# Tutorials for the Digraph3 Python software resources

*Release 3.7-3200+

Raymond Bisdorff

July 15, 2019

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Preface</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Working with the Digraph3 software resources</strong></td>
<td>4</td>
</tr>
<tr>
<td>2.1 Purpose</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Downloading of the Digraph3 resources</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Starting a Python3 session</td>
<td>5</td>
</tr>
<tr>
<td>2.4 Digraph object structure</td>
<td>5</td>
</tr>
<tr>
<td>2.5 Permanent storage</td>
<td>6</td>
</tr>
<tr>
<td>2.6 Inspecting a Digraph object</td>
<td>6</td>
</tr>
<tr>
<td>2.7 Special classes</td>
<td>8</td>
</tr>
<tr>
<td><strong>3 Manipulating Digraph objects</strong></td>
<td>8</td>
</tr>
<tr>
<td>3.1 Random digraph</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Graphviz drawings</td>
<td>11</td>
</tr>
<tr>
<td>3.3 Asymmetric and symmetric parts</td>
<td>12</td>
</tr>
<tr>
<td>3.4 Fusion by epistemic disjunction</td>
<td>13</td>
</tr>
<tr>
<td>3.5 Dual, converse and codual digraphs</td>
<td>13</td>
</tr>
<tr>
<td>3.6 Symmetric and transitive closures</td>
<td>14</td>
</tr>
<tr>
<td>3.7 Strong components</td>
<td>14</td>
</tr>
<tr>
<td>3.8 CSV storage</td>
<td>15</td>
</tr>
<tr>
<td>3.9 Complete, empty and indeterminate digraphs</td>
<td>16</td>
</tr>
<tr>
<td><strong>4 Computing the winner of an election</strong></td>
<td>18</td>
</tr>
<tr>
<td>4.1 Linear voting profiles</td>
<td>18</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.2 Computing the winner</td>
<td>19</td>
</tr>
<tr>
<td>4.3 The Condorcet winner</td>
<td>19</td>
</tr>
<tr>
<td>4.4 Cyclic social preferences</td>
<td>20</td>
</tr>
<tr>
<td>5 Working with the <code>outrankingDigraphs</code> module</td>
<td>22</td>
</tr>
<tr>
<td>5.1 Outranking digraph</td>
<td>22</td>
</tr>
<tr>
<td>5.2 Browsing the performances</td>
<td>23</td>
</tr>
<tr>
<td>5.3 Valuation semantics</td>
<td>25</td>
</tr>
<tr>
<td>5.4 Pairwise comparisons</td>
<td>25</td>
</tr>
<tr>
<td>5.5 Recoding the valuation</td>
<td>26</td>
</tr>
<tr>
<td>5.6 Codual digraph</td>
<td>26</td>
</tr>
<tr>
<td>5.7 XMCDA 2.0</td>
<td>27</td>
</tr>
<tr>
<td>6 Generating random performance tableaux</td>
<td>28</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>28</td>
</tr>
<tr>
<td>6.2 Generating standard random performance tableaux</td>
<td>28</td>
</tr>
<tr>
<td>6.3 Generating random Cost-Benefit tableaux</td>
<td>30</td>
</tr>
<tr>
<td>6.4 Generating three objectives tableaux</td>
<td>33</td>
</tr>
<tr>
<td>6.5 Generating random linearly ranked performances</td>
<td>36</td>
</tr>
<tr>
<td>7 Ranking with multiple incommensurable criteria</td>
<td>36</td>
</tr>
<tr>
<td>7.1 The ranking problem</td>
<td>37</td>
</tr>
<tr>
<td>7.2 The Copeland ranking</td>
<td>39</td>
</tr>
<tr>
<td>7.3 The Net-Flows ranking</td>
<td>39</td>
</tr>
<tr>
<td>7.4 Kemeny rankings</td>
<td>40</td>
</tr>
<tr>
<td>7.5 Slater rankings</td>
<td>42</td>
</tr>
<tr>
<td>7.6 Kohler’s ranking-by-choosing rule</td>
<td>42</td>
</tr>
<tr>
<td>7.7 Tideman’s Ranked-Pairs rule</td>
<td>43</td>
</tr>
<tr>
<td>7.8 Ranking big performance tableaux</td>
<td>44</td>
</tr>
<tr>
<td>8 HPC ranking with big outranking digraphs</td>
<td>45</td>
</tr>
<tr>
<td>8.1 C-compiled Python modules</td>
<td>46</td>
</tr>
<tr>
<td>8.2 Big Data performance tableaux</td>
<td>46</td>
</tr>
<tr>
<td>8.3 C-implemented integer-valued outranking digraphs</td>
<td>47</td>
</tr>
<tr>
<td>8.4 The sparse outranking digraph implementation</td>
<td>48</td>
</tr>
<tr>
<td>8.5 Ranking big sets of decision alternatives</td>
<td>51</td>
</tr>
<tr>
<td>8.6 HPC quantiles ranking records</td>
<td>53</td>
</tr>
<tr>
<td>9 Computing a best choice recommendation</td>
<td>57</td>
</tr>
<tr>
<td>9.1 What site to choose ?</td>
<td>58</td>
</tr>
<tr>
<td>9.2 Performance tableau</td>
<td>59</td>
</tr>
<tr>
<td>9.3 Outranking digraph</td>
<td>60</td>
</tr>
<tr>
<td>9.4 Rubis best choice recommendations</td>
<td>61</td>
</tr>
<tr>
<td>9.5 Rubis strict best choice recommendation</td>
<td>64</td>
</tr>
<tr>
<td>9.6 Weakly ordering</td>
<td>66</td>
</tr>
<tr>
<td>10 Rating with learned quantile norms</td>
<td>67</td>
</tr>
<tr>
<td>10.1 Introduction</td>
<td>67</td>
</tr>
<tr>
<td>10.2 Incremental learning of historical performance quantiles</td>
<td>68</td>
</tr>
<tr>
<td>10.3 Rating new performances with quantile norms</td>
<td>70</td>
</tr>
<tr>
<td>11 Working with the <code>graphs</code> module</td>
<td>74</td>
</tr>
<tr>
<td>11.1 Structure of a <code>Graph</code> object</td>
<td>76</td>
</tr>
<tr>
<td>11.2 q-coloring of a graph</td>
<td>78</td>
</tr>
<tr>
<td>11.3 MIS and clique enumeration</td>
<td>79</td>
</tr>
<tr>
<td>11.4 Line graphs and maximal matchings</td>
<td>80</td>
</tr>
<tr>
<td>11.5 Grids and the Ising model</td>
<td>82</td>
</tr>
<tr>
<td>11.6 Simulating Metropolis random walks</td>
<td>84</td>
</tr>
</tbody>
</table>
1 Preface

This PDF document contains a set of tutorials introducing the main objects like digraphs, outranking digraphs, performance tableaux and voting profiles available in the Digraph3 Python software resources (see html documentation here <https://digraph3.readthedocs.io/en/latest/index.html> for the corresponding HTML version).

Some of the tutorials are 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 ranking 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 in the n-cycle graph. A last tutorial is finally devoted on how to compute kernels in graphs, digraphs and more specifically in outranking digraphs.

Author Raymond Bisdorff, Emeritus Professor, University of Luxembourg FSTC/CSC
Version Revision: Python 3.7
Copyright R. Bisdorff (https://leopold-loewenheim.uni.lu/bisdorff/) 2013-2019
2 Working with the Digraph3 software resources

- Purpose (page 4)
- Downloading of the Digraph3 resources (page 5)
- Starting a Python3 session (page 5)
- Digraph object structure (page 5)
- Permanent storage (page 6)
- Inspecting a Digraph object (page 6)
- Special classes (page 8)

2.1 Purpose

The basic idea of the Digraph3 Python resources is to make easy python interactive sessions or write short Python3 scripts for computing all kind of results from a bipolar-valued digraph or graph. These include such features as maximal independent or irredundant choices, maximal dominant or absorbent choices, rankings, outrankings, linear ordering, etc. Most of the available computing resources are meant to illustrate the 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.
2.2 Downloading of the Digraph3 resources

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

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

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

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

2. Or, from sourceforge.net

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

3. or a subversion client

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

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

2.3 Starting a Python3 session

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

```
$HOME/.../Digraph3$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> ...
```

```python
>>> 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
Attributes : ['actions', 'valuationdomain', 'relation', 'order', 'name', 'gamma', 'notGamma']
>>> dg.save('tutorialDigraph')
--- Saving digraph in file: <tutorialDigraph.py> ---
>>> ...
```

2.4 Digraph object structure

All digraphs.Digraph objects contain at least the following attributes:
0. A **name** attribute, holding usually the actual name of the stored instance that was used to create the instance;

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

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

3. A logical characteristic **valuationdomain**, a dictionary with three decimal entries: the minimum (-1.0, means certainly false), the median (0.0, means missing information) and the maximum characteristic value (+1.0, means certainly true);

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

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

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

### 2.5 Permanent storage

The `dg.save('tutorialDigraph')` command (see Line 19 above) stores the digraph `dg` in a file named `tutorialDigraph.py` with the following content:

```python
# Saved digraph instance
actions = {
'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 = {'hasIntegerValuation': True, 'min': Decimal('-1.0'), 'med': Decimal('0.0'), 'max': Decimal('1.0')}
relation = {
'a1': {'a1':-1.0, 'a2':-1.0, 'a3':1.0, 'a4':-1.0, 'a5':-1.0},
'a2': {'a1':1.0, 'a2':-1.0, 'a3':-1.0, 'a4':1.0, 'a5':1.0},
'a3': {'a1':1.0, 'a2':1.0, 'a3':-1.0, 'a4':1.0, 'a5':1.0},
'a4': {'a1':1.0, 'a2':1.0, 'a3':1.0, 'a4':-1.0, 'a5':-1.0},
'a5': {'a1':1.0, 'a2':1.0, 'a3':1.0, 'a4':-1.0, 'a5':-1.0},
}
```

### 2.6 Inspecting a Digraph object

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

```python
>>> dg = Digraph('tutorialDigraph')
>>> dg.showAll()
***** show detail ******
Digraph : tutorialDigraph
***** Actions *****
['a1', 'a2', 'a3', 'a4', 'a5']
***** Characteristic valuation domain *****
{'hasIntegerValuation': True, 'min': Decimal('-1.0'), 'med': Decimal('0.0'), 'max': Decimal('1.0')}
***** Relation Table *****
S | 'a1' 'a2' 'a3' 'a4' 'a5'
```


<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>-1</th>
<th>1</th>
<th>-1</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a1'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'a2'</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>'a3'</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>'a4'</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>'a5'</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Valuation domain: \([-1;+1]\)

*--- Connected Components ---*

1: ['a1', 'a2', 'a3', 'a4', 'a5']

Neighborhoods:

**Gamma:**

- 'a1': in => {'a2', 'a4', 'a3', 'a5'}, out => {'a3'}
- 'a2': in => {'a5', 'a4'}, out => {'a1', 'a4', 'a5'}
- 'a3': in => {'a1', 'a4'}, out => {'a1', 'a4', 'a4'}
- 'a4': in => {'a2', 'a3'}, out => {'a1', 'a3', 'a2'}
- 'a5': in => {'a2'}, out => {'a1', 'a3', 'a2'}

**Not Gamma:**

- '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 `digraphs.Digraph.exportGraphViz()` method generates in the current working directory a tutorial.dot file and a tutorialdigraph.png picture of the tutorial digraph g (see Fig.1), if the graphviz (https://graphviz.org/) tools are installed on your system¹.

```python
>>> 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: The tutorial digraph](image)

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

```python
>>> dg.showStatistics()
***** general statistics ************
for digraph : <tutorialDigraph.py>
order : 5 nodes
```

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.
2.7 Special classes

Some special classes of digraphs, like the `digraphs.CompleteDigraph`, the `digraphs.EmptyDigraph` or the oriented `digraphs.GridDigraph` class for instance, are readily available (see Fig.2).

```python
>>> 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
```

Back to Tutorials of the Digraph3 resources (page 3)
Fig. 2: The tutorial grid graph
3.1 Random digraph

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

```python
>>> from randomDigraphs import RandomValuationDigraph
>>> dg = RandomValuationDigraph(order=7, Normalized=True)
```

With the `save()` method (see Line 3) we may keep a backup version for future use of \(dg\) which will be stored in a file called `tutRandValDigraph.py` in the current working directory. The Digraph class now provides some generic methods for exploring a given Digraph object, like the `showShort()`, `showAll()`, `showRelationTable()` and the `showNeighborhoods()` methods.

```python
>>> dg.showShort()
*----- show summary --------------*
Digraph : randomValuationDigraph
*---- Actions ----*
['1', '2', '3', '4', '5', '6', '7']
*---- Characteristic valuation domain ----*
{'med': Decimal('0.0'), 'hasIntegerValuation': False, 'min': Decimal('-1.0'), 'max': Decimal('1.0')}
*--- Connected Components ---*
1: ['1', '2', '3', '4', '5', '6', '7']
```

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

```python
>>> dg.showNeighborhoods()
Neighborhoods observed in digraph 'randomdomValuation'
Gamma :
'1': in => {'5', '7', '4'}, out => {'5', '7', '6', '3', '4'}
'2': in => {'7', '3'}, out => {'5', '7', '4'}
'3': in => {'7', '1'}, out => {'6', '2', '4'}
'4': in => {'5', '7', '1', '2', '3'}, out => {'5', '7', '1', '6'}
'5': in => {'1', '2', '4'}, out => {'1', '4'}
'6': in => {'7', '1', '3', '4'}, out => set()
'7': in => {'1', '2', '4'}, out => {'1', '2', '3', '4', '6'}
Not Gamma :
'1': in => {'6', '2', '3'}, out => {'2'}
'2': in => {'5', '1', '4'}, out => {'1', '6', '3'}
```
Warning: Notice that most Digraph class methods will ignore the reflexive couples by considering that the relation is indeterminate, i.e. the characteristic value $r(x S x)$ for all action $x$ is put to the median, i.e. indeterminate, value in this case (see [BIS-2004] (page 139)).

3.2 Graphviz drawings

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

```python
>>> dg.exportGraphViz('tutRandValDigraph')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to tutRandValDigraph.dot
```

dot -Grankdir=BT -Tpng tutRandValDigraph.dot -o tutRandValDigraph.png

Fig. 3: 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. 3).
3.3 Asymmetric and symmetric parts

We may now extract both this symmetric as well as this asymmetric part of digraph $dg$ with the help of two corresponding constructors (see Fig. 4).

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

Fig. 4: Asymmetric and symmetric part of the tutorial random valuation digraph

**Note:** Notice that the partial objects $asymDg$ and $symDg$ put to the indeterminate characteristic value all *not-asymmetric*, respectively *not-symmetric* links between nodes (see Fig. 4).

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

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

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

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

3.4 Fusion by epistemic disjunction

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

```python
>>> from digraphs import FusionDigraph
>>> fusDg = FusionDigraph(asymDg, symDg, operator='o-max')
>>> fusDg.showRelationTable()
```

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

3.5 Dual, converse and codual digraphs

We may as readily compute the dual, the converse and the codual (dual and converse) of \( dg \).

```python
>>> from digraphs import DualDigraph, ConverseDigraph, CoDualDigraph
>>> ddg = DualDigraph(dg)
>>> ddg.showRelationTable()
```

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

```python
>>> cdg = ConverseDigraph(dg)
>>> cdg.showRelationTable()
```

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

```python
>>> cddg = CoDualDigraph(dg)
>>> cddg.showRelationTable()
```

```
* ---- Relation Table -----
  -r(ySx) | '1' '2' '3' '4' '5' '6' '7'
  --------|------------------------------------------
  '1'    | 0.00 -0.22 -0.42 0.44 0.32 -0.84 0.88
  '2'    | -0.48 0.00 0.08 -0.40 -0.48 0.00 0.72
  '3'    | -0.70 -0.38 0.00 -0.62 -0.46 -0.40 0.82
  '4'    | 0.86 0.50 0.70 0.00 0.64 -0.96 0.52
  '5'    | 0.30 0.80 -0.56 0.04 0.00 -0.18 -0.84
  '6'    | 0.38 -0.54 0.84 0.66 -0.22 0.00 0.04
  '7'    | 0.44 0.02 -1.00 0.76 -0.52 -0.22 0.00
```
Computing the dual, respectively the converse, may also be done with prefixing the \texttt{\_\_neg\_} (-) or the \texttt{\_\_invert\_} (\~) operator. The codual of a Digraph object may, hence, as well be computed with a \texttt{composition} (in either order) of both operations.

```python
>>> ddg = -dg    # dual of dg
>>> cdg = -dg    # converse of dg
>>> cddg = ~(~dg)    # = -(~(dg)) codual of dg

>>> cddg.showRelationTable()
```

### 3.6 Symmetric and transitive closures

Symmetric and transitive closure in-site constructors are also available (see Fig. 5). Note that it is a good idea, before going ahead with these in-site operations who irreversibly modify the original dg object, to previously make a backup version of dg. The simplest storage method, always provided by the generic \texttt{digraphs.Digraph.save()}, writes out in a named file the python content of the Digraph object in string representation.

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

### 3.7 Strong components

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

```python
>>> from digraphs import StrongComponentsCollapsedDigraph
>>> sc = StrongComponentsCollapsedDigraph(dg)
>>> sc.showAll()
```
Fig. 5: Symmetric and transitive closure of the tutorial random valuation digraph

3.8 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.

It is possible to reload a Digraph instance from its previously saved CSV file content.
It is as well possible to show a colored version of the valued relation table in a system browser window tab (see Fig. 6).

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

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

3.9 Complete, empty and indeterminate digraphs

Let us finally mention some special universal classes of digraphs that are readily available in the digraphs module, like the digraphs.CompleteDigraph, the digraphs.EmptyDigraph and the digraphs.IndeterminateDigraph classes, which put all characteristic values respectively to the maximum, the minimum or the median indeterminate characteristic value.
... >> e = EmptyDigraph(order=5)
>>> e.showRelationTable()
* ---- Relation Table -----
  S | '1' '2' '3' '4' '5'
---- |---------------------------------------
'1' | -1.00 -1.00 -1.00 -1.00 -1.00
'2' | -1.00 -1.00 -1.00 -1.00 -1.00
'3' | -1.00 -1.00 -1.00 -1.00 -1.00
'4' | -1.00 -1.00 -1.00 -1.00 -1.00
'5' | -1.00 -1.00 -1.00 -1.00 -1.00
>>>
>>> e.showNeighborhoods()
Neighborhoods:
  Gamma :
  '1': in => set(), out => set()
  '2': in => set(), out => set()
  '5': in => set(), out => set()
  '3': in => set(), out => set()
  '4': in => set(), out => set()
Not Gamma :
  '1': in => {'2', '4', '5', '3'}, out => {'2', '4', '5', '3'}
  '2': in => {'1', '4', '5', '3'}, out => {'1', '4', '5', '3'}
  '5': in => {'1', '2', '4', '3'}, out => {'1', '2', '4', '3'}
  '3': in => {'1', '2', '4', '5'}, out => {'1', '2', '4', '5'}
  '4': in => {'1', '2', '5', '3'}, out => {'1', '2', '5', '3'}
>>>
>>> i = IndeterminateDigraph()
* ---- Relation Table -----
  S | '1' '2' '3' '4' '5'
---- |--------------------------------------
'1' | 0.00 0.00 0.00 0.00 0.00
'2' | 0.00 0.00 0.00 0.00 0.00
'3' | 0.00 0.00 0.00 0.00 0.00
'4' | 0.00 0.00 0.00 0.00 0.00
'5' | 0.00 0.00 0.00 0.00 0.00
>>>
>>> i.showNeighborhoods()
Neighborhoods:
  Gamma :
  '1': in => set(), out => set()
  '2': in => set(), out => set()
  '5': in => set(), out => set()
  '3': in => set(), out => set()
  '4': in => set(), out => set()
Not Gamma :
  '1': in => set(), out => set()
  '2': in => set(), out => set()
  '5': in => set(), out => set()
  '3': in => set(), out => set()
  '4': in => set(), out => set()

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

Back to Tutorials of the Digraph3 resources (page 3)
4 Computing the winner of an election

• Linear voting profiles (page 18)
• Computing the winner (page 19)
• The Condorcet winner (page 19)
• Cyclic social preferences (page 20)

4.1 Linear voting profiles

The votingProfiles module provides resources for handling election results [ADT-L2] (page 138), like the votingProfiles.LinearVotingProfile class. We consider an election involving a finite set of candidates and finite set of weighted voters, who express their voting preferences in a complete linear ranking (without ties) of the candidates. The data is internally stored in two ordered dictionaries, one for the voters and another one for the candidates. The linear ballots are stored in a standard dictionary:

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

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

```python
>>> from votingProfiles import RandomLinearVotingProfile
>>> v = RandomLinearVotingProfile(numberOfVoters=5, ... numberOfCandidates=3 ... votersWeights=[2,3,1,5,4])
>>> v.candidates
OrderedDict([('a1',{'name':'a1}), ('a2',{'name':'a2')}), ('a3':{'name':'a3'})])
>>> v.voters
OrderedDict([['v1',{'weight': 2}), ('v2':{'weight': 3}), ...
('v5',{'weight': 4})])
>>> v.linearBallot
{'v4': ['a1', 'a3', 'a2'], 'v3': ['a1', 'a3', 'a2'], 'v1': ['a1', 'a2', 'a3'],...
'v5': ['a2', 'a3', 'a1'], 'v2': ['a3', 'a2', 'a1']} >>> ...
```

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

```python
>>> v.showLinearBallots()
voters (weight)  candidates rankings
v1(2):  ['a2', 'a1', 'a3']
v2(3):  ['a3', 'a1', 'a2']
v3(1):  ['a1', 'a3', 'a2']
v4(5):  ['a1', 'a2', 'a3']
v5(4):  ['a3', 'a1', 'a2']
# voters: 15
>>> ...
```
Editing of the linear voting profile may be achieved by storing the data in a file, edit it, and reload it again.

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

### 4.2 Computing the winner

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

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

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] (page 138)).

