Lecture 11: Ranking big multiple criteria performance tableaux

Raymond Bisdorff

University of Luxembourg
MICS ADT Course 2020

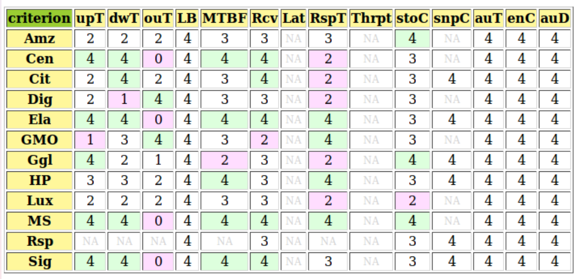
May 18, 2020

Contents

Contents

Motivation: Showing a performance tableau

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

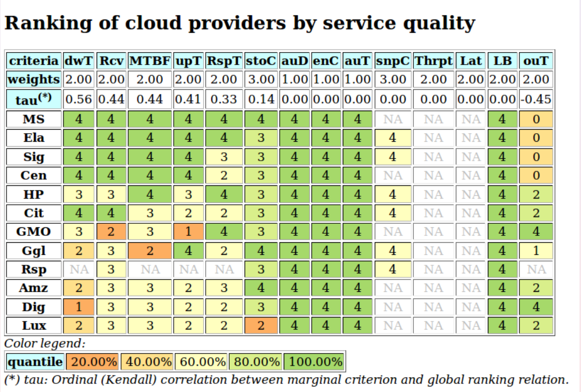


criterion	upT	dwT	ouT	LB	MTBF	Rcv	Lat	RspT	Thrpt	stoC	snpC	auT	enC	auD
Amz	2	2	2	4	3	3	NA	3	NA	4	NA	4	4	4
Cen	4	4	0	4	4	4	NA	2	NA	3	NA	4	4	4
Cit	2	4	2	4	3	4	NA	2	NA	3	4	4	4	4
Dig	2	1	4	4	3	3	NA	2	NA	3	NA	4	4	4
Ela	4	4	0	4	4	4	NA	4	NA	3	4	4	4	4
GMO	1	3	4	4	3	2	NA	4	NA	3	NA	4	4	4
Ggl	4	2	1	4	2	3	NA	2	NA	4	4	4	4	4
HP	3	3	2	4	4	3	NA	4	NA	3	4	4	4	4
Lux	2	2	2	4	3	3	NA	2	NA	2	NA	4	4	4
MS	4	4	0	4	4	4	NA	4	NA	4	NA	4	4	4
Rsp	NA	NA	NA	4	NA	3	NA	NA	NA	3	4	4	4	4
Sig	4	4	0	4	4	4	NA	3	NA	3	4	4	4	4

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).



Ranking of cloud providers by service quality

criteria	dwT	Rcv	MTBF	upT	RspT	stoC	auD	enC	auT	snpC	Thrpt	Lat	LB	ouT
weights	2.00	2.00	2.00	2.00	2.00	3.00	1.00	1.00	1.00	3.00	2.00	2.00	2.00	2.00
tau$^{(*)}$	0.56	0.44	0.44	0.41	0.33	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-0.45
MS	4	4	4	4	4	4	4	4	4	NA	NA	NA	4	0
Ela	4	4	4	4	4	3	4	4	4	4	NA	NA	4	0
Sig	4	4	4	4	3	3	4	4	4	4	NA	NA	4	0
Cen	4	4	4	4	2	3	4	4	4	NA	NA	NA	4	0
HP	3	3	4	3	4	3	4	4	4	4	NA	NA	4	2
Cit	4	4	3	2	2	3	4	4	4	4	NA	NA	4	2
GMO	3	2	3	1	4	3	4	4	4	NA	NA	NA	4	4
Ggl	2	3	2	4	2	4	4	4	4	4	NA	NA	4	1
Rsp	NA	3	NA	NA	NA	3	4	4	4	4	NA	NA	4	NA
Amz	2	3	3	2	3	4	4	4	4	NA	NA	NA	4	2
Dig	1	3	3	2	2	3	4	4	4	NA	NA	NA	4	4
Lux	2	3	3	2	2	2	4	4	4	NA	NA	NA	4	2

Color legend:

quantile	20.00%	40.00%	60.00%	80.00%	100.00%

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

How to rank big performance tableaux ?

