Documentation of the DIGRAPH3 software collection

Tutorials and Pearls

Raymond BISDORFF

Luxembourg, February 2020

Last updated: February 21, 2020

This documentation is dedicated to late Marc ROUBENS who, through his expertise in fuzzy sets and systems, greatly stimulated our research in bipolar-valued epistemic logic.

More documents are freely available here
Contents

1 Working with the Digraph3 software resources
   1.1 Purpose ................................................. 4
   1.2 Downloading of the Digraph3 resources .......... 4
   1.3 Starting a Python3 session ......................... 5
   1.4 Digraph object structure ......................... 5
   1.5 Permanent storage ................................. 6
   1.6 Inspecting a Digraph object ..................... 6
   1.7 Special classes ..................................... 9

2 Manipulating Digraph objects .......................... 10
   2.1 Random digraph ................................... 11
   2.2 Graphviz drawings ............................... 12
   2.3 Asymmetric and symmetric parts .................. 13
   2.4 Border and inner parts ............................ 15
   2.5 Fusion by epistemic disjunction ................. 16
   2.6 Dual, converse and co dual digraphs ............ 16
   2.7 Symmetric and transitive closures ............... 18
   2.8 Strong components ................................ 18
   2.9 CSV storage ........................................ 19
   2.10 Complete, empty and indeterminate digraphs .... 20

3 Computing the winner of an election ................. 22
   3.1 Linear voting profiles ............................ 22
   3.2 Computing the winner ............................. 23
   3.3 The Condorcet winner ............................ 24
   3.4 Cyclic social preferences ........................ 26

4 Working with the outrankingDigraphs module ....... 28
   4.1 Outranking digraph ............................... 29
   4.2 Browsing the performances ....................... 30
   4.3 Valuation semantics ............................. 32
   4.4 Pairwise comparisons ............................ 32
   4.5 Recoding the valuation ........................... 34
   4.6 Codual digraph .................................... 34
   4.7 XMCDA 2.0 .......................................... 35

5 Generating random performance tableaux ............. 36
   5.1 Introduction ...................................... 36
   5.2 Generating standard random performance tableaux 37
   5.3 Generating random Cost-Benefit tableaux ......... 39
   5.4 Generating three objectives tableaux ............. 43
   5.5 Generating random linearly ranked performances ... 47

6 Ranking with multiple incommensurable criteria .... 47
   6.1 The ranking problem .............................. 48
   6.2 The Copeland ranking ............................ 51
6.3 The NetFlows ranking ........................................ 52
6.4 Kemeny rankings ............................................. 53
6.5 Slater rankings ............................................... 56
6.6 Kohler's ranking-by-choosing rule ......................... 58
6.7 Tideman's ranked-pairs rule ............................... 58
6.8 Ranking big performance tableaux ......................... 59

7 HPC ranking with big outranking digraphs ............... 61
7.1 C-compiled Python modules ............................... 62
7.2 Big Data performance tableaux ............................ 62
7.3 C-implemented integer-valued outranking digraphs ...... 64
7.4 The sparse outranking digraph implementation .......... 65
7.5 Ranking big sets of decision alternatives ............... 70
7.6 HPC quantiles ranking records ........................... 72

8 Computing a best choice recommendation .................. 78
8.1 What site to choose? ...................................... 78
8.2 Performance tableau ...................................... 80
8.3 Outranking digraph ........................................ 82
8.4 Rubis best choice recommendations ........................ 84
8.5 Computing strict best choice recommendations ......... 86
8.6 Weakly ordering the outranking digraph ................ 87

9 Rating with learned quantile norms ........................ 89
9.1 Introduction ............................................... 89
9.2 Incremental learning of historical performance quantiles 90
9.3 Rating new performances with quantile norms ........ 93

10 Working with the graphs module .......................... 100
10.1 Structure of a Graph object ................................ 100
10.2 q-coloring of a graph .................................... 103
10.3 MIS and clique enumeration .............................. 104
10.4 Line graphs and maximal matchings ..................... 105
10.5 Grids and the Ising model .............................. 108
10.6 Simulating Metropolis random walks ................... 109

11 Computing the non isomorphic MISs of the 12-cycle graph ......................................................... 111
11.1 Introduction ............................................... 111
11.2 Computing the maximal independent sets (MISs) ...... 112
11.3 Computing the automorphism group ..................... 114
11.4 Computing the isomorphic MISs ......................... 114

12 On computing digraph kernels .............................. 116
12.1 What is a graph kernel? .................................. 116
12.2 Initial and terminal kernels ............................. 121
12.3 Kernels in lateralized digraphs .......................... 126
12.4 Computing good and bad choice recommendations .... 129
12.5 Tractability ............................................... 133
13 About split, interval and permutation graphs 135
13.1 A multiply perfect graph ........................... 135
13.2 Who is the liar ? ................................ 137
13.3 Generating permutation graphs ................. 140
13.4 Recognizing permutation graphs ............... 143

14 On tree graphs and graph forests 148
14.1 Generating random tree graphs ............... 149
14.2 Recognizing tree graphs ....................... 151
14.3 Spanning trees and forests .................... 152
14.4 Maximum determined spanning forests ...... 154

15 Bibliography 156

References 156

Tutorials of the Digraph3 Resources

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

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

Some of the tutorials are decision problem oriented and show how to compute the potential winner(s) of an election, how to build a best choice recommendation, or how to rate or linearly rank with multiple incommensurable performance criteria.

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

Another tutorial is furthermore devoted on how to generally compute kernels in graphs, digraphs and, more specifically, initial and terminal kernels in outranking digraphs. Special tutorials are devoted to perfect graphs, like split, interval and permutation graphs, and to tree-graphs and forests.
1 Working with the *Digraph3* software resources

- **Purpose**
- **Downloading of the Digraph3 resources**
- **Starting a Python3 session**
- **Digraph object structure**
- **Permanent storage**
- **Inspecting a Digraph object**
- **Special classes**

1.1 Purpose

The basic idea of the Digraph3 Python resources is to make easy python interactive sessions or write short Python3 scripts for computing all kind of results from a bipolar-valued digraph or graph. These include such features as maximal independent, maximal dominant or absorbent choices, rankings, outrankings, linear ordering, etc. Most of the available computing resources are meant to illustrate a Master Course on *Algorithmic Decision Theory* given at the University of Luxembourg in the context of its *Master in Information and Computer Science* (MICS).

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

1.2 Downloading of the Digraph3 resources

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

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

2. Or, from sourceforge.net
...$ git clone https://git.code.sf.net/p/digraph3/code Digraph3

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

1.3 Starting a Python3 session

You may start an interactive Python3 session in the Digraph3 directory for exploring the classes and methods provided by the 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.

>>> ...

>>> 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> ---*

1.4 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 **valuation domain**, a dictionary with three decimal entries: the minimum (-1.0, means certainly false), the median (0.0, means missing information) and the maximum characteristic value (+1.0, means certainly true);

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

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

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

### 1.5 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': -1.0, 'med': 0.0, 'max': 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},
}
```

### 1.6 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()
```
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'
------|------------------------
'a1' | -1 -1 1 -1 -1
'a2' | 1 -1 -1 1 1
'a3' | 1 -1 -1 1 -1
'a4' | 1 1 1 -1 -1
'a5' | 1 1 1 -1 -1

Valuation domain: [-1;+1]

*--- 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', 'a5'}, out => {'a1', '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 (see Fig. 1.1), if the graphviz (https://graphviz.org/) tools are installed on your system.

```python
>>> dg.exportGraphViz('tutorialDigraph')

*---- exporting a dot file do GraphViz tools --------*
Exporting to tutorialDigraph.dot
dot -Grankdir=BT -Tpng tutorialDigraph.dot -o tutorialDigraph.png
```

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.
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
size : 12 arcs
# undetermined : 0 arcs
determinateness : 1.00
arc density : 0.60
double arc density : 0.40
single arc density : 0.40
absence density : 0.20
strict single arc density: 0.40
strict absence density : 0.20
# components : 1
# strong components : 1
transitivity degree : 0.48
: [0, 1, 2, 3, 4, 5]
outdegrees distribution : [0, 1, 1, 3, 0, 0]
indegrees distribution : [0, 1, 2, 1, 1, 0]
mean outdegree : 2.40
mean indegree : 2.40
: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
symmetric degrees dist. : [0, 0, 0, 0, 1, 4, 0, 0, 0, 0, 0]
mean symmetric degree : 4.80
outdegrees concentration index : 0.1667
indegrees concentration index : 0.2333
symdegrees concentration index : 0.0333
: [0, 1, 2, 3, 4, 'inf']
neighbourhood depths distribution: [0, 1, 4, 0, 0, 0]
mean neighbourhood depth : 1.80
digraph diameter : 2
```
1.7 Special classes

Some special classes of digraphs, like the `digraphs.CompleteDigraph`, the `digraphs.EmptyDigraph` or the oriented `digraphs.GridDigraph` class for instance, are readily available (see Fig. 1.2).

```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
```
```bash
dot -Grankdir=BT -Tpng TutorialGrid.dot -o tutorialGrid.png
```
2 Manipulating Digraph objects

- Random digraph
- Graphviz drawings
- Asymmetric and symmetric parts
- Border and inner parts
- Fusion by epistemic disjunction
- Dual, converse and codual digraphs
- Symmetric and transitive closures
- Strong components
2.1 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 randomDigraphs.RandomValuationDigraph constructor.

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

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']
>>> dg.showRelationTable(ReflexiveTerms=False)
* ---- Relation Table ----- 
<table>
<thead>
<tr>
<th>r(xSy)</th>
<th>'1'</th>
<th>'2'</th>
<th>'3'</th>
<th>'4'</th>
<th>'5'</th>
<th>'6'</th>
<th>'7'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'1'</td>
<td>-</td>
<td>-0.48</td>
<td>0.70</td>
<td>0.86</td>
<td>0.30</td>
<td>0.38</td>
<td>0.44</td>
</tr>
<tr>
<td>'2'</td>
<td>-0.22</td>
<td>-0.38</td>
<td>0.50</td>
<td>0.80</td>
<td>-0.54</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>'3'</td>
<td>-0.42</td>
<td>0.08</td>
<td>-0.70</td>
<td>-0.56</td>
<td>0.84</td>
<td>-1.00</td>
<td></td>
</tr>
<tr>
<td>'4'</td>
<td>0.44</td>
<td>-0.40</td>
<td>-0.62</td>
<td>-0.04</td>
<td>0.66</td>
<td>0.76</td>
<td></td>
</tr>
<tr>
<td>'5'</td>
<td>0.32</td>
<td>-0.48</td>
<td>-0.46</td>
<td>0.64</td>
<td>-0.22</td>
<td>-0.52</td>
<td></td>
</tr>
<tr>
<td>'6'</td>
<td>-0.84</td>
<td>0.00</td>
<td>-0.40</td>
<td>-0.96</td>
<td>-0.18</td>
<td>-0.22</td>
<td></td>
</tr>
<tr>
<td>'7'</td>
<td>0.88</td>
<td>0.72</td>
<td>0.82</td>
<td>0.52</td>
<td>-0.84</td>
<td>0.04</td>
<td></td>
</tr>
</tbody>
</table>
>>> dg.showNeighborhoods()
Neighborhoods observed in digraph 'randomValuation'
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'}
```
Warning: Mind that most Digraph class methods will ignore the reflexive couples by considering that the reflexive relations are indeterminate, i.e. the characteristic value \( r(x S x) \) for all action \( x \) is put to the median, i.e. indeterminate value 0 in this case (see [BIS-2004]).

2.2 Graphviz drawings

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

```python
>>> dg.exportGraphViz('tutRandValDigraph')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to tutRandValDigraph.dot
dot -Grankdir=BT -Tpng tutRandValDigraph.dot -o tutRandValDigraph.png
```
Double links are drawn in bold black with an arrowhead at each end, whereas single asymmetric links are drawn in black with an arrowhead showing the direction of the link. Notice the undetermined relational situation ($r(6, S, 2) = 0.00$) observed between nodes ‘6’ and ‘2’. The corresponding link is marked in gray with an open arrowhead in the drawing (see Fig. 2.1).

### 2.3 Asymmetric and symmetric parts

We may now extract both the *symmetric* as well as the *asymmetric* part of digraph $dg$ with the help of two corresponding constructors (see Fig. 2.2).
Fig. 2.2: Asymmetric and symmetric part of the tutorial random valuation digraph

Note: Notice that the partial objects \texttt{asymDg} and \texttt{symDg} put to the indeterminate characteristic value all not-asymmetric, respectively not-symmetric links between nodes (see Fig. 2.2).

Here below, for illustration the source code of relation constructor of the \texttt{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]
                    elif relationIn[a][b] <= Med and relationIn[b][a] >= Med:
                        relationOut[a][b] = relationIn[a][b]
                    else:
                        # (continues on next page)
```

(continues on next page)
2.4 Border and inner parts

We may also extract the border -i.e. the part of the digraph induced by the union of its initial and terminal prekernels (see *On computing digraph kernels*)- as well as the inner part -i.e. the complement of the border part- of digraph $dg$ with the help of two corresponding class constructors (see Fig. 2.3).

```python
from digraphs import GraphBorder, GraphInner
bg = GraphBorder(dg)
bg.exportGraphViz()
ig = GraphInner(dg)
ig.exportGraphViz()
```

![Fig. 2.3: Border and inner part of the tutorial random valuation digraph](Rubis Python Server (graphviz), R. Bisdorff, 2008)

**Note:** Notice that the partial objects $bg$ and $ig$ put to the indeterminate characteristic value all *not border*, respectively *not inner* links between nodes (see Fig. 2.3).
2.5 Fusion by epistemic disjunction

We may recover object \(dg\) from both partial objects \(asymDg\) and \(symDg\), or as well from the border \(bg\) and the inner \(ig\), with a bipolar fusion constructor, also called epistemic disjunction, available via the digraphs.FusionDigraph class. The epistemic disjunction operation \(o\text{-}max\) works as follows:

Let \(r\) and \(r’\) characterise two bipolar-valued epistemic situations.

- \(o\text{-}max(r, r’) = \max(r, r’)\) when both \(r\) and \(r’\) are validated (positive);
- \(o\text{-}max(r, r’) = \min(r, r’)\) when both \(r\) and \(r’\) are invalidated (negative);
- \(o\text{-}max(r, r’) = \text{indeterminate}\) otherwise.

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

<table>
<thead>
<tr>
<th>(r(xSy))</th>
<th>'1'</th>
<th>'2'</th>
<th>'3'</th>
<th>'4'</th>
<th>'5'</th>
<th>'6'</th>
<th>'7'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>-0.22</td>
<td>-0.22</td>
<td>-0.22</td>
<td>-0.22</td>
<td>-0.22</td>
<td>-0.22</td>
<td>-0.22</td>
<td>-0.22</td>
</tr>
<tr>
<td>-0.42</td>
<td>-0.42</td>
<td>-0.42</td>
<td>-0.42</td>
<td>-0.42</td>
<td>-0.42</td>
<td>-0.42</td>
<td>-0.42</td>
</tr>
<tr>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
</tr>
<tr>
<td>-0.84</td>
<td>-0.84</td>
<td>-0.84</td>
<td>-0.84</td>
<td>-0.84</td>
<td>-0.84</td>
<td>-0.84</td>
<td>-0.84</td>
</tr>
<tr>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
</tr>
</tbody>
</table>
```

2.6 Dual, converse and codual digraphs

We may as readily compute the dual (negated relation\(^{14}\)), the converse (transposed relation) and the codual (transposed and negated relation) of the digraph instance \(dg\).