```plaintext
>>> v.computeSimpleMajorityWinner()
['a3']
```

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

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

The Borda rank analysis table may be printed out with a corresponding show command.

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

### 4.3 The Condorcet winner

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

```plaintext
>>> from votingProfiles import CondorcetDigraph
>>> cdg = CondorcetDigraph(v,hasIntegerValuation=True)
>>> cdg.showAll()
```
A candidate \( x \), showing a positive majority margin \( M(x,y) \), is beating candidate \( y \) with an absolute majority in a pairwise voting. Hence, a candidate showing only positive terms in her row in the Condorcet digraph relation table, beats all other candidates with absolute majority of votes. Condorcet recommends to declare this candidate (is always unique, why?) the winner of the election. Here we are lucky, it is again candidate \( a1 \) who is hence the Condorcet winner.

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

4.4 Cyclic social preferences

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

voters (weight)  candidates rankings
v1(1):  ['a1', 'a3', 'a5', 'a2', 'a4']
v2(1):  ['a1', 'a2', 'a4', 'a3', 'a5']
v3(1):  ['a5', 'a2', 'a4', 'a3', 'a1']
v4(1):  ['a3', 'a4', 'a1', 'a5', 'a2']
v5(1):  ['a4', 'a2', 'a3', 'a5', 'a1']
v6(1):  ['a2', 'a4', 'a5', 'a1', 'a3']
v7(1):  ['a5', 'a4', 'a3', 'a1', 'a2']
v8(1):  ['a2', 'a4', 'a5', 'a1', 'a3']
v9(1):  ['a5', 'a3', 'a4', 'a1', 'a2']

>>> cdg = CondorcetDigraph(v)

>>> cdg.showRelationTable()

* ---- Relation Table ----

<table>
<thead>
<tr>
<th></th>
<th>'a1'</th>
<th>'a2'</th>
<th>'a3'</th>
<th>'a4'</th>
<th>'a5'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a1'</td>
<td>-</td>
<td>0.11</td>
<td>-0.11</td>
<td>-0.56</td>
<td>-0.33</td>
</tr>
<tr>
<td>'a2'</td>
<td>-0.11</td>
<td>-</td>
<td>0.11</td>
<td>-0.11</td>
<td>-0.33</td>
</tr>
<tr>
<td>'a3'</td>
<td>0.11</td>
<td>-0.11</td>
<td>-</td>
<td>0.33</td>
<td>-0.11</td>
</tr>
<tr>
<td>'a4'</td>
<td>0.56</td>
<td>-0.11</td>
<td>0.33</td>
<td>-</td>
<td>0.11</td>
</tr>
<tr>
<td>'a5'</td>
<td>0.33</td>
<td>0.11</td>
<td>0.11</td>
<td>-0.11</td>
<td>-</td>
</tr>
</tbody>
</table>

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

>>> cdg.exportGraphViz('cycles')

*---- exporting a dot file for GraphViz tools ---------*

Exporting to cycles.dot

dot -Grankdir=BT -Tpng cycles.dot -o cycles.png

Fig. 8: Cyclic social preferences

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

>>> cdg.computeChordlessCircuits()

[(\['a2', 'a3', 'a1'\], frozenset({'a2', 'a3', 'a1'}))],
[(\['a2', 'a4', 'a5'\], frozenset({'a2', 'a5', 'a4'}))],
[(\['a2', 'a4', 'a1'\], frozenset({'a2', 'a1', 'a4'}))]

Condorcet’s approach for determining the winner of an election is hence not decisive in all circumstances and we
need to exploit more sophisticated approaches for finding the winner of the election on the basis of the majority margins of the given linear ballots (see [BIS-2008] (page 138)).

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

Back to Tutorials of the Digraph3 resources (page 3)

5 Working with the outrankingDigraphs module

- Outranking digraph (page 22)
- Browsing the performances (page 23)
- Valuation semantics (page 25)
- Pairwise comparisons (page 25)
- Recoding the valuation (page 26)
- Codual digraph (page 26)
- XMCDA 2.0 (page 27)

See also the technical documentation of the outrankingDigraphs-label.

5.1 Outranking digraph

In this Digraph3 module, the main outrankingDigraphs.BipolarOutrankingDigraph class provides a generic bipolar-valued outranking digraph model. A given object of this class consists in

1. a potential set of decision actions: a dictionary describing the potential decision actions or alternatives with ‘name’ and ‘comment’ attributes,

2. a coherent family of criteria: a dictionary of criteria functions used for measuring the performance of each potential decision action with respect to the preference dimension captured by each criterion,

3. the evaluations: a dictionary of performance evaluations for each decision action or alternative on each criterion function.

4. the digraph valuationdomain, a dictionary with three entries: the minimum (-100, means certainly no link), the median (0, means missing information) and the maximum characteristic value (+100, means certainly a link),

5. the outranking relation: a double dictionary defined on the Cartesian product of the set of decision alternatives capturing the credibility of the pairwise outranking situation computed on the basis of the performance differences observed between couples of decision alternatives on the given family if criteria functions.

With the help of the outrankingDigraphs.RandomBipolarOutrankingDigraph class (of type outrankingDigraphs.BipolarOutrankingDigraph), let us generate for illustration a random bipolar-valued outranking digraph consisting of 7 decision actions denoted \(a_01, a_02, \ldots, a_{07}\).

```python
>>> from outrankingDigraphs import RandomBipolarOutrankingDigraph
>>> odg = RandomBipolarOutrankingDigraph()
>>> odg.showActions()
*----- show digraphs actions --------------*
key: a01
name: random decision action
comment: RandomPerformanceTableau() generated.
key: a02
name: random decision action
```
In this example we consider furthermore a family of seven equisignificant cardinal criteria functions $g_{01}, g_{02}, \ldots, g_{07}$, measuring the performance of each alternative on a rational scale from 0.0 to 100.00. In order to capture the evaluation’s uncertainty and imprecision, each criterion function $g_l$ to $g_7$ admits three performance discrimination thresholds of 10, 20 and 80 pts for warranting respectively any indifference, preference and veto situations.

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

5.2 Browsing the performances

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

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

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

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

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

Considering the given performance tableau, the \texttt{outrankingDigraphs.BipolarOutrankingDigraph} class constructor computes the characteristic value $r(x S y)$ of a pairwise outranking relation "$x S y$" (see [BIS-2013] (page 138), [ADT-L7] (page 138)) in a default valuation domain [-100.0,+100.0] with the median value 0.0 acting as indeterminate characteristic value. The semantics of $r(x S y)$ are the following.

1. If $r(x S y) > 0.0$ it is more \textit{True} than \textit{False} that $x$ outranks $y$, i.e. alternative $x$ is at least as well performing than alternative $y$ \textit{and} there is no considerable negative performance difference observed in disfavour of $x$,

2. If $r(x S y) < 0.0$ it is more \textit{False} than \textit{True} that $x$ outranks $y$, i.e. alternative $x$ is \textit{not} at least as well performing than alternative $y$ \textit{and} there is no considerable positive performance difference observed in favour of $x$,

3. If $r(x S y) = 0.0$ it is \textit{indeterminate} whether $x$ outranks $y$ or not.

The resulting bipolar-valued outranking relation may be inspected with the following command.

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

5.4 Pairwise comparisons

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

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

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

```python
>>> odg.showPairwiseComparison('a03','a02')
*------------ pairwise comparison ---- *
Comparing actions : (a03, a02)
```
This time, we observe a considerable out-performance of \( a_{03} \) against \( a_{02} \) on criterion \( g_{02} \) (see second row in the relation table above). We therefore notice a positively polarized \textit{certainly confirmed} outranking situation in this case \cite{BIS-2013} (page 138).

### 5.5 Recoding the valuation

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

```python
>>> odg.recodeValuation(-37,+37)
>>> odg.valuationdomain['hasIntegerValuation'] = True
```

```
* ---- Relation Table -----  
<table>
<thead>
<tr>
<th></th>
<th>'a01'</th>
<th>'a02'</th>
<th>'a03'</th>
<th>'a04'</th>
<th>'a05'</th>
<th>'a06'</th>
<th>'a07'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a01'</td>
<td>0</td>
<td>+11</td>
<td>-11</td>
<td>+5</td>
<td>+17</td>
<td>+14</td>
<td></td>
</tr>
<tr>
<td>'a02'</td>
<td>+5</td>
<td>0</td>
<td>-37</td>
<td>+13</td>
<td>+5</td>
<td>+15</td>
<td></td>
</tr>
<tr>
<td>'a03'</td>
<td>+31</td>
<td>+37</td>
<td>0</td>
<td>+34</td>
<td>+31</td>
<td>+31</td>
<td></td>
</tr>
<tr>
<td>'a04'</td>
<td>+9</td>
<td>+18</td>
<td>-21</td>
<td>0</td>
<td>+9</td>
<td>+19</td>
<td></td>
</tr>
<tr>
<td>'a05'</td>
<td>+19</td>
<td>+37</td>
<td>-26</td>
<td>+27</td>
<td>0</td>
<td>+19</td>
<td></td>
</tr>
<tr>
<td>'a06'</td>
<td>+6</td>
<td>+27</td>
<td>-19</td>
<td>+13</td>
<td>+12</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>'a07'</td>
<td>+25</td>
<td>+17</td>
<td>-9</td>
<td>+9</td>
<td>+9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Valuation domain: {'hasIntegerValuation': True, 'min': Decimal('-37'), 'max': Decimal('37'), 'med': Decimal('0.000')}

```python
>>> ...
```

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

### 5.6 Codual digraph

From the theory (see \cite{BIS-2013} (page 138), \cite{ADT-L7} (page 138) ) we know that a bipolar-valued outranking relation is \textit{weakly complete}, i.e. if \( r(xSy) < 0 \) then \( r(ySx) >= 0 \). From this property follows that a bipolar-valued outranking relation verifies the \textit{coduality} principle: the dual (-) of the converse (~) of the outranking relation corresponds to its strict outranking part. We may visualize the codual (strict) outranking digraph with a graphviz drawing$^{1}$.

```python
>>> cdodg = -(~odg)
>>> cdodg.exportGraphViz('codualOdg')
```

------ exporting a dot file for GraphViz tools --------
It becomes readily clear now from the picture above that alternative $a_{03}$ strictly outranks in fact all the other alternatives. Hence, $a_{03}$ appears as Condorcet winner and may be recommended as best decision action in this illustrative preference modelling exercise.

### 5.7 XMCDA 2.0

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

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

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

```
>>> pt = XMCDA2PerformanceTableau('tutorialPerfTab')
>>> odg = BipolarOutrankingDigraph(pt)
>>> odg.showRelationTable()
  *---- Relation Table ----*
  S | 'a01'  'a02'  'a03'  'a04'  'a05'  'a06'  'a07'
  --------------------|--------------------------------------------------
 'a01' | +0.00  +29.73 -29.73 +13.51 +48.65 +40.54 +48.65
 'a02' | +13.51 +0.00 -100.00 +37.84 +13.51 +43.24 -37.84
 'a03' | +83.78 +100.00 +0.00 +91.89 +83.78 +83.78 +70.27
 'a04' | +24.32 +48.65 -56.76 +0.00 +24.32 +51.35 +24.32
 'a05' | +51.35 +100.00 -70.27 +72.97 +0.00 +51.35 +32.43
 'a06' | +16.22 +72.97 -51.35 +35.14 +32.43 +0.00 +37.84
```
We recover the original bipolar-valued outranking characteristics, and we may restart again the preference mod-
eling process.

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

Back to Tutorials of the Digraph3 resources (page 3)

6 Generating random performance tableaux

6.1 Introduction

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

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

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

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

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

6.2 Generating standard random performance tableaux

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

- numberOfActions := nbr of decision actions.
- numberOfCriteria := number performance criteria.
• weightDistribution := ‘random’ (default) | ‘fixed’ | ‘equisignificant’:
  If ‘random’, weights are uniformly selected randomly
  from the given weight scale;
  If ‘fixed’, the weightScale must provided a corresponding weights
distribution;
  If ‘equisignificant’, all criterion weights are put to unity.

• weightScale := [Min,Max] (default = (1,numberOfCriteria).

• IntegerWeights := True (default) | False (normalized to proportions of 1.0).

• commonScale := [a,b]; common performance measuring scales (default = [0.0,100.0])

• commonThresholds := [(q0,q1),(p0,p1),(v0,v1)]; common indifference(q), preference (p) and considerable
  performance difference discrimination thresholds. For each threshold type \(x\) in \(\{q, p, v\}\), the float \(x_0\) value
  represents a constant percentage of the common scale and the float \(x_1\) 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,None)): (‘uniform’,None,None), uniformly distributed float values on the given common scales’ range
  [Min,Max].
  (‘normal’,*mu*,*sigma*), truncated Gaussian distribution, by default \(mu = (b-a)/2\) and \(sigma =
  (b-a)/4\).
  (‘triangular’,*mode*,*repartition*), generalized triangular distribution with a probability
  repartition parameter specifying the probability mass accumulated until the mode value. By
  default, \(mode = (b-a)/2\) and \(repartition = 0.5\).
  (‘beta’,None,(alpha,beta)), a beta generator with default alpha=2 and beta=2 parameters.

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

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

Code example.

```
>>> 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
 ... },
... }
```
Fig. 12: Browser view on random performance tableau instance

<table>
<thead>
<tr>
<th>criteria</th>
<th>g01</th>
<th>g02</th>
<th>g03</th>
<th>g04</th>
<th>g05</th>
<th>g06</th>
<th>g07</th>
<th>g08</th>
<th>g09</th>
<th>g10</th>
<th>g11</th>
<th>g12</th>
<th>g13</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>a01</td>
<td>15.17</td>
<td>46.37</td>
<td>82.88</td>
<td>41.14</td>
<td>59.94</td>
<td>41.19</td>
<td>58.68</td>
<td>44.73</td>
<td>22.19</td>
<td>64.64</td>
<td>34.93</td>
<td>42.36</td>
<td>17.55</td>
</tr>
<tr>
<td>a02</td>
<td>44.51</td>
<td>16.22</td>
<td>41.66</td>
<td>53.58</td>
<td>31.39</td>
<td>65.22</td>
<td>71.96</td>
<td>57.84</td>
<td>78.08</td>
<td>77.37</td>
<td>8.30</td>
<td>63.41</td>
<td>61.55</td>
</tr>
<tr>
<td>a03</td>
<td>NA</td>
<td>21.53</td>
<td>12.82</td>
<td>56.93</td>
<td>26.80</td>
<td>48.03</td>
<td>54.35</td>
<td>62.42</td>
<td>94.27</td>
<td>73.37</td>
<td>71.11</td>
<td>21.81</td>
<td>56.80</td>
</tr>
<tr>
<td>a04</td>
<td>58.00</td>
<td>51.16</td>
<td>21.92</td>
<td>65.57</td>
<td>59.02</td>
<td>44.77</td>
<td>37.49</td>
<td>58.39</td>
<td>80.79</td>
<td>55.39</td>
<td>46.44</td>
<td>19.57</td>
<td>39.22</td>
</tr>
<tr>
<td>a05</td>
<td>24.22</td>
<td>77.01</td>
<td>75.74</td>
<td>83.87</td>
<td>40.85</td>
<td>8.55</td>
<td>85.44</td>
<td>67.34</td>
<td>57.40</td>
<td>39.08</td>
<td>64.83</td>
<td>29.37</td>
<td>96.39</td>
</tr>
<tr>
<td>a06</td>
<td>29.10</td>
<td>39.35</td>
<td>15.45</td>
<td>34.99</td>
<td>49.12</td>
<td>11.49</td>
<td>28.44</td>
<td>52.89</td>
<td>64.24</td>
<td>62.92</td>
<td>58.28</td>
<td>32.02</td>
<td>10.25</td>
</tr>
<tr>
<td>a07</td>
<td>96.58</td>
<td>32.06</td>
<td>6.05</td>
<td>49.56</td>
<td>NA</td>
<td>66.06</td>
<td>41.64</td>
<td>13.08</td>
<td>38.31</td>
<td>24.82</td>
<td>48.39</td>
<td>57.03</td>
<td>42.91</td>
</tr>
<tr>
<td>a08</td>
<td>82.29</td>
<td>47.67</td>
<td>9.96</td>
<td>79.43</td>
<td>29.45</td>
<td>84.17</td>
<td>31.99</td>
<td>90.68</td>
<td>39.58</td>
<td>50.78</td>
<td>61.88</td>
<td>44.40</td>
<td>48.26</td>
</tr>
<tr>
<td>a09</td>
<td>43.90</td>
<td>14.81</td>
<td>60.55</td>
<td>42.37</td>
<td>6.72</td>
<td>56.14</td>
<td>34.20</td>
<td>51.54</td>
<td>21.79</td>
<td>79.13</td>
<td>50.95</td>
<td>83.16</td>
<td>13.89</td>
</tr>
<tr>
<td>a10</td>
<td>38.75</td>
<td>79.76</td>
<td>27.88</td>
<td>42.39</td>
<td>71.88</td>
<td>66.09</td>
<td>58.33</td>
<td>58.88</td>
<td>17.10</td>
<td>44.25</td>
<td>48.73</td>
<td>30.63</td>
<td>52.73</td>
</tr>
<tr>
<td>a11</td>
<td>35.84</td>
<td>67.48</td>
<td>38.81</td>
<td>33.75</td>
<td>26.87</td>
<td>64.10</td>
<td>71.95</td>
<td>62.72</td>
<td>85.80</td>
<td>53.97</td>
<td>48.33</td>
<td>19.33</td>
<td></td>
</tr>
<tr>
<td>a12</td>
<td>29.12</td>
<td>13.97</td>
<td>67.45</td>
<td>38.60</td>
<td>48.30</td>
<td>11.87</td>
<td>NA</td>
<td>57.76</td>
<td>74.86</td>
<td>26.57</td>
<td>48.80</td>
<td>43.57</td>
<td>7.68</td>
</tr>
<tr>
<td>a13</td>
<td>34.79</td>
<td>90.72</td>
<td>38.93</td>
<td>57.38</td>
<td>6.14</td>
<td>97.86</td>
<td>91.16</td>
<td>43.80</td>
<td>33.68</td>
<td>38.98</td>
<td>28.87</td>
<td>63.36</td>
<td>60.83</td>
</tr>
<tr>
<td>a14</td>
<td>62.22</td>
<td>80.16</td>
<td>19.26</td>
<td>62.34</td>
<td>60.96</td>
<td>24.72</td>
<td>73.63</td>
<td>71.21</td>
<td>56.43</td>
<td>46.12</td>
<td>26.09</td>
<td>51.43</td>
<td>12.86</td>
</tr>
<tr>
<td>a15</td>
<td>44.23</td>
<td>69.62</td>
<td>94.95</td>
<td>34.95</td>
<td>63.46</td>
<td>52.97</td>
<td>98.84</td>
<td>78.74</td>
<td>36.64</td>
<td>65.12</td>
<td>22.46</td>
<td>55.52</td>
<td>68.79</td>
</tr>
<tr>
<td>a16</td>
<td>19.10</td>
<td>45.49</td>
<td>65.63</td>
<td>64.96</td>
<td>50.57</td>
<td>55.91</td>
<td>10.02</td>
<td>34.70</td>
<td>29.31</td>
<td>50.15</td>
<td>70.68</td>
<td>62.57</td>
<td>71.09</td>
</tr>
<tr>
<td>a17</td>
<td>27.73</td>
<td>22.03</td>
<td>48.00</td>
<td>79.38</td>
<td>23.35</td>
<td>74.03</td>
<td>58.74</td>
<td>59.42</td>
<td>50.95</td>
<td>82.27</td>
<td>49.20</td>
<td>43.27</td>
<td>38.61</td>
</tr>
<tr>
<td>a18</td>
<td>41.46</td>
<td>33.83</td>
<td>7.97</td>
<td>75.11</td>
<td>49.00</td>
<td>55.70</td>
<td>64.99</td>
<td>38.47</td>
<td>49.86</td>
<td>17.45</td>
<td>28.08</td>
<td>35.21</td>
<td>67.81</td>
</tr>
<tr>
<td>a19</td>
<td>22.41</td>
<td>NA</td>
<td>34.86</td>
<td>49.30</td>
<td>65.18</td>
<td>39.84</td>
<td>81.16</td>
<td>NA</td>
<td>55.99</td>
<td>66.55</td>
<td>55.38</td>
<td>43.08</td>
<td>29.72</td>
</tr>
<tr>
<td>a20</td>
<td>21.52</td>
<td>69.98</td>
<td>71.81</td>
<td>43.74</td>
<td>24.53</td>
<td>55.39</td>
<td>52.67</td>
<td>13.67</td>
<td>68.80</td>
<td>57.46</td>
<td>70.81</td>
<td>5.41</td>
<td>76.05</td>
</tr>
<tr>
<td>a21</td>
<td>56.90</td>
<td>48.80</td>
<td>31.66</td>
<td>15.31</td>
<td>40.57</td>
<td>58.14</td>
<td>70.19</td>
<td>67.23</td>
<td>61.10</td>
<td>31.04</td>
<td>60.72</td>
<td>22.39</td>
<td>70.38</td>
</tr>
</tbody>
</table>

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

### 6.3 Generating random Cost-Benefit tableaux

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

- We distinguish three types of decision actions: *cheap*, *neutral* and *expensive* ones with an equal proportion of 1/3. We also distinguish two types of weighted criteria: *cost* criteria to be minimized, and *benefit* criteria to be maximized; in the proportions 1/3 respectively 2/3.