- *Copeland*'s, as well as the *NetFlows* ranking rule, are of complexity $\mathcal{O}(n^2)$;
- 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 shall present hereafter a sparse model of the outranking digraph, where we only keep a linearly ordered list of diagonal quantiles equivalence classes with local outranking content.

Performance quantiles

- Let X be the set of n potential decision alternatives evaluated on a single real performance criteria.
- We denote $x, y, \ldots$ the performances observed of the potential decision actions in X.
- We call quantile $q(p)$ the performance such that $p\%$ of the observed n performances in X are less or equal to $q(p)$.
- We consider a series: $p_k = k/q$ for $k = 0, \ldots q$ of $q + 1$ equally spaced quantiles like
  - quartiles: $0.00, 0.25, 0.50, 0.75, 1.00$,
  - quintiles: $0.00, 0.20, 0.40, 0.60, 0.80, 1.00$,
  - heptiles (7): $0.00, 0.14, 0.29, 0.43, 0.57, 0.71, 0.86, 1.00$, etc
- The quantile $q(p_k)$ is estimated by linear interpolation from the cumulative distribution of the performances in X.

Upper- and lower-closed quantile classes

- The upper-closed q^k class corresponds to the interval $]q(p_{k-1}); q(p_k)]$, for $k = 2, \ldots, q$, where $q(p_q) = \max_X x$ and the first class gathers all data below $q(p_1)$: $]-\infty; q(p_1)]$.
- The lower-closed q_k class corresponds to the interval $[q(p_{k-1}); q(p_k)[$,for $k = 1, \ldots, q-1$, where $q(p_0) = \min_X x$ and the last class gathers all data above $q(p_{q-1})$: $[q(p_{q-1}), +\infty[$.
- We call q-tiles a complete series of $k = 1, \ldots, q$ *upper-closed* q^k, resp. *lower-closed* q_k, quantile classes.

Multiple criteria q-tiles sorting

- $X = \{x, y, z, ...\}$ is a finite set of n objects to be sorted.
- $F = \{1, ..., m\}$ is a finite and coherent family of m performance criteria.
- For each criterion j in F, the objects are evaluated on a real performance scale $[0; M_j]$, supporting
  1. an indifference threshold ind_j,
  2. a preference threshold pr_j,
  3. a veto threshold v_j,
- such that $0 \leqslant ind_j < pr_j \ll v_j \leqslant M_j$.
- Each criterion j in F carries a rational significance w_j such that $0 < w_j < 1.0$ and $\sum_{j \in F} w_j = 1.0$.

---

q-tiles sorting with bipolar outrankings

From an epistemic point of view, we say that:

1. object x *outranks* object y, denoted $(x \succsim y)$, if
   1.1 a significant majority of criteria validates a global outranking situation between x and y, i.e. $(x \succcurlyeq y)$ and
   1.2 no veto $(x \lll y)$ is observed on a discordant criterion,
2. object x *does not outrank* object y, denoted $(x \not\succsim y)$, if
   2.1 a significant majority of criteria invalidates a global outranking situation between x and y, i.e. $(x \not\succcurlyeq y)$ and
   2.2 no counter-veto $(x \ggg y)$ observed on a concordant criterion.
3. The bipolar characteristic of x belonging to *upper-closed* q-tiles class q^k, resp. *lower-closed* class q_k, may hence, in a multiple criteria outranking approach, be assessed as follows:

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

$$r(x \in q_k) = \min \left[r\big(x \succsim \mathrm{q}(p_{k-1})\big),\ -r\big(x \succsim \mathrm{q}(p_k)\big) \right]$$

The multicriteria q-tiles sorting algorithm

```
1. Input: a set $X$ of $n$ objects with a performance table on a
   family of $m$ criteria and a set $\mathcal{Q}$ of $k = 1,..,q$ empty $q$-tiles
   equivalence classes.
2. For each object $x \in X$ and each $q$-tiles class $q^k \in \mathcal{Q}$
     if upper-closed quantiles (default):
       $r(x \in q^k) \leftarrow \min [ -r(\mathrm{q}(p_{k-1}) \succsim x), r(\mathrm{q}(p_k) \succsim x) ]$
       if $r(x \in q^k) \geqslant 0$:
         add $x$ to $q$-tiles class $q^k$
     else:
       $r(x \in q_k) \leftarrow \min [ r(x \succsim \mathrm{q}(p_{k-1})), -r(x \succsim \mathrm{q}(p_k)) ]$
       if $r(x \in q_k) \geqslant 0$:
         add $x$ to $q$-tiles class $q_k$
3. Output: $\mathcal{Q}$
```