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

<table>
<thead>
<tr>
<th>(-r(xSy))</th>
<th>'1'</th>
<th>'2'</th>
<th>'3'</th>
<th>'4'</th>
<th>'5'</th>
<th>'6'</th>
<th>'7'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
</tr>
<tr>
<td>0.42</td>
<td>0.42</td>
<td>0.42</td>
<td>0.42</td>
<td>0.42</td>
<td>0.42</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>-0.44</td>
<td>-0.44</td>
<td>-0.44</td>
<td>-0.44</td>
<td>-0.44</td>
<td>-0.44</td>
<td>-0.44</td>
<td>-0.44</td>
</tr>
<tr>
<td>-0.32</td>
<td>-0.32</td>
<td>-0.32</td>
<td>-0.32</td>
<td>-0.32</td>
<td>-0.32</td>
<td>-0.32</td>
<td>-0.32</td>
</tr>
<tr>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
</tr>
<tr>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
</tr>
</tbody>
</table>
```

\(^{14}\) Not to be confused with the dual graph of a plane graph \(g\) that has a vertex for each face of \(g\). Here we mean the less than (strict converse) relation corresponding to a greater or equal relation, or the less than or equal relation corresponding to a (strict) better than relation.
Computing the dual, respectively the converse, may also be done with prefixing the `__neg__` (-) or the `__invert__` (~) operator. The co-dual of a Digraph object may, hence, as well be computed with a composition (in either order) of both operations.

```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

>>> 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
```
2.7 Symmetric and transitive closures

Symmetric and transitive closure in-site constructors are also available (see Fig. 2.4). Note that it is a good idea, before going ahead with these in-site operations who irreversibly modify the original \( dg \) object, to previously make a backup version of \( dg \). The simplest storage method, always provided by the generic `digraphs.Digraph.save()`, writes out in a named file the python content of the Digraph object in string representation.

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

Fig. 2.4: Symmetric and transitive closure of the tutorial random valuation digraph

2.8 Strong components

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

*----- show detail -----*

Digraph : tutRandValDigraph_Scc

*---- Actions ----*

['_7_1_2_6_5_3_4_']

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

S | 'Scc_1'
---|--------
'Scc_1' | 0.00

short content

Neighborhoods:

Gamma :

'frozenset({7, 1, 2, 6, 5, 3, 4}): in => set(), out => set()'

Not Gamma :

'frozenset({7, 1, 2, 6, 5, 3, 4}): in => set(), out => set()'

2.9 CSV storage

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

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

# content of file tutRandValDigraph.csv

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

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

>>> dgcsv = CSVDigraph('tutRandValDigraph')
>>> dgcsv.showRelationTable(ReflexiveTerms=False)

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

r(xSy) | '1' '2' '3' '4' '5' '6' '7'
-------|------------------------------------------------------------
'Scc_1' | -0.48 0.70 0.86 0.30 0.38 0.44
'1' | -0.22 -0.38 0.50 0.80 -0.54 0.02
'2' | -0.42 0.08 -0.70 -0.56 0.84 -0.04 -1.0

(continues on next page)
It is as well possible to show a colored version of the valued relation table in a system browser window tab (see Fig. 2.5).

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

**Tutorial random digraph**

<table>
<thead>
<tr>
<th>(x)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>-0.48</td>
<td>0.70</td>
<td>0.86</td>
<td>0.30</td>
<td>0.38</td>
<td>0.44</td>
</tr>
<tr>
<td>2</td>
<td>-0.22</td>
<td>0.00</td>
<td>-0.38</td>
<td>0.50</td>
<td>0.80</td>
<td>-0.54</td>
<td>0.02</td>
</tr>
<tr>
<td>3</td>
<td>-0.42</td>
<td>0.08</td>
<td>0.00</td>
<td>0.70</td>
<td>-0.56</td>
<td>0.84</td>
<td>-1.00</td>
</tr>
<tr>
<td>4</td>
<td>0.44</td>
<td>-0.40</td>
<td>-0.62</td>
<td>0.00</td>
<td>0.04</td>
<td>0.66</td>
<td>0.76</td>
</tr>
<tr>
<td>5</td>
<td>0.32</td>
<td>-0.48</td>
<td>-0.46</td>
<td>0.64</td>
<td>0.00</td>
<td>-0.22</td>
<td>-0.52</td>
</tr>
<tr>
<td>6</td>
<td>-0.84</td>
<td>0.00</td>
<td>-0.40</td>
<td>-0.96</td>
<td>-0.18</td>
<td>-0.22</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>0.88</td>
<td>0.72</td>
<td>0.82</td>
<td>0.52</td>
<td>-0.84</td>
<td>0.04</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Fig. 2.5: The valued relation table shown in a browser window

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

### 2.10 Complete, empty and indeterminate digraphs

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

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

Help on class `CompleteDigraph` in module `digraphs`:

```
 Parameters:
   order > 0; valuationdomain=(Min,Max).
 Specialization of the general Digraph class for generating
```
temporary complete graphs of order 5 in \{-1,0,1\} by default.

Method resolution order:

CompleteDigraph
Digraph
builtins.object

...  

```
>>> e = EmptyDigraph(order=5)
>>> e.showRelationTable()
* ---- Relation Table -----  
S | '1' | '2' | '3' | '4' | '5'
---- -|----------------------------------------
'1' | -1.00 -1.00 -1.00 -1.00 -1.00
'2' | -1.00 -1.00 -1.00 -1.00 -1.00
'3' | -1.00 -1.00 -1.00 -1.00 -1.00
'4' | -1.00 -1.00 -1.00 -1.00 -1.00
'5' | -1.00 -1.00 -1.00 -1.00 -1.00
```

```
>>> e.showNeighborhoods()
Neighborhoods:

Gamma :  
'1' : in => set(), out => set()
'2' : in => set(), out => set()
'3' : in => set(), out => set()
'4' : in => set(), out => set()
'5' : 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'\}
'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()
'3' : in => set(), out => set()
'4' : in => set(), out => set()
'5' : in => set(), out => set()

Not Gamma :  
'1' : in => set(), out => set()
```

(continues on next page)
3 Computing the winner of an election

3.1 Linear voting profiles

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

```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,
...     RandomWeights=True)
>>> v.candidates
OrderedDict([('a1', {'name': 'a1'}), ('a2', {'name': 'a2'}),
             ('a3', {'name': 'a3'})])
>>> v.voters
OrderedDict([('v1', {'weight': 2}), ('v2', {'weight': 3}),
             ('v3', {'weight': 1}), ('v4', {'weight': 5}),
             ('v5', {'weight': 4})])
>>> v.linearBallot
{'v4': ['a1', 'a3', 'a2'], 'v3': ['a1', 'a3', 'a2'], 'v1': ['a1', 'a2', 'a3'],
 'v5': ['a2', 'a3', 'a1'], 'v2': ['a3', 'a2', 'a1']}

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

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

### 3.2 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.

```python
>>> v.computeUninominalVotes()
{'a2': 2, 'a1': 6, 'a3': 7}
>>> v.computeSimpleMajorityWinner()
['a3']
```
As we observe no absolute majority (8/15) of votes for any of the three candidates, we may look for the instant runoff winner instead (see [ADT-L2]).

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

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

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

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

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

### 3.3 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 $M(a1,a2) = 4 - 1 = +3$. 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.

```python
>>> from votingProfiles import CondorcetDigraph
>>> cdg = CondorcetDigraph(v,hasIntegerValuation=True)
>>> cdg.showAll()
*----- show detail -------------*
Digraph : rel_randLinearProfile
*----- Actions ----*
['a1', 'a2', 'a3']
*----- Characteristic valuation domain ----*
{'max': Decimal('15.0'), 'med': Decimal('0')},
```

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

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

By seeing the majority margins like a bipolar-valued characteristic function for a global preference relation defined on the set of candidates, we may use all operational resources of the generic Digraph class (see *Working with the Digraph3 software resources*), and especially its `exportGraphViz` method¹, for visualizing an election result.

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

![Fig. 3.1: Visualizing an election result](image.png)
### 3.4 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.

```python
>>> v.showLinearBallots()
voters(\text{weight}) \hspace{1cm} \text{candidates rankings}
\begin{align*}
v1(1): & \quad ['a1', 'a3', 'a5', 'a2', 'a4'] \\
v2(1): & \quad ['a1', 'a2', 'a4', 'a3', 'a5'] \\
v3(1): & \quad ['a5', 'a2', 'a4', 'a3', 'a1'] \\
v4(1): & \quad ['a3', 'a4', 'a1', 'a5', 'a2'] \\
v5(1): & \quad ['a4', 'a2', 'a3', 'a5', 'a1'] \\
v6(1): & \quad ['a2', 'a4', 'a5', 'a1', 'a3'] \\
v7(1): & \quad ['a5', 'a4', 'a3', 'a1', 'a2'] \\
v8(1): & \quad ['a2', 'a4', 'a5', 'a1', 'a3'] \\
v9(1): & \quad ['a5', 'a3', 'a4', 'a1', 'a2']
\end{align*}
```

```python
>>> cdg = CondorcetDigraph(v)
>>> cdg.showRelationTable()
*---- Relation Table ----*
\begin{tabular}{|c|cccc|}
\hline
S & 'a1' & 'a2' & 'a3' & 'a4' & 'a5' \\
\hline
'a1' & - & 0.11 & -0.11 & -0.56 & -0.33 \\
'a2' & -0.11 & - & 0.11 & 0.11 & -0.11 \\
'a3' & 0.11 & -0.11 & - & -0.33 & -0.11 \\
'a4' & 0.56 & -0.11 & 0.33 & - & 0.11 \\
'a5' & 0.33 & 0.11 & 0.11 & -0.11 & - \\
\hline
\end{tabular}
```

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.

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

```bash
dot -Grankdir=BT -Tpng cycles.dot -o cycles.png
```
But, there may be many cycles appearing in a digraph, and, we may detect and enumerate all minimal chordless circuits in a Digraph instance with the `computeChordlessCircuits()` method.

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

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

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

```python
>>> v.showHTMLVotingHeatmap(rankingRule='NetFlows')
```
Notice that the importance weights of the voters are negative, which means that the preference direction of the criteria (in this case the individual voters) is decreasing, i.e. goes from lowest (best) rank to highest (worst) rank. Notice also, that the compromise NetFlows ranking \( [a_4, a_5, a_2, a_1, a_3] \), shown in this heat map (see Fig. 3.3) results in an optimal ordinal correlation index of +0.778 with the pairwise majority voting margins (see tutorials OrdinalCorrelation-Tutorial-label and Ranking with multiple incommensurable criteria).

Back to Content Table

4 Working with the outrankingDigraphs module

- Outranking digraph
- Browsing the performances
- Valuation semantics
- Pairwise comparisons
- Recoding the valuation
- Co-dual digraph
- XMCDA 2.0

See also the technical documentation of the outrankingDigraphs-label.
4.1 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 a01, a02, ..., a07.

```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
comment: RandomPerformanceTableau() generated.
...
key: a07
name: random decision action
comment: RandomPerformanceTableau() generated.
```

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

4.2 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 \( a_{03} \) is in fact best performing on four out of seven criteria.

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

There is no doubt that action \( a_{03} \), with a performance in the highest quintile in five out
of seven criteria, appears definitely to be best performing. Action $a05$ shows a more or less average performance on most criteria, whereas action $a02$ appears to be the weakest alternative.

### 4.3 Valuation semantics

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

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

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

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

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

```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

>>> odg.valuationdomain
{ 'min': Decimal('-100.0'), 'max': Decimal('100.0'),
  'med': Decimal('0.0')}
```

### 4.4 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 $a02$. 

32
The outranking valuation characteristic appears as majority margin resulting from the difference of the weights of the criteria in favor of the statement that alternative \(a01\) is at least well performing as alternative \(a02\). No considerable performance difference being observed, no veto or counter-veto situation is triggered in this pairwise comparison. Such a case is, however, observed for instance when we pairwise compare the performances of alternatives \(a03\) and \(a02\).

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

All outranking digraphs, being of root type `digraphs.Digraph`, inherit the methods available under this class. The characteristic valuation domain of an outranking digraph may be recoded with the `digraphs.Digraph.recodeValuation()` 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
```

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

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

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

4.6 Codual digraph

From the theory (see [BIS-2013], [ADT-L7]) we know that a bipolar-valued outranking relation is weakly complete, i.e. if \(r(xSy) < 0.0\) then \(r(ySx) >= 0.0\). From this property follows that a bipolar-valued outranking relation verifies the **coduality** principle: the dual (strict negation -14) of the converse (inverse ~) of the outranking relation corresponds to its strict outranking part. We may visualize the codual (strict) outranking digraph with a graphviz drawing.

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

*---- exporting a dot file for GraphViz tools ---------*
```
Exporting to codualOdg.dot
```
```
dot -Grankdir=BT -Tpng codualOdg.dot -o codualOdg.png
```
It becomes readily clear now from the picture above that alternative $a_{03}$ 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.

### 4.7 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 2.0 (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/).

```python
>>> 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. Re-instantiating again a corresponding updated $odg$ object goes like follow.

```python
>>> 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
```

(continues on next page)
We recover the original bipolar-valued outranking characteristics, and we may restart again the preference modelling process.

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

Back to Content Table

5 Generating random performance tableaux

- Introduction
- Generating standard random performance tableaux
- Generating random Cost-Benefit tableaux
- Generating three objectives tableaux
- Generating random linearly ranked performances

5.1 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.

5.2 Generating standard random performance tableaux

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

- numberOfActions := nbr of decision actions.
- numberOfCriteria := number performance criteria.
- weightDistribution := ‘random’ (default) | ‘fixed’ | ‘equisignificant’:
  - If ‘random’, weights are uniformly selected randomly from the given weight scale;
  - If ‘fixed’, the weightScale must provide a corresponding weights distribution;
  - If ‘equisignificant’, all criterion weights are put to unity.
- weightScale := [Min,Max] (default = (1,numberOfCriteria)).
- IntegerWeights := True (default) | False (normalized to proportions of 1.0).
- commonScale := [a,b]; common performance measuring scales (default = [0.0,100.0])
- commonThresholds := [(q0,q1),(p0,p1),(v0,v1)]; common indifference(q), preference (p) and considerable performance difference discrimination thresholds. For each threshold type x in \{q,p,v\}, the float x0 value represents a constant percentage of the common scale and the float x1 value a proportional value of the actual performance measure. Default values are [(2.5,0.0,0.0),(5.0,0.0),(60.0,0.0,0.0)].
- commonMode := common random distribution of random performance measurements (default = (‘beta’,None,(2,2))):
  - (‘uniform’,None,None), uniformly distributed float values on the given common scales’ range [Min,Max].
  - (‘normal’,*mu*,*sigma*), truncated Gaussian distribution, by default \(mu = (b-a)/2\) 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, \( \text{mode} = \frac{b-a}{2} \) and \( \text{repartition} = 0.5 \).

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

- \( \text{valueDigits} := <\text{integer}> \), precision of performance measurements (2 decimal digits by default).
- \( \text{missingDataProbability} := 0 <= \text{float} <= 1.0 \); probability of missing performance evaluation on a criterion for an alternative (default 0.025).

Code example.

```python
>>> from randomPerfTabs import RandomPerformanceTableau
>>> t = RandomPerformanceTableau(numberOfActions=21,numberOfCriteria=13, seed=100)
>>> t.actions
{'a01': {'comment': 'RandomPerformanceTableau() generated.',
'name': 'random decision action'},
'a02': {... },
...
}
>>> t.criteria
{'g01': {'thresholds': {'ind': (Decimal('10.0'), Decimal('0.0'))},
'veto': (Decimal('80.0'), Decimal('0.0'))},
'pref': (Decimal('20.0'), Decimal('0.0'))},
'scale': [0.0, 100.0],
'weight': Decimal('1'),
'name': 'digraphs.RandomPerformanceTableau() instance',
'comment': 'Arguments: ; weightDistribution=random;
weightScale=(1, 1); commonMode=None'},
'g02': { ... },
...
}
>>> t.evaluation
{'g01': {'a01': Decimal('15.17'),
'a02': Decimal('44.51'),
'a03': Decimal('-999'), # missing evaluation
... },
...
}
>>> t.showHTMLPerformanceTableau()
```
5.3 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.