- Random performances on each type of criteria are drawn, either from an ordinal scale [0;10], or from a cardinal scale [0.0;100.0], following a parametric triangular law of mode: 30% performance for cheap, 50% for neutral, and 70% performance for expensive decision actions, with constant probability repartition 0.5 on each side of the respective mode.

- Cost criteria use mostly cardinal scales (3/4), whereas benefit criteria use mostly ordinal scales (2/3).

- The sum of weights of the cost criteria by default equals the sum weights of the benefit criteria: `weighDistribution` = ‘equiobjectives’.
On cardinal criteria, both of cost or of benefit type, we observe following constant preference discrimination quantiles: 5% indifferent situations, 90% strict preference situations, and 5% veto situation.

Parameters:

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

**Warning:** Minimal number of decision actions required is 3!

Example Python session:

```python
>>> from randomPerfTabs import RandomCBPerformanceTableau
>>> t = RandomCBPerformanceTableau(
...    numberOfActions=7,
...    numberOfCriteria=5,
...    weightDistribution='equiobjectives',
...    commonPercentiles={'ind':5,'pref':10,'veto':95},
...    seed=100)

>>> t.showActions()

*----- show decision action --------------*
  key: a1
    short name: a1
    name: random cheap decision action
  key: a2
    short name: a2
    name: random neutral decision action
  ...  
  key: a7
    short name: a7
    name: random advantageous decision action

>>> t.showCriteria()

*---- criteria ----- *
  g1 'random ordinal benefit criterion'
    Scale = (0, 10)
    Weight = 0.167
  g2 'random cardinal cost criterion'
    Scale = (0.0, 100.0)
    Weight = 0.250
    Threshold ind : 1.76 + 0.00x ; percentile: 0.095
    Threshold pref : 2.16 + 0.00x ; percentile: 0.143
    Threshold veto : 73.19 + 0.00x ; percentile: 0.952

...```

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

```python
>>> t.showStatistics()

*-------- Performance tableau summary statistics -------*
Instance name : randomCBperftab
#Actions : 7
#Criteria : 5

*Statistics per Criterion*
Criterion name : g1
  Criterion weight : 2
criterion scale : 0.00 - 10.00
  mean evaluation : 5.14
  standard deviation : 2.64
  maximal evaluation : 8.00
  quantile Q3 (x_75) : 8.00
  median evaluation : 6.50
  quantile Q1 (x_25) : 3.50
  minimal evaluation : 1.00
  mean absolute difference : 2.94
  standard difference deviation : 3.74

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

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

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

![Heatmap of performance tableau](attachment:Fig_13.png)

Fig. 13: Unranked heat map of a random Cost-Benefit performance tableau
Such a performance tableau may be stored and re-accessed in the XMCDA2 encoded format.

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

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

```python
>>> t.saveCSV('temp')
* --- Storing performance tableau in CSV format in file 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,-34.91,79.24,9.00
"a7",4.00,-91.11,-7.44,48.22,7.00
```

Back to [Tutorials of the Digraph3 resources](#) (page 3)

### 6.4 Generating three objectives tableaux

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

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

Generator directives are the following:

- `numberOfActions` = 20 (default),
- `numberOfCriteria` = 13 (default),
- `weightDistribution` = 'equiobjectives' (default) | 'random' | 'equisignificant',
- `weightScale` = (1,numberOfCriteria): only used when random criterion weights are requested,
- `integerWeights` = True (default): False gives 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,
- `missingProbability` = 0.05 (default): random insertion of missing values with given probability,
- `seed`= None.

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

```python
>>> from randomPerfTabs import Random3ObjectivesPerformanceTableau
>>> t = Random3ObjectivesPerformanceTableau{
    numberOfActions=31,
    numberOfCriteria=13,
    weightDistribution='equiobjectives',
    seed=120)
>>> t.showObjectives()
*------ show objectives -------
Eco: Economical aspect
  g04 criterion of objective Eco 20
  g05 criterion of objective Eco 20
  g08 criterion of objective Eco 20
  g11 criterion of objective Eco 20
  Total weight: 80.00 (4 criteria)
Soc: Societal aspect
  g06 criterion of objective Soc 16
  g07 criterion of objective Soc 16
  g09 criterion of objective Soc 16
  g10 criterion of objective Soc 16
  g13 criterion of objective Soc 16
  Total weight: 80.00 (5 criteria)
Env: Environmental aspect
  g01 criterion of objective Env 20
  g02 criterion of objective Env 20
  g03 criterion of objective Env 20
  g12 criterion of objective Env 20
  Total weight: 80.00 (4 criteria)

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

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

The three partial digraphs: \texttt{geco}, \texttt{gsoc} and \texttt{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.

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

Decision action \texttt{a26} (Eco+ Soc+ Env-) appears dominating the other decision alternatives, whereas decision action \texttt{a30} (Eco- Soc- Env-) appears to be dominated by all the others.
6.5 Generating random linearly ranked performances

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

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

**Parameters:**

- number of actions,
- number of performance criteria,
- weightDistribution := equisignificant | random (default, see above above)
- weightScale := (1, 1 | numberOfCriteria (default when random))
- integerWeights := Boolean (True = default)
- commonThresholds (default) := {
  'ind':(0,0),
  'pref':(1,0),
  'veto':(numberOfActions,0)
} (default)

Back to *Tutorials of the Digraph3 resources* (page 3)

7 Ranking with multiple incommensurable criteria

- *The ranking problem* (page 37)
- *The Copeland ranking* (page 39)
7.1 The ranking problem

We need to rank without ties a set $X$ of items (usually decision alternatives) that are evaluated on multiple in-commensurable performance criteria; yet, for which we may know their pairwise valued outranking situation characteristics, i.e. $r(x, y)$ for all $x, y$ in $X$ (see [BIS-2013] (page 138)). Unfortunately, the Condorcet digraph, associated with such a given outranking digraph, presents only exceptionally a linear ordering. Usually, pairwise majority comparisons do not render even a complete or, at least, a transitive partial outranking relation.

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

```python
>>> from outrankingDigraphs import *

>>> t = RandomCBPerformanceTableau(numberOfActions=9,
...   numberOfCriteria=13,seed=2)

>>> g = BipolarOutrankingDigraph(t,Normalized=True)

>>> g.showRelationTable(ReflexiveTerms=False)

* ---- Relation Table ----
S | 'a1' 'a2' 'a3' 'a4' 'a5' 'a6' 'a7' 'a8' 'a9'
-----|-----------------------------------------------------------------------------------
'a1' | - +0.00 +0.24 +0.00 +0.17 +0.26 +0.07 +0.00
'a2' | +0.00 - -0.50 +0.00 -0.13 +0.00 +0.00 -0.02 +0.00
'a3' | +0.14 +0.50 - +0.40 +0.36 +0.50 +0.71 +0.69 +1.00
'a4' | +0.05 +0.00 -0.40 - +0.00 +0.21 +0.26 -0.10 +0.10
'a5' | +0.00 +0.36 -0.36 +0.00 - +0.26 +0.00 +0.26 -1.00
'a6' | -0.10 +0.00 -0.29 -0.07 +0.02 - +0.24 +0.19 +0.04
'a7' | -0.26 +0.00 -0.29 +0.00 -0.10 - +0.00 -1.00
'a8' | -0.07 +0.33 -0.24 +0.10 +0.05 +0.29 +0.00 -0.02 +0.00
'a9' | +0.00 +0.00 -1.00 -0.10 +1.00 +0.33 +1.00 +0.02 -
```

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

```python
>>> c = PolarisedOutrankingDigraph(g,level=0,KeepValues=True,
...   StrictCut=True)

>>> c.showRelationTable(ReflexiveTerms=False,IntegerValues=True)

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

To estimate how difficult this ranking problem may be, we can have a look at the corresponding strict outranking digraph graphviz drawing.
The shown strict outranking relation is apparently not transitive: for instance, alternative $a_9$ outranks alternative $a_5$ and alternative $a_5$ outranks $a_2$, however $a_9$ does not outrank $a_2$. We may compute the transitivity degree of the outranking digraph, i.e. the ratio of the number of outranking arcs over the number of arcs of the transitive closure of the digraph $gcd$.

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

There is one chordless circuit detected in the given strict outranking digraph $gcd$, namely $a_4$ outranks $a_9$, the latter outranks $a_8$, and $a_8$ outranks again $a_4$. Any potential linear ordering of these three alternatives will, in fact, always contradict somehow the given outranking relation.

Several heuristic ranking rules have been proposed for constructing a linear ordering which is closest in some specific sense to a given outranking relation.
The Digraph3 resources provide some of the most common of these ranking rules, like Copeland’s, Kemeny’s, Slater’s, Kohler and Tideman’s ranking rules.

7.2 The Copeland ranking

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

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

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

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

Copeland’s rule actually renders a linear order which is indeed highly correlated, in the ordinal Kendall sense (see [BIS-2012] (page 138)), with the given pairwise outranking relation.

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

7.3 The Net-Flows ranking

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

```python
>>> from linearOrders import NetFlowsOrder
>>> nf = NetFlowsOrder(g)
>>> nf.netFlows
[(Decimal('7.143'), 'a3'),
(Decimal('2.155'), 'a9'),
(Decimal('1.214'), 'a1'),
(Decimal('-0.429'), 'a4'),
(Decimal('-0.690'), 'a8'),
(Decimal('-1.631'), 'a6'),
(Decimal('-1.774'), 'a5'),
(Decimal('-1.845'), 'a2'),
(Decimal('-4.143'), 'a7')]`
The Net-Flows ranking is here, in this didactic example, not as much correlated with the given outranking relation as its crisp cousin ranking.

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

### 7.4 Kemeny rankings

A Kemeny ranking is a linear order which is closest, in the sense of the ordinal Kendall distance (see [BIS-2012] (page 138)), to the given valued outranking digraph $g$.

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

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

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

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

```
>>> from weakOrders import KemenyWeakOrder
>>> wke = KemenyWeakOrder(g, orderLimit=9)
>>> wke.exportGraphViz('tutorialKemeny')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to tutorialKemeny.dot
```

It is interesting to notice that all seven Kemeny rankings place alternative $a1$ at rank 1 before alternative $a3$. This is precisely the only inversion that separates the Copeland ranking (see above) from being optimal in the Kemeny sense.
Fig. 16: Epistemic disjunction of Kemeny rankings
7.5 Slater rankings

The Slater ranking rule is similar to Kemeny’s, but it is working, instead, on the associated crisp Condorcet digraph \( c \). It renders here the following results.

```python
In [1]: sl = KemenyOrder(c, orderLimit=9)
In [2]: len(sl.maximalRankings)
Out[2]: 174
In [3]: sl.showRanking()
Out[3]: ['a1', 'a3', 'a8', 'a4', 'a6', 'a9', 'a5', 'a2', 'a7']
In [4]: corr = g.computeOrdinalCorrelation(sl)
In [5]: print("Fitness of Slater's ranking: \%.3f\n" % corr['correlation'])
Fitness of Slater's ranking: 0.844
In [6]: slw = KemenyWeakOrder(c, orderLimit=9)
In [7]: slw.exportGraphViz('tutorialSlater')
```

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

![Fig. 17: Epistemic disjunction of Slater rankings](image)

What precise ranking result should we hence adopt?

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

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

7.6 Kohler’s ranking-by-choosing rule

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

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

1. Compute for each row of the valued outranking relation table (see above) the smallest value;
2. Select the row where this minimum is maximal. Ties are resolved in lexicographic order;
3. Put the selected decision alternative at rank \( r \).
4. Delete the corresponding row and column from the relation table and restart until the table is empty.

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

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

### 7.7 Tideman’s Ranked-Pairs rule

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

1. Rank the ordered pairs \((x, y)\) of alternatives in decreasing order of the outranking characteristic values \(r(x S y)\);

2. Consider the pairs in that order (ties are resolved by a lexicographic rule):
   - if the next pair does not create a cycle with the pairs already blocked, block this pair;
   - if the next pair creates a cycle with the already blocked pairs, skip it.

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

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

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

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

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

### 7.8 Ranking big performance tableaux

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

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

```python
>>> from sparseOutrankingDigraphs import PreRankedOutrankingDigraph
>>> tp = RandomCBPerformanceTableau(numberOfActions=100,BigData=True,
... Thread=MP,
... seed=100)

>>> bg = PreRankedOutrankingDigraph(tp,quantiles=20,
... LowerClosed=False,
... minimalComponentSize=1,
... Threading=True)

>>> print(bg)
*----- show short --------------*
Instance name : randomCBperftab_mp
# Actions : 100
# Criteria : 7
Sorting by : 10-Tiling
Ranking rule : Copeland
# Components : 20
Minimal order : 1
Maximal order : 20
Average order : 5.0
fill rate : 10.061%
==== Constructor run times (in sec.) ====
Total time : 0.17790
QuantilesSorting : 0.09019
Preordering : 0.00043
Decomposing : 0.08522
Ordering : 0.00000
<class 'sparseOutrankingDigraphs.PreRankedOutrankingDigraph'> instance

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

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

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

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

This ranking heuristic is readily scalable with ad hoc HPC tuning to several millions of decision alternatives (see [BIS-2016] (page 138)).

8 HPC ranking with big outranking digraphs
8.1 C-compiled Python modules

The Digraph3 collection provides cythonized\(^6\), i.e. C-compiled and optimised versions of the main python modules for tackling multiple criteria decision problems facing very large sets of decision alternatives (\(> 10000\)). Such problems appear usually with a combinatorial organisation of the potential decision alternatives, as is frequently the case in bioinformatics for instance. If HPC facilities with nodes supporting numerous cores (\(> 20\)) and big RAM (\(> 50GB\)) are available, ranking up to several millions of alternatives (see [BIS-2016] (page 138)) becomes effectively tractable.

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

- cRandPerfTabs.pyx
- cIntegerOutrankingDigraphs.pyx
- cIntegerSortingDigraphs.pyx
- cSparseIntegerOutrankingDigraphs.pyx

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

8.2 Big Data performance tableaux

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

```
>>> 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'}])
```

\(^6\) See https://cython.org/
Conversions from the Big Data model to the standard model and vice versa are provided:

```
>>> t1 = t.convert2Standard()
>>> t1.convertWeight2Decimal()
>>> t1.convertEvaluation2Decimal()
```

8.3 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
```

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

```python
>>> from outrankingDigraphs import BipolarOutrankingDigraph
>>> 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('-100.0'),
                   'med': Decimal('0.0'),
                   'max': Decimal('100.0'),
                   '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.

```python
>>> 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.

### 8.4 The sparse outranking digraph implementation

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

```python
>>> 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 quartile limits.

```python
>>> 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', '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)

Legend:
* outranking for certain ( )
* outranked for certain ( )
* more or less outranking (+)
* more or less outranked (-)
* indeterminate ( )

With a fill rate of 25%, the memory occupation of this sparse outranking digraph sg instance takes now only 769kB, compared to the 1.7MB required by a corresponding standard IntegerBipolarOutrankingDigraph instance.
Fig. 18: Sparse quartiles-sorting decomposed outranking relation (extract).
For sparse outranking digraphs, the adjacency table is implemented as a dynamic `self.relation(x, y)` function instead of a double dictionary `self.relation[x][y]`:

```python
def relation(self, int x, int y):
    
    ***
    *Parameters*:
    * x (int action key),
    * y (int action key).
    Dynamic construction of the global outranking characterstic 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(y):
                return Max
            else:
                return Min
    elif rx > ry:
        return Min
    else:
        return Max
```

### 8.5 Ranking big sets of decision alternatives

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

```bash
>>> 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 `cSparseIntegerOutrankingDigraphs.cQuantilesRankingDigraph` class.

```python
>>> 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
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.

```python
>>> 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]}),
...]
```

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

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 optimised sparse 4-tiling correlation of +0.81.

8.6 HPC quantiles ranking records

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

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, and
- on the 3TB -bigmem Gaia-183 node with 64 cores,

by running the cythonized python modules in an Intel compiled virtual Python 3.6.5 environment [GCC Intel(R) 170.1 -enable-optimizations c++ gcc 6.3 mode] on Debian 8 Linux.

Example python session on the HPC-UL Iris-126 -skylake node:

```
>>> 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(ig.boostedRanking)))['correlation'])))
Sparse 4-tiling : +0.7172
>>> print('Optimized sparse 4-tiling: %+.4f'
...       % (ig.computeOrderCorrelation(
...           list(reversed(ig.boostedRanking)))['correlation'])))
Optimized sparse 4-tiling: +0.7051
```
### Fig. 19: HPC-UL Ranking Performance Records (Spring 2018)

<table>
<thead>
<tr>
<th>( N \times V )</th>
<th>outranking relation order</th>
<th>size</th>
<th>( q )</th>
<th>fill rate</th>
<th>n. cores</th>
<th>run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>25 ( \times ) 10^4</td>
<td>4</td>
<td>0.005%</td>
<td>28</td>
<td>0.5’</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>1 ( \times ) 10^5</td>
<td>4</td>
<td>0.001%</td>
<td>28</td>
<td>1’</td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td>1 ( \times ) 10^10</td>
<td>5</td>
<td>0.002%</td>
<td>28</td>
<td>10’</td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td>1 ( \times ) 10^12</td>
<td>6</td>
<td>0.001%</td>
<td>64</td>
<td>2’</td>
<td></td>
</tr>
<tr>
<td>3000000</td>
<td>9 ( \times ) 10^12</td>
<td>15</td>
<td>0.004%</td>
<td>64</td>
<td>13’</td>
<td></td>
</tr>
<tr>
<td>6000000</td>
<td>36 ( \times ) 10^12</td>
<td>15</td>
<td>0.002%</td>
<td>64</td>
<td>41’</td>
<td></td>
</tr>
</tbody>
</table>

---

On this 2x14c Intel Xeon Gold 6132 @ 2.6 GHz equipped HPC node with 132GB RAM, 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)
```

<table>
<thead>
<tr>
<th>criteria</th>
<th>weights</th>
<th>#773909</th>
<th>#668947</th>
<th>#567308</th>
<th>#578560</th>
<th>#426464</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Ec01'</td>
<td>42</td>
<td>969.81</td>
<td>844.71</td>
<td>917.00</td>
<td>NA</td>
<td>808.35</td>
</tr>
<tr>
<td>'So02'</td>
<td>48</td>
<td>NA</td>
<td>891.52</td>
<td>836.43</td>
<td>NA</td>
<td>899.22</td>
</tr>
<tr>
<td>'En03'</td>
<td>56</td>
<td>687.10</td>
<td>NA</td>
<td>503.38</td>
<td>873.90</td>
<td>NA</td>
</tr>
<tr>
<td>'So04'</td>
<td>48</td>
<td>455.05</td>
<td>845.29</td>
<td>866.16</td>
<td>800.39</td>
<td>956.14</td>
</tr>
<tr>
<td>'En05'</td>
<td>56</td>
<td>809.60</td>
<td>846.87</td>
<td>939.46</td>
<td>851.83</td>
<td>950.51</td>
</tr>
<tr>
<td>'Ec06'</td>
<td>42</td>
<td>919.62</td>
<td>802.45</td>
<td>717.39</td>
<td>832.44</td>
<td>974.63</td>
</tr>
<tr>
<td>'Ec07'</td>
<td>42</td>
<td>889.01</td>
<td>722.09</td>
<td>606.11</td>
<td>902.28</td>
<td>574.08</td>
</tr>
<tr>
<td>'So08'</td>
<td>48</td>
<td>862.19</td>
<td>699.38</td>
<td>907.34</td>
<td>571.18</td>
<td>943.34</td>
</tr>
<tr>
<td>'En09'</td>
<td>56</td>
<td>857.34</td>
<td>817.44</td>
<td>819.92</td>
<td>674.60</td>
<td>376.70</td>
</tr>
<tr>
<td>'Ec10'</td>
<td>42</td>
<td>NA</td>
<td>874.86</td>
<td>NA</td>
<td>847.75</td>
<td>739.94</td>
</tr>
<tr>
<td>'En11'</td>
<td>56</td>
<td>NA</td>
<td>824.24</td>
<td>855.76</td>
<td>NA</td>
<td>953.77</td>
</tr>
<tr>
<td>'Ec12'</td>
<td>42</td>
<td>802.18</td>
<td>871.06</td>
<td>488.76</td>
<td>841.41</td>
<td>599.17</td>
</tr>
<tr>
<td>'En13'</td>
<td>56</td>
<td>827.73</td>
<td>839.70</td>
<td>864.48</td>
<td>720.31</td>
<td>877.23</td>
</tr>
<tr>
<td>'So14'</td>
<td>48</td>
<td>943.31</td>
<td>580.69</td>
<td>827.45</td>
<td>815.18</td>
<td>461.04</td>
</tr>
<tr>
<td>'En15'</td>
<td>56</td>
<td>794.57</td>
<td>801.44</td>
<td>924.29</td>
<td>938.70</td>
<td>863.72</td>
</tr>
<tr>
<td>'Ec16'</td>
<td>42</td>
<td>581.15</td>
<td>599.87</td>
<td>949.84</td>
<td>367.34</td>
<td>859.70</td>
</tr>
<tr>
<td>'So17'</td>
<td>48</td>
<td>881.55</td>
<td>856.05</td>
<td>NA</td>
<td>796.10</td>
<td>655.37</td>
</tr>
<tr>
<td>'Ec18'</td>
<td>42</td>
<td>863.44</td>
<td>520.24</td>
<td>919.75</td>
<td>865.14</td>
<td>914.32</td>
</tr>
<tr>
<td>'So19'</td>
<td>48</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>790.43</td>
<td>842.85</td>
</tr>
<tr>
<td>'Ec20'</td>
<td>42</td>
<td>852.52</td>
<td>831.93</td>
<td>820.92</td>
<td>881.68</td>
<td>864.81</td>
</tr>
<tr>
<td>'So21'</td>
<td>48</td>
<td>880.87</td>
<td>NA</td>
<td>628.96</td>
<td>746.67</td>
<td>863.82</td>
</tr>
</tbody>
</table>

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 28)).

