(fileName='bestDeterminedspanningTree',
...                    WithSpanningTree=True)
 *---- exporting a dot file for GraphViz tools ---------*
 Exporting to spanningTree.dot
 [['v4', 'v2', 'v1', 'v3', 'v1', 'v2', 'v5', 'v2', 'v4']]
 neato -Tpng bestDeterminedSpanningTree.dot -o␣
 ↪bestDeterminedSpanningTree.png
```

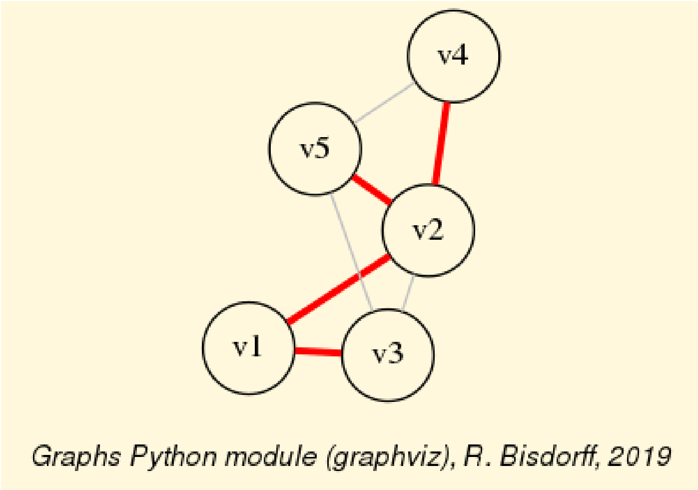


Fig. 4.38: Best determined spanning tree

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 *Content Table* (page 1)

---

5 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-2004a] 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))

[BIS-2004b] Bisdorff R. (2004) "Concordant Outranking with multiple criteria of ordinal significance". 4OR, *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, Springer-Verlag, Issue: Volume 2, Number 4, December 2004, Pages: 293 - 308. [ISSN: 1619-4500 (Paper) 1614-2411 (Online)] Electronic version: DOI: 10.1007/s10288-004-0053-7 (PDF file 137.1Kb for downloading (http://hdl.handle.net/10993/23721))

[BER-1963] Berge C (1963) Perfect graphs. Six Papers on Graph Theory. Calcutta: Indian Statistical Institute pp 1–21 333

[CHU-2006] Chudnovsky M, Robertson N, Seymour P, Robin T (2006) The strong perfect graph theorem. Annals of Mathematics 164(1):51–229 333

[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.

[GAL-1962] Gale D. and Shapley L. S. (1962). "College Admissions and the Stability of Marriage". *American Mathematical Monthly.* **69** (1): 9–14. doi:10.2307/2312726. JSTOR 2312726. Archived from the original on 2017-09-25. Retrieved 2019-11-20