<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>51.39</td>
<td>65.22</td>
<td>71.96</td>
<td>57.04</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>73.57</td>
<td>71.11</td>
<td>21.81</td>
<td>56.90</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>62.29</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>90.88</td>
<td>39.56</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>93.16</td>
<td>81.89</td>
</tr>
<tr>
<td>a10</td>
<td>38.75</td>
<td>79.70</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>NA</td>
<td>85.80</td>
<td>58.37</td>
<td>49.33</td>
<td>NA</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>64.14</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>60.03</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>88.79</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`.

Fig. 5.1: Browser view on random performance tableau instance
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: weightDistribution = ‘equiobjectives’.

On cardinal criteria, both of cost or of benefit type, we observe following constant preference discrimination quantiles: 5% indifferent situations, 90% strict preference situations, and 5% veto situation.

Parameters:

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

Warning: Minimal number of decision actions required is 3!

Example Python session

```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
```

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

>>> t.showCriteria()

*---- criteria -----*

- g1 'random ordinal benefit criterion'
  Scale = (0, 10)
  Weight = 0.167
- g2 'random cardinal cost criterion'
  Scale = (0.0, 100.0)
  Weight = 0.250

Threshold ind : 1.76 + 0.00x ; percentile: 0.095
Threshold pref : 2.16 + 0.00x ; percentile: 0.143
Threshold veto : 73.19 + 0.00x ; percentile: 0.952

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

>>> t.showStatistics()

*-------- Performance tableau summary statistics -------*

Instance name : randomCBperftab
#Actions : 7
#Criteria : 5

*Statistics per Criterion*

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

- Criterion name : g2
  Criterion weight : 3
  criterion scale : -100.00 - 0.00
  mean evaluation : -49.32
  standard deviation : 27.59
  maximal evaluation : 0.00
  quantile Q3 (x_75) : -27.51

(continues on next page)
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**

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

*Color legend:*

| quantile | 0.20% | 0.40% | 0.60% | 0.80% | 1.00% |

Fig. 5.2: 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 !
```  
```python
>>> 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
```

...$ less temp.csv

"actions", "g1", "g2", "g3", "g4", "g5"
5.4 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,
- `missingDataProbability = 0.05` (default): random insertion of missing values with given probability,
- `seed= None.`

**Note:** If the mode of the _triangular_ distribution is set to ‘variable’, three modes at 0.3 (-), 0.5 (~), respectively 0.7 (+) of the common scale span are set at random for each coalition and action.
Warning: Minimal number of decision actions required is 3!

Example Python session

```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.

```python
>>> t.showActions()
key: a01
  name: random decision action Eco+ Soc- Env+
  profile: {'Eco': 'good', 'Soc': 'weak', 'Env': 'good'}
key: a02
  ...
key: a26
  name: random decision action Eco+ Soc+ Env-
  profile: {'Eco': 'good', 'Soc': 'good', 'Env': 'weak'}
... 
key: a30
```
Variable triangular modes (0.3, 0.5 or 0.7 of the span of the measure scale) for each objective result in different performance status for each decision action with respect to the three objectives. Action \textit{a01}, for instance, will probably show \textit{good} performances wrt the \textit{economic} and environmental aspects, and \textit{weak} performances wrt the \textit{societal} aspect.

For testing purposes we provide a special \texttt{perfTabs.PartialPerformanceTableau} class for extracting a \textbf{partial performance tableau} from a given tableau instance. In the example blow, we construct the partial performance tableaux corresponding to each on of the three decision objectives.

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

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

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

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

```python
>>> from digraphs import FusionLDigraph
>>> gfus = FusionLDigraph([geco,gsoc,genv])
>>> gfus.strongComponents()
{'a30'},
  frozenset({'a10', 'a03', 'a19', 'a08', 'a07', 'a04', 'a21', 'a20',
    'a13', 'a23', 'a16', 'a12', 'a24', 'a02', 'a31', 'a29',
    'a05', 'a09', 'a28', 'a25', 'a17', 'a14', 'a15', 'a06',
    'a01', 'a27', 'a11', 'a18', 'a22'}),
  frozenset({'a26'})}
```

```python
>>> from digraphs import StrongComponentsCollapsedDigraph
>>> scc = StrongComponentsCollapsedDigraph(gfus)
>>> scc.showActions()
*----- show digraphs actions --------------*
key: frozenset({'a30'})
```
A graphviz drawing illustrates the apparent preferential links between the strong components.

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

Decision action \textit{a26} (Eco+ Soc+ Env-) appears dominating the other decision alternatives, whereas decision action \textit{a30} (Eco- Soc- Env-) appears to be dominated by all the others.
5.5 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 `votingDigraphs.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 Content Table

6 Ranking with multiple incommensurable criteria

- The ranking problem
- The Copeland ranking
- The NetFlows ranking
- Kemeny rankings
- Slater rankings
- Kohler ‘s ranking-by-choosing rule
- Tideman ‘s ranked-pairs rule
- Ranking big performance tableaux
6.1 The ranking problem

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

Let us consider a didactic outranking digraph generated from a random Cost-Benefit performance tableau concerning 9 decision alternatives evaluated on 13 performance criteria. We may compute the corresponding strict outranking digraph as follows.

Listing 6.1: Random bipolar-valued strict outranking relation characteristics

```python
>>> from outrankingDigraphs import *
>>> t = RandomCBPerformanceTableau(numberOfActions=9,
...                                   numberOfCriteria=13,seed=200)
>>> g = BipolarOutrankingDigraph(t,Normalized=True)
>>> gcd = ~(-g)  # codual digraph
>>> gcd.showRelationTable(ReflexiveTerms=False)
```

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

Listing 6.2: Median cut polarised strict outranking relation characteristics

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

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

To estimate how difficult this ranking problem here 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 $a_8$ outranks alternative $a_6$ and alternative $a_6$ outranks $a_4$, however $a_8$ does not outrank $a_4$ (see Fig. 6.1). 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 $gcd$.

```python
>>> gcd.computeTransitivityDegree(Comments=True)
Transitivity degree of graph <converse-dual_rel_randomCBperftab> : 0.46
```

The strict 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.

```python
>>> gcd.computeChordlessCircuits()
>>> gcd.showChordlessCircuits()
```

(continues on next page)
There is one chordless circuit detected in the given strict outranking digraph $gcd$, namely $a6$ outranks $a7$, the latter outranks $a8$, and $a8$ outranks again $a6$ (see Fig. 6.1). 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’s or Tideman’s ranking rule.

### 6.2 The Copeland ranking

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

**Listing 6.3: Computing a Copeland Ranking**

```python
>>> from linearOrders import CopelandOrder
>>> cop = CopelandOrder(gcd, Comments=True)
```

Copeland scores

```plaintext
a5 : 12
a1 : 2
a6 : 2
a7 : 2
a8 : 0
a4 : -3
a9 : -3
a3 : -5
a2 : -7
```

Copeland Ranking:

```plaintext
['a5', 'a1', 'a6', 'a7', 'a8', 'a4', 'a9', 'a3', 'a2']
```

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

The Copeland scores deliver actually only a unique weak order, i.e. a ranking with potential ties. This weak order may be constructed with the transitiveDigraphs. WeakCopelandOrder class. Notice by the way that Copeland scores here are in fact
invariant under a codual, i.e. the negation of the converse \((\sim g)\) transform of the outranking digraph.

Copeland’s ranking rule actually renders a linear order which is rather correlated (+0.463) with the given pairwise outranking relation in the ordinal Kendall sense (see Listing 6.4 and [BIS-2012]).

Listing 6.4: Checking the quality of the Copeland Ranking

```python
>>> corr = gcd.computeRankingCorrelation(cop.copelandRanking)
>>> gcd.showCorrelation(corr)

Correlation indexes:
Extended Kendall tau index : +0.463
Epistemic determination level : 0.230
Bipolar-valued equivalence : +0.107
```

With an epistemic determination level of 0.230, the extended Kendall tau index is in fact computed on 61.50% of the pairwise strict outranking comparisons. Furthermore, the bipolar-valued relational equivalence characteristics between the strict outranking relation and the Copeland ranking equals +0.107, i.e. a majority of 55.35% of the criteria significance supports the relational equivalence between the given strict outranking relation and the corresponding Copeland ranking (see [BIS-2012]).

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

### 6.3 The NetFlows ranking

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

```python
>>> from linearOrders import NetFlowsOrder
>>> nf = NetFlowsOrder(gcd,Comments=True)

Net Flows:
a5 : 3.600
a7 : 2.800
a6 : 1.300
a3 : 0.033
a1 : -0.400
a8 : -0.567
a4 : -1.283
a9 : -2.600
a2 : -2.883

NetFlows Ranking:
['a5', 'a7', 'a6', 'a3', 'a1', 'a8', 'a4', 'a9', 'a2']
```

The net-flow scores deliver in this example a ranking without ties. It may happen often, however, that we obtain, as with the Copeland scores above, only a ranking with ties,
which may then be resolved again by following a lexicographic rule. In such cases, it is possible to construct again the corresponding *weak order* with the `transitiveDigraphs. WeakNetFlowsOrder` class.

The resulting *NetFlows* ranking is here, in this didactic example, slightly better correlated (+0.638) with the given outranking relation as its crisp cousin, the *Copeland* ranking.

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

Indeed, the extended Kendall tau index of +0.638 leads to a bipolar-valued equivalence characteristics of 0.147, i.e. a *majority* of 57.35% of the criteria significance supports the relational equivalence between the given strict outranking relation and the corresponding *Copeland* ranking.

To appreciate the respective quality of both the *Copeland* and the *NetFlows* rankings, it is useful to consider *Kemeny* ’s and *Slater* ’s *optimal* ranking rules.

### 6.4 Kemeny rankings

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

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

So, *+0.779* is the highest possible ordinal correlation (fitness) any potential ranking one can achieve with the given pairwise strict 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 here, we obtain in fact two optimal *Kemeny* rankings with a same maximal *Kemeny* index of 12.92.

Listing 6.5: Optimal Kemeny rankings

```
>>> ke.maximalRankings
[['a5', 'a6', 'a7', 'a3', 'a8', 'a9', 'a4', 'a1', 'a2'],
 ['a5', 'a6', 'a7', 'a3', 'a9', 'a4', 'a1', 'a8', 'a2']]
```
We may visualize the partial order defined by the *epistemic disjunction* of both *Kemeny* rankings (see transitiveDigraphs module and Fig. 6.2) as follows.

```python
>>> from transitiveDigraphs import KemenyOrdersFusion
>>> wke = KemenyOrdersFusion(gcd, orderLimit=9)
>>> wke.exportGraphViz(fileName='tutorialKemeny')
*---- exporting a dot file for GraphViz tools ----------*
Exporting to tutorialKemeny.dot
0 { rank = same; a5; }
1 { rank = same; a6; }
2 { rank = same; a7; }
3 { rank = same; a3; }
4 { rank = same; a9; a8; }
5 { rank = same; a4; }
6 { rank = same; a1; }
7 { rank = same; a2; }
dot -Grankdir=TB -Tpng tutorialKemeny.dot -o tutorialKemeny.png
```
Fig. 6.2: Epistemic disjunction of Kemeny rankings

It is interesting to notice in Listing 6.5, that both Kemeny rankings only differ in their respective ranking of alternative $a_8$; either before or after alternatives $a_9$, $a_4$, and $a_1$. 
### 6.5 Slater rankings

The Slater ranking rule is similar to Kemeny’s, but it is working, instead, on the associated median cut strict outranking digraph $ccd$. It renders here the following results.

```python
>>> sl = KemenyOrder(ccd,orderLimit=9)
>>> sl.showRanking()
['a5', 'a6', 'a4', 'a1', 'a3', 'a7', 'a8', 'a9', 'a2']
```

```python
>>> corr = gsd.computeOrdinalCorrelation(sl)
Correlation indexes:
Extended Kendall tau : +0.676
Epistemic determination : 0.230
Bipolar-valued equivalence : +0.156
```

```python
>>> len(sl.maximalRankings)
7
```

We notice that the first crisp Slater ranking is a rather good fit ($+0.676$), slightly better apparently than the NetFlows ranking ($+0.638$). However, there are apparently 7 such potentially optimal Slater rankings. The corresponding epistemic disjunction gives the following partial ordering (see Fig. 6.3).

```python
>>> slw = KemenyOrdersFusion(ccd,orderLimit=9)
>>> slw.exportGraphViz(fileName='tutorialSlater')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to tutorialSlater.dot
do...
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 (< 20 objects).

More efficient ranking heuristics, like the *Copeland* and the *NetFlows* rules, are therefore needed in practice.
6.6 Kohler’s ranking-by-choosing rule

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

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

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

2. Select the row where this minimum is maximal. Ties are resolved in lexicographic order;

3. Put the selected decision alternative at rank $i$;

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

```python
>>> from linearOrders import KohlerOrder
>>> ko = KohlerOrder(gcd)
>>> ko.showRanking()
[['a5', 'a7', 'a6', 'a3', 'a9', 'a8', 'a4', 'a1', 'a2']
 ['a5', 'a6', 'a7', 'a3', 'a8', 'a9', 'a4', 'a1', 'a2']
>>> corr = gcd.computeOrdinalCorrelation(ko)
>>> gcd.showCorrelation(corr)
Correlation indexes:
Extended Kendall tau : +0.747
Epistemic determination : 0.230
Bipolar-valued equivalence : +0.172

With this min-max ranking-by-choosing strategy, we find a correlation result (+0.747) that is clearly the nearest to an optimal Kemeny ranking (see Listing 6.5). Only two adjacent pairs: (‘a6’,’a7’) and (‘a8’,’a9’) are actually inverted here.

Kohler’s ranking has a dual version, the prudent Arrow-Raynaud ordering-by-choosing rule, where a corresponding max-min strategy, when used on the non-strict outranking digraph, for ordering the from the worst to the best produces the same eventual ranking result (see [DIA-2010]).

6.7 Tideman’s ranked-pairs rule

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

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

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)
>>> rp.showRanking()
['a5', 'a6', 'a7', 'a3', 'a8', 'a9', 'a4', 'a1', 'a2']
```

The RankedPairs ranking rule actually renders in fact an optimal Kemeny rankings as we may verify below.

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

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

Unfortunately, the ranking-by-choosing rules, as well as their dual ordering-by-choosing ones, are not efficiently scalable to outranking digraphs of larger orders (> 1000). For such outranking digraphs, with several hundred or thousands of alternatives, only the Copeland and, more accurately, the NetFlows ranking rules, with a polynomial complexity of \(O(n^2)\) where \(n\) is the order of the strict outranking digraph, remain in fact computationally efficient.

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

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

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

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

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

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

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

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

Back to Content Table

7 HPC ranking with big outranking digraphs

- C-compiled Python modules
- Big Data performance tableaux
- C-implemented integer-valued outranking digraphs
- The sparse outranking digraph implementation
7.1 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 (\( > 50 \text{GB} \)) are available, ranking up to several millions of alternatives (see [BIS-2016]) becomes effectively tractable.

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

- \texttt{cRandPerfTabs.pyx}
- \texttt{cIntegerOutrankingDigraphs.pyx}
- \texttt{cIntegerSortingDigraphs.pyx}
- \texttt{cSparseIntegerOutrankingDigraphs.pyx}

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

7.2 Big Data performance tableaux

In order to efficiently type the C variables, the \texttt{cRandPerfTabs} module provides the usual random performance tableau models, but, with \texttt{integer} action keys, \texttt{float} performance evaluations, \texttt{integer} criteria weights and \texttt{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 \texttt{actions} and \texttt{criteria} dictionaries.