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

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 #426464 #298061 #155874 #815552 #279729 #928564

--#928564 | +22 +228 -14 +246 +36 +78 +56 +110 +318

#773909 | - +390 +90 +270 -50 +340 +220 +60 +116

#668947 | +78 - +42 +250 -22 +218 +56 +172 +74

#567308 | +70 +418 - +180 +156 +174 +266 +78 +256

#578560 | -4 +78 +28 - +100 -48 +154 -110

#426464 | +202 +258 +284 +138 - +416 +312 +382 +534

#298061 | -48 +68 +172 +32 -42 - +54 +48 +248

#155874 | +72 +378 +322 +174 +274 +466 - +212 +308

#815552 | +78 +126 +272 +318 +54 +194 +172 - -14

#279729 | +240 +230 -110 +290 +72 +140 +388 +62 -

#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()

Decimal('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 (ie the codual ~(-g) of g) shown in Fig. 20.

#(-g).exportGraphViz()

*---- exporting a dot file for GraphViz tools ---------*

Exporting to converse-dual_rel_best10.dot

dot -Tpng converse-dual_rel_best10.dot -o converse-dual_rel_best10.png

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, 773909, 155874, 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]
```
Fig. 20: Validated strict outranking situations between the ten best-ranked alternatives

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, different global ranking results.

Back to Tutorials of the Digraph3 resources (page 3)

9 Computing a best choice recommendation

- What site to choose? (page 58)
- Performance tableau (page 59)
- Outranking digraph (page 60)
- Rubis best choice recommendations (page 61)
- Rubis strict best choice recommendation (page 64)
- Weakly ordering (page 66)

See also the lecture 7 notes from the MICS Algorithmic Decision Theory course: [ADT-L7] (page 138).
9.1 What site to choose?

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

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

Three decision objectives are guiding the CEO’s choice:
1. minimize the yearly costs induced by the moving,
2. maximize the future turnover of the SME,
3. maximize the new working conditions.

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

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

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

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

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

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

Concerning annual costs, we notice that the CEO is indifferent up to a performance difference of 1000€, and he actually prefers a site if there is at least a positive difference of 2500€. The grades observed on the six qualitative
criteria (measured in percentages of satisfaction) are very subjective and rather imprecise. The CEO is hence indifferent up to a satisfaction difference of 10%, and he claims a significant preference when the satisfaction difference is at least of 20%. Furthermore, a satisfaction difference of 80% represents for him a considerably large performance difference, triggering a veto situation the case given (see [BIS-2013] (page 138)).

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

9.2 Performance tableau

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

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

```python
from perfTabs import *
t = XMCDA2PerformanceTableau('officeChoice')
help(t) # for discovering all the methods available
t.showPerformanceTableau()
```

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

```python
>>> t.showCriteria()
```

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

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

Fig. 21: Heatmap of the office choice performance tableau

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

9.3 Outranking digraph

To help now the CEO choosing the best site, we are going to compute pairwise outrankings (see [BIS-2013] (page 138)) 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] (page 138)), denoted $r(x \ S \ y)$, is measured in a bipolar significance valuation $[-100.00, 100.00]$, where positive terms $r(x \ S \ y) > 0.0$ indicate a validated, and negative terms $r(x \ S \ y) < 0.0$ indicate a non-validated outrankings; whereas the median value $r(x \ S \ y) = 0.0$ represents an indeterminate situation (see [BIS-2004] (page 139)).

For computing such a bipolar-valued outranking digraph from the given performance tableau $t$, we use the BipolarOutrankingDigraph constructor from the outrankingDigraphs-label module. The Digraph.showHTMLRelationTable method shows here the resulting bipolar-valued adjacency matrix in a system browser window (see Fig. 22).
Valued Adjacency Matrix

<table>
<thead>
<tr>
<th>r(x S y)</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.00</td>
<td>0.00</td>
<td>100.00</td>
<td>11.11</td>
<td>55.56</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>B</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-55.56</td>
<td>0.00</td>
<td>100.00</td>
<td>-55.56</td>
</tr>
<tr>
<td>C</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>33.33</td>
<td>0.00</td>
<td>100.00</td>
<td>11.11</td>
</tr>
<tr>
<td>D</td>
<td>33.33</td>
<td>55.56</td>
<td>11.11</td>
<td>0.00</td>
<td>33.33</td>
<td>100.00</td>
<td>22.22</td>
</tr>
<tr>
<td>E</td>
<td>55.56</td>
<td>0.00</td>
<td>0.00</td>
<td>-11.11</td>
<td>0.00</td>
<td>100.00</td>
<td>-11.11</td>
</tr>
<tr>
<td>F</td>
<td>0.00</td>
<td>-100.00</td>
<td>-100.00</td>
<td>-100.00</td>
<td>0.00</td>
<td>-100.00</td>
<td>-100.00</td>
</tr>
<tr>
<td>G</td>
<td>0.00</td>
<td>77.78</td>
<td>-11.11</td>
<td>100.00</td>
<td>55.56</td>
<td>100.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Fig. 22: The office choice outranking digraph

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

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

We may get even more insight in the apparent outranking situations when looking at the Condorcet digraph (see Fig.23).

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

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

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

9.4 Rubis best choice recommendations

Following the Rubis outranking method (see [BIS-2008] (page 138)), potential best choice recommendations are determined by the outranking prekernels (weakly independent and strictly outranking choices) of the chordless odd circuits augmented outranking digraph. As we observe no circuits here, we may directly compute the prekernels of $g$ (see the tutorial On computing digraph kernels (page 88)).

```
>>> g.showPreKernels()
*--- Computing preKernels ----*
Dominant preKernels :
['D']
  independence : 100.0
  dominance : 11.111
  absorbency : -100.0
  covering : 1.000
```
We notice three potential best choice recommendations: the Condorcet winner $D$ (Line 4), the triplet $B$, $C$ and $E$ (Line 9), and finally the pair $A$ and $G$ (Line 14). The Rubis best choice recommendation is given by the most determined prekernel; the one supported by the most significant criteria coalition. This result is shown with the...
We notice in Line 6 above that the most significantly supported best choice recommendation is indeed the Condorcet winner \( D \) with a majority of 56% of the criteria significance (see Line 12). Both other potential best choice recommendations, as well as the potential worst choice recommendation, are not positively validated as best, resp. worst choices. They may or may not be considered so. Alternative \( A \), with extreme contradictory performances, appears both, in a best and a worst choice recommendation (see Lines 27 and 37) and seems hence not actually comparable to its competitors.

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

\begin{verbatim}
>>> g.showBestChoiceRecommendation(CoDual=False)
******************************
Rubis best choice recommendation(s) (BCR)
(in decreasing order of determinateness)
Credibility domain: [-100.00,100.00]

--- >> potential best choice(s)
  * choice : ['D']
  +-irredundancy : 100.00
  independence : 100.00
  dominance : 11.11
  absorbency : -100.00
  covering (%) : 100.00
  determinateness (%) : 55.56
- most credible action(s) = { 'D': 2.07, }

--- >> potential best choice(s)
  * choice : ['A', 'G']
  +-irredundancy : 0.00
  independence : 0.00
  dominance : 55.56
  absorbency : 0.00
  covering (%) : 70.00
  determinateness (%) : 50.00
- most credible action(s) = { }

--- >> potential worst choice(s)
  * choice : ['B', 'C', 'E']
  +-irredundancy : 0.00
  independence : 0.00
  dominance : 11.11
  absorbency : -100.00
  covering (%) : 50.00
  determinateness (%) : 50.00
- most credible action(s) = { }

Execution time: 0.014 seconds

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

We notice in Line 6 above that the most significantly supported best choice recommendation is indeed the Condorcet winner \( D \) with a majority of 56% of the criteria significance (see Line 12). Both other potential best choice recommendations, as well as the potential worst choice recommendation, are not positively validated as best, resp. worst choices. They may or may not be considered so. Alternative \( A \), with extreme contradictory performances, appears both, in a best and a worst choice recommendation (see Lines 27 and 37) and seems hence not actually comparable to its competitors.

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

\begin{verbatim}
>>> import xmcda

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

And, in a system browser window, browse the solution file.
\end{verbatim}
The `valuationType` parameter allows to work:

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

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

```
>>> from outrankingDigraphs import RubisRestServer
>>> solver = RubisRestServer()
>>> solver.ping()
*************************************************
* This is the Leopold-Loewenheim Apache Server *
* of the University of Luxembourg. *
* Welcome to the Rubis XMCDA 2.0 Web service *
* R. Bisdorff (c) 2009-2013 *
* November 2013, version REST/D4 1.1 *
*************************************************
```

We may submit the given performance tableau.

```
>>> t = XMCDA2PerformanceTableau('officeChoice')
>>> solver.submitProblem(t)
The problem submission was successful !
Server ticket: lB3yGVvV866hSNZo
```

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

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

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

Here, we find confirmed again that alternative $D$, indeed, appears to be the most significant best choice candidate. Yet, what about alternative $G$, the other good compromise best choice we have noticed from the performance heat map shown above?

## 9.5 Rubis strict best choice recommendation

When comparing the performances of alternatives $D$ and $G$ on a pairwise perspective, we notice that, with the given preference discrimination thresholds, alternative $G$ is actually **certainly at least as good as** alternative $D$ ($r(G \text{ outranks } D) = 100.0$).
Valuation in range: -9.00 to +9.00; global concordance: +9.00

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

```
>>> g.showPairwiseComparison('C','G')
*------------ pairwise comparison ---- *
Comparing actions : (C, G)/(G, C)
crit. wght. g(x) g(y) diff. | ind. pref. (C,G)/(G,C) |
=============================================================================
C  3.00 -6700.00 -12000.00 +5300.00 | 1000.00 2500.00 +3.00/-3.00 |
Cf 1.00  10.00  50.00 -40.00 | 10.00  20.00 -1.00/+1.00 |
P  1.00 100.00  80.00 +20.00 | 10.00  20.00 +1.00/-1.00 |
Pr  1.00  80.00  60.00 +20.00 | 10.00  20.00 +1.00/-1.00 |
St  1.00  0.00  20.00 -20.00 | 10.00  20.00 -1.00/+1.00 |
V  1.00  70.00 100.00 -30.00 | 10.00  20.00 -1.00/+1.00 |
W  1.00  0.00  50.00 -50.00 | 10.00  20.00 -1.00/+1.00 |
=============================================================================
Valuation in range: -9.00 to +9.00; global concordance: +1.00/-1.00
```

To model these strict outranking situations, we may compute the Rubis best choice recommendation on the codual, the converse (~) of the dual (-), of the outranking digraph instance g (see [BIS-2013] (page 138)), as follows.

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

It is interesting to notice that the strict best choice recommendation consists in the set of weak Condorcet winners: 'A', 'C' and 'D' (see Line 6). In the corresponding characteristic vector (see Line 14-15), representing the bipolar credibility degree with which each alternative may indeed be considered a best choice (see the tutorial on Bipolar-valued kernel membership characteristic vectors (page 132) and [BIS-2006a] (page 139), [BIS-2006b] (page 139)), we find confirmed that alternative D is the only positively validated one, whereas both extreme alternatives - A (the most expensive) and C (the cheapest) - stay in an indeterminate situation. They may be potential best choice candidates besides D. Notice furthermore that compromise alternative G, while not actually included in the crisp best choice recommendation, shows as well an indeterminate situation with respect to being or not a potential best choice candidate.

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

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

```python
>>> from weakOrders import RankingByChoosingDigraph
>>> gcd = ~(-g)
>>> rbc = RankingByChoosingDigraph(gcd)

Exiting computing threads

>>> rbc.showRankingByChoosing()
Ranking by Choosing and Rejecting
1st ranked ['D'] (0.28)
2nd ranked ['C', 'G'] (0.17)
2nd last ranked ['B', 'C', 'E'] (0.22)
1st last ranked ['A', 'F'] (0.50)

>>> rbc.exportGraphViz('officeChoiceRanking')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to officeChoiceRanking.dot
0 { rank = same; A; C; D; }
1 { rank = same; G; }
2 { rank = same; E; B; }
3 { rank = same; F; }
dot -Grankdir=TB -Tpng officeChoiceRanking.dot -o officeChoiceRanking.png
```

Fig. 24: Ranking-by-choosing from the office choice outranking digraph

In this ranking-by-choosing method, where we operate the epistemic fusion of iterated (strict) best and worst choices, compromise alternative D is indeed ranked before compromise alternative G. If the computing node supports multiple processor cores, best and worst choosing iterations are run in parallel. The overall partial ordering result shows again the important fact that the most expensive site A, and the cheapest site C, both appear incomparable with most of the other alternatives, as is apparent from the Hasse diagram (see above) of the ranking-by-choosing relation.
The best choice recommendation appears hence depending on the very importance the CEO is attaching to each of the three objectives he is considering. In the setting here, where he considers all three objectives to be equally important (minimize costs = 3.0, maximize turnover = 3.0, and maximize working conditions = 3.0), site D represents actually the best compromise. However, if Costs do not play much 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 on the one hand this importance balance in the XMCDA data file by lowering the significance of the Costs criterion; all criteria are considered equi-significant (weight = 1.0) for instance. It may as well be opportune, on the other hand, to rank the importance of the three objectives as follows: minimize costs (weight = 9.0) > maximize turnover (weight = 3 x 2.0) > maximize working conditions (weight = 3 x 1.0). What will become the best choice recommendation under both working hypotheses?

Back to Tutorials of the Digraph3 resources (page 3)

10 Rating with learned quantile norms

- Introduction (page 67)
- Incremental learning of historical performance quantiles (page 68)
- Rating new performances with quantile norms (page 70)

10.1 Introduction

In this tutorial we address the problem of rating multiple criteria performances of a set of potential decision alternatives with respect to empirical order statistics, ie performance quantiles learned from historical performance data gathered from similar decision alternatives observed in the past (see [CPSTAT-L5] (page 138)).

To illustrate the decision problem we face, consider for a moment that, in a given decision aid study, we observe, for instance in the Table below, the multi-criteria performances of two potential decision alternatives, named a1001 and a1010, marked on 7 incommensurable preference criteria: 2 costs criteria c1 and c2 (to minimize) and 6 benefits criteria b1 to b5 (to maximize).

<table>
<thead>
<tr>
<th>Criterion</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
<th>b5</th>
<th>c1</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>a1001</td>
<td>37.0</td>
<td>2</td>
<td>2</td>
<td>61.0</td>
<td>31.0</td>
<td>-4</td>
<td>-40.0</td>
</tr>
<tr>
<td>a1010</td>
<td>32.0</td>
<td>9</td>
<td>6</td>
<td>55.0</td>
<td>51.0</td>
<td>-4</td>
<td>-35.0</td>
</tr>
</tbody>
</table>

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

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

The non trivial decision problem we now face here, is to decide, how the multiple criteria performances of a1001, respectively a1010, may be rated (excellent?, good?, or fair?; perhaps even, weak? or very weak?) in an order statistical sense, when compared with all potential similar multi-criteria performances one has already encountered in the past.

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

See also the technical documentation of the performanceQuantiles-label.

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] (page 139) and [NR3-2007] (page 138), who present an efficient algorithm for incrementally updating a quantile-binned cumulative distribution function (CDF) with newly observed CDFs.

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

- Ordered objectives and a criteria dictionaries from a valid performance tableau instance;
- A list quantileFrequencies of quantile frequencies like quartiles [0.0, 0.25, 0.5, 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.

```python
>>> 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, 
... numberOfBins = 'quartiles', 
... LowerClosed=True)
```
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 column 2). The preference direction of the *costs* criteria $c_1$ and $c_2$ is **negative**: the lesser the costs the better it is, whereas all the *benefits* criteria $b_1$ to $b_5$ show **positive** preference directions, ie the worst, resp. best performance observed so far on each criterion. Column ‘0.5’ shows the **median** ($Q_2$) observed on the criteria.

New decision alternatives with random multiple criteria performance vectors from the same random performance tableau model may now be generated with ad hoc random performance generators. We provide for experimental purpose, in the `randomPerfTabs` module, three such generators: one for the standard `randomPerfTabs.RandomPerformanceTableau` model, one for the two objectives `randomPerfTabs.RandomCBPerformanceTableau` Cost-Benefit model, and one for the `randomPerfTabs.Random3ObjectivesPerformanceTableau` model with three objectives concerning respectively economic, environmental or social aspects.

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

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

Parameter `historySize` (see Line 6) of the `performanceQuantiles.PerformanceQuantiles.updateQuantiles()` method allows to **balance** the new evaluations against the historical ones. With `historySize = None` (the default setting), the balance in the example above is 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. 25).

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

---

**Performance quantiles**

**Sampling sizes between 986 and 995.**

```
<table>
<thead>
<tr>
<th>criterion</th>
<th>0.00</th>
<th>0.25</th>
<th>0.50</th>
<th>0.75</th>
<th>1.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>1.99</td>
<td>28.77</td>
<td>49.63</td>
<td>75.27</td>
<td>99.83</td>
</tr>
<tr>
<td>b2</td>
<td>0.00</td>
<td>2.94</td>
<td>4.92</td>
<td>6.72</td>
<td>10.00</td>
</tr>
<tr>
<td>b3</td>
<td>0.00</td>
<td>2.90</td>
<td>4.86</td>
<td>8.01</td>
<td>10.00</td>
</tr>
<tr>
<td>b4</td>
<td>3.27</td>
<td>35.91</td>
<td>59.58</td>
<td>72.00</td>
<td>96.05</td>
</tr>
<tr>
<td>b5</td>
<td>0.85</td>
<td>32.84</td>
<td>48.09</td>
<td>69.75</td>
<td>99.00</td>
</tr>
<tr>
<td>c1</td>
<td>-10.00</td>
<td>-7.35</td>
<td>-5.39</td>
<td>-3.38</td>
<td>0.00</td>
</tr>
<tr>
<td>c2</td>
<td>-96.37</td>
<td>-72.22</td>
<td>-52.27</td>
<td>-33.99</td>
<td>-1.43</td>
</tr>
</tbody>
</table>
```

Fig. 25: Showing the updated quartiles limits
10.3 Rating new performances with quantile norms

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

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

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

We reconsider the `PerformanceQuantiles` object instance `pq` as computed in the previous section. Let `newActions` be a list of 10 new decision alternatives generated with the same random performance tableau model and including the two decision alternatives `a1001` and `a1010` mentioned at the beginning.