Example of upper-closed quintiles sorting

```
>>> from randomPerfTabs import *
>>> t = RandomPerformanceTableau(numberOfActions=50,seed=5)
>>> from sortingDigraphs import QuantilesSortingDigraph
>>> qs = QuantilesSortingDigraph(t,limitingQuantiles=5)
>>> qs.showSorting()
*--- Sorting results in descending order ---*
]0.80 - 1.00]:   ['a22']
]0.60 - 0.80]:   ['a03', 'a07', 'a08', 'a11', 'a14', 'a17',
                  'a19', 'a20', 'a29', 'a32', 'a33', 'a37',
                  'a39', 'a41', 'a42', 'a49']
]0.40 - 0.60]:   ['a01', 'a02', 'a04', 'a05', 'a06', 'a08',
                  'a09', 'a16', 'a17', 'a18', 'a19', 'a21',
                  'a24', 'a27', 'a28', 'a30', 'a31', 'a35',
                  'a36', 'a40', 'a43', 'a46', 'a47', 'a48',
                  'a49', 'a50']
]0.20 - 0.40]:   ['a04', 'a10', 'a12', 'a13', 'a15', 'a23',
                  'a25', 'a26', 'a34', 'a38', 'a43', 'a44',
                  'a45', 'a49']
]   < - 0.20]:   ['a44']
```

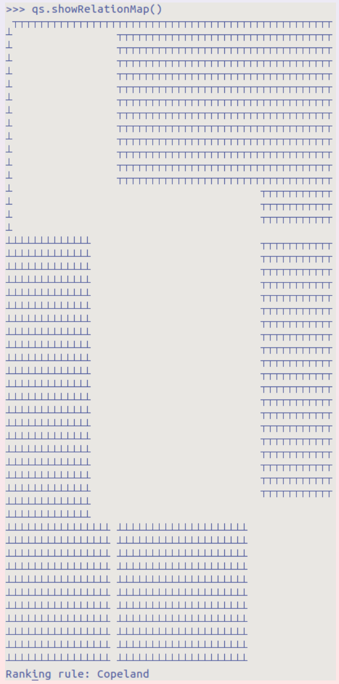
Properties of q-tiles sorting algorithm

1. *Coherence*: Each object is always sorted into a non-empty subset of adjacent q-tiles classes.
2. *Separability*: Computing the sorting result for object x is independent from the computing of the other objects' sorting results.
3. The complexity of the q-tiles sorting algorithm is $\mathcal{O}(nmq)$; linear in the number of decision actions (n), criteria (m) and quantile classes (q).

Comment

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

Relation map of the quintiles sorting result



```
>>> qs.showRelationMap()
```

Ranking rule: Copeland

Symbol legend

⊤ outranking for certain

' ' indeterminate

⊥ outranked for certain

Sorted digraph *qs*:

# Actions : 50

# Criteria : 7

Sorted by : 5-Tiling

Ranking rule : Copeland

Ordering by : average

# tiles : 5

Minimal order : 1

Maximal order : 26

Average order : 15.2

Content

1. Pre-ranking a q-tiled performance tableau
    - How to rank a big performance tableau ?
    - q-tiling a performance tableau
    - Properties of the q-tiles sorting
2. Sparse outranking digraphs
    - Pre-ranking the q-tiles sorting result
    - The `PreRankedOutrankingDigraph` class
    - The q-tiles local ranking algorithm
3. HPC-ranking a big performance tableau
    - Multithreading the sorting&ranking procedure
    - Ranking Performance using the UL HPC clusters
    - Profiling the HPC sorting&ranking procedure

Example of upper-closed quintiles sorting

Quantile class	Content
]0.80 - 1.00]:	[$a22$]
]0.60 - 0.80]:	[$a03$, $a07$, $a08$, $a11$, $a14$, $a17$,$a19$, $a20$, $a29$, $a32$, $a33$, $a37$, $a39$, $a41$, $a42$, $a49$]
]0.40 - 0.60]:	[$a01$, $a02$, $a04$, $a05$, $a06$, $a08$, $a09$, $a16$, $a17$, $a18$, $a19$, $a21$, $a24$, $a27$, $a28$, $a30$, $a31$, $a35$, $a36$, $a40$, $a43$, $a46$, $a47$, $a48$, $a49$, $a50$]
]0.20 - 0.40]:	[$a04$, $a10$, $a