```python
>>> 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
```

\(^6\) See \url{https://cython.org/}

Attributes : ['randomSeed', 'name', 'actions', 'criteria', 'evaluation', 'weightPreorder']

>>> t.actions
OrderedDict([(1, {'name': '#1'}), (2, {'name': '#2'}),
             (3, {'name': '#3'}), (4, {'name': '#4'})])

>>> t.criteria
OrderedDict([
    ('g1', {'name': 'RandomPerformanceTableau() instance',
             'comment': 'Arguments: ; weightDistribution=equisignificant;
                         weightScale=(1, 1); commonMode=None',
             'thresholds': {'ind': (10.0, 0.0),
                           'pref': (20.0, 0.0),
                           'veto': (80.0, 0.0)},
             'scale': (0.0, 100.0),
             'weight': 1,
             'preferenceDirection': 'max'}),
    ('g2', {'name': 'RandomPerformanceTableau() instance',
             'comment': 'Arguments: ; weightDistribution=equisignificant;
                         weightScale=(1, 1); commonMode=None',
             'thresholds': {'ind': (10.0, 0.0),
                           'pref': (20.0, 0.0),
                           'veto': (80.0, 0.0)},
             'scale': (0.0, 100.0),
             'weight': 1,
             'preferenceDirection': 'max'})])

>>> t.evaluation
{'g1': {1: 35.17, 2: 56.4, 3: 1.94, 4: 5.51},
 'g2': {1: 95.12, 2: 90.54, 3: 51.84, 4: 15.42}}

>>> t.showPerformanceTableau()
Criteria | 'g1'   'g2'
Actions   | 1 1
----------|--------
#1        | 91.18  90.42
#2        | 66.82  41.31
#3        | 35.76  28.86
#4        | 7.78   37.64

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

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

*-------- PerformanceTableau instance description -------*
Instance class : PerformanceTableau
Seed : None
Instance name : std_cRandomperftab
# Actions : 4
# Criteria : 2
7.3 C-implemented integer-valued outranking digraphs

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

```python
>>> t = cRandomPerformanceTableau(numberOfActions=1000,numberOfCriteria=2)
>>> from cIntegerOutrankingDigraphs import *
>>> g = IntegerBipolarOutrankingDigraph(t,Threading=True,nbrCores=4)
>>> g

*------- Object instance description ------*
Instance class : IntegerBipolarOutrankingDigraph
Instance name : rel_cRandomperftab
# Actions : 1000
# Criteria : 2
Size : 465024
Determinateness : 56.877
Valuation domain : {'min': -2, 'med': 0, 'max': 2,
'hasIntegerValuation': True}
---- Constructor run times (in sec.) ----
Total time : 4.23880
Data input : 0.01203
Compute relation : 3.60788
Gamma sets : 0.61889
#Threads : 4
Attributes : ['name', 'actions', 'criteria', 'totalWeight',
'valuationdomain', 'methodData', 'evaluation',
'order', 'runTimes', 'nbrThreads', 'relation',
'gamma', 'notGamma']
```

On a classic intel-i7 equipped PC with four single threaded cores, the `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 \( g_1 \). The standard `Decimal` valued `BipolarOutrankingDigraph` instance \( g_1 \) 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.

### 7.4 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 quantile limits.
```python
>>> from cSparseIntegerOutrankingDigraphs import *
>>> sg = SparseIntegerOutrankingDigraph(t, quantiles=4)

*----- 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.

```
follows.

```python
>>> sg.showRelationMap(toIndex=75)
```
Fig. 7.1: Sparse quartiles-sorting decomposed outranking relation (extract).

Legend:
• outranking for certain (⊤)
• outranked for certain (⊥)
• more or less outranking (+)
• more or less outranked (-)
• indeterminate ()

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

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

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

```python
def relation(self, int x, int y):
    """
    *Parameters*:
    * x (int action key),
    * y (int action key).
    Dynamic construction of the global outranking characteristic function \( r(x S y) \).
    """
    cdef int Min, Med, Max, rx, ry
    Min = self.valuationdomain['min']
    Med = self.valuationdomain['med']
    Max = self.valuationdomain['max']
    if x == y:
        return Med
    cx = self.actions[x]['component']
    cy = self.actions[y]['component']
    #print(self.components)
    rx = self.components[cx]['rank']
    ry = self.components[cy]['rank']
    if rx == ry:
        try:
            rxpg = self.components[cx]['subGraph'].relation
            return rxpg[x][y]
        except AttributeError:
            componentRanking = self.components[cx]['componentRanking']
            if componentRanking.index(x) < componentRanking.index(y):
                return Max
            else:
                return Min
    elif rx > ry:
        return Min
    else:
        return Max
```
7.5 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.

```
>>> 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.

```
>>> 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'],
(continues on next page)
```
'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]}),
    ...,
    ...,
    ('c45', {'rank': 45,
             'lowQtileLimit': '<',
             'highQtileLimit': '0.25',
             'componentRanking': [79, 83, 84]}),
    ('c46', {'rank': 46,
             'lowQtileLimit': '<',
             'highQtileLimit': '0.25',
             'componentRanking': [52, 16, 86]}),
    ('c47', {'rank': 47,
             'lowQtileLimit': '<',
             'highQtileLimit': '0.25',
             'componentRanking': [79, 83, 84]}))

>>> print('%.0f kB % (total_size(qr)/1024)')
208kB
```

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

```python
>>> from cIntegerOutrankingDigraphs import *
>>> ig = IntegerBipolarOutrankingDigraph(t)
```
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.

7.6 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\(^7\), and
- on the 3TB-bigmem Gaia-183 node with 64 cores\(^8\),

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

\(^7\) See https://hpc.uni.lu/systems/iris/
\(^8\) See https://hpc.uni.lu/systems/gaia/
Example python session on the HPC-UL Iris-126 -skylake node

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

```python
>>> from cRandPerfTabs import
cRandom3ObjectivesPerformanceTableau as cR3ObjPT
>>> pt = cR3ObjPT(numberOfActions=1000000,
... numberofCriteria=21,
... weightDistribution='equiobjectives',
... commonScale = (0.0,1000.0),
... commonThresholds = [(2.5,0.0),(5.0,0.0),(75.0,0.0)],
... commonMode = ['beta','variable',None],
... missingDataProbability=0.05,
... seed=16)
>>> import cSparseIntegerOutrankingDigraphs as iBg
>>> qr = iBg.cQuantilesRankingDigraph(pt,quantiles=10,
... quantilesOrderingStrategy='optimal',
... minimalComponentSize=1,
... componentRankingRule='NetFlows',
... LowerClosed=False,
... Threading=True,
... tempDir='/tmp',
... nbrOfCPUs=28)
```

--- Object instance description ---------------------
Instance class : cQuantilesRankingDigraph
Instance name : random3ObjectivesPerfTab_mp
# Actions : 1000000
# Criteria : 21
Sorting by : 10-Tiling
Ordering strategy : optimal
Ranking rule : NetFlows
# Components : 233645
Minimal order : 1

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

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

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

Let us inspect the 21 marginal performances of the five best-ranked alternatives listed below.
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 on Generating random performance tableaux).

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])
>>> from cIntegerOutrankingDigraphs import *
>>> g = IntegerBipolarOutrankingDigraph(best10)
>>> g.valuationdomain
{'min': -1008, 'med': 0, 'max': 1008, 'hasIntegerValuation': True}
>>> g.showRelationTable(ReflexiveTerms=False)
* ---- Relation Table -----
  r(x>y) | #773909 #668947 #567308 #578560 #426464 #298061 #155874 #815552 ...
  ——|---------------------------------------------------------------
#773909 | - +390 +90 +270 -50 +340 +220 +60 ...
#668947 | +116 +222 ...
#567308 | ...
#578560 | ...
#426464 | ...
```

(continues on next page)
Three alternatives - #155874, #426464 and #567308- qualify as Condorcet winners, i.e. they each **positively outrank** all the other nine alternatives. No chordless outranking circuits are detected, yet the transitivity of the apparent outranking relation is not given. And, no clear ranking alignment hence appears when inspecting the **strict** outranking digraph (i.e. the codual \(~(-g)~\) of \(g\)) shown in Fig. 7.3.

```plaintext
>>> (~(-g)).exportGraphViz()
*---- exporting a dot file for GraphViz tools ---------*
Exporting to converse-dual_rel_best10.dot
```
Fig. 7.3: Validated strict outranking situations between the ten best-ranked alternatives

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

```
>>> g.computeCopelandRanking()
[426464, 567308, 155874, 279729, 773909, 298061, 668947, 815552, 928564, 578560]
```

```
>>> g.computeNetFlowsRanking()
[426464, 155874, 773909, 567308, 815552, 279729, 928564, 298061, 668947, 578560]
```

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

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

Back to *Content Table*
8 Computing a best choice recommendation

- What site to choose?
- Performance tableau
- Outranking digraph
- Rubis best choice recommendations
- Computing strict best choice recommendations
- Weakly ordering the outranking digraph

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

8.1 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.
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]).

In view of this performance tableau, what is now the office site we may recommend to the CEO as *best choice*?
8.2 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()

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

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

>>> t.showCriteria()

*---- criteria -----*
    C 'Costs'
    Scale = (Decimal('0.00'), Decimal('50000.00'))
    Weight = 0.333
    Threshold ind : 1000.00 + 0.00x ; percentile: 0.095
    Threshold pref : 2500.00 + 0.00x ; percentile: 0.143
    Cf 'Comfort'
    Scale = (Decimal('0.00'), Decimal('100.00'))
    Weight = 0.111
    Threshold ind : 10.00 + 0.00x ; percentile: 0.095
    Threshold pref : 20.00 + 0.00x ; percentile: 0.286
    Threshold veto : 80.00 + 0.00x ; percentile: 0.905
```

(continues on next page)
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.

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

### Heatmap of performance tableau officeChoice.xml

<table>
<thead>
<tr>
<th>criteria</th>
<th>C</th>
<th>W</th>
<th>V</th>
<th>St</th>
<th>Pr</th>
<th>P</th>
<th>Cf</th>
</tr>
</thead>
<tbody>
<tr>
<td>weights</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></td>
</tr>
<tr>
<td>A</td>
<td>-35000.00</td>
<td>75.00</td>
<td>60.00</td>
<td>100.00</td>
<td>100.00</td>
<td>90.00</td>
<td>0.00</td>
</tr>
<tr>
<td>B</td>
<td>-17800.00</td>
<td>30.00</td>
<td>80.00</td>
<td>10.00</td>
<td>20.00</td>
<td>30.00</td>
<td>100.00</td>
</tr>
<tr>
<td>C</td>
<td>-6700.00</td>
<td>0.00</td>
<td>70.00</td>
<td>0.00</td>
<td>80.00</td>
<td>100.00</td>
<td>10.00</td>
</tr>
<tr>
<td>D</td>
<td>-14100.00</td>
<td>55.00</td>
<td>50.00</td>
<td>30.00</td>
<td>70.00</td>
<td>90.00</td>
<td>30.00</td>
</tr>
<tr>
<td>E</td>
<td>-34800.00</td>
<td>100.00</td>
<td>60.00</td>
<td>90.00</td>
<td>40.00</td>
<td>70.00</td>
<td>60.00</td>
</tr>
<tr>
<td>F</td>
<td>-18600.00</td>
<td>0.00</td>
<td>0.00</td>
<td>70.00</td>
<td>0.00</td>
<td>0.00</td>
<td>80.00</td>
</tr>
<tr>
<td>G</td>
<td>-12000.00</td>
<td>50.00</td>
<td>100.00</td>
<td>20.00</td>
<td>60.00</td>
<td>80.00</td>
<td>50.00</td>
</tr>
</tbody>
</table>

*Color legend:*

- quantile 0.2
- quantile 0.4
- quantile 0.6
- quantile 0.8
- quantile 1.0

**Fig. 8.1:** Heat map 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.
8.3 Outranking digraph

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

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

The credibility of each pairwise outranking situation (see [BIS-2013]), denoted $r(x \ S \ y)$, is measured in a bipolar significance valuation $[-100.00, 100.00]$, 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]).

![Valued Adjacency Matrix](image)

**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>0.00</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. 8.2: The office choice outranking digraph

For computing such a bipolar-valued outranking digraph from the given performance tableau $t$, we use the `BipolarOutrankingDigraph` constructor from the `outrankingDigraphs` module. The `Digraph.showHTMLRelationTable` method shows here the resulting bipolar-valued adjacency matrix in a system browser window (see Fig. 8.2).

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

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**.

```python
>>> 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. 8.3).

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

Fig. 8.3: The office choice outranking digraph

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

```python
>>> g.computeChordlessCircuits()
[]
```
```bash
>>> g.showChordlessCircuits()
No circuits observed in this digraph.
*---- Chordless circuits -----*
0 circuits.
```
8.4 Rubis best choice recommendations

Following the Rubis outranking method (see [BIS-2008]), potential best choice recommendations are determined by the outranking prekernels—weakly independent and strictly outranking choices—of the outranking digraph (see the tutorial on *On computing digraph kernels*). The case given, we previously need to break open all chordless circuits at their weakest link. As we observe no such chordless circuits here, we may directly compute the prekernels of \( g \).

```plaintext
>>> g.showPreKernels()
*--- Computing preKernels ---*
Dominant preKernels :

[D]

independence : 100.0
dominance : 11.111
absorbency : -100.0
covering : 1.000

[B, E, C]

independence : 0.0
dominance : 11.111
absorbency : -100.0
covering : 0.500

[A, G]

independence : 0.0
dominance : 55.556
absorbency : 0.0
covering : 0.700

Absorbent preKernels :

[F, A]

independence : 0.0
dominance : 0.0
absorbency : 100.0
covering : 0.700

*----- statistics -----*
graph name: rel_officeChoice.xml
number of solutions
dominant kernels : 3
absorbent kernels: 1
cardinality frequency distributions
cardinality : [0, 1, 2, 3, 4, 5, 6, 7]
dominant kernel : [0, 1, 1, 1, 0, 0, 0, 0]
absorbent kernel: [0, 0, 1, 0, 0, 0, 0, 0]
Execution time : 0.00018 sec.
Results in sets: dompreKernels and abspreKernels.
```

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

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

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

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

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

To model these strict outranking situations, we may compute the Rubis best choice recommendation on the codual, the converse (\(\sim\)) of the dual (\(-\))\textsuperscript{14}, of the outranking digraph instance $g$ (see \cite{bis-2013}), as follows.

```
>>> g.showBestChoiceRecommendation(CoDual=True,ChoiceVector=True)
* --- Best and worst 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']
```
It is interesting to notice that the strict best choice recommendation consists in the set of weak Condorcet winners: ‘A’, ‘C’ and ‘D’ (see Line 6). In the corresponding characteristic vector (see Line 14-15), representing the bipolar credibility degree with which each alternative may indeed be considered a best choice (see [BIS-2006a], [BIS-2006b]), we find confirmed that alternative D is the only positively validated one, whereas both extreme alternatives - A (the most expensive) and C (the cheapest) - stay in an indeterminate situation. They may be potential best choice candidates besides D. Notice furthermore that compromise alternative G, while not actually included in 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 of alternative A.