```python
>>> from sortingDigraphs import NormedQuantilesRatingDigraph
>>> newActions = rpg.randomActions(10)
>>> nqr = NormedQuantilesRatingDigraph(pq,newActions,rankingRule='best')
>>> nqr

*----- Object instance description ----------- *
Instance class : NormedQuantilesRatingDigraph
Instance name : normedRatingDigraph
# Criteria : 7
# Quantile profiles : 4
# New actions : 10
Size : 93
Determinateness (%) : 52.17
Attributes: ['runTimes', 'objectives', 'criteria',
'LowerClosed', 'quantilesFrequencies', 'limitingQuantiles',
'historySizes', 'cdf', 'name', 'newActions', 'evaluation',
'categories', 'criteriaCategoryLimits', 'profiles', 'profileLimits',
'hasNoVeto', 'actions', 'completeRelation', 'relation',
'concordanceRelation', 'valuationdomain', 'order', 'gamma',
'notGamma', 'rankingRule', 'rankingCorrelation', 'rankingScores',
'actionsRanking', 'ratingCategories', 'ratingRelation',
'relationOrig', 'rankingByBestChoosing']
----------- Constructor run times (in sec.) -----------
#Threads : 1
Total time : 0.01636
Data input : 0.00051
Quantile classes : 0.00006
Compute profiles : 0.00005
Compute relation : 0.01420
Compute rating : 0.00154
```

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

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

```python
>>> nqr.showPerformanceTableau(actionsSubset=nqr.newActions)

*---- performance tableau ----- *
criteria | a1001 a1002 a1003 a1004 a1005 a1006 a1007 a1008 a1009 a1010
-----------|-------------------------------
```
Among the 10 new incoming decision alternatives (see below), we recognize alternatives a1001 (see column 2) and a1010 (see last column) we have mentioned in our introduction.

The NormedQuantilesRatingDigraph instance’s actions dictionary also contains the closed lower limits of the four quartile classes: \( m_1 = [0.0-[], m_2 = [0.25 -[, m_3 = [0.5-[, m_4 = [0.75 -]. \)

The main run time (see Lines 23-29 of the object description above) is spent by the class constructor in computing a bipolar-valued outranking relation on the extended actions set including both the new alternatives as well as the quartile class limits. In case of large volumes, ie many new decision alternatives and centile classes for instance, a multi-threading version may be used when multiple processing cores are available (see the technical description of the sortingDigraphs.NormedQuantilesRatingDigraph class).

The actual rating procedure will rely on a complete ranking of the new decision alternatives as well as the quartile class limits obtained from the corresponding bipolar-valued outranking digraph. Two efficient and scalable ranking rules, the Copeland and its valued version, the Netflows rule may be used for this purpose. The rankingRule parameter allows to choose one of both. With rankingRule='best' (see Line 2 above) the NormedQuantilesRatingDigraph constructor will choose the ranking rule that results in the highest ordinal correlation with the given outranking relation (see [BIS-2012] (page 138)).

In this rating example, the NetFlows rule appears to be the more appropriate ranking rule.

We achieve here a linear ranking without ties (from best to worst) of the digraph’s actions, ie including the new decision alternatives as well as the quartile limits \( m_1 \) to \( m_4 \), which is very close in an ordinal sense (\( \tau = 0.94 \)) to the underlying valued outranking relation.

The eventual rating procedure is based on the lower quantile limits, such that we may collect the quartile classes’ contents in increasing order of the quartiles’ lower limits.
We notice above that no new decision alternative is rated in the lowest [0.0-0.25], respectively highest [0.75- [quartile class. Indeed, the rating result is shown, in descending order, as follows:

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

The same result may even more conveniently be consulted in a browser view via a specialised rating heatmap format (see `perfTabs:PerformanceTableau.showHTMLPerformanceHeatmap()` method (see Fig. 26).

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

### Heatmap of Quartiles rating

**Ranking rule: NetFlows; Ranking correlation: 0.938**

<table>
<thead>
<tr>
<th>criteria</th>
<th>weights</th>
<th>b3</th>
<th>c2</th>
<th>b2</th>
<th>b1</th>
<th>c1</th>
<th>b5</th>
<th>b4</th>
</tr>
</thead>
<tbody>
<tr>
<td>tau(*)</td>
<td></td>
<td>0.62 0.56 0.43 0.42 0.41 0.37 0.29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0.75 - 0.50]</td>
<td>[8.01 3.99 6.72 75.27 -3.38 69.75 72.00]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1005c</td>
<td>0.00 24.00 3.00 42.00 -1.00 30.00 28.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1010n</td>
<td>6.00 -35.00 9.00 32.00 -4.00 51.00 55.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1008n</td>
<td>6.00 -43.00 5.00 64.00 -6.00 96.00 49.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1002c</td>
<td>4.00 -23.00 5.00 27.00 -6.00 63.00 54.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1006c</td>
<td>3.00 -27.00 3.00 33.00 -5.00 39.00 20.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0.50 - 0.25]</td>
<td>[4.86 -52.27 4.92 49.63 -5.39 48.09 58.58]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1003a</td>
<td>2.00 -37.00 8.00 24.00 -8.00 61.00 74.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1001c</td>
<td>2.00 -40.00 2.00 37.00 -4.00 31.00 61.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1007c</td>
<td>2.00 -73.00 6.00 39.00 -1.00 16.00 20.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1004c</td>
<td>1.00 -37.00 3.00 16.00 -5.00 48.00 25.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1009n</td>
<td>6.00 -94.00 4.00 42.00 -6.00 57.00 44.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0.25 - 0.00]</td>
<td>[2.90 -72.22 2.94 28.77 -7.35 32.84 35.91]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0.00 -]</td>
<td>[0.00 -96.37 0.00 1.99 -10.00 0.85 3.27]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Color legend:**

<table>
<thead>
<tr>
<th>quantile</th>
<th>20.00%</th>
<th>40.00%</th>
<th>60.00%</th>
<th>80.00%</th>
<th>100.00%</th>
</tr>
</thead>
</table>

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

Fig. 26: Heatmap of normed quartiles ranking

Using furthermore a specialised version of the `weakOrders.WeakOrder.exportGraphViz()` method allows drawing the same rating result in a Hasse diagram format (see Fig. 27).

```python
>>> nqr.exportRatingGraphViz('normedRatingDigraph')
""" exporting a dot file for GraphViz tools """
Exporting to normedRatingDigraph.dot
    dot -Grankdir=TB -Tpng normedRatingDigraph.dot -o normedRatingDigraph.png"""
Fig. 27: Normed quartiles rating digraph
We may now answer the normed rating decision problem stated at the beginning. Decision alternative $a1001$ is rated in quartile $Q2$ and alternative $a1010$ in quartile $Q3$ (see Table below). Indeed, the performances of decision alternative $a1001$ were generated with a triangular law at a low mode, i.e. low costs but also low benefits, whereas the performances of alternative $a1010$ were generated with a median mode.

<table>
<thead>
<tr>
<th>Rating</th>
<th>Criterion</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
<th>b5</th>
<th>c1</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>quartiles</td>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Q2</td>
<td>$a1001$</td>
<td>37.0</td>
<td>2</td>
<td>2</td>
<td>61.0</td>
<td>31.0</td>
<td>-4</td>
<td>-40.0</td>
</tr>
<tr>
<td>Q3</td>
<td>$a1010$</td>
<td>32.0</td>
<td>9</td>
<td>6</td>
<td>55.0</td>
<td>51.0</td>
<td>-4</td>
<td>-35.0</td>
</tr>
</tbody>
</table>

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

Compared with the quartiles rating result, we notice that the five alternatives ($a1002$, $a1005$, $a1006$, $a1008$, and $a1010$), rated before into the third quartile class [0.50-0.75], are now divided up: alternatives $a1002$, $a1005$, $a1008$, and $a1010$ attain the 7th decile class [0.6-0.7], whereas alternative $a1006$ attains only the the 6th decile class [0.5-0.6]. Of the five $Q2$ [0.25-0.5] rated alternatives ($a1001$, $a1003$, $a1004$, $a1006$ and $a1007$), alternatives $a1001$ and $a1003$ are now rated in the 6th decile class [0.5 - 0.6], whereas $a1004$ and $a1007$ are rated the 5th decile class [0.4-0.5] and $a1009$ is lowest rated in the 4th decile class [0.3 - 0.4].

A browser view may again more conveniently illustrate this preciser deciles rating result (see Fig. 28).

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

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 with dodeciles or even centiles. Especially if tail rating results, i.e. distinguishing very best, or very worst multiple criteria performances, becomes a critical purpose. Similarly, the historySize parameter may be used for monitoring on the fly unstable random multiple criteria performance data.

11 Working with the graphs module

- Structure of a Graph object (page 76)
- $q$-coloring of a graph (page 78)
- MIS and clique enumeration (page 79)
- Line graphs and maximal matchings (page 80)
- Grids and the Ising model (page 82)
**Heatmap of Deciles rating**

*Ranking rule: NetFlows, Ranking correlation: **0.960***

<table>
<thead>
<tr>
<th>criteria</th>
<th>weights</th>
<th>tau(*)</th>
<th>c2</th>
<th>b3</th>
<th>c1</th>
<th>b1</th>
<th>b5</th>
<th>b2</th>
<th>b4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>quantile</th>
<th>20.00%</th>
<th>40.00%</th>
<th>60.00%</th>
<th>80.00%</th>
<th>100.00%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Fig. 28: Heatmap of mormed deciles rating
11.1 Structure of a Graph object

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

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

See the technical documentation of the graphs-label.

Example Python3 session

```python
>>> 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:

```python
# Graph instance saved in Python format
vertices = {
    'v1': {'shortName': 'v1', 'name': 'random vertex'},
    'v2': {'shortName': 'v2', 'name': 'random vertex'},
    'v3': {'shortName': 'v3', 'name': 'random vertex'},
    'v4': {'shortName': 'v4', 'name': 'random vertex'},
    'v5': {'shortName': 'v5', 'name': 'random vertex'},
    'v6': {'shortName': 'v6', 'name': 'random vertex'},
    'v7': {'shortName': 'v7', 'name': 'random vertex'},
}
valuationDomain = {'min':-1,'med':0,'max':1}
edges = {
    frozenset(['v1', 'v2']): -1,
    frozenset(['v1', 'v3']): -1,
    frozenset(['v1', 'v4']): -1,
    frozenset(['v1', 'v5']): 1,
    frozenset(['v1', 'v6']): -1,
    frozenset(['v1', 'v7']): -1,
    frozenset(['v2', 'v3']): 1,
    frozenset(['v2', 'v4']): 1,
    frozenset(['v2', 'v5']): -1,
    frozenset(['v2', 'v6']): 1,
    frozenset(['v2', 'v7']): -1,
    frozenset(['v3', 'v4']): -1,
    frozenset(['v3', 'v5']): -1,
    frozenset(['v3', 'v6']): -1,
    frozenset(['v3', 'v7']): -1,
    frozenset(['v4', 'v5']): 1,
    frozenset(['v4', 'v6']): -1,
    frozenset(['v4', 'v7']): 1,
    frozenset(['v5', 'v6']): 1,
    frozenset(['v5', 'v7']): -1,
    frozenset(['v6', 'v7']): 1,
}
```
The stored graph can be recalled and plotted with the generic `graphs.Graph.exportGraphViz()` method as follows.

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

![Graph diagram](image)

**Fig. 29:** Tutorial graph instance

Properties, like the gamma function and vertex degrees and neighbourhood depths may be shown with a `graphs.Graph.showShort()` method.

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

A `Graph` instance corresponds bijectively to a symmetric `Digraph` instance and we may easily convert from one to the other with the `graphs.Graph.graph2Digraph()` and vice versa with the `digraphs.Digraph.digraph2Graph()` method. Thus, all resources of the `digraphs.Digraph` class, suitable for symmetric digraphs, become readily available, and vice versa.
```python
>>> 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 - -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]
bnh depths : [0, 1, 2, 3, 4, 5, 6, 'inf. ']
distribution : [0, 0, 1, 4, 2, 0, 0, 0]

11.2 q-coloring of a graph

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

```
11.3 MIS and clique enumeration

2-colorings define independent sets of vertices that are maximal in cardinality; for short called a MIS. Computing such MISs in a given Graph instance may be achieved by the `graphs.Graph.showMIS()` method.
A MIS in the dual of a graph instance \( g \) (its negation \(-g\) ), corresponds to a maximal \textit{clique}, i.e., a maximal complete subgraph in \( g \). Maximal cliques may be directly enumerated with the \texttt{graphs.Graph.showClique()} method.
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.

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.
11.5 Grids and the Ising model

Special classes of graphs, like \( n \times m \) rectangular or triangular grids (graphs.GridGraph and graphs.IsingModel) are available in the graphs module. For instance, we may use a Gibbs sampler again for simulating an Ising Model on such a grid.
Fig. 33: Ising model of the 15x15 grid graph
11.6 Simulating Metropolis random walks

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

```python
>>> from graphs import MetropolisChain
>>> g = Graph(numberOfVertices=5, edgeProbability=0.5)
>>> g.showShort()
+---- short description of the graph ----+
| Name : 'randomGraph'
| Vertices : ['v1', 'v2', 'v3', 'v4', 'v5']
| Valuation domain : {'max': 1, 'med': 0, 'min': -1}
| Gamma function :
| v1 -> ['v2', 'v3', 'v4']
| v2 -> ['v1', 'v4']
| v3 -> ['v5', 'v1']
| v4 -> ['v2', 'v5', 'v1']
| v5 -> ['v3', 'v4']

>>> probs = {}  # 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()
... probs[v] = (n - i)/(n*(n+1)/2)
... i += 1
>>> met = MetropolisChain(g, probs)
>>> frequency = met.checkSampling(verticesList[0], nSim=30000)
>>> for v in verticesList:
...    print(v, probs[v], frequency[v])
...    v1 0.3333 0.3343
...    v2 0.2666 0.2680
...    v3 0.2 0.2030
...    v4 0.1333 0.1311
...    v5 0.0666 0.0635

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

For more technical information and more code examples, look into the technical documentation of the graphs-label. For the readers interested in algorithmic applications of Markov Chains we may recommend consulting O. Häggström’s 2002 book: [FMCAA] (page 139).

Back to Tutorials of the Digraph3 resources (page 3)
12 Computing the non isomorphic MISs of the n-cycle graph

Due to the public success of our common 2008 publication with Jean-Luc Marichal [ISOMIS-08] (page 138), we present in this tutorial an example Python session for computing the non isomorphic maximal independent sets (MISs) from the 12-cycle graph, i.e., a digraphs.CirculantDigraph class instance of order 12 and symmetric circulants 1 and -1.

```python
>>> from digraphs import CirculantDigraph
>>> c12 = CirculantDigraph(order=12, circulants=[1, -1])
>>> # 12-cycle digraph instance
```

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

```python
>>> from graphs import CycleGraph
>>> cg12 = CycleGraph(order=12)
>>> # 12-cycle graph instance
```

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

```python
>>> c12.showMIS(withListing=False)
```

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 along with all the Digraphs3 python modules for computing the set of MISs observed in the graph.

```bash
...$ echo 12 | /usr/local/bin/perrinMIS
```

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

---

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

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

```python
>>> 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'}),
...
...}
```

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

```python
>>> c12.automorphismGenerators()
```
```
{'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'}
```

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

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

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

The corresponding group stabilizers’ sizes and frequencies – orbit 1 with 12 symmetry axes, orbit 2 with 8 symmetry axes, and orbits 3 and 4 both with one symmetry axis (see Lines 11-13), are illustrated in the corresponding unlabelled graphs of Fig. 35 below.

---

2 Dependency: The `digraphs.Digraph.automorphismGenerators()` method uses the shell `dreadnaut` command from the `nauty` software package. See https://www3.cs.stonybrook.edu/~algorith/implement/nauty/implement.shtml. On Mac OS there exist dmg installers and on Ubuntu Linux or Debian, one may easily install it with `$ sudo apt-get install nauty`.
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] (page 138)).

Back to Tutorials of the Digraph3 resources (page 3)

13 On computing digraph kernels

- What is a graph kernel? (page 88)
- Initial and terminal kernels (page 93)
- Kernels in lateralized digraphs (page 95)
- Computing good and bad choice recommendations (page 97)
- Tractability (page 100)

13.1 What is a graph kernel?

We call choice in a graph, respectively a digraph, a subset of its vertices, resp. of its nodes or actions. A choice $Y$ is called internally stable or independent when there exist no links (edges) or relations (arcs) between its members. Furthermore, a choice $Y$ is called externally stable when for each vertex, node or action $x$ not in $Y$, there exists at least a member $y$ of $Y$ such that $x$ is linked or related to $y$. Now, an internally and externally stable choice is called a kernel.
A first trivial example is immediately given by the maximal independent vertices sets (MISs) of the n-cycle graph (see Computing the non isomorphic MISs of the n-cycle graph (page 85)). Indeed, each MIS in the n-cycle graph is by definition independent, ie internally stable, and each non selected vertex in the n-cycle graph is in relation with either one or even two members of the MIS. See, for instance, the four non isomorphic MISs of the 12-cycle graph as shown in Fig. 35.

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

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

```python
>>> from graphs import RandomRegularGraph
>>> g = RandomRegularGraph(order=12,degree=3,seed=100)
>>> g.exportGraphViz('random3RegularGraph')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to random3RegularGraph.dot
fdp -Tpng random3RegularGraph.dot -o random3RegularGraph.png
```

Fig. 36: A random 3-regular graph instance

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

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

It is easily verified in Fig. 37 above, that the computed MIS renders indeed a valid kernel of the given graph. The complete set of kernels of this 3-regular graph instance coincides hence with the set of its MISs.

```python
>>> g.showMIS()
*--- Maximal Independent Sets ---*
['a01', 'a02', 'a03', 'a07']
['a01', 'a04', 'a05', 'a08']
```
We cannot resist in looking in this 3-regular graph for non isomorphic kernels (MISs, see previous tutorial). To do so we must first, convert the given graph instance into a digraph instance. Then, compute its automorphism generators, and finally, identify the isomorphic kernel orbits.
In our random 3-regular graph instance (see Fig. 36), we may thus find eleven non isomorphic kernels with orbit sizes equal to two. We illustrate below the isomorphic twin of the random MIS example shown in Fig. 37.

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

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

```python
>>> from graphs import CycleGraph
>>> c8 = CycleGraph(order=8)
>>> maxMatching = c8.computeMaximumMatching()
```
In the context of digraphs, i.e. oriented graphs, the kernel concept gets much richer and separates from the symmetric MIS concept.
13.2 Initial and terminal kernels

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

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

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

A both internally and dominant, resp. absorbent stable choice is called a dominant or initial, resp. an absorbent or terminal kernel. From a topological perspective, the initial kernel concept looks from the outside of the digraph into its interior, whereas the terminal kernel looks from the interior of a digraph toward its outside. From an algebraic perspective, the initial kernel is a prefix operand, and the terminal kernel is a postfix operand in the Berge kernel equation (see [BIS-2006a] (page 139)).

Furthermore, as the kernel concept involves conjointly a positive logical refutation (the internal stability) and a positive logical affirmation (the external stability), it appeared rather quickly necessary in our operational developments to adopt a bipolar characteristic [-1,1] valuation domain, modelling negation by change of numerical sign and including explicitly a third median logical value (0) expressing logical indeterminateness (neither positive, nor negative, see [BIS-2000] (page 139) and [BIS-2004] (page 139)).

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

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

```python
>>> from digraphs import CompleteDigraph
>>> u = CompleteDigraph(order=5)
>>> u

*------- Digraph instance description ------*
Instance class : CompleteDigraph
Instance name : complete
Digraph Order : 5
Digraph Size : 20
Valuation domain : [-1.00 ; 1.00]
---------------------------------

>>> u.showPreKernels()

*--- Computing preKernels ---*
Dominant kernels :
['1'] independence: 1.0; dominance : 1.0; absorbency : 1.0
['2'] independence: 1.0; dominance : 1.0; absorbency : 1.0
['3'] independence: 1.0; dominance : 1.0; absorbency : 1.0
['4'] independence: 1.0; dominance : 1.0; absorbency : 1.0
['5'] independence: 1.0; dominance : 1.0; absorbency : 1.0
Absorbent kernels :
['1'] independence: 1.0; dominance : 1.0; absorbency : 1.0
['2'] independence: 1.0; dominance : 1.0; absorbency : 1.0
['3'] independence: 1.0; dominance : 1.0; absorbency : 1.0
['4'] independence: 1.0; dominance : 1.0; absorbency : 1.0
['5'] independence: 1.0; dominance : 1.0; absorbency : 1.0

*----- statistics -----*
graph name: complete
number of solutions
dominant kernels : 5
absorbent kernels: 5
cardinality frequency distributions
cardinality : [0, 1, 2, 3, 4, 5]
dominant kernel : [0, 5, 0, 0, 0, 0]
absorbent kernel: [0, 5, 0, 0, 0, 0]
```
In a complete digraph, each single node is indeed both an initial and a terminal kernel candidate and there is no definite begin or end of the digraph to be detected. Laterality is here entirely relative to a specific singleton chosen as reference point of view. The same absence of laterality is apparent in two other universal digraph models, the empty and the indeterminate digraph.

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

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

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

Chordless circuits of even length 2 x k, with k > 1, contain however two isomorphic kernels of cardinality k which qualify conjointly as initial and terminal candidates.
Chordless circuits of even length may thus be indifferently oriented along two opposite directions. Notice by the way that the duals of all chordless circuits of odd or even length, i.e. filled circuits also called anti-holes (see Fig. 40), never contain any potential kernel candidates.

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

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

```python
>>> from randomDigraphs import RandomDigraph
>>> rd = RandomDigraph(order=7, arcProbability=0.3, seed=5)
>>> rd.exportGraphViz('randomLaterality')
*---- exporting a dot file for GraphViz tools -----------*
Exporting to randomLaterality.dot
```

Fig. 41: A random digraph instance of order 7 and arc probability 0.3

The random digraph shown in Fig. 41 above has no apparent special properties, except from being connected.

The given digraph instance is neither asymmetric (a3 <-> a6) nor symmetric (a2 -> a1, a1 -> a2); there are no chordless circuits (see Line 5 above); and, the digraph is not transitive (a5 -> a2 -> a1, but a5 -> a1). More than half of the required transitive closures are missing (see Line 8).

Now, we know that its potential prekernels must be among its set of maximal independent choices.
Among the six MISs contained in this random digraph (see above Lines 3-8) we discover two initial and two terminal kernels (Lines 12-34). Notice by the way the covering values (between 0.0 and 1.0) shown by the Digraph.showPreKernels() method (Lines 17, 22, 28 and 33). The higher this value, the more the corresponding kernel candidate makes apparent the digraph’s laterality. We may hence redraw the same digraph in Fig. 42 by looking into its interior via the best covering initial kernel candidate: the dominant choice {'3','4'} (coloured in yellow), and looking out of it via the best covered terminal kernel candidate: the absorbent choice {'1','6'} (coloured in blue).

In algorithmic decision theory, initial and terminal prekernels may provide convincing best, resp. worst, choice recommendations (see Computing a best choice recommendation (page 57)).

