Motivation: Showing a performance tableau

Consider a performance tableau showing the service quality of 12 commercial cloud providers measured by an external auditor on 14 incommensurable performance criteria.

Legend: 0 = 'very weak', 1 = 'weak', 2 = 'fair', 3 = 'good', 4 = 'very good', 'NA' = missing data; 'green' and 'red' mark the best, respectively the worst, performances on each criterion.

Motivation: showing an ordered heat map

The same performance tableau may be optimistically colored with the highest 7-tiles class of the marginal performances and presented like a heat map, eventually linearly ordered, following for instance the Copeland ranking rule, from the best to the worst performing alternatives (ties are lexicographically resolved).

How to rank big performance tableaux?

- The Copeland ranking rule is based on crisp net flows requiring the in- and out-degree of each node in the outranking digraph;
- When the order $n$ of the outranking digraph becomes big (several thousand or millions of alternatives), this requires handling a huge set of $n^2$ pairwise outranking situations;
- We use instead a sparse model of the outranking digraph, where we only keep a linearly ordered list of diagonal multicriteria quantiles equivalence classes with local outranking content.
Example of sparse outranking Digraph

```python
>>> from sparseOutrankingDigraphs import *
>>> t = RandomPerformanceTableau(numberOfActions=50)
>>> bg = PreRankedOutrankingDigraph(t,quantiles=5)
>>> bg.showDecomposition()
--- quantiles decomposition in decreasing order---
c1. [0.60-0.80[ : ['a22','a24','a32']
c2. [0.40-0.80[ : ['a16', 'a28','a31','a40']
c3. [0.40-0.60[ : ['a01','a02','a05','a06','a10',
'a13','a15','a25','a27','a35','a36','a37','a39','a41','a48']
c4. [0.20-0.60[ : ['a09','a14','a18','a20','a26',
'a38','a43','a45','a49']
c5. [0.20-0.40[ : ['a03','a04','a07','a08','a11',
'a12','a17','a21','a29','a30','a33','a34','a42','a44','a47']
c6. [0.00-0.40[ : ['a46','a50']
c7. [0.00-0.20[ : ['a19','a23']
```

Sparse versus standard outranking digraph of order 50

Symbol legend

⊤ outranking for certain
+ more or less outranking
= indeterminate
− more or less outranked
⊥ outranked for certain

Sparse digraph bg:
# Actions : 50
# Criteria : 7
Sorted by : 5-Tiling
Ranking rule : Copeland
# Components : 7
Minimal order : 1
Maximal order : 15
Average order : 7.1
Fill rate : 20.980%
correlation : +0.7563

Properties of $q$-tiles sorting result

1. **Coherence:** Each object is always sorted into a non-empty subset of adjacent $q$-tiles classes.
2. **Uniqueness:** If the $q$-tiles classes represent a discriminated partition of the measurement scales on each criterion and $r \neq 0$, then every object is sorted into exactly one $q$-tiles class.
3. **Separability:** Computing the sorting result for object $x$ is independent from the computing of the other objects’ sorting results.

Comment

The separability property gives us access to efficient parallel processing of class membership characteristics $r(x \in q^h)$ for all $x \in X$ and $q^h$ in $Q$.

Multithreading the $q$-tiles sorting & ranking procedures

1. 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.
2. Furthermore, the ranking procedure being local to each diagonal component, these procedures may as well be safely processed in parallel threads on each restricted outranking digraph $G|q^h$. 

Multithreading the $q$-tiles sorting & ranking procedures

1. 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.
2. Furthermore, the ranking procedure being local to each diagonal component, these procedures may as well be safely processed in parallel threads on each restricted outranking digraph $G|q^h$. 

Multithreading the $q$-tiles sorting & ranking procedures
Generic algorithm design for parallel processing

```python
from multiprocessing import Process, active_children
class myThread(Process):
    def __init__(self, threadID, ...):
        Process.__init__(self)
        self.threadID = threadID
        ...
    def run(self):
        ... task description ...
        ...
nbrOfJobs = ...
for job in range(nbrOfJobs):
    ... pre-threading tasks per job
    print('iteration = ',job+1,end=" ")
splitThread = myThread(job, ...)
splitThread.start()
while active_children() != []:
    pass
print('Exiting computing threads')
for job in range(nbrOfJobs):
    ... post-threading tasks per job
```

HPC performance measurements

<table>
<thead>
<tr>
<th>digraph order</th>
<th>digraph standard model</th>
<th>sparse model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#c.</td>
<td>τ_g</td>
</tr>
<tr>
<td>1000</td>
<td>118</td>
<td>6&quot;</td>
</tr>
<tr>
<td>2000</td>
<td>118</td>
<td>15&quot;</td>
</tr>
<tr>
<td>2500</td>
<td>118</td>
<td>27&quot;</td>
</tr>
<tr>
<td>10000</td>
<td>118</td>
<td>7&quot;</td>
</tr>
<tr>
<td>15000</td>
<td>118</td>
<td>12&quot;</td>
</tr>
<tr>
<td>25000</td>
<td>118</td>
<td>36&quot;</td>
</tr>
<tr>
<td>50000</td>
<td>118</td>
<td>2&quot;</td>
</tr>
<tr>
<td>100000</td>
<td>(size = [10^{10}])</td>
<td>118</td>
</tr>
<tr>
<td>1732051</td>
<td>(size = [3 \times 10^{12}])</td>
<td>118</td>
</tr>
</tbody>
</table>

Legend:
- #c. = number of cores;
- g: standard outranking digraph, bg: the sparse outranking digraph;
- τ_g, resp. τ_bg, are the corresponding constructor run times;
- τ_g, resp. τ_bg are the ordinal correlation of the Copeland ordering with the given outranking relation.

Gaia-80 November 2016 ranking record

```
Results with 118 cores on gaia-80, seed=185
model: Obj, equijectives, ('beta', 'variable', None)
Tue Nov 22 07:27:51 2016
perfTab: 625.218357 sec., 563959984 bytes

Instance name: randomObjectivesPerTab_mp
# Actions: 25000000
# Criteria: 21
Sorting by: Ordering strategy: average
Local ranking rule: copeland
# Components: 200000
Minimal size: 1
Maximal order: 543
Average order: 12.5
Fill rate: 0.0088
** Constructor run times (in sec.) **
# Threads: 118
Total time: 10664.60302
Quantiles sorting: 0.021160085
Preordering: 0.570926
Decomposing: 352.83330
Ordering: 0.00007
```

Concluding ...

- We implement a sparse outranking digraph model coupled with a linearly ordering algorithm based on quantiles-sorting & local-ranking procedures;
- Global ranking result fits apparently well with the given outranking relation;
- Independent sorting and local ranking procedures allow effective multiprocessing strategies;
- Efficient scalability allows hence the linear ranking of very large sets of potential decision actions (millions of nodes) graded on multiple incommensurable criteria;
- Good perspectives for further optimization with cPython and HPC ad hoc tuning.

Python and cython HPC modules available under:
http://github.com/rbisdorff/Digraph3