### 8.6 Weakly ordering the outranking digraph

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

```python
>>> from transitiveDigraphs import RankingByChoosingDigraph
>>> gcd = ~(-g)
>>> rbc = RankingByChoosingDigraph(gcd)
Threading ... # multiprocessing if 2 cores are available
Exiting computing threads
>>> rbc.showRankingByChoosing()
```
Ranking by Choosing and Rejecting

1st ranked ['D'] (0.28)

2nd ranked ['C', 'G'] (0.17)

2nd last ranked ['B', 'C', 'E'] (0.22)

1st last ranked ['A', 'F'] (0.50)

```python
>>> 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. 8.4: Ranking-by-choosing from the office choice outranking digraph

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

The best choice recommendation appears hence depending on the very importance the CEO is attaching to each of the three 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?

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

Back to Content Table

9 Rating with learned quantile norms

- Introduction
- Incremental learning of historical performance quantiles
- Rating new performances with quantile norms

9.1 Introduction

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

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

<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.
The performances on the $c_2$ criterion are again measured on a cardinal negative scale from -100.00 (worst) to 0.0 (best).

The importance (sum of weights) of the $c_2$ criteria is equal to the importance (sum of weights) of the $b_2$ criteria taken all together.

The non trivial decision problem we now face here, is to decide, how the multiple criteria performances of $a_{1001}$, respectively $a_{1010}$, 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.

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

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

- Ordered **objectives** and a **criteria** dictionaries from a valid performance tableau instance;
- A list **quantileFrequencies** of quantile frequencies like quartiles [0.0, 0.25, 05, 0.75,1.0], quintiles [0.0, 0.2, 0.4, 0.6, 0.8, 1.0] or deciles [0.0, 0.1, 0.2, ... 1.0] for instance;
- An ordered dictionary **limitingQuan tiles** 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 $a_{1001}$ and $a_{1010}$ above.

```python
>>> from performanceQuantiles import PerformanceQuantiles
>>> from randomPerfTabs import RandomCBPerformanceTableau
>>> nbrAction=900
>>> nbrCrit = 7
```
>>> seed = 100
>>> tp = RandomCBPerformanceTableau(numberOfActions=nbrActions,\
...       numberOfCriteria=nbrCrit,seed=seed)
>>> pq = PerformanceQuantiles(tp,\
...       numberOfBins = 'quartiles',\
...       LowerClosed=True)
>>> pq.__dict__.keys()
{'objectives', 'LowerClosed', 'name', 'quantilesFrequencies', 'criteria', 'historySizes', 'limitingQuantiles', ...

The performanceQuantiles.PerformanceQuantiles class parameter `numberOfBins` (see Line 9 above), choosing the wished number of quantile frequencies, may be either quartiles (4 bins), quintiles (5 bins), deciles (10 bins), dodeciles (20 bins) or any other integer number of quantile bins. The quantile bins may be either lower closed (default) or upper-closed.

>>> # Printing out the estimated quartile limits
>>> pq.showLimitingQuantiles(ByObjectives=True)
-- Historical performance quantiles ----

Costs
---
criteria | weights | '0.00' '0.25' '0.50' '0.75' '1.00'
------ | ------- | -------------------------------
'c1' | 5 | -10 -7 -5 -3 0
'c2' | 5 | -96.37 -70.65 -50.10 -30.00 -1.43

Benefits
---
criteria | weights | '0.00' '0.25' '0.50' '0.75' '1.00'
------ | ------- | -------------------------------
'b1' | 2 | 1.99 29.82 49.44 70.73 99.83
'b2' | 2 | 0 3 5 7 10
'b3' | 2 | 0 3 5 7 10
'b4' | 2 | 3.27 30.10 50.82 70.89 98.05
'b5' | 2 | 0.85 29.08 48.55 69.98 97.56

Both objectives are equi-important; the sum of weights (10) of the costs criteria balance the sum of weights (10) of the benefits criteria (see column 2). The preference direction of the costs criteria `c1` and `c2` is negative; the lesser the costs the better it is, whereas all the benefits criteria `b1` to `b5` show positive preference directions, i.e. the higher the benefits the better it is. The columns entitled ‘0.0’, resp. ‘1.0’ show the quartile Q0, resp. Q4, i.e. the worst, resp. best performance observed so far on each criterion. Column ‘0.5’ shows the median (Q2) observed on the criteria.

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

```python
>>> # 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` 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. 9.1).

```python
>>> # 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>58.58</td>
<td>72.00</td>
<td>98.05</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. 9.1: Showing the updated quartiles limits
9.3 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 *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')
>>> print(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

(continues on next page)
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
---------|-------------------------------------------------------------
 b1'    |  37.0  27.0  24.0  16.0  42.0  33.0  39.0  64.0  42.0  32.0
 b2'    |   2.0   5.0   8.0   3.0   3.0   2.0   6.0   5.0   4.0   9.0
 b3'    |   2.0   4.0   2.0   1.0   6.0   3.0   2.0   6.0   6.0   6.0
 b4'    |  61.0  54.0  74.0  25.0  28.0  20.0  20.0  49.0  44.0  55.0
 b5'    |  31.0  63.0  61.0  48.0  30.0  39.0  16.0  96.0  57.0  51.0
 c1'    | -4.0  -6.0  -8.0  -5.0  -1.0  -5.0  -1.0  -6.0  -6.0  -4.0
 c2'    | -40.0 -23.0 -37.0 -37.0 -37.0 -24.0 -27.0 -73.0 -43.0 -94.0 -35.0
```

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: \( m_1 = [0.0-[, \ m_2 = [0.25-[, \ m_3 = [0.5-[, \ m_4 = [0.75-[, \)

```python
>>> nqr.showPerformanceTableau(actionsSubset=nqr.profiles)
*---- Quartiles limit profiles -----*
criteria | 'm1' 'm2' 'm3' 'm4'
---------|----------------------------
 b1'    | 2.0 28.8 49.6 75.3
 b2'    | 0.0  2.9  4.9  6.7
 b3'    | 0.0  2.9  4.9  8.0
 b4'    | 3.3 35.9 58.6 72.0
 b5'    | 0.8 32.8 48.1 69.7
 c1'    | -10.0 -7.4 -5.4 -3.4
 c2'    | -96.4 -72.2 -52.3 -34.0
```

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, i.e. many new decision alternatives and centile classes for instance, a multi-threading version may be used when multiple processing cores are available (see the technical description of the `sortingDigraphs.NormedQuantilesRatingDigraph` class).
The actual rating procedure will rely on a complete ranking of the new decision alternatives as well as the quantile class limits obtained from the corresponding bipolar-valued outranking digraph. Two efficient and scalable ranking rules, the Copeland and its valued version, the Netflows rule may be used for this purpose. The rankingRule parameter allows to choose one of both. With rankingRule='best' (see Line 2 above) the NormedQuantilesRatingDigraph constructor will choose the ranking rule that results in the highest ordinal correlation with the given outranking relation (see [BIS-2012]).

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

>>> print('Ranking rule :', nqr.rankingRule)
Ranking rule : NetFlows
>>> print('Actions ranking :', nqr.actionsRanking)
Actions ranking :
[ 'm4', 'a1005', 'a1010', 'a1008', 'a1002', 'a1006',
 'm3', 'a1003', 'a1001', 'a1007', 'a1004', 'a1009',
 'm2', 'm1']
>>> print('Ranking correlation : %+.2f' %
... (nqr.rankingCorrelation['correlation'] ))
Ranking correlation : +0.938

We achieve here a linear ranking without ties (from best to worst) of the digraph’s actions, i.e. including the new decision alternatives as well as the quartile limits m1 to m4, which is very close in an ordinal sense (tau = 0.94) 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.

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

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:

>>> 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 heat map format (see perfTabs:PerformanceTableau. showHTMLPerformanceHeatmap() method (see Fig. 9.2).

>>> 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>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><strong>weights</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>tau</strong>(*)</td>
<td>0.62</td>
<td>0.56</td>
<td>0.43</td>
<td>0.42</td>
<td>0.41</td>
<td>0.37</td>
<td>0.29</td>
</tr>
<tr>
<td>[0.75 -</td>
<td>8.01</td>
<td>33.99</td>
<td>6.72</td>
<td>75.27</td>
<td>-3.38</td>
<td>69.75</td>
<td>72.00</td>
</tr>
<tr>
<td>a1005c</td>
<td>6.00</td>
<td>-24.00</td>
<td>3.00</td>
<td>42.00</td>
<td>-1.00</td>
<td>30.00</td>
<td>28.00</td>
</tr>
<tr>
<td>a1010n</td>
<td>6.00</td>
<td>-35.00</td>
<td>9.00</td>
<td>32.00</td>
<td>-4.00</td>
<td>51.00</td>
<td>55.00</td>
</tr>
<tr>
<td>a1008n</td>
<td>6.00</td>
<td>-43.00</td>
<td>5.00</td>
<td>64.00</td>
<td>-6.00</td>
<td>96.00</td>
<td>49.00</td>
</tr>
<tr>
<td>a1002c</td>
<td>4.00</td>
<td>-23.00</td>
<td>5.00</td>
<td>27.00</td>
<td>-6.00</td>
<td>63.00</td>
<td>54.00</td>
</tr>
<tr>
<td>a1006c</td>
<td>3.00</td>
<td>-27.00</td>
<td>3.00</td>
<td>33.00</td>
<td>-5.00</td>
<td>39.00</td>
<td>20.00</td>
</tr>
<tr>
<td>[0.50 -</td>
<td>4.86</td>
<td>-52.27</td>
<td>4.92</td>
<td>49.63</td>
<td>-5.39</td>
<td>48.09</td>
<td>58.58</td>
</tr>
<tr>
<td>a1003a</td>
<td>2.00</td>
<td>-37.00</td>
<td>8.00</td>
<td>24.00</td>
<td>-8.00</td>
<td>61.00</td>
<td>74.00</td>
</tr>
<tr>
<td>a1001c</td>
<td>2.00</td>
<td>-40.00</td>
<td>2.00</td>
<td>37.00</td>
<td>-4.00</td>
<td>31.00</td>
<td>61.00</td>
</tr>
<tr>
<td>a1007c</td>
<td>2.00</td>
<td>-73.00</td>
<td>6.00</td>
<td>39.00</td>
<td>-1.00</td>
<td>16.00</td>
<td>20.00</td>
</tr>
<tr>
<td>a1004c</td>
<td>1.00</td>
<td>-37.00</td>
<td>3.00</td>
<td>16.00</td>
<td>-5.00</td>
<td>48.00</td>
<td>25.00</td>
</tr>
<tr>
<td>a1009n</td>
<td>6.00</td>
<td>-94.00</td>
<td>4.00</td>
<td>42.00</td>
<td>-6.00</td>
<td>57.00</td>
<td>44.00</td>
</tr>
<tr>
<td>[0.25 -</td>
<td>2.90</td>
<td>-72.22</td>
<td>2.94</td>
<td>28.77</td>
<td>-7.35</td>
<td>32.84</td>
<td>35.91</td>
</tr>
<tr>
<td>[0.00 -</td>
<td>0.00</td>
<td>-96.37</td>
<td>0.00</td>
<td>1.99</td>
<td>-10.00</td>
<td>0.85</td>
<td>3.27</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.

Fig. 9.2: Heat map of normed quartiles ranking

Using furthermore a specialised version of the transitiveDigraph. `TransitiveDigraph.exportGraphViz()` method allows drawing the same rating result in a Hasse diagram format (see Fig. 9.3).

```python
>>> nqr.exportRatingGraphViz('normedRatingDigraph')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to normedRatingDigraph.dot
  dot -Grankdir=TB -Tpng normedRatingDigraph.dot -o normedRatingDigraph.png
```
We may now answer the normed rating decision problem stated at the beginning. Decision alternative $a_{1001}$ is rated in quartile Q2 and alternative $a_{1010}$ in quartile Q3 (see Table below). Indeed, the performances of decision alternative $a_{1001}$ were generated with a triangular law at a low mode, i.e. low costs but also low benefits, whereas the performances of alternative $a_{1010}$ were generated with a median mode.
A more precise rating result may be achieved when we use *deciles* instead of *quartiles* for estimating the historical cumulative distribution functions.

```python
>>> pq1 = PerformanceQuantiles(tp, numberOfBins = 'deciles',
...                            LowerClosed=True)
>>> nqr1 = NormedQuantilesRatingDigraph(pq1,newActions,rankingRule='best')
>>> nqr1.showQuantilesRating()

*-------- Deciles rating result ---------
[0.60 - 0.70[ ['a1005', 'a1010', 'a1008', 'a1002']
[0.50 - 0.60[ ['a1006', 'a1001', 'a1003']
[0.40 - 0.50[ ['a1007', 'a1004']
[0.30 - 0.40[ ['a1009']
```

Compared with the quartiles rating result, we notice 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 [0.25-0.50] 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. 9.4).

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

<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>5</td>
<td>5</td>
<td></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>
Fig. 9.4: Heat map of normed deciles rating

<table>
<thead>
<tr>
<th>criteria</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>weights</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>tau(*)</td>
<td>0.67</td>
<td>0.65</td>
<td>0.58</td>
<td>0.57</td>
<td>0.53</td>
<td>0.53</td>
<td>0.48</td>
</tr>
<tr>
<td>[0.90]</td>
<td>-20.32</td>
<td>7.73</td>
<td>-2.53</td>
<td>86.83</td>
<td>82.16</td>
<td>7.66</td>
<td>82.04</td>
</tr>
<tr>
<td>[0.80]</td>
<td>-29.70</td>
<td>7.26</td>
<td>-3.35</td>
<td>79.30</td>
<td>75.15</td>
<td>6.64</td>
<td>74.66</td>
</tr>
<tr>
<td>[0.70]</td>
<td>-37.97</td>
<td>6.67</td>
<td>-4.14</td>
<td>70.95</td>
<td>60.20</td>
<td>5.88</td>
<td>69.76</td>
</tr>
<tr>
<td>a1005c</td>
<td>-24.00</td>
<td>6.00</td>
<td>-1.00</td>
<td>42.00</td>
<td>30.00</td>
<td>3.00</td>
<td>28.00</td>
</tr>
<tr>
<td>a1010n</td>
<td>-35.00</td>
<td>6.00</td>
<td>-4.00</td>
<td>32.00</td>
<td>51.00</td>
<td>9.00</td>
<td>55.00</td>
</tr>
<tr>
<td>a1008n</td>
<td>-43.00</td>
<td>6.00</td>
<td>-6.00</td>
<td>64.00</td>
<td>96.00</td>
<td>5.00</td>
<td>49.00</td>
</tr>
<tr>
<td>a1002c</td>
<td>-23.00</td>
<td>4.00</td>
<td>-6.00</td>
<td>27.00</td>
<td>63.00</td>
<td>5.00</td>
<td>54.00</td>
</tr>
<tr>
<td>[0.60]</td>
<td>-44.23</td>
<td>3.92</td>
<td>-5.04</td>
<td>60.56</td>
<td>56.01</td>
<td>5.37</td>
<td>62.23</td>
</tr>
<tr>
<td>a1006c</td>
<td>-27.00</td>
<td>3.00</td>
<td>-5.00</td>
<td>33.00</td>
<td>39.00</td>
<td>3.00</td>
<td>20.00</td>
</tr>
<tr>
<td>a1001c</td>
<td>-40.00</td>
<td>2.00</td>
<td>-4.00</td>
<td>37.00</td>
<td>31.00</td>
<td>2.00</td>
<td>61.00</td>
</tr>
<tr>
<td>a1003a</td>
<td>-37.00</td>
<td>2.00</td>
<td>-8.00</td>
<td>24.00</td>
<td>61.00</td>
<td>8.00</td>
<td>74.00</td>
</tr>
<tr>
<td>[0.50]</td>
<td>-52.22</td>
<td>4.64</td>
<td>-6.02</td>
<td>49.56</td>
<td>48.07</td>
<td>4.83</td>
<td>58.45</td>
</tr>
<tr>
<td>a1007c</td>
<td>-73.00</td>
<td>2.00</td>
<td>-1.00</td>
<td>39.00</td>
<td>16.00</td>
<td>6.00</td>
<td>20.00</td>
</tr>
<tr>
<td>a1004c</td>
<td>-37.00</td>
<td>1.00</td>
<td>-5.00</td>
<td>48.00</td>
<td>3.00</td>
<td>25.00</td>
<td></td>
</tr>
<tr>
<td>[0.40]</td>
<td>-60.50</td>
<td>3.84</td>
<td>-6.69</td>
<td>39.61</td>
<td>40.16</td>
<td>4.25</td>
<td>49.82</td>
</tr>
<tr>
<td>a1009n</td>
<td>-94.00</td>
<td>6.00</td>
<td>-6.00</td>
<td>42.00</td>
<td>57.00</td>
<td>4.00</td>
<td>44.00</td>
</tr>
<tr>
<td>[0.30]</td>
<td>-67.14</td>
<td>3.12</td>
<td>-7.32</td>
<td>30.85</td>
<td>34.33</td>
<td>3.30</td>
<td>40.89</td>
</tr>
<tr>
<td>[0.20]</td>
<td>-77.07</td>
<td>2.55</td>
<td>-7.94</td>
<td>23.84</td>
<td>29.57</td>
<td>2.27</td>
<td>30.45</td>
</tr>
<tr>
<td>[0.10]</td>
<td>-83.04</td>
<td>1.99</td>
<td>-8.48</td>
<td>16.64</td>
<td>16.91</td>
<td>1.58</td>
<td>24.78</td>
</tr>
<tr>
<td>[0.00]</td>
<td>-96.37</td>
<td>0.00</td>
<td>10.00</td>
<td>9.99</td>
<td>9.85</td>
<td>0.00</td>
<td>3.27</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.