### 13.4 Computing good and bad choice recommendations

To illustrate this idea, let us finally compute good and bad choice recommendations in the following random bipolar-valued outranking digraph.
Fig. 42: A random digraph oriented by best covering initial and best covered terminal kernel

Fig. 43: The performance tableau of a random outranking digraph instance

The underlying random performance tableau (see Fig. 43) shows the performance grading of 7 potential decision
actions with respect to 7 decision criteria supporting each an increasing performance scale from 0 to 100. Notice
the missing performance data concerning decision actions ‘a2’ and ‘a5’. The resulting strict outranking - ie a
weighted majority supported - better than without considerable counter-performance - digraph is shown in Fig.
44 below.

```plaintext
1 >>> gcd = ~(-g) # Codual: the converse of the negation
2 >>> gcd.exportGraphViz(fileName='tutOutRanking')
3 *---- exporting a dot file for GraphViz tools ---------*
4 Exporting to tutOutranking.dot
5 dot -Grankdir=BT -Tpng tutOutranking.dot -o tutOutranking.png
```

![Fig. 44: A random strict outranking digraph instance](image)

Fig. 44: A random strict outranking digraph instance

All decision actions appear strictly better performing than action ‘a7’. We call it a Condorcet looser and it is an
evident terminal prekernel candidate. On the other side, three actions: ‘a1’, ‘a2’ and ‘a4’ are not dominated. They
give together an initial prekernel candidate.

```plaintext
1 >>> gcd.showPreKernels()
2 *--- Computing preKernels ----*
3 Dominant preKernels:
4 ['a1', 'a2', 'a4']
5  independence :  0.00
6  dominance :  6.98
7  absorbency :  -48.84
8  covering :  0.667
9 Absorbent preKernels:
10  ['a3', 'a7']
11  independence :  0.00
12  dominance :  -74.42
13  absorbency :  16.28
14  covered :  0.800
```

With such unique disjoint initial and terminal prekernels (see Line 4 and 10), the given digraph instance is hence
clearly lateralized. Indeed, these initial and terminal prekernels of the codual outranking digraph reveal best, resp. worst, choice recommendations one may formulate on the basis of a given outranking digraph instance.

```python
>>> g.showRubisBestChoiceRecommendation()
*---------------------------------------------------------------*
Rubis best choice recommendation(s) (BCR)
(in decreasing order of determinateness)
Credibility domain: [-100.00,100.00]
--- >> potential best choice(s)
* choice : ['a1', 'a2', 'a4']
+-irredundancy : 0.00
independence : 0.00
dominance : 6.98
absorbency : -48.84
covering (%) : 66.67
determinateness (%) : 57.97
- most credible action(s) = { 'a4': 20.93, 'a2': 20.93, }
--- >> potential worst choice(s)
* choice : ['a3', 'a7']
+-irredundancy : 0.00
independence : 0.00
dominance : -74.42
absorbency : 16.28
covered (%) : 80.00
determinateness (%) : 64.62
- most credible action(s) = { 'a7': 48.84, }
```

Notice that solving the valued Berge kernel equations (IBIS-2006a) (page 139)) provides furthermore a positive characterization of the most credible decision actions in each respective choice recommendation (see Lines 14 and 23 above). Actions ‘a2’ and ‘a4’ are equivalent candidates for a unique best choice, and action ‘a7’ is clearly confirmed as the worst choice.

In Fig. 45 below, we orient the drawing of the strict outranking digraph instance with the help of these best and worst choice recommendations.

```python
>>> gcd.exportGraphViz(fileName='bestWorstOrientation',
... bestChoice=['a2','a4'], worstChoice=['a7'])
*---- exporting a dot file for GraphViz tools ---------*
Exporting to bestWorstOrientation.dot
dot -Grankdir=BT -Tpng bestWorstOrientation.dot -o bestWorstOrientation.png
```

The grey arrows in Fig. 45, like the one between actions ‘a4’ and ‘a1’, represent indeterminate preferential situations. Action ‘a1’ appears hence to be rather incomparable to all the other, except action ‘a7’. It may be interesting to compare this result with a Copeland ranking of the underlying performance tableau (see Ranking with multiple incommensurable criteria (page 36)).

```python
>>> g.showHTMLPerformanceHeatmap(colorLevels=5, ndigits=0,
... Correlations=True, rankingRule='Copeland')
```

In the resulting linear ranking (see Fig. 46), action ‘a4’ is set at first rank, followed by action ‘a2’. This makes sense as ‘a4’ shows three performances in the first quintile, whereas ‘a2’ is only partially evaluated and shows only two such excellent performances. But ‘a4’ also shows a very weak performance in the first quintile. Both decision actions, hence, don’t show eventually a performance profile that would make apparent a clear preference situation in favour of one or the other. In this sense, the prekernels based best choice recommendations may appear more faithful with respect to the actually definite strict outranking relation than any ‘forced’ linear ranking result as shown in Fig. 46 above.

### 13.5 Tractability

Finally, let us give some hints on the tractability of kernel computations. Detecting all (pre)kernels in a digraph is a famously NP-hard computational problem. Checking external stability conditions for an independent choice
Fig. 45: The strict outranking digraph oriented by its best and worst choice recommendations

Heatmap of Performance Tableau 'randomOutranking'

<table>
<thead>
<tr>
<th>criteria</th>
<th>g4</th>
<th>g7</th>
<th>g3</th>
<th>g6</th>
<th>g1</th>
<th>g2</th>
<th>g3</th>
</tr>
</thead>
<tbody>
<tr>
<td>weights</td>
<td>9</td>
<td>10</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>tau(1)</td>
<td>+0.64</td>
<td>+0.40</td>
<td>+0.29</td>
<td>+0.17</td>
<td>+0.02</td>
<td>-0.05</td>
<td>-0.10</td>
</tr>
<tr>
<td>a4</td>
<td>96</td>
<td>32</td>
<td>89</td>
<td>60</td>
<td>47</td>
<td>92</td>
<td>0</td>
</tr>
<tr>
<td>a2</td>
<td>87</td>
<td>82</td>
<td>69</td>
<td>7</td>
<td>NA</td>
<td>NA</td>
<td>62</td>
</tr>
<tr>
<td>a6</td>
<td>68</td>
<td>70</td>
<td>36</td>
<td>68</td>
<td>54</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>a1</td>
<td>98</td>
<td>31</td>
<td>94</td>
<td>15</td>
<td>65</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>a5</td>
<td>54</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>NA</td>
<td>77</td>
<td>87</td>
</tr>
<tr>
<td>a3</td>
<td>29</td>
<td>48</td>
<td>97</td>
<td>30</td>
<td>11</td>
<td>28</td>
<td>13</td>
</tr>
<tr>
<td>a7</td>
<td>20</td>
<td>6</td>
<td>17</td>
<td>34</td>
<td>57</td>
<td>80</td>
<td>22</td>
</tr>
</tbody>
</table>

Color legend:
- Quantile: 20.00% 40.00% 60.00% 80.00% 100.00%

(*) tau: Ordinal (Kendall) correlation between marginal criterion and global ranking relation
Ranking rule: Copeland
Ordinal (Kendall) correlation between global ranking and global outranking relation: +0.848

Fig. 46: Heatmap with Copeland ranking of the performance tableau
is equivalent to checking its maximality and may be done in the linear complexity of the order of the digraph. However, checking all independent choices contained in a digraph may get hard already for tiny sparse digraphs of order \( n > 20 \) (see [BIS-2006b] (page 139)). Indeed, the worst case is given by an empty or indeterminate digraph where the set of all potential independent choices to check is in fact the power set of the vertices.

```python
>>> e = EmptyDigraph(order=20)
>>> e.showMIS()  # by visiting all 2**20 independent choices
*** Maximal independent choices ***
[ '1', '2', '3', '4', '5', '6', '7', '8', '9', '10',
  '11', '12', '13', '14', '15', '16', '17', '18', '19', '20']
number of solutions: 1
execution time: 1.47640 sec.  ###
```

Now, there exist more efficient specialized algorithms for directly enumerating MISs and dominant or absorbent kernels contained in specific digraph models without visiting all independent choices (see [BIS-2006b] (page 139)). Alain Hertz provided kindly such a MISs enumeration algorithm for the Digraph3 project (see digraphs.Digraph.showMIS_AH()). When the number of independent choices is big compared to the actual number of MISs, like in very sparse or empty digraphs, the performance difference may be dramatic (see Line 7 above and Line 15 below).

```python
>>> e.showMIS_AH()  # by visiting only maximal independent choices
*---------------------------------------------------*
* Python implementation of Hertz’s *               *
* algorithm for generating all MISs *              *
* R.B. version 7(6)-25-Apr-2006 *                 *
*---------------------------------------------------*
===>>> Initial solution :
[ '1', '2', '3', '4', '5', '6', '7', '8', '9', '10',
  '11', '12', '13', '14', '15', '16', '17', '18', '19', '20']
*---- results ----*
[ '1', '2', '3', '4', '5', '6', '7', '8', '9', '10',
  '11', '12', '13', '14', '15', '16', '17', '18', '19', '20']
*---- statistics ----*
mis solutions : 1
execution time : 0.00026 sec.  ###
iteration history: 1
```

For more or less dense strict outranking digraphs of modest order, as facing usually in algorithmic decision theory applications, enumerating all independent choices remains however in most cases tractable, especially by using a very efficient Python generator (see digraphs.Digraph.independentChoices() below):

```python
def independentChoices(self,U):
    ***
    Generator for all independent choices with associated dominated, absorbed and independent neighborhoods of digraph instance self.
    Initiate with U = self.singletons().
    Yields [(independent choice, domnb, absnb, indnb)].
    ***
    if U == []:
        yield [(frozenset(),set(),set(),set(self.actions))]
    else:
        x = list(U.pop())
        for S in self.independentChoices(U):
            yield S
            if x[0] <= S[0][3]:
                Sxgdom = S[0][1] | x[1]
                Sxgabs = S[0][2] | x[2]
                Sxindep = S[0][3] & x[3]
                Sxchoice = S[0][0] | x[0]
```
Sx = [(Sxchoice, Sxgdom, Sxgamabs, Sxindep)]

yield Sx

And, checking maximality of independent choices via the external stability conditions during their enumeration (see digraphs.Digraph.computePreKernels() below):

```python
def computePreKernels(self):
    """
    computing dominant and absorbent preKernels:
    Result in self.dompreKernels and self.abspreKernels
    """
    actions = set(self.actions)
    n = len(actions)
    dompreKernels = set()
    abspreKernels = set()
    for choice in self.independentChoices(self.singletons()):
        restactions = actions - choice[0][0]
        if restactions <= choice[0][1]:
            dompreKernels.add(choice[0][0])
        if restactions <= choice[0][2]:
            abspreKernels.add(choice[0][0])
    self.dompreKernels = dompreKernels
    self.abspreKernels = abspreKernels
```

provides the effective advantage of computing all initial and terminal prekernels in a single loop (see Line 10 and [BIS-2006b] (page 139)).

Back to Tutorials of the Digraph3 resources (page 3)

## 14 About split, interval and permutation graphs

- A multiply perfect graph (page 103)
- Who is the liar ? (page 105)
- Generating permutation graphs (page 107)
- Recognizing permutation graphs (page 110)

### 14.1 A multiply perfect graph

Following Martin Golumbic (see [GOL-2004] (page 139) p. 149), we call a given graph \( g \):

- **Comparability graph** when \( g \) is transitively orientable.
- **Triangulated graph** when \( g \) does not contain any chordless cycle of length 4 and more.
- **Interval graph** when \( g \) is triangulated and its dual \(-g\) is a comparability graph.
- **Permutation graph** \( g \) and its dual \(-g\) are both comparability graphs.
- **Split graph** when \( g \) and its dual \(-g\) are both triangulated graphs.

To illustrate these perfect graph classes, we will generate from 8 intervals, randomly chosen in the default integer range [0,10], a RandomIntervalIntersectionsGraph instance \( g \) (see Line 2 below). With seed = 100, we obtain an interval graph which is conjointly a triangulated, a comparability, a split and a permutation graph.

```python
>>> from graphs import RandomIntervalIntersectionsGraph
>>> g = RandomIntervalIntersectionsGraph(order=8,seed=100)
```
In Fig. 47 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, ie \(-g\) is a comparability graph;
3. \( g \) is a triangulated graph;
4. \( g \) is a cotriangulated graph, ie \(-g\) is a comparability graph;

are independent of one another (see [GOL-2004] (page 139) p. 275).

14.2 Who is the lier?

*Claude Berge*’s famous mystery story (see [GOL-2004] (page 139) p.20) may well illustrate the importance of being an interval graph.

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

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

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.
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] (page 139) Chapter 7, pp 157-170).

By using color sorting queues, the minimal vertex coloring for a permutation graph is computable in $O(n \log(n))$ (see [GOL-2004] (page 139)).
Fig. 50: The default permutation graph

```
...  WithVertexColoring=True)
*---- exporting a dot file for GraphViz tools ---------*
Exporting to coloredPermutationGraph.dot
fdp -Tpng coloredPermutationGraph.dot -o coloredPermutationGraph.png
```

Fig. 51: 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. 52).

```
>>> 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
```

As mentioned before, a permutation graph and its dual are transitive orientable. The graphs. PermutationGraph.transitiveOrientation() method constructs from a given permutation graph a digraph where each edge of the permutation graph is converted into an arc oriented in increasing alphabetic order of the adjacent vertices’ keys (see [GOL-2004] (page 139)). This orientation of the edges of a permutation graph is always transitive and delivers a weak ordering of the vertices.

```
>>> dg = g.transitiveOrientation()
>>> dg
*------- Digraph instance description ------*
Instance class : WeakOrder
Instance name : oriented_permutationGraph
Digraph Order : 6
Digraph Size : 9
Valuation domain : [-1.00; 1.00]
```
The dual of a permutation graph is again a permutation graph and as such also transitively orientable.
14.4 Recognizing permutation graphs

Now, a given graph \( g \) is a permutation graph if and only if both \( g \) and \(-g\) are transitively orientable. This property gives a polynomial test procedure (in \( O(n^3) \) due to the transitivity check) for recognizing permutation graphs.

Let us consider, for instance, the following random graph of order 8 generated with an edge probability of 40% and a random seed equal to 4335.

```python
>>> 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
Valuation domain : [-1.00; 1.00]
Attributes : ['name', 'order', 'vertices', 'valuationDomain', 'seed', 'edges', 'size', 'gamma', 'edgeProbability']

>>> g.exportGraphViz()

Fig. 54: Random graph of order 8 generated with edge probability 0.4

If the random graph instance \( g \) (see Fig. 54) is a permutation graph, \( g \) and its dual \(-g\) must be transitively orientable, i.e., comparability graphs (see [GOL-2004] page 139). With the graphs.Graph.isComparabilityGraph() test, we may easily check this fact. This method proceeds indeed by trying to construct a transitive neighbourhood decomposition of a given graph instance and, if successful, stores the resulting edge orientations into a self.edgeOrientations attribute (see [GOL-2004] page 139 p.129-132).

```
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.
hoods marked in black, red and blue (see Fig.56).

Let us recheck these facts by explicitly constructing transitively oriented digraph instances with the graphs.Graph.computeTransitivelyOrientedDigraph() method.

```python
>>> og = g.computeTransitivelyOrientedDigraph(PartiallyDetermined=True)
>>> print('Transitivity degree: %.3f' % (og.transitivityDegree))
Transitivity degree: 1.000
>>> ogd = (-g).computeTransitivelyOrientedDigraph(PartiallyDetermined=True)
>>> print('Transitivity degree: %.3f' % (ogd.transitivityDegree))
Transitivity degree: 1.000
```

The PartiallyDetermined=True flag (see Lines 1 and 5) is required here in order to orient only the actual edges of the graphs. Relations between vertices not linked by an edge will be put to the indeterminate characteristic value 0. This will allow us to compute, later on, convenient disjunctive digraph fusions.

As both graphs are indeed transitively orientable (see Lines 3 and 6 above), we may conclude that the given random graph \( g \) is actually a permutation graph instance. Yet, we still need to find now its corresponding permutation. We therefore implement a recipe given by Martin Golumbic [GOL-2004] (page 139).

We will first fuse both \( og \) and \( ogd \) orientations above with an epistemic disjunction (see the digraphsTools.omax() operator), hence, the partially determined orientations requested above.

```python
>>> from digraphs import FusionDigraph
>>> f1 = FusionDigraph(og,ogd,operator='o-max')
>>> s1 = f1.computeCopelandRanking()
>>> print(s1)
['v5', 'v7', 'v1', 'v6', 'v8', 'v4', 'v3', 'v2']
```

We obtain by the Copeland ranking rule (see Ranking with multiple incommensurable criteria (page 36) and the digraphs.Digraph.computeCopelandRanking() method) a linear ordering of the vertices (see 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).

```python
>>> f2 = FusionDigraph((-og),ogd,operator='o-max')
>>> s2 = f2.computeCopelandRanking()
>>> print(s2)
['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).

```python
>>> permutation = [0 for j in range(g.order)]
>>> for j in range(g.order):
...     permutation[s2.index(s1[j])] = j+1
>>> print(permutation)
[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] (page 139)). In our case here, we observe two different permutations and their reverses:

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

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

And:

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

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

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

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

```python
>>> 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')
```

And, we recover indeed two isomorphic copies of the original random graph (see Fig. 54).

Back to Tutorials of the Digraph3 resources (page 3)

### 15 On tree graphs and graph forests

- **Generating random tree graphs** (page 114)
- **Recognizing tree graphs** (page 116)
- **Spanning trees and forests** (page 118)
- **Maximum determined spanning forests** (page 119)
15.1 Generating random tree graphs

Using the `graphs.RandomTree` class, we may, for instance, generate a random tree graph with 9 vertices.

```python
>>> 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
```

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 -ie 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] (page 139)).

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] (page 139)).
Fig. 58: Random Tree instance of order 9
>>> verticesList = ['v1','v2','v3','v4','v5','v6','v7']
>>> n = len(verticesList)
>>> from random import seed, choice
>>> seed(101)
>>> code = []
>>> for k in range(n-2):
...     code.append(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. 59: 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?

### 15.2 Recognizing tree graphs

Given a graph $g$ of order $n$ and size $s$, the following 5 assertions $A1, A2, A3, A4$ and $A5$ are all equivalent (see [BAR-1991] (page 139)):

- $A1$: $g$ 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.

```python
>>> from graphs import RandomGraph
>>> g = RandomGraph(order=6, edgeProbability=0.3, seed=62)
>>> if g.order == (g.size + 1) and g.isConnected():
...     print('The graph is a tree ?', True)
... else:
...     print('The graph is a tree ?', False)
The graph is a tree ? True
```

The random graph of order 6 and edge probability 30%, generated with seed 62, is actually a tree graph instance, as we may readily confirm from its graphviz drawing in Fig. 60 (see also the `graphs.Graph.isTree()` method for an implemented alternative test).

```python
>>> g.exportGraphViz()
**** exporting a dot file for GraphViz tools ********
Exporting to test62.dot
fdp -Tpng test62.dot -o test62.png
```

Yet, we still have to recover its corresponding Prüfer code. Therefore, we may use the `graphs.RandomTree.tree2Pruefer()` method.

```python
>>> from graphs import RandomGraph
>>> RandomTree.tree2Pruefer(g)
['v6', 'v1', 'v2', 'v1', 'v2', 'v5']
```

Let us now turn toward a major application of tree graphs, namely spanning trees and forests related to graph traversals.
15.3 Spanning trees and forests

With the `graphs.RandomSpanningTree` class we may generate, from a given connected graph `g` instance, uniform random instances of a spanning tree by using Wilson's algorithm [WIL-1996] (page 139).

Note: Wilson's algorithm only works for connected graphs\(^4\).

More general, and in case of a not connected graph, we may generate with the `graphs.RandomSpanningForest` class a not necessarily uniform random instance of a spanning forest -one or more random tree graphs- generated from a random depth first search of the graph components' traversals.

15.4 Maximum determined spanning forests

In case of valued graphs supporting weighted edges, we may finally construct a most determined spanning tree (or forest if not connected) using Kruskal’s greedy minimum-spanning-tree algorithm on the dual valuation of the graph [KRU-1956] (page 139).

We consider, for instance, a randomly valued graph with five vertices and seven edges bipolar-valued in [-1.0; 1.0].

To inspect the edges’ actual weights, we first transform the graph into a corresponding digraph (see Line 1 below) and use the digraphs.Digraph.showRelationTable() method (see Line 2 below) for printing its symmetric adjacency matrix.

---

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.
To compute the most determined spanning tree or forest, we may use the `graphs.BestDeterminedSpanningForest` class constructor.

```python
>>> mt = BestDeterminedSpanningForest(g)
```

The given graph is connected and, hence, admits a single spanning tree (see Fig. 63) of maximum mean determination \(= (0.47 + 0.91 + 0.90 + 0.34)/4 = 0.655\) (see Lines 9, 6 and 10 in the relation table above).

```python
>>> mt.exportGraphViz(fileName='bestDeterminedspanningTree',
                      WithSpanningTree=True)
```

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.
16 Pearls of bipolar-valued epistemic logic

- Coping with missing data and indeterminateness (page 121)
- Ordinal correlation equals bipolar-valued relational equivalence (page 126)
- Bipolar-valued kernel membership characteristic vectors (page 132)

16.1 Coping with missing data and indeterminateness

In a stubborn keeping with a two-valued logic, where every argument can only be true or false, there is no place for efficiently taking into account missing data or logical indeterminateness. These cases are seen as problematic and, at best are simply ignored. Worst, in modern data science, missing data get often replaced with fictive values, potentially falsifying hence all subsequent computations.

In social choice problems like elections, abstentions are, however, frequently observed and represent a social expression that may be significant for revealing non represented social preferences.

In marketing studies, interviewees will not always respond to all the submitted questions. Again, such abstentions do sometimes contain nevertheless valid information concerning consumer preferences.

Let us take an example performance tableau from a Movie magazine’s evaluation of movies that could be seen in town⁹ (see Fig. 64).

```python
>>> from outrankingDigraphs import *
>>> t = XMCDA2PerformanceTableau('graffiti07')
>>> t.showHTMLPerformanceTableau(ndigits=0)
```

15 journalists and movie critics provide here their rating of 25 movies: 5 stars (masterpiece), 4 stars (must be seen), 3 stars (excellent), 2 stars (good), 1 star (could be seen), -1 star (I do not like), -2 (I hate), NA (not seen).

To aggregate all the critics’ rating opinions, the Graffiti magazine provides for each movie a global score computed as an average grade, just ignoring the not seen data. These averages are thus not computed on comparable denominators; some critics do indeed use a more or less extended range of grades. The movies not seen by critic SJ, for instance, are favored, as this critic is more severe than others in her grading. Dropping the movies that were not seen by all the critics is here not possible either, as no one of the 25 movies was actually seen by all the critics. Providing any value for the missing data will as well always somehow falsify any global value scoring.

What to do ?