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 *deciles* 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.

Back to *Content Table*
10 Working with the graphs module

- Structure of a Graph object
- q-coloring of a graph
- MIS and clique enumeration
- Line graphs and maximal matchings
- Grids and the Ising model
- Simulating Metropolis random walks

See also the technical documentation of the graphs-label.

10.1 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
```
{'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 \texttt{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
```

![Graphs Python module (graphviz). R. Bisdorf, 2011](image)

Fig. 10.1: Tutorial graph instance

Properties, like the gamma function and vertex degrees and neighbourhood depths may be shown with a \texttt{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']
```

(continues on next page)
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 ----- 
   | 'v1' 'v2' 'v3' 'v4' 'v5' 'v6' 'v7'
------|------------------------------------------
'v1'  | - -1 -1 -1 1 -1 -1
'v2'  | -1 -1 1 1 -1 1 -1
'v3'  | -1 1 -1 -1 -1 -1 -1
'v4'  | -1 1 -1 -1 1 -1 -1
'v5'  | 1 -1 -1 1 -1 -1 -1
'v6'  | -1 1 -1 -1 1 -1 -1
'v7'  | -1 -1 -1 1 -1 -1 -1

>>> g1 = dg.digraph2Graph()
>>> g1.showShort()
*---- short description of the graph ----*
Name : 'tutorialGraph'
Vertices : ['v1', 'v2', 'v3', 'v4', 'v5', 'v6', 'v7']
Valuation domain : {'med': 0, 'min': -1, 'max': 1}
Gamma function :
  v1 -> ['v5']
  v2 -> ['v3', 'v6', 'v4']
  v3 -> ['v2']
  v4 -> ['v5', 'v7', 'v2']
  v5 -> ['v6', 'v1', 'v4']
  v6 -> ['v5', 'v2']
  v7 -> ['v4']
degrees : [0, 1, 2, 3, 4, 5, 6]
distribution : [0, 3, 1, 3, 0, 0, 0]
nbh depths : [0, 1, 2, 3, 4, 5, 6, 'inf. ']
distribution : [0, 0, 1, 4, 2, 0, 0, 0]
```
### 10.2 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.

```python
>>> from graphs import Q_Coloring
>>> qc = Q_Coloring(g)
Running a Gibbs Sampler for 42 steps!
The q-coloring with 3 colors is feasible!!
>>> qc.showConfiguration()
```

- $v_5$ lightblue
- $v_3$ gold
- $v_7$ gold
- $v_2$ lightblue
- $v_4$ lightcoral
- $v_1$ gold
- $v_6$ lightcoral

```python
>>> qc.exportGraphViz('tutorial-3-coloring')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to tutorial-3-coloring.dot
```

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

---

**Fig. 10.2: 3-Coloring of the tutorial graph**

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

```python
>>> qc = Q_Coloring(g, colors=['gold', 'coral'])
Running a Gibbs Sampler for 42 steps!
The q-coloring with 2 colors is feasible!!
>>> qc.showConfiguration()
```

- $v_5$ gold
- $v_3$ coral
- $v_7$ gold


10.3 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 Graph.showMIS() method.

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

Fig. 10.3: 2-coloring of the tutorial graph
A MIS in the dual of a graph instance $g$ (its negation $-g^{14}$), corresponds to a maximal clique, i.e. a maximal complete subgraph in $g$. Maximal cliques may be directly enumerated with the `graphs.Graph.showCliques()` method.

```python
>>> g.showCliques()

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

```python
>>> g.cliques

[frozenseq(('v2', 'v3')), frozenseq(('v4', 'v7')),
frozenseq(('v2', 'v4'))],
[frozenseq(('v1', 'v5'))],
frozenseq(('v6', 'v5'))]
```

### 10.4 Line graphs and maximal matchings

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

```python
>>> g = CycleGraph(order=5)
>>> g

*------- Graph instance description ------*
Instance class : CycleGraph
Instance name : cycleGraph
Graph Order : 5
Graph Size : 5
Valuation domain : [-1.00; 1.00]
Attributes : ['name', 'order', 'vertices', 'valuationDomain', 'edges', 'size', 'gamma']
(continues on next page)
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.

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

Fig. 10.4: A perfect maximum matching of the 8-cycle graph
10.5 Grids and the Ising model

Special classes of graphs, like \( n \times m \) rectangular or triangular grids (\texttt{graphs.GridGraph} and \texttt{graphsISINGModel}) are available in the \texttt{graphs} module. For instance, we may use a Gibbs sampler again for simulating an Ising Model on such a grid.

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

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

```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']
```
(continues on next page)
v3 -> ['v5', 'v1']
v4 -> ['v2', 'v5', 'v1']
v5 -> ['v3', 'v4']

>>> probs = {}  # initialize a potential stationary probability vector
>>> n = g.order  # for instance: probs[v_i] = n-i/Sum(1:n) for i in 1:n
>>> i = 0

>>> verticesList = [x for x in g.vertices]
>>> verticesList.sort()

>>> for v in verticesList:
...    probs[v] = (n - i)/(n*(n+1)/2)
...    i += 1

>>> met = MetropolisChain(g,probs)

>>> frequency = met.checkSampling(verticesList[0],nSim=30000)

>>> for v in verticesList:
...    print(v,probs[v],frequency[v])

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

>>> met.showTransitionMatrix()

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].

Back to **Content Table**
11 Computing the non isomorphic MISs of the 12-cycle graph

- Introduction
- Computing the maximal independent sets (MISs)
- Computing the automorphism group
- Computing the isomorphic MISs

11.1 Introduction

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

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

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)
```
11.2 Computing the maximal independent sets (MISs)

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

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

*** Maximal independent choices ***
number of solutions: 29

cardinality distribution
  card.: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
  freq.: [0, 0, 0, 0, 3, 24, 2, 0, 0, 0, 0, 0, 0]

Results in c12.misset
```

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

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

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

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

\(^3\) The `perrinMIS` shell command may be installed system wide with the command `.../Digraph3$ make installPerrin` from the main Digraph3 directory. It is stored by default into `<usr/local/bin/>`. This may be changed with the `INSTALLDIR` flag. The command `.../Digraph3$ make installPerrinUser` installs it instead without sudo into the user’s private `<$Home/.bin>` directory.
11.3 Computing the automorphism group

For computing the corresponding non isomorphic MISs, we actually need the automorphism group of the c12-cycle graph. The `digraphs.Digraph.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()
```

```python
Permutations

{1': '1', '2': '12', '3': '11', '4': '10', '5':
     '9', '6': '8', '7': '7', '8': '6', '9': '5', '10':
     '4', '11': '3', '12': '2'}
{1': '2', '2': '1', '3': '12', '4': '11', '5': '10',
     '6': '9', '7': '8', '8': '7', '9': '6', '10': '5',
     '11': '4', '12': '3'}
```

```python
>>> print('grpsize = ', c12.automorphismGroupSize)
groups = 24
```

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

11.4 Computing the isomorphic MISs

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

```python
>>> c12.showOrbits(c12.misset, withListing=False)
```

```python
*---- Global result ----
Number of MIS: 29
```

---

2 Dependency: The `digraphs.Digraph.automorphismGenerators()` method uses the shell `dreadnaut` command from the `nauty` software package. See [https://www3.cs.stonybrook.edu/~algorith/implement/nauty/implement.shtml](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`. 

114
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. 11.2 below.

Fig. 11.2: The symmetry axes of the four non isomorphic MISs of the 12-cycle graph

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

Back to Content Table
12 On computing digraph kernels

- What is a graph kernel?
- Initial and terminal kernels
- Kernels in lateralized digraphs
- Computing good and bad choice recommendations
- Tractability

12.1 What is a graph kernel?

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

A first trivial example is immediately given by the maximal independent vertices sets (MISs) of the n-cycle graph (see Computing the non isomorphic MISs of the 12-cycle graph). Indeed, each MIS in the n-cycle graph is by definition independent, i.e. internally stable, and each non selected vertex in the n-cycle graph is in relation with either one or even two members of the MIS. See, for instance, the four non isomorphic MISs of the 12-cycle graph as shown in Fig. 11.2.

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

We may illustrate this coincidence between MISs and kernels in graphs and symmetric digraphs with the following random 3-regular graph instance (see Fig. 12.1).

```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
```
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. 12.2 above, that the computed MIS renders indeed a valid
kernel of the given graph. The complete set of kernels of this 3-regular graph instance
coincides hence with the set of its MISs.

```python
>>> g.showMIS()
*--- Maximal Independent Sets ---*
['a01', 'a02', 'a03', 'a07']
['a01', 'a04', 'a05', 'a08']
['a04', 'a05', 'a08', 'a09']
['a01', 'a04', 'a05', 'a10']
['a04', 'a05', 'a09', 'a10']
['a02', 'a03', 'a07', 'a12']
['a01', 'a03', 'a07', 'a11']
['a05', 'a08', 'a09', 'a11']
['a03', 'a07', 'a11', 'a12']
['a07', 'a09', 'a11', 'a12']
['a08', 'a09', 'a11', 'a12']
['a04', 'a05', 'a06', 'a08']
['a04', 'a05', 'a06', 'a10']
['a02', 'a04', 'a06', 'a10']
['a02', 'a03', 'a06', 'a12']
['a02', 'a06', 'a10', 'a12']
['a01', 'a02', 'a04', 'a07', 'a10']
['a02', 'a04', 'a07', 'a09', 'a10']
['a02', 'a07', 'a09', 'a10', 'a12']
['a01', 'a03', 'a05', 'a08', 'a11']
['a03', 'a05', 'a06', 'a08', 'a11']
['a03', 'a06', 'a08', 'a11', 'a12']

number of solutions: 22
cardinality distribution

card.: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
freq.: [0, 0, 0, 0, 16, 6, 0, 0, 0, 0, 0, 0, 0]
execution time: 0.00045 sec.

Results in self.misset

>>> g.misset
frozenset({'a02', 'a01', 'a07', 'a03'}),
frozenset({'a04', 'a01', 'a08', 'a05'}),
frozenset({'a09', 'a04', 'a08', 'a05'}),
...

frozenset({'a06', 'a02', 'a12', 'a10'}),
frozenset({'a06', 'a11', 'a08', 'a03', 'a05'}),
frozenset({'a03', 'a06', 'a11', 'a12', 'a08'})
```

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

```python
>>> dg = g.toDigraph()
>>> dg.automorphismGenerators()
```

(continues on next page)
*----- saving digraph in nauty dre format -----*

Actions index:

```python
1 : a01
2 : a02
3 : a03
4 : a04
5 : a05
6 : a06
7 : a07
8 : a08
9 : a09
10 : a10
11 : a11
12 : a12
```

```python
{'1': 'a01', '2': 'a02', '3': 'a03', '4': 'a04', '5': 'a05',
 '6': 'a06', '7': 'a07', '8': 'a08', '9': 'a09', '10': 'a10',
 '11': 'a11', '12': 'a12'}
```

# automorphisms extraction from dre file #

```bash
echo '
```

```bash
# permutation = 1['1', '11', '7', '5', '4', '9', '3', '10', '6', '8', '2', '12
```

```bash
-->
```
```bash

```bash
>>> dg.showOrbits(g.misset)
```

*-- Isomorphic reduction of choices

```python
... current representative: frozenset({'a09', 'a11', 'a12', 'a08'})
```

```python
length : 4
```

```python
number of isomorph choices 2
```

```python
isomorph choices
```

```python
['a06', 'a02', 'a12', 'a10'] # <= the random MIS shown above
```

```python
['a09', 'a11', 'a12', 'a08']
```

*----------------------*

*----- Global result ----

Number of choices: 22
Number of orbits : 11

Labelled representatives:

```python
['a06', 'a04', 'a10', 'a05']
['a09', 'a07', 'a10', 'a04', 'a02']
['a06', 'a11', 'a12', 'a08', 'a03']
['a04', 'a01', 'a10', 'a05']
['a07', 'a02', 'a12', 'a03']
['a09', 'a11', 'a12', 'a07']
['a06', 'a04', 'a08', 'a05']
['a06', 'a04', 'a02', 'a10']
['a01', 'a11', 'a07', 'a03']
['a01', 'a11', 'a08', 'a03', 'a05']
['a09', 'a11', 'a12', 'a08']
```

```python
Symmetry vector
```

```python
stabilizer size : [1, 2]
```

(continued on next page)
In our random 3-regular graph instance (see Fig. 12.1), we may thus find eleven non isomorphic kernels with orbit sizes equal to two. We illustrate below the isomorphic twin of the random MIS example shown in Fig. 12.2.

Fig. 12.3: Two isomorphic kernels of the random 3-regular graph instance

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

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

```python
>>> from graphs import CycleGraph
>>> c8 = CycleGraph(order=8)
>>> maxMatching = c8.computeMaximumMatching()
>>> c8.exportGraphViz(fileName='maxMatchingcycleGraph',
    matching=maxMatching)
... exporting a dot file for GraphViz tools ---------*
Exporting to maxMatchingcyleGraph.dot
Matching: {frozenset({‘v1’, ‘v2’}), frozenset({‘v5’, ‘v6’}),
    frozenset({‘v3’, ‘v4’}), frozenset({‘v7’, ‘v8’}) }
circo -Tpng maxMatchingcyleGraph.dot -o maxMatchingcyleGraph.png
```
In the context of digraphs, i.e. oriented graphs, the kernel concept gets much richer and separates from the symmetric MIS concept.

12.2 Initial and terminal kernels

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

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

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

A both internally and dominant, resp. absorbent stable choice is called a dominant or initial, resp. an absorbent or terminal kernel. From a topological perspective, the initial kernel concept looks from the outside of the digraph into its interior, whereas the terminal kernel looks from the interior of a digraph toward its outside. From an algebraic perspective, the initial kernel is a prefix operand, and the terminal kernel is a postfix operand in the Berge kernel equation systems.