A better approach is to rank the movies on the basis of pairwise bipolar-valued at least as well rated as opinions. Under this epistemic argumentation approach, missing data are naturally treated as opinion abstentions and hence do not falsify the logical computations. Such a ranking (see the Ranking with multiple incommensurable criteria (page 36) tutorial) of the 25 movies is provided, for instance, by the heatmap view shown in Fig. 65.

```python
>>> t.showHTMLPerformanceHeatmap(Correlations=True,
                                        rankingRule='NetFlows',
                                        ndigits=0)
```

There is no doubt that movie mv_QS, with 6 ‘must be seen’ marks, is correctly best-ranked and the movie mv_TV is worst-ranked with five ‘don’t like’ marks.

Let us explicitly construct the corresponding bipolar-valued outranking digraph and consult in Fig. 66 the pairwise characteristic values we observe between the two best-ranked movies, namely mv_QS and mv_RR.

<table>
<thead>
<tr>
<th>criteria</th>
<th>AP</th>
<th>AS</th>
<th>CF</th>
<th>CS</th>
<th>DR</th>
<th>FG</th>
<th>GS</th>
<th>JH</th>
<th>JPT</th>
<th>MR</th>
<th>RR</th>
<th>SF</th>
<th>SJ</th>
<th>TD</th>
<th>VT</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>3.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mv_AL</td>
<td>3</td>
<td>-1</td>
<td>2</td>
<td>-1</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>2</td>
</tr>
<tr>
<td>mv_BI</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>mv_CM</td>
<td>NA</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>NA</td>
</tr>
<tr>
<td>mv_DF</td>
<td>NA</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>-1</td>
<td>NA</td>
</tr>
<tr>
<td>mv_DG</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>-1</td>
<td>NA</td>
</tr>
<tr>
<td>mv_DI</td>
<td>1</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>1</td>
<td>NA</td>
</tr>
<tr>
<td>mv_DF</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
<td>NA</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>mv_FC</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>mv_FF</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mv_GG</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>-1</td>
<td>NA</td>
<td>-1</td>
<td>2</td>
<td>NA</td>
<td>-1</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td>mv_GH</td>
<td>1</td>
<td>NA</td>
<td>1</td>
<td>-1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>mv_HN</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>NA</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>NA</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>mv_HP</td>
<td>1</td>
<td>NA</td>
<td>3</td>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>NA</td>
<td>1</td>
<td>2</td>
<td>NA</td>
</tr>
<tr>
<td>mv_HS</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>NA</td>
<td>1</td>
<td>3</td>
<td>NA</td>
</tr>
<tr>
<td>mv_JB</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>3</td>
</tr>
<tr>
<td>mv_MB</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>NA</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>mv_NPC</td>
<td>NA</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
</tr>
<tr>
<td>mv_PE</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>1</td>
<td>NA</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>3</td>
<td>NA</td>
<td>3</td>
</tr>
<tr>
<td>mv_QS</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>4</td>
<td>3</td>
<td>NA</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>NA</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>mv_RG</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
<td>NA</td>
</tr>
<tr>
<td>mv_RR</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>4</td>
<td>NA</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>NA</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>mv_SM</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>NA</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>NA</td>
</tr>
<tr>
<td>mv_TF</td>
<td>-1</td>
<td>NA</td>
<td>1</td>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>2</td>
<td>NA</td>
<td>-1</td>
<td>NA</td>
<td>-1</td>
<td>NA</td>
<td>1</td>
<td>NA</td>
</tr>
<tr>
<td>mv_TM</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>4</td>
<td>NA</td>
</tr>
<tr>
<td>mv_TP</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>NA</td>
<td>2</td>
</tr>
</tbody>
</table>

Fig. 64: Graffiti magazine’s movie ratings from September 2007
### Heatmap of Performance Tableau 'grafittiPerfTab.xml'

<table>
<thead>
<tr>
<th>criteria</th>
<th>JH</th>
<th>JPT</th>
<th>AP</th>
<th>DR</th>
<th>MR</th>
<th>VT</th>
<th>GS</th>
<th>CS</th>
<th>SJ</th>
<th>RR</th>
<th>TD</th>
<th>CF</th>
<th>SF</th>
<th>AS</th>
<th>FG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>weights</strong></td>
<td>+3.00</td>
<td>+1.00</td>
<td>+1.00</td>
<td>+1.00</td>
<td>+3.00</td>
<td>+1.00</td>
<td>+1.00</td>
<td>+1.00</td>
<td>+1.00</td>
<td>+1.00</td>
<td>+1.00</td>
<td>+1.00</td>
<td>+1.00</td>
<td>+1.00</td>
<td></td>
</tr>
<tr>
<td>tau(*)</td>
<td>+0.50</td>
<td>+0.43</td>
<td>+0.32</td>
<td>+0.26</td>
<td>+0.25</td>
<td>+0.23</td>
<td>+0.16</td>
<td>+0.14</td>
<td>+0.14</td>
<td>+0.13</td>
<td>+0.11</td>
<td>+0.11</td>
<td>+0.10</td>
<td>+0.08</td>
<td>+0.03</td>
</tr>
<tr>
<td>mv_QS</td>
<td>4</td>
<td>4</td>
<td>NA</td>
<td>3</td>
<td>4</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>mv_RR</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>NA</td>
<td>3</td>
<td>4</td>
<td>NA</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>mv_DG</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>NA</td>
<td>3</td>
<td>-1</td>
<td>NA</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>2</td>
</tr>
<tr>
<td>mv_NP</td>
<td>3</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>mv_HN</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>NA</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>NA</td>
<td>1</td>
<td>NA</td>
</tr>
<tr>
<td>mv_HS</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>mv_SM</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>NA</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>mv_JB</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>mv_PE</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
<td>NA</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>mv_FC</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>mv_TP</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>NA</td>
<td>3</td>
<td>NA</td>
</tr>
<tr>
<td>mv_CM</td>
<td>2</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>NA</td>
</tr>
<tr>
<td>mv_DF</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>mv_TM</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>NA</td>
</tr>
<tr>
<td>mv_DJ</td>
<td>NA</td>
<td>2</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>mv_AL</td>
<td>NA</td>
<td>2</td>
<td>3</td>
<td>NA</td>
<td>2</td>
<td>3</td>
<td>-1</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>-1</td>
<td>NA</td>
</tr>
<tr>
<td>mv_RG</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
</tr>
<tr>
<td>mv_MB</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>1</td>
<td>NA</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>mv_GH</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>NA</td>
<td>1</td>
<td>-1</td>
<td>NA</td>
<td>4</td>
<td>NA</td>
<td>3</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
</tr>
<tr>
<td>mv_HP</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>NA</td>
<td>3</td>
<td>NA</td>
</tr>
<tr>
<td>mv_BI</td>
<td>NA</td>
<td>2</td>
<td>1</td>
<td>-1</td>
<td>NA</td>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>-1</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
</tr>
<tr>
<td>mv_DI</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>mv_FF</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>NA</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mv_GG</td>
<td>-1</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>NA</td>
<td>3</td>
<td>NA</td>
</tr>
<tr>
<td>mv_TF</td>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>

**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: **0.780**

Fig. 65: *Graffiti* magazine’s ordered movie ratings from September 2007
```python
>>> g = BipolarOutrankingDigraph(t)
>>> g.recodeValuation(-19,19)  # integer characteristic values
>>> g.showHTMLPairwiseOutrankings('mv_QS','mv_RR')
```

Fig. 66: Pairwise comparison of the two best-ranked movies

Six out of the fifteen critics have not seen one or the other of these two movies. Notice the higher significance (3) that is granted to two locally renowned movie critics, namely JH and VT. Their opinion counts for three times the opinion of the other critics. All nine critics that have seen both movies, except critic MR, state that mv_QS is rated at least as well as mv_RR and the balance of positive against negative opinions amounts to +11, a characteristic value which positively validates the outranking situation with a majority of \((11/19 + 1.0) / 2.0 = 79\%\).

The complete table of pairwise majority margins of global ‘at least as well rated as’ opinions, ranked by the same rule as shown in the heat map above (see Fig. 65), may be shown as follows.

```python
>>> ranking = g.computeNetFlowsRanking()
>>> g.showHTMLRelationTable(actionsList=ranking, ndigits=0,
... tableTitle='Bipolar characteristic values of
... "rated at least as good as" situations')
```

Positive characteristic values, validating a global ‘at least as well rated as’ opinion are marked in light green (see Fig. 67). Whereas negative characteristic values, invalidating such a global opinion, are marked in light red. We may by the way notice that the best-ranked movie mv_QS is indeed a Condorcet winner, i.e. better rated than all the other movies by a 65\% majority of critics. This majority may be assessed from the average determinateness of the given bipolar-valued outranking digraph g.

```python
>>> print( '%.0f%%' % g.computeDeterminateness(InPercents=True) )
65%
```

Notice also the indeterminate situation we observe, for instance, when comparing movie mv_PE with movie mv_NP.

```python
>>> g.showHTMLPairwiseComparison('mv_PE','mv_NP')
```

Only eight, out of the fifteen critics, have seen both movies and the positive opinions do neatly balance the negative ones. A global statement that mv_PE is ‘at least as well rated as’ mv_NP may in this case hence neither be validated, nor invalidated; a preferential situation that cannot be modelled with any scoring approach.

It is fair, however, to eventually mention here that the Graffiti magazine’s average scoring method is actually showing a very similar ranking. Indeed, average scores usually confirm well all evident pairwise comparisons, yet enforce comparability for all less evident ones.
### Bipolar characteristic values of “rated at least as good as” situations

<table>
<thead>
<tr>
<th>EGS</th>
<th>mv_QS</th>
<th>mv_QM</th>
<th>mv_RR</th>
<th>mv_DG</th>
<th>mv_NP</th>
<th>mv_HN</th>
<th>mv_HK</th>
<th>mv_MG</th>
<th>mv_MH</th>
<th>mv_TA</th>
<th>mv_TM</th>
<th>mv_DG</th>
<th>mv_PQ</th>
<th>mv_TP</th>
<th>mv_CM</th>
<th>mv_IH</th>
<th>mv_JH</th>
<th>mv_BE</th>
<th>mv_MR</th>
<th>mv_GR</th>
<th>mv_GH</th>
<th>mv_IP</th>
<th>mv_BB</th>
<th>mv_BB</th>
<th>mv_FF</th>
<th>mv_FF</th>
<th>mv_FF</th>
<th>mv_FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>14</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>9</td>
<td>11</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>8</td>
<td>11</td>
<td>3</td>
<td>9</td>
<td>8</td>
<td>13</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>6</td>
<td>10</td>
<td>10</td>
<td>6</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>6</td>
<td>10</td>
<td>10</td>
<td>6</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>14</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>9</td>
<td>11</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>11</td>
<td>7</td>
<td>11</td>
<td>12</td>
<td>7</td>
<td>11</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>9</td>
<td>8</td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

Valuation domain: [-19.00; +19.00]

**Fig. 67**: Pairwise majority margins of ‘at least as well rated as’ rating opinions

### Pairwise Comparison

**Comparing actions**: (mv_PE,mv_NP)

<table>
<thead>
<tr>
<th>crit.</th>
<th>wght.</th>
<th>g(x)</th>
<th>g(y)</th>
<th>diff</th>
<th>ind</th>
<th>wp</th>
<th>p</th>
<th>concord</th>
<th>wv</th>
<th>polarisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>1.00</td>
<td>3.00</td>
<td>NA</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AS</td>
<td>1.00</td>
<td>4.00</td>
<td>1.00</td>
<td>+3.00</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>+1.00</td>
</tr>
<tr>
<td>CF</td>
<td>1.00</td>
<td>2.00</td>
<td>3.00</td>
<td>-1.00</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>1.00</td>
<td>NA</td>
<td>1.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DR</td>
<td>1.00</td>
<td>3.00</td>
<td>NA</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PG</td>
<td>1.00</td>
<td>1.00</td>
<td>3.00</td>
<td>-2.00</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td>1.00</td>
<td>NA</td>
<td>1.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JH</td>
<td>3.00</td>
<td>2.00</td>
<td>3.00</td>
<td>-1.00</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>-3.00</td>
</tr>
<tr>
<td>JPT</td>
<td>1.00</td>
<td>4.00</td>
<td>3.00</td>
<td>+1.00</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>+1.00</td>
</tr>
<tr>
<td>MR</td>
<td>1.00</td>
<td>2.00</td>
<td>3.00</td>
<td>-1.00</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>1.00</td>
<td>NA</td>
<td>2.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SF</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
<td>+0.00</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>+1.00</td>
</tr>
<tr>
<td>SJ</td>
<td>1.00</td>
<td>3.00</td>
<td>NA</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TD</td>
<td>1.00</td>
<td>NA</td>
<td>3.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VT</td>
<td>3.00</td>
<td>3.00</td>
<td>3.00</td>
<td>+0.00</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>+3.00</td>
</tr>
</tbody>
</table>

Valuation in range: -19.00 to +19.00; global concordance: +0.00

**Fig. 68**: Indeterminate pairwise comparison example
Notice finally the ordinal correlation figures \( \tau \) in Fig. 65 3rd row. How may we compute these ordinal correlation indexes?

### 16.2 Ordinal correlation equals bipolar-valued relational equivalence

#### Kendall’s \( \tau \) index

M. G. Kendall ([KEN-1938](#)) defined his ordinal correlation index \( \tau \) for linear orders of dimension \( n \) as a balancing of the number \( \#C \) of correctly oriented pairs against the number \( \#I \) of incorrectly oriented pairs. The total number of irreflexive pairs being \( n(n-1) \), in the case of linear orders, \( \#C + \#I = n(n-1) \). Hence \( \tau = (\#C / n(n-1)) - (\#I / n(n-1)) \). In case \( \#I \) is zero, \( \tau = +1 \) (all pairs are equivalently oriented); inversely, in case \( \#C \) is zero, \( \tau = -1 \) (all pairs are differently oriented).

Noticing that \( (\#C / n(n-1)) = 1 - (\#I / n(n-1)) \), and recalling that the bipolar-valued negation is operated by changing the sign of the characteristic value, Kendall’s original \( \tau \) definition implemented in fact the bipolar-valued negation of the non equivalence of two linear orders:

\[
\tau = 1 - 2(\#I / n(n-1)) = -[2(\#I / n(n-1)) - 1] = 2(\#C / n(n-1)) - 1,
\]

i.e. the normalized majority margin of equivalently oriented irreflexive pairs.

Let \( R_1 \) and \( R_2 \) be two random crisp relations defined on a same set of 5 alternatives. We may compute Kendall’s \( \tau \) index as follows.

```python
>>> from digraphs import *

>>> R1 = RandomDigraph(order=5,Bipolar=True)

>>> R2 = RandomDigraph(order=5,Bipolar=True)

>>> E = EquivalenceDigraph(R1,R2)

>>> E.showRelationTable(ReflexiveTerms=False)
```

| r(<=>)| 'a1'  'a2'  'a3'  'a4'  'a5'
|-------|-------------------------------------------
| 'a1'  | - -1.00 1.00 -1.00 1.00 |
| 'a2'  | -1.00 - -1.00 1.00 -1.00 |
| 'a3'  | -1.00 -1.00 - 1.00 1.00 |
| 'a4'  | -1.00 1.00 -1.00 - | 1.00 |
| 'a5'  | -1.00 1.00 -1.00 1.00 - |

Valuation domain: \([-1.00;1.00]\]

```python

>>> E.correlation

{'correlation': -0.1, 'determination': 1.0}
```

In the table of the equivalence relation \( R_1<=>R_2 \) above, we observe that the normalized majority margin of equivalent versus non equivalent irreflexive pairs amounts to \((9 - 11)/20 = -0.1\), i.e. the value of Kendall’s \( \tau \) index in this plainly determined crisp case.

What happens now with more or less determined and even partially indeterminate relations? May we proceed in a similar way?

#### Bipolar-valued relational equivalence

Let us now consider two randomly bipolar-valued digraphs \( R_1 \) and \( R_2 \) of order five.

```python
>>> R1 = RandomValuationDigraph(order=5,seed=1)

>>> R1.showRelationTable(ReflexiveTerms=False)
```

| r(R1)| 'a1'  'a2'  'a3'  'a4'  'a5'
|-------|-------------------------------------------
| 'a1'  | - -0.66 0.44 0.94 -0.84 |
| 'a2'  | -0.36 - -0.70 0.26 0.94 |
| 'a3'  | 0.14 0.20 - 0.66 -0.04 |
| 'a4'  | -0.48 -0.76 0.24 - | -0.94 |
```
Valuation domain: \([-1.00;1.00]\)

```python
>>> R2 = RandomValuationDigraph(order=5, seed=2)
>>> R2.showRelationTable(ReflexiveTerms=False)

* ---- Relation Table ----*

<table>
<thead>
<tr>
<th>r(R2)</th>
<th>'a1'</th>
<th>'a2'</th>
<th>'a3'</th>
<th>'a4'</th>
<th>'a5'</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>-------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'a1'</td>
<td>- -0.86 -0.78 -0.80 -0.08</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'a2'</td>
<td>-0.58 - -0.88 0.70 -0.22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'a3'</td>
<td>-0.36 0.54 - -0.46 0.54</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'a4'</td>
<td>-0.92 0.48 0.74 - -0.60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'a5'</td>
<td>0.10 0.62 0.00 0.84 -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Valuation domain: \([-1.00;1.00]\)
```

We may notice in the relation tables shown above that 9 pairs, like \((a1,a2)\) or \((a3,a2)\) for instance, appear equivalently oriented. The `digraphs.EquivalenceDigraph` class implements this relational equivalence relation between digraphs \(R1\) and \(R2\).

```python
>>> eq = EquivalenceDigraph(R1, R2)
>>> eq.showRelationTable(ReflexiveTerms=False)

* ---- Relation Table ----*

<table>
<thead>
<tr>
<th>r(&lt;=&gt;)</th>
<th>'a1'</th>
<th>'a2'</th>
<th>'a3'</th>
<th>'a4'</th>
<th>'a5'</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>-------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'a1'</td>
<td>- 0.66 -0.44 -0.80 0.08</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'a2'</td>
<td>0.36 - -0.70 0.26 -0.22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'a3'</td>
<td>-0.14 0.20 - -0.46 -0.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'a4'</td>
<td>0.48 -0.48 0.24 - 0.60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'a5'</td>
<td>-0.02 0.10 0.00 0.84 -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Valuation domain: \([-1.00;1.00]\)
```

In our bipolar-valued epistemic logic, logical disjunctions and conjunctions are implemented as \(\text{max}\), respectively \(\text{min}\) operators. Notice also that the logical equivalence \((R1<=>R2)\) corresponds to a double implication \((R1 => R2)\) and \((R2 => R1)\) and that the implication \((R1 => R2)\) is logically equivalent to the disjunction \((\text{not } R1 \text{ or } R2)\).

If \(r(x R1 y)\) and \(r(x R2 y)\) denote the bipolar-valued characteristic values of relation \(R1\), resp. \(R2\), we may hence compute as follows a majority margin \(M(R1<=>R2)\) between equivalently and not equivalently oriented irreflexive pairs \((x,y)\).

\[
M(R1<=>R2)* = \sum_{(x,y)} \{ \min [\max(-r(x R1 y), r(x R2 y)), \max(-r(x R2 y), r(x R1 y))] \}.
\]

\(M(R1<=>R2)\) is thus given by the sum of the non reflexive terms of the relation table of \(eq\), the relation equivalence digraph computed above.

In the crisp case, \(M(R1<=>R2)\) is now normalized with the maximum number of possible irreflexive pairs, namely \(n(n-1)\). In a generalized \(r\)-valued case, the maximal possible equivalence majority margin \(M\) corresponds to the sum \(D\) of the conjoint determinations of \((x R1 y)\) and \((x R2 y)\) (see [BIS-2012] (page 138)).

\[
D = \sum_{(x,y)} \min [\abs(r(x R1 y)), \abs(r(x R2 y))].
\]

Thus, we obtain in the general \(r\)-valued case:

\[
\tau(R1,R2) = \frac{M(R1<=>R2)}{D}.
\]

\(\tau(R1,R2)\) corresponds thus to a classical ordinal correlation index, but restricted to the conjointly determined parts of the given relations \(R1\) and \(R2\). In the limit case of two crisp linear orders, \(D\) equals \(n(n-1)\), i.e. the number of irreflexive pairs, and we recover hence Kendall’s original \(\tau\) index definition.

It is worthwhile noticing that the ordinal correlation index \(\tau(R1,R2)\) we obtain above corresponds to the ratio of
• $r(R_1<=>R_2) = M(R_1<=>R_2) / n(n-1)$: The normalized majority margin of the pairwise relational equivalence statements, also called valued ordinal correlation, and

• $d = D / n(n-1)$: The normalized determination of the corresponding pairwise relational equivalence statements, in fact de determinateness of the relational equivalence digraph.

We have thus successfully out-factored the determination effect from the correlation effect. With completely determined relations, $\tau(R_1,R_2) = r(R_1<=>R_2)$. By convention, we set the ordinal correlation with a completely indeterminate relation, i.e. when $D = 0$, to the indeterminate correlation value 0.0. With uniformly chosen random r-valued relations, the expected $\tau$ index is 0.0, denoting in fact an indeterminate correlation. The corresponding expected normalized determination $d$ is about 0.333 (see [BIS-2012] (page 138)).

We may verify these relations with help of the corresponding equivalence digraph $eq$ (see above).