Furthermore, as the kernel concept involves conjointly a positive logical refutation (the internal stability) and a positive logical affirmation (the external stability), it appeared rather quickly necessary in our operational developments to adopt a bipolar characteristic [-1,1] valuation domain, modelling negation by change of numerical sign and including explicitly a third median logical value (0) expressing logical indeterminateness (neither positive, nor negative, see [BIS-2000] and [BIS-2004]).
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 ‘laternity’, 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]
Execution time : 0.00004 sec.
Results in sets: dompreKernels and abspreKernels.
```

In a complete digraph, each single node is indeed both an initial and a terminal kernel candidate and there is no definite begin or end of the digraph to be detected. Laternity 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.
>>> ed = EmptyDigraph(order=5)
>>> ed.showPreKernels()
*--- Computing preKernels ---*
Dominant kernel :
['1', '2', '3', '4', '5']
  independence : 1.0
  dominance     : 1.0
  absorbency    : 1.0
Absorbent kernel :
['1', '2', '3', '4', '5']
  independence : 1.0
  dominance     : 1.0
  absorbency    : 1.0

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.

>>> from digraphs import IndeterminateDigraph
>>> id = IndeterminateDigraph(order=5)
>>> id.showPreKernels()
*--- Computing preKernels ---*
Dominant prekernel :
['1', '2', '3', '4', '5']
  independence : 0.0 # <== indeterminate
  dominance     : 1.0
  absorbency    : 1.0
Absorbent prekernel :
['1', '2', '3', '4', '5']
  independence : 0.0 # <== indeterminate
  dominance     : 1.0
  absorbency    : 1.0

Both these results make sense, as in a completely empty or indeterminate digraph, there is no interior of the digraph defined, only a border which is hence at the same time an initial and terminal 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, i.e. holes surrounded by an oriented cycle -a circuit- of odd length. They do not admit in fact any initial or terminal kernel.

>>> from digraphs import CirculantDigraph
>>> c5 = CirculantDigraph(order=5,circulants=[1])
>>> c5.showPreKernels()
*----- statistics -----*
digraph name: c5
number of solutions
  dominant prekernels : 0
  absorbent prekernels: 0
Chordless circuits of even length $2 \times k$, 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. 12.5), never contain any potential kernel candidates.

![Fig. 12.5: The dual of the chordless 6-circuit](dualChordlessCircuit.png)
We call weak, a chordless circuit with indeterminate inner part. The digraphs.
CirculantDigraph class provides a parameter for constructing such a kind of weak chordless circuits.

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

It is worth noticing that the dual version \(^{(14)}\) of a weak circuit corresponds to its converse version, i.e. \(-c6 = \overline{\overline{c6}}\) (see Fig. 12.6).

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

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

Fig. 12.6: Dual and converse of the weak 6-circuit

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

\(^{13}\) The class of self-codual bipolar-valued digraphs consists of all weakly asymmetric digraphs, i.e. digraphs containing only asymmetric and/or indeterminate links. Limit cases consists of, on the one side, full tournaments with indeterminate reflexive links, and, on the other side, fully indeterminate digraphs. In this class, the converse (inverse \(\overline{\overline{\cdot}}\)) operator is indeed identical to the dual (negation \(-\)) one.
12.3 Kernels in laterized digraphs

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

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

```python
>>> from randomDigraphs import RandomDigraph
>>> rd = RandomDigraph(order=7, arcProbability=0.3, seed=5)
>>> rd.exportGraphViz('randomLaterality')
Exporting to randomLaterality.dot
dot -Grankdir=BT -Tpng randomLaterality.dot -o randomLaterality.png
```

Fig. 12.7: A random digraph instance of order 7 and arc probability 0.3
The random digraph shown in Fig. 12.7 above has no apparent special properties, except from being connected.

```python
>>> rd.showComponents()
*--- Connected Components ---*
1: ['a1', 'a2', 'a3', 'a4', 'a5', 'a6', 'a7']
```

```python
>>> rd.computeChordlessCircuits()
[]  # no chordless circuits detected
```

```python
>>> print('Transitivity degree: %.2f %

... (rd.computeTransitivityDegree()*100))
Transitivity degree: 48.39%
```

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.

```python
>>> rd.showMIS()
*--- Maximal independent choices ---*
['a2', 'a4', 'a6']
['a6', 'a1']
['a5', 'a1']
['a3', 'a1']
['a4', 'a3']
['a7']
```

```python
>>> rd.showPreKernels()
*--- Computing preKernels ---*
Dominant preKernels :
['a2', 'a4', 'a6']
  independence : 1.0
dominance : 1.0
  absorbency : -1.0
covering : 0.500
['a4', 'a3']
  independence : 1.0
dominance : 1.0
  absorbency : -1.0
covering : 0.600  # <=
Absorbent preKernels :
['a3', 'a1']
  independence : 1.0
dominance : -1.0
  absorbency : 1.0
covering : 0.500
['a6', 'a1']
  independence : 1.0
dominance : -1.0
```

(continues on next page)
Among the six MISs contained in this random digraph (see above Lines 3-8) we discover two initial and two terminal kernels (Lines 12-34). Notice by the way the covering values (between 0.0 and 1.0) shown by the `digraphs.Digraph.showPreKernels()` method (Lines 17, 22, 28 and 33). The higher this value, the more the corresponding kernel candidate makes apparent the digraph’s *laterality*. We may hence redraw the same digraph in Fig. 12.8 by looking into its interior via the best covering initial kernel candidate: the dominant choice ‘{a3’,‘4a’} (coloured in yellow), and looking out of it via the best covered terminal kernel candidate: the absorbent choice ‘{a1’,‘a6’} (coloured in blue).

```python
>>> rd.exportGraphViz(fileName='orientedLaterality',
... bestChoice=set([‘a3’, ‘a4’]),
... worstChoice=set([‘a1’, ‘a6’]))

*---- exporting a dot file for GraphViz tools ---------*
Exporting to orientedLaterality.dot
dot -Grankdir=BT -Tpng orientedLaterality.dot -o orientedLaterality.png
```

Fig. 12.8: A random digraph oriented by best covering initial and best covered terminal kernel

In algorithmic decision theory, initial and terminal prekernels may provide convincing best, resp. worst, choice recommendations (see *Computing a best choice recommendation*).
12.4 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.

```python
>>> from outrankingDigraphs import *

>>> g = RandomBipolarOutrankingDigraph(seed=5)

>>> g

*------- Object instance description -------*
Instance class : RandomBipolarOutrankingDigraph
Instance name : randomOutranking
# Actions : 7
# Criteria : 7
Size : 26
Determinateness : 34.275
Valuation domain : {'min': -100.0, 'med': 0.0, 'max': 100.0}

>>> g.showHTMLPerformanceTableau()

```

Performance table randomOutranking

<table>
<thead>
<tr>
<th>criterion</th>
<th>g1</th>
<th>g2</th>
<th>g3</th>
<th>g4</th>
<th>g5</th>
<th>g6</th>
<th>g7</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>64.9</td>
<td>1.31</td>
<td>13.88</td>
<td>98.24</td>
<td>94.10</td>
<td>14.57</td>
<td>31.00</td>
</tr>
<tr>
<td>a2</td>
<td>NA</td>
<td>NA</td>
<td>61.75</td>
<td>87.24</td>
<td>69.06</td>
<td>6.51</td>
<td>81.85</td>
</tr>
<tr>
<td>a3</td>
<td>11.32</td>
<td>27.95</td>
<td>12.67</td>
<td>28.93</td>
<td>96.66</td>
<td>30.14</td>
<td>48.07</td>
</tr>
<tr>
<td>a4</td>
<td>46.91</td>
<td>91.63</td>
<td>0.18</td>
<td>96.15</td>
<td>89.37</td>
<td>60.31</td>
<td>31.58</td>
</tr>
<tr>
<td>a5</td>
<td>NA</td>
<td>76.57</td>
<td>87.14</td>
<td>53.92</td>
<td>29.88</td>
<td>0.34</td>
<td>48.12</td>
</tr>
<tr>
<td>a6</td>
<td>54.38</td>
<td>15.96</td>
<td>20.95</td>
<td>67.78</td>
<td>36.12</td>
<td>67.79</td>
<td>70.47</td>
</tr>
<tr>
<td>a7</td>
<td>57.39</td>
<td>79.71</td>
<td>21.55</td>
<td>20.48</td>
<td>16.60</td>
<td>33.79</td>
<td>5.70</td>
</tr>
</tbody>
</table>

Fig. 12.9: The performance tableau of a random outranking digraph instance

The underlying random performance tableau (see Fig. 12.9) shows the performance grading of 7 potential decision actions with respect to 7 decision criteria supporting each an increasing performance scale from 0 to 100. Notice the missing performance data concerning decision actions ‘a2’ and ‘a5’. The resulting strict outranking - i.e. a weighted majority supported - better than without considerable counter-performance - digraph is shown in Fig. 12.10 below.

```python
>>> gcd = ~(-g)  # Codual: the converse of the negation

>>> gcd.exportGraphViz(fileName='tutOutRanking')

*---- exporting a dot file for GraphViz tools ----*
Exporting to tutOutranking.dot

dot -G rankdir=BT -Tpng tutOutranking.dot -o tutOutranking.png

```
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.

```python
>>> gcd.showPreKernels()
*--- Computing preKernels ---*
Dominant preKernels :
['a1', 'a2', 'a4']
  independence :  0.00
dominance      :  6.98
absorbency      : -48.84
covering        :  0.667
Absorbent preKernels :
['a3', 'a7']
  independence :  0.00
dominance      : -74.42
absorbency      :  16.28
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 co-dual outranking digraph reveal best, resp. worst, choice recommendations one may formulate on the basis of a given outranking digraph instance.
Rubis best choice recommendation(s) (BCR)
(in decreasing order of determinateness)
Credibility domain: \([-100.00, 100.00]\)

*** >> potential best choice(s)
* choice : ['a1', 'a2', 'a4']
  independence : 0.00
  dominance : 6.98
  absorbency : -48.84
  covering (%) : 66.67
  determinateness (%) : 57.97
- most credible action(s) = {'a4': 20.93, 'a2': 20.93, }

*** >> potential worst choice(s)
* choice : ['a3', 'a7']
  independence : 0.00
  dominance : -74.42
  absorbency : 16.28
  covered (%) : 80.00
  determinateness (%) : 64.62
- most credible action(s) = {'a7': 48.84, }

Notice that solving the valued Berge kernel equations (see the Pearls of bipolar-valued epistemic logic) 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. 12.11 below, we orient the drawing of the strict outranking digraph instance with the help of these best and worst choice recommendations.
Fig. 12.11: The strict outranking digraph oriented by its best and worst choice recommendations

The gray arrows in Fig. 12.11, like the one between actions ‘a4’ and ‘a1’, represent indeterminate preferential situations. Action ‘a1’ appears hence to be rather incomparable to all the other, except action ‘a7’. It may be interesting to compare this result with a Copeland ranking of the underlying performance tableau (see Ranking with multiple incommensurable criteria).

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

Fig. 12.12: Heat map with Copeland ranking of the performance tableau
In the resulting linear ranking (see Fig. 12.12), action ‘a4’ is set at first rank, followed by action ‘a2’. This makes sense as ‘a4’ shows three performances in the first quintile, whereas ‘a2’ is only partially evaluated and shows only two such excellent performances. But ‘a4’ also shows a very weak performance in the first quintile. Both decision actions, hence, don’t show eventually a performance profile that would make apparent a clear preference situation in favour of one or the other. In this sense, the prekernels based best choice recommendations may appear more faithful with respect to the actually definite strict outranking relation than any ‘forced’ linear ranking result as shown in Fig. 12.12 above.

12.5 Tractability

Finally, let us give some hints on the tractability of kernel computations. Detecting all (pre)kernels in a digraph is a famously NP-hard computational problem. Checking external stability conditions for an independent choice is equivalent to checking its maximality and may be done in the linear complexity of the order of the digraph. However, checking all independent choices contained in a digraph may get hard already for tiny sparse digraphs of order $n > 20$ (see [BIS-2006b]). Indeed, the worst case is given by an empty or indeterminate digraph where the set of all potential independent choices to check is in fact the power set of the vertices.

```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.  # <<== !!!!

>>> 2**20
1048576
```

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

```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']
```

(continues on next page)
*---- 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]:
                Sxg = [S[0][1] | x[1],
                Sxga = S[0][2] | x[2],
                Sxin = S[0][3] & x[3],
                Sxcho = S[0][0] | x[0]
                Sx = [(Sxcho, Sxg, Sxga, Sxin)]
                yield Sx
```

And, checking maximality of independent choices via the external stability conditions
during their enumeration (see digraphs.Digraph.computePreKernels() below) provides
the effective advantage of computing all initial and terminal prekernels in a single
loop (see Line 10 and [BIS-2006b]).

```python
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()

(continues on next page)
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

13 About split, interval and permutation graphs

- A multiply perfect graph
- Who is the liar?
- Generating permutation graphs
- Recognizing permutation graphs

13.1 A multiply perfect graph

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

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

To illustrate these perfect graph classes, we will generate from 8 intervals, randomly chosen in the default integer range $[0,10]$, a graphs.RandomIntervalIntersectionsGraph instance $g$ (see Line 2 below).

```python
>>> from graphs import RandomIntervalIntersectionsGraph
>>> g = RandomIntervalIntersectionsGraph(order=8,seed=100)
>>> g
*------- Graph instance description ------*
Instance class : RandomIntervalIntersectionsGraph
Instance name : randIntervalIntersections
Seed : 100
```
Graph Order : 8
Graph Size : 23
Valuation domain : [-1.0; 1.0]
Attributes : ['seed', 'name', 'order', 'intervals', 'vertices', 'valuationDomain', 'edges', 'size', 'gamma']

```
>>> print(g.intervals)
[(2, 7), (2, 7), (5, 6), (6, 8), (1, 8), (1, 1), (4, 7), (0, 10)]
```

With seed = 100, we obtain here an interval graph, in fact a perfect graph, which is conjointly a triangulated, a comparability, a split and a permutation graph.

```
>>> g.isPerfectGraph(Comments=True)
Graph randIntervalIntersections is perfect !
```

```
>>> g.isIntervalGraph(Comments=True)
Graph 'randIntervalIntersections' is triangulated.
Graph 'dual_randIntervalIntersections' is transitively orientable.
=> Graph 'randIntervalIntersections' is an interval graph.
```

```
>>> g.isSplitGraph(Comments=True)
Graph 'randIntervalIntersections' is triangulated.
Graph 'dual_randIntervalIntersections' is triangulated.
=> Graph 'randIntervalIntersections' is a split graph.
```

```
>>> g.isPermutationGraph(Comments=True)
Graph 'randIntervalIntersections' is transitively orientable.
Graph 'dual_randIntervalIntersections' is transitively orientable.
=> Graph 'randIntervalIntersections' is a permutation graph.
```

```
>>> print(g.computePermutation())
[v5, v6, v4, v2, v1, v3, v7, v8]
[v8, v6, v1, v2, v4, v3, v7, v5]
[8, 2, 6, 5, 7, 4, 3, 1]
```

```
>>> g.exportGraphViz('randomSplitGraph')
*---- exporting a dot file for GraphViz tools --------*
Exporting to randomSplitGraph.dot
fdp -Tpng randomSplitGraph.dot -o randomSplitGraph.png
```
In Fig. 13.1 we may readily recognize the essential characteristic of split graphs, namely being always splittable into two disjoint sub-graphs: an independent choice \((v6)\) and a clique \((v1, v2, v3, v4, v5, v7, v8)\); which explains their name.