```
>>> eq = EquivalenceDigraph(R1,R2)
>>> M = Decimal('0'); D = Decimal('0')
>>> n2 = eq.order*(eq.order - 1)
>>> for x in eq.actions:
...   for y in eq.actions:
...     M += eq.relation[x][y]
...     D += abs(eq.relation[x][y])
>>> print('r(R1<=>R2) = %.3f, d = %.3f, tau = %.3f' % (M/n2,D/n2,M/D))
```

```
1>> r(R1<=>R2) = +0.026, d = 0.356, tau = +0.073
```

In general we simply use the `digraphs.Digraph.computeOrdinalCorrelation()` method which renders a dictionary with a ‘correlation’ ($\tau$) and a ‘determination’ ($d$) attribute. We may recover $r(<=>)$ by multiplying $\tau$ with $d$.

```
>>> corr = R1.computeOrdinalCorrelation(R2)
>>> tau = corr['correlation']
>>> d = corr['determination']
>>> r = tau * d
>>> print('\tau(R1,R2) = %.3f, d = %.3f, r(R1<=>R2) = %.3f' % (tau, d, r))
```

```
1>> \tau(R1,R2) = +0.073, d = 0.356, r(R1<=>R2) = +0.026
```

We may now illustrate the quality of the global ranking of the movies shown with the heat map in Fig. 65.

**Fitness of ranking heuristics**

We reconsider the bipolar-valued outranking digraph $g$ modelling the pairwise global 'at least as well rated as' relation among the 25 movies seen above.

```
>>> g = BipolarOutrankingDigraph(t,Normalized=True)
```

```
1+------- Object instance description -------+
2Instance class : BipolarOutrankingDigraph
3Instance name : rel_grafittiPerfTab.xml
4# Actions : 25
5# Criteria : 15
6Size : 390
7Determinateness : 65%
8Valuation domain : {'min': Decimal('-1.0'),
9  'med': Decimal('0.0'),
10  'max': Decimal('1.0')},
11```

 Out of the 25 x 24 = 600 irreflexive movie pairs, digraph $g$ contains 390 positively validated, 188 positively invalidated outranking situations, and 22 indeterminate outranking situations (see the zero-valued cells in Fig. 67).

Let us now compute the normalized majority margin $r(<=>)$ of the equivalence between the marginal critic’s pairwise ratings and the global Net-Flows ranking shown in the ordered heat map (se Fig.65).
We recover above the relational equivalence characteristic values shown in the third row of the table in Fig. 65. The global Net-Flows ranking represents obviously a rather balanced compromise with respect to all movie critics’ opinions as there appears no valued negative correlation with anyone of them. The Net-Flows ranking apparently takes also correctly in account that the journalist JH, a locally renowned movie critic, shows a higher significance weight.

The ordinal correlation between the global Net-Flows ranking and the digraph $g$ may be furthermore computed as follows:

We notice that the correlation $\tau$ index between the Net-Flows ranking and the determined part of the outranking digraph is quite high (+0.78). Due to the rather high number of missing data, the $r$-valued relational equivalence between the $nf$ and the $g$ digraph, with a characteristics value of only +0.234, may be misleading. Yet, +0.234 still corresponds to an epistemic majority support of nearly 62% of the movie critics’ rating opinions.

It would be interesting to compare similarly the correlations one may obtain with other global ranking heuristics, like the Copeland or the Kohler ranking rule.

Illustrating preference divergences

The valued relational equivalence index gives us a further measure for studying how divergent appear the rating opinions expressed by the movie critics.
It is remarkable that, due to the quite numerous missing data, all pairwise valued ordinal correlation indexes $r(x<=y)$ appear to be of low value, except the diagonal ones. These reflexive indexes $s(x<=x)$ would trivially all amount to +1.0 in a plainly determined case. Here they indicate a reflexive normalized determination score $d$, i.e. the proportion of pairs of movies each critic did evaluate. Critic JPT (the editor of the Graffiti magazine), for instance, evaluated all but one ($d = 24*23/600 = 0.92$), whereas critic FG evaluated only 10 movies among the 25 in discussion ($d = 10*9/600 = 0.15$).

To get a picture of the actual divergence of rating opinions concerning jointly seen pairs of movies, we may develop a Principal Component Analysis (10) of the corresponding tau correlation matrix. The 3D plot of the first 3 principal axes is shown below.

The first 3 principal axes support together about 70% of the total inertia. Most excentric and opposed in their

---

10 The 3D PCA plot method requires a running R statistics software (https://www.r-project.org/) installation and the Calmat matrix calculator (see the calmat directory in the Digraph3 resources)
respective rating opinions appear, on the first principal axis with 27.2% inertia, the conservative daily press against labour and public press. On the second principal axis with 23.7% inertia, it is the people press versus the cultural critical press. And, on the third axis with still 19.3% inertia, the written media appear most opposed to the radio media.

**Exploring the better rated and the as well as rated opinions**

In order to furthermore study the quality of a ranking result, it may be interesting to have a separate view on the asymmetric and symmetric parts of the ‘at least as well rated as’ opinions (see the tutorial Working with the Digraph3 software resources (page 4)).

Let us first have a look at the pairwise asymmetric part, namely the ‘better rated than’ and ‘less well rated than’ opinions of the movie critics.

```python
>>> ag = AsymmetricPartialDigraph(g)
```
```python
>>> ag.showHTMLRelationTable(actionsList=g.computeNetFlowsRanking(),ndigits=0)
```

**Valued Adjacency Matrix**

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0.5</td>
</tr>
</tbody>
</table>
```

We notice here that the Net-Flows ranking rule inverts in fact just three ‘less well ranked than’ opinions and four ‘better ranked than’ ones. A similar look at the symmetric part, the pairwise ‘as well rated as’ opinions, suggests a preordered preference structure in several equivalently rated classes.

```python
>>> sg = SymmetricPartialDigraph(g)
```
```python
>>> sg.showHTMLRelationTable(actionsList=g.computeNetFlowsRanking(),ndigits=0)
```
Such a preordering of the movies may, for instance, be computed with the `digraphs.Digraph.computeRankingByChoosing()` method, where we iteratively extract dominant kernels -best remaining choices- and absorbent kernels -worst remaining choices- (see the tutorial "On computing digraph kernels" (page 88)). We operate herefore on the asymmetric ‘better rated than’, i.e. the codual\(^{11}\) of the ‘at least as well rated as’ opinions.

```python
>>> from weakOrders import RankingByChoosingDigraph
>>> rbc = RankingByChoosingDigraph(g, CoDual=True)
>>> rbc.showRankingByChoosing()
Ranking by Choosing and Rejecting
1st Best Choice ['mv_QS']
2nd Best Choice ['mv_DG', 'mv_FC', 'mv_HN', 'mv_HS', 'mv_NP', 'mv_PE', 'mv_RR', 'mv_SM']
3rd Best Choice ['mv_CM', 'mv_JB', 'mv_TM']
4th Best Choice ['mv_AL', 'mv_TP']
4th Worst Choice ['mv_AL', 'mv_TP']
3rd Worst Choice ['mv_GH', 'mv_MB', 'mv_RG']
2nd Worst Choice ['mv_DF', 'mv_DJ', 'mv_FF', 'mv_GG']
1st Worst Choice ['mv_BI', 'mv_DM', 'mv_HP', 'mv_TF']
```

In the tutorial "On computing digraph kernels" (page 88), we thoroughly discuss the computation of kernels in bipolar-valued digraphs. Yet, we do not tackle there the problem of computing the corresponding bipolar-valued kernel membership characteristic vectors. This is the topic for the next pearl of bipolar-valued epistemic logic.

### 16.3 Bipolar-valued kernel membership characteristic vectors

**Claude Berge’s Kernel Equation Systems**

Let \(G(X,R)\) be a crisp irreflexive digraph defined on a finite set \(X\) of nodes and where \(R\) is the corresponding \([-1,+1]\)-valued adjacency matrix. Let \(Y\) be the \([-1,+1]\)-valued membership characteristic (row) vector of a choice in \(X\). When \(Y\) satisfies the following equation system

\[
Y \circ R = -Y,
\]

where for all \(x\) in \(X\),

\[
(Y \circ R)(x) = \max_{y \in X, x \neq y} \left( \min(Y(x), R(x, y)) \right)
\]

than \(Y\) characterises an initial kernel ([BER-1958] (page 139)). Transposing the membership characteristic vector into a column vector \(Y^t\), makes Berge’s equation system similarly characterise a terminal kernel.

\[11\] A kernel in a digraph \(g\) is a clique in the dual digraph \(-g\).
\[ R \circ Y^t = -Y^t. \]

Let us verify this result on a tiny example.

```python
>>> from digraphs import *

>>> g = RandomDigraph(order=3, seed=1)

* ---- Relation Table ----- 
  R | 'a1' 'a2' 'a3'
  ----|---------------------
  'a1' | -1 +1 -1
  'a2' | -1 -1 +1
  'a3' | +1 +1 -1

>>> g.showPreKernels()

*--- Computing preKernels ---*

Dominant preKernels :
['a3']
  independence : 1.0
  dominance : 1.0
  absorbency : -1.0
  covering : 1.000

Absorbent preKernels :
['a2']
  independence : 1.0
  dominance : -1.0
  absorbency : 1.0
  covered : 1.000
```

It is easy to verify that the characteristic vector \([-1, -1, +1]\] satisfies the initial kernel equation system; \(a3\) gives an initial kernel. Similarly, the characteristic vector \([-1, +1, -1]\] verifies indeed the terminal kernel equation system and hence \(a2\) gives a terminal kernel.

We succeeded now in generalizing Berge’s kernel equation systems to genuine bipolar-valued digraphs ([BIS-2006a] (page 139)). The constructive proof, found by M. Pirlot, is based on the following fixpoint equation that may be used for computing bipolar-valued kernel membership vectors,

\[
T(Y) := -(Y \circ R) = Y ,
\]

Solving bipolar-valued kernel equation systems

John von Neumann ([SCH-1985] (page 139)) showed indeed that, when a digraph \(G(X,R)\) is acyclic with a unique initial kernel \(K\) characterised by its membership characteristics vector \(Yk\), then the following dual bipolar-valued fixpoint equation

\[
T^2(Y) := -(Y \circ R) \circ R = Y.
\]

will admit a stable high and a stable low fixpoint solution that converge both to \(Yk\).

Inspired by this crisp dual fixpoint equation, we observed that for a given bipolar-valued digraph \(G(X,R)\), each of its dominant or absorbent prekernels \(Ki\) in \(X\) determines an induced partial graph \(G(X,R/Ki)\) which is acyclic and admits \(Ki\) as unique kernel (see [BIS-2006b] (page 139)).

Following the von Neumann fixpoint algorithm, a similar bipolar-valued extended dual fixpoint algorithm, applied to \(G(X,R/Ki)\), allows to compute hence the associated bipolar-valued kernel characteristic vectors \(Yi\) in polynomial complexity.

Algorithm

\[ \text{in} : \text{bipolar-valued digraph } G(X,R), \]
\[ \text{out} : \text{set } \{ Y1, Y2, \ldots \} \text{ of bipolar-valued kernel membership characteristic vectors.} \]

1. enumerate all initial and terminal crisp prekernels \(K, K2, \ldots\) in the given bipolar-valued digraph (see the ‘On computing digraph kernels’ (page 88)’ tutorial);
2. for each crisp initial kernel \(Ki\):
(a) construct a partially determined subgraph $G(X,R/K_i)$ supporting exactly this unique initial kernel $K_i$;

(b) Use the dual fixpoint equation $T_2$ with the partially determined adjacency matrix $R/K_i$ for computing a stable low and a stable high fixpoint;

(c) Determine the bipolar-valued $K_i$-membership characteristic vector $Y_i$ with an epistemic disjunction of the previous low and high fixpoints;

3. repeat step (2) for each terminal kernel $K_j$ by using the dual fixpoint equation with the transpose of the adjacency matrix $R/K_j$.

Time for a practical illustration.

```python
>>> from outrankingDigraphs import *
>>> g = RandomBipolarOutrankingDigraph(Normalized=True, seed=5)
>>> g
*------- Object instance description ------*
Instance class : RandomBipolarOutrankingDigraph
Instance name : rel_randomperftab
# Actions : 7
# Criteria : 7
Size : 26
Determinateness (%) : 67.14
Valuation domain : [-1.0;1.0]
Attributes : ['name', 'actions', 'criteria', 'evaluation',
'relation', 'valuationdomain', 'order',
'gamma', 'notGamma']
```

The random outranking digraph $g$, we consider here for illustration, models the pairwise outranking situations between seven decision alternatives evaluated on seven incommensurable performance criteria. We compute its corresponding bipolar-valued prekernels on the associated codual digraph $gcd$.

```python
>>> gcd = ~(-g) # strict outranking digraph
>>> gcd
>>> gcd.showPreKernels()
*--- Computing preKernels ---*
Dominant preKernels :
['a1', 'a4', 'a2']
  independence : +0.000
  dominance : +0.070
  absorbency : -0.488
  covering : +0.667
Absorbent preKernels :
['a7', 'a3']
  independence : +0.000
  dominance : -0.744
  absorbency : +0.163
  covered : +0.800
*----- statistics -----*
graph name: converse-dual_rel_randomperftab
number of solutions
  dominant kernels : 1
  absorbent kernels: 1
cardinality frequency distributions
  cardinality : [0, 1, 2, 3, 4, 5, 6, 7]
  dominant kernel : [0, 0, 1, 0, 0, 0, 0, 0]
  absorbent kernel: [0, 0, 1, 0, 0, 0, 0, 0]
Execution time : 0.00022 sec.
```

The codual outranking digraph, modelling a strict outranking relation, admits an initial prekernel $[a1, a2, a4]$ and a terminal one $[a3, a7]$ (see above the indeterminate independence qualifications in Line 7 and 13).

Let us compute the initial prekernel restricted adjacency table with the `digraphs.Digraph`. 

```python
>>> g = RandomBipolarOutrankingDigraph(Normalized=True, seed=5)
>>> g
```
We first notice that this initial prekernel is indeed only weakly independent: The outranking situation between $a_4$ and $a_1$ appears indeterminate. The corresponding initial prekernel membership characteristic vector may be computed with the `digraphs.Digraph.computeKernelVector()` method.

We start the fixpoint computation with an empty set characterisation as first low vector and a complete set $X$ characterising high vector. After each iteration, the low vector is set to the negation of the previous high vector and the high vector is set to the negation of the previous low vector.

A unique stable prekernel characteristic vector $Y_1$ is here attained at the fourth iteration with positive members $a_2: +0.21$ and $a_4: +0.21$ (60.5% criteria significance majority); $a_1: 0.00$ being an ambiguous potential member. Alternatives $a_3$, $a_5$, $a_6$ and $a_7$ are all negative members, i.e. positive non members of this outranking prekernel.

Let us now compute the restricted adjacency table for the outranked, i.e. the terminal prekernel $[a_3, a_7]$. 

---

Here is the code used to generate the initial kernel restricted adjacency table:

```python
>>> k1Relation = gcd.domkernelrestrict(['a1','a2','a4'])
>>> gcd.showHTMLRelationTable(...     actionsList=['a1','a2','a4','a3','a5','a6','a7'],...
...     relation=k1Relation, ...
...     tableTitle='K1 restricted adjacency table')
```

---

**K1 restricted adjacency table**

<table>
<thead>
<tr>
<th>r(x,y)</th>
<th>a1</th>
<th>a2</th>
<th>a4</th>
<th>a3</th>
<th>a5</th>
<th>a6</th>
<th>a7</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>-</td>
<td>-0.23</td>
<td>-1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.16</td>
</tr>
<tr>
<td>a2</td>
<td>-0.21</td>
<td>-</td>
<td>-0.21</td>
<td>0.21</td>
<td>0.44</td>
<td>0.05</td>
<td>0.49</td>
</tr>
<tr>
<td>a4</td>
<td>0.00</td>
<td>-0.21</td>
<td>-</td>
<td>0.21</td>
<td>0.00</td>
<td>0.07</td>
<td>0.58</td>
</tr>
<tr>
<td>a3</td>
<td>-0.28</td>
<td>-0.21</td>
<td>-0.74</td>
<td>-</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>a5</td>
<td>-0.26</td>
<td>-0.67</td>
<td>0.00</td>
<td>0.00</td>
<td>-</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>a6</td>
<td>-0.12</td>
<td>-0.49</td>
<td>-0.49</td>
<td>0.00</td>
<td>0.00</td>
<td>-</td>
<td>0.00</td>
</tr>
<tr>
<td>a7</td>
<td>-0.51</td>
<td>-0.49</td>
<td>-0.86</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-</td>
</tr>
</tbody>
</table>

Valuation domain: [-1.00; +1.00]

---

Fig. 69: Initial kernel $[a_1, a_2, a_4]$ restricted adjacency table

---

We start the fixpoint computation with an empty set characterisation as first low vector and a complete set $X$ characterising high vector. After each iteration, the low vector is set to the negation of the previous high vector and the high vector is set to the negation of the previous low vector.

A unique stable prekernel characteristic vector $Y_1$ is here attained at the fourth iteration with positive members $a_2: +0.21$ and $a_4: +0.21$ (60.5% criteria significance majority); $a_1: 0.00$ being an ambiguous potential member. Alternatives $a_3$, $a_5$, $a_6$ and $a_7$ are all negative members, i.e. positive non members of this outranking prekernel.

Let us now compute the restricted adjacency table for the outranked, i.e. the terminal prekernel $[a_3, a_7]$. 

---

Here is the code used to generate the restricted adjacency table:

```python
>>> gcd.computeKernelVector(['a1', 'a2', 'a4'], Initial=True, Comments=True)
--> Initial prekernel: {'a1', 'a4', 'a2'}
initial low vector : [-1.00, -1.00, -1.00, -1.00, -1.00, -1.00, -1.00]
initial high vector: [+1.00, +1.00, +1.00, +1.00, +1.00, +1.00, +1.00]
1st low vector : [ 0.00, +0.21, -0.21, 0.00, -0.44, -0.07, -0.58]
1st high vector: [+1.00, +1.00, +1.00, +1.00, +1.00, +1.00, +1.00]
2nd low vector: [ 0.00, +0.21, -0.21, 0.00, -0.44, -0.07, -0.58]| 2nd high vector: [+1.00, +1.00, +1.00, +1.00, +1.00, +1.00, +1.00]
3rd low vector : [ 0.00, +0.21, -0.21, +0.21, -0.21, -0.05, -0.21]
3rd high vector: [+1.00, +1.00, +1.00, +1.00, +1.00, +1.00, +1.00]
4th low vector: [ 0.00, +0.21, -0.21, +0.21, -0.21, -0.07, -0.21]
4th high vector: [+1.00, +1.00, +1.00, +1.00, +1.00, +1.00, +1.00]
# iterations : 4
low & high fusion : [ 0.00, +0.21, -0.21, +0.21, -0.21, -0.07, -0.21]
Choice vector for initial prekernel: {'a1', 'a2', 'a4'}

a2: +0.21
a4: +0.21
a1: 0.00
a6: -0.07
a3: -0.21
a5: -0.21
a7: -0.21

---
Again, we notice that this terminal prekernel is indeed only weakly independent. The corresponding bipolar-valued characteristic vector $Y_2$ may be computed as follows.

A unique stable bipolar-valued high and low fixpoint is attained at the third iteration with $a7$ positively confirmed (about 75% criteria significance majority) as member of this terminal prekernel, whereas the membership of $a3$ in this prekernel appears indeterminate. All the remaining nodes have negative membership characteristic values and are hence positively excluded from this prekernel.

When we reconsider the graphviz drawing of this outranking digraph in Fig. 71 (see the tutorial ‘On computing digraph kernels’ (page 88) Fig. 45), it becomes obvious why alternative $a1$ is neither included nor excluded from the initial prekernel. Same observation is applicable to alternative $a3$ which can neither be included nor excluded from the terminal prekernel. It may even happen, in case of more indeterminate outranking situations, that no alternative is positively included or excluded from a weakly independent prekernel; the corresponding bipolar-valued membership characteristic vector being completely indeterminate (see for instance the tutorial ‘Computing a best choice recommendation’).
Fig. 71: The strict outranking digraph oriented by the positive members of its initial and terminal prekernels (page 57').

To illustrate finally why sometimes we need to operate an epistemic disjunctive fusion of unequal stable low and high membership characteristics vectors (see Step 2.c.), let us consider, for instance, the following crisp 7-cycle graph.

```python
>>> g = CirculantDigraph(order=7,circulants=[-1,1])
>>> g

*------- Digraph instance description ------*
Instance class : CirculantDigraph
Instance name : c7
Digraph Order : 7
Digraph Size : 14
Valuation domain : [-1.00;1.00]
Determinateness (%) : 100.00
Attributes : ['name', 'order', 'circulants', 'actions', 'valuationdomain', 'relation', 'gamma', 'notGamma']

Digraph c7 is a symmetric crisp digraph showing, among others, the maximal independent set {'2','5','7'}, i.e. an initial as well as terminal kernel. We may compute the corresponding initial kernel characteristic vector.

```
Notice that the stable low vector characterises the **negative membership** part, whereas, the stable high vector characterises the **positive membership** part (see Lines 9-10 above). The bipolar **disjunctive fusion** assembles eventually both stable parts into the correct prekernel characteristic vector (Line 12).

The adjacency matrix of a symmetric digraph staying *unchanged* by the transposition operator, the previous computations, when qualifying the same kernel as a *terminal* instance, will hence produce exactly the same result.

It is worthwhile noticing again the essential computational role, the logical **indeterminate value 0.0** is playing in this dual fixpoint algorithm. To implement such kind of algorithms without a logical **neutral term** would be like implementing numerical algorithms without a possible usage of the number 0. Infinitely many trivial **impossibility theorems** and **dubious logical results** come up.

Back to *Tutorials of the Digraph3 resources* (page 3)

## 17 Bibliography

### References