Notice however that the four properties:

1. \(g\) is a comparability graph;
2. \(g\) is a cocomparability graph, i.e. \(-g\) is a comparability graph;
3. \(g\) is a triangulated graph;
4. \(g\) is a cotriangulated graph, i.e. \(-g\) is a comparability graph;

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

### 13.2 Who is the liar?

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

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

```python
>>> from graphs import Graph
>>> g = Graph('berge')
>>> g.showShort()

*---- short description of the graph ----*
Name : 'berge'
Vertices : ['A', 'B', 'C', 'D', 'E', 'I']
Valuation domain : {'min': -1, 'med': 0, 'max': 1}
Gamma function :
A -> ['D', 'B', 'E']
B -> ['E', 'I', 'A']
C -> ['E', 'D', 'I']
D -> ['C', 'I', 'A']
E -> ['C', 'B', 'I', 'A']
I -> ['C', 'E', 'B', 'D']

>>> g.exportGraphViz('berge1')

*---- exporting a dot file for GraphViz tools -------*
Exporting to berge1.dot
fdp -Tpng berge1.dot -o berge1.png
Fig. 13.2: Graph representation of the testimonies of the professors

From graph theory we know that time interval intersections graphs must in fact be interval graphs, i.e. triangulated and co-comparative graphs. The testimonies graph should therefore not contain any chordless cycle of four and more vertices. Now, the presence or not of such chordless cycles in the testimonies graph may be checked as follows.

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

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

And it is D; if we put to doubt his testimony that he saw A (see Line 1 below), we obtain indeed a triangulated graph instance whose dual is a comparability graph.

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

Graph 'berge' is triangulated.

(continues on next page)
Graph 'dual_berge' is transitively orientable.

\[ \Rightarrow \text{Graph 'berge' is an interval graph.} \]

```python
>>> g.exportGraphViz('berge2')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to berge2.dot
fdp -Tpng berge2.dot -o berge2.png
```

![Graph Diagram](image)

**Fig. 13.3: The triangulated testimonies graph**

### 13.3 Generating permutation graphs

A graph is called a **permutation** or **inversion** graph if there exists a permutation of its list of vertices such that the graph is isomorphic to the inversions operated by the permutation in this list (see [GOL-2004] Chapter 7, pp 157-170). This kind is also part of the class of perfect graphs.

```python
>>> from graphs import PermutationGraph
>>> g = PermutationGraph(permutation = [4, 3, 6, 1, 5, 2])
>>> g
*------- Graph instance description ------*
Instance class : PermutationGraph
Instance name : permutationGraph
Graph Order : 6
Permutation : [4, 3, 6, 1, 5, 2]
Graph Size : 9
Valuation domain : [-1.00; 1.00]
Attributes : ['name', 'vertices', 'order', 'permutation',
             'valuationDomain', 'edges', 'size', 'gamma']
>>> g.isPerfectGraph()
True
>>> g.exportGraphViz()
*---- exporting a dot file for GraphViz tools ---------*
```
Fig. 13.4: The default permutation graph

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

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

Fig. 13.5: Minimal vertex coloring of the permutation graph

The correspondingly colored **matching diagram** of the nine **inversions** -the actual **edges** of the permutation graph-, which are induced by the given permutation $[4, 3, 6,$
1, 5, 2], may as well be drawn with the graphviz `neato` layout and explicitly positioned horizontal lists of vertices (see Fig. 13.6).

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

**Fig. 13.6: Colored matching diagram of the permutation [4, 3, 6, 1, 5, 2]**

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]). This orientation of the edges of a permutation graph is always transitive and delivers a transitive ordering of the vertices.

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

>>> g.exportGraphViz()

Fig. 13.8: Random graph of order 8 generated with edge probability 0.4

If the random perfect graph instance \( g \) (see Fig. 13.8) is indeed a permutation graph, \( g \) and its dual \(-g\) should be transitively orientable, i.e. comparability graphs (see [GOL-2004]). With the \texttt{graphs.Graph.isComparabilityGraph()} test, we may easily check this fact. This method proceeds indeed by trying to construct a transitive neighborhood decomposition of a given graph instance and, if successful, stores the resulting edge orientations into a \texttt{self.edgeOrientations} attribute (see [GOL-2004] p.129-132).

```python
>>> if g.isComparabilityGraph():
...
    print(g.edgeOrientations)
    (('v1', 'v1'): 0, ('v1', 'v2'): 1, ('v2', 'v1'): -1, ('v1', 'v3'): 1,
     ('v3', 'v1'): -1, ('v1', 'v4'): 1, ('v4', 'v1'): -1, ('v1', 'v5'): 0,
     ('v5', 'v1'): 0, ('v1', 'v6'): 1, ('v6', 'v1'): -1, ('v1', 'v7'): 0,
     ('v7', 'v1'): 0, ('v1', 'v8'): 1, ('v8', 'v1'): -1, ('v2', 'v2'): 0,
     ('v2', 'v3'): 0, ('v3', 'v2'): 0, ('v2', 'v4'): 0, ('v4', 'v2'): 0,
     ('v4', 'v5'): 0, ('v5', 'v4'): 0, ('v2', 'v6'): 0, ('v6', 'v2'): 0,
     ('v2', 'v7'): 0, ('v7', 'v2'): 0, ('v2', 'v8'): 0, ('v8', 'v2'): 0,
     ('v3', 'v3'): 0, ('v3', 'v4'): 0, ('v4', 'v3'): 0, ('v3', 'v5'): 0,
     ('v5', 'v3'): 0, ('v3', 'v6'): 0, ('v6', 'v3'): 0, ('v3', 'v7'): 0,
```
The resulting orientation of the edges of $g$ (see Fig. 13.9) is indeed transitive. The same procedure applied to the dual graph $gd = -g$ gives a transitive orientation to the edges of $-g$.

```python
>>> gd = -g
>>> if gd.isComparabilityGraph():
...    print(gd.edgeOrientations)
{('v1', 'v1'): 0, ('v1', 'v2'): 0, ('v2', 'v1'): 0, ('v1', 'v3'): 0,
 ('v3', 'v1'): 0, ('v1', 'v4'): 0, ('v4', 'v1'): 0, ('v1', 'v5'): 1,
 ('v5', 'v1'): -1, ('v1', 'v6'): 0, ('v6', 'v1'): 0, ('v1', 'v7'): 1,
 ('v7', 'v1'): -1, ('v1', 'v8'): 0, ('v8', 'v1'): 0, ('v2', 'v2'): 0,
 ('v2', 'v3'): -2, ('v3', 'v2'): 2, ('v2', 'v4'): -3, ('v4', 'v2'): 3,
 ('v2', 'v5'): 1, ('v5', 'v2'): -1, ('v2', 'v6'): 1, ('v6', 'v2'): -1,
 ('v2', 'v7'): 1, ('v7', 'v2'): -1, ('v2', 'v8'): 1, ('v8', 'v2'): -1,
 ('v3', 'v2'): 0, ('v2', 'v3'): -3, ('v4', 'v3'): 3, ('v3', 'v3'): 1,
 ('v3', 'v5'): -1, ('v5', 'v3'): 1, ('v5', 'v6'): -1, ('v5', 'v7'): 1,
 ('v7', 'v3'): -1, ('v3', 'v8'): 1, ('v8', 'v3'): -1, ('v4', 'v3'): 0,
 ('v4', 'v5'): -1, ('v5', 'v4'): -1, ('v4', 'v6'): 1, ('v6', 'v4'): -1,
 ('v4', 'v7'): 1, ('v7', 'v4'): -1, ('v4', 'v8'): 1, ('v8', 'v4'): -1,
 ('v7', 'v4'): 1, ('v4', 'v7'): 1, ('v7', 'v8'): -1, ('v8', 'v7'): -1,
 ('v8', 'v4'): 1, ('v4', 'v8'): 1, ('v8', 'v4'): 1}
```
It is worthwhile noticing that the orientation of $g$ is achieved with a single neighbourhood decomposition, covering all the vertices. Whereas, the orientation of the dual $-g$ needs a decomposition into three subsequent neighbourhoods marked in black, red and blue (see Fig. 13.10).

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

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] p.159.

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

>>> 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).

>>> 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]). In our case here, we observe two different permutations and their reverses:

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

And:

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

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

And, we recover indeed two isomorphic copies of the original random graph (compare Fig. 13.11 with Fig. 13.8).

Back to Content Table

14 On tree graphs and graph forests

- Generating random tree graphs
- Recognizing tree graphs
- Spanning trees and forests
- Maximum determined spanning forests
14.1 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', 'v7', 'v6', 'v7']
>>> t.exportGraphViz('tutRandomTree')
*---- exporting a dot file for GraphViz tools ---------*
Exporting to tutRandomTree.dot
neato -Tpng tutRandomTree.dot -o tutRandomTree.png
```

![Graphs Python module (graphviz), R. Bisendorf, 2019](image)

Fig. 14.1: Random Tree instance of order 9

A tree graph of order $n$ contains $n-1$ edges (see Line 8 and 9) and we may distinguish
vertices like $v_1$, $v_2$, $v_4$, $v_5$ or $v_9$ of degree 1, called the leaves of the tree, and vertices like $v_3$, $v_6$, $v_7$ or $v_8$ of degree 2 or more, called the nodes of the tree.

The structure of a tree of order $n > 2$ is entirely characterised by a corresponding Prüfer code - i.e. a list of vertices keys- of length $n-2$. See, for instance in Line 12 the code ['v3', 'v8', 'v8', 'v3', 'v7', 'v6', 'v7'] corresponding to our sample tree graph $t$.

Each position of the code indicates the parent of the remaining leaf with the smallest vertex label. Vertex $v_3$ is thus the parent of $v_1$ and we drop leaf $v_1$, $v_8$ is now the parent of leaf $v_2$ and we drop $v_2$, vertex $v_8$ is again the parent of leaf $v_4$ and we drop $v_4$, vertex $v_3$ is the parent of leaf $v_5$ and we drop $v_5$, $v_7$ is now the parent of leaf $v_3$ and we may drop $v_3$, $v_6$ becomes the parent of leaf $v_8$ and we drop $v_8$, $v_7$ becomes now the parent of leaf $v_6$ and we may drop $v_6$. The two eventually remaining vertices, $v_7$ and $v_9$, give the last link in the reconstructed tree (see [BAR-1991]).

It is as well possible to first, generate a random Prüfer code of length $n-2$ from a set of $n$ vertices and then, construct the corresponding tree of order $n$ by reversing the procedure illustrated above (see [BAR-1991]).
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?

### 14.2 Recognizing tree graphs

Given a graph \( g \) of order \( n \) and size \( s \), the following 5 assertions \( A_1, A_2, A_3, A_4 \) and \( A_5 \) are all equivalent (see [BAR-1991]):

- \( A_1 \): \( g \) is a tree;
- \( A_2 \): \( g \) is without (chordless) cycles and \( n = s + 1 \);
- \( A_3 \): \( g \) is connected and \( n = s + 1 \);
- \( A_4 \): Any two vertices of \( g \) are always connected by a unique path;
- \( A_5 \): \( g \) is connected and dropping any single edge will always disconnect \( g \).

Assertion \( A_3 \), 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. 14.3 (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
```

![GraphViz drawing](image)

**Fig. 14.3: Recognizing a tree instance**

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

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

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

### 14.3 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]
Wilson’s algorithm only works for connected graphs\(^4\).
>>> g = RandomGraph(order=15, edgeProbability=0.1, seed=140)
>>> g.computeComponents()
[['v12', 'v01', 'v13'], ['v02', 'v06'],
 ['v08', 'v03', 'v07'], ['v15', 'v11', 'v10', 'v04', 'v05'],
 ['v09', 'v14']]

>>> spf = RandomSpanningForest(g, seed=100)
>>> spf.exportGraphViz(fileName='spanningForest', WithSpanningTree=True)
*---- exporting a dot file for GraphViz tools --------*
Exporting to spanningForest.dot
[['v03', 'v07', 'v08', 'v07', 'v03'],
 ['v13', 'v12', 'v13', 'v01', 'v13'],
 ['v02', 'v06', 'v02'],
 ['v15', 'v11', 'v04', 'v11', 'v15', 'v10', 'v05', 'v10', 'v15'],
 ['v09', 'v14', 'v09']]
neato -Tpng spanningForest.dot -o spanningForest.png

Fig. 14.5: Random spanning forest instance

14.4 Maximum determined spanning forests

In case of valued graphs supporting weighted edges, we may finally construct a most determined spanning tree (or forest if not connected) using Kruskal’s greedy minimum-spanning-tree algorithm\(^5\) on the dual valuation of the graph [KRU-1956].

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

\(^5\) 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.
```
>>> from graphs import *
>>> g = RandomValuationGraph(seed=2)
>>> print(g)

*------- Graph instance description ------*
Instance class : RandomValuationGraph
Instance name : randomGraph
Graph Order : 5
Graph Size : 7
Valuation domain : [-1.00; 1.00]
Attributes : ['name', 'order', 'vertices', 'valuationDomain',
      'edges', 'size', 'gamma']

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

```
>>> dg = g.graph2Digraph()
>>> dg.showRelationTable()

* ---- Relation Table -----*
<table>
<thead>
<tr>
<th>S</th>
<th>'v1'</th>
<th>'v2'</th>
<th>'v3'</th>
<th>'v4'</th>
<th>'v5'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'v1'</td>
<td>0.00</td>
<td>0.91</td>
<td>0.90</td>
<td>-0.89</td>
<td>-0.83</td>
</tr>
<tr>
<td>'v2'</td>
<td>0.91</td>
<td>0.00</td>
<td>0.67</td>
<td>0.47</td>
<td>0.34</td>
</tr>
<tr>
<td>'v3'</td>
<td>0.90</td>
<td>0.67</td>
<td>0.00</td>
<td>-0.38</td>
<td>0.21</td>
</tr>
<tr>
<td>'v4'</td>
<td>-0.89</td>
<td>0.47</td>
<td>-0.38</td>
<td>0.00</td>
<td>0.21</td>
</tr>
<tr>
<td>'v5'</td>
<td>-0.83</td>
<td>0.34</td>
<td>0.21</td>
<td>0.21</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Valuation domain: [-1.00;1.00]
```

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

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

*------- Graph instance description ------*
Instance class : BestDeterminedSpanningForest
Instance name : randomGraph_randomSpanningForest
Graph Order : 5
Graph Size : 4
Valuation domain : [-1.00; 1.00]
Attributes : ['name', 'vertices', 'order', 'valuationDomain',
      'edges', 'size', 'gamma', 'dfs',
      'date', 'averageTreeDetermination']

*---- best determined spanning tree specific data ----*
Depth first search path(s) :
[[['v1', 'v2', 'v4', 'v2', 'v5', 'v2', 'v1', 'v3', 'v1']]]
Average determination(s) : [Decimal('0.655')]

The given graph is connected and, hence, admits a single spanning tree (see Fig. 14.6) 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).
```
Fig. 14.6: Best determined spanning tree

One may easily verify that all other potential spanning trees, including instead the edges \{v3, v5\} and/or \{v4, v5\} - will show a lower average determination.

Back to Content Table

15 Bibliography

References


[BIS-2016] Bis dorff R. (2016). “Computing linear rankings from trillions of pairwise outranking situations”. In Proceedings of DA2PL’2016 From Multiple Criteria Decision Aid to Preference Learning, R. Busa-Fekete, E. Hüllermeier, V. Mousseau and K. Pfannschmidt (Eds.), University of Paderborn (Germany), Nov. 7-8 2016: 1-6 (downloadable PDF file 451.4 kB (http://hdl.handle.net/10993/28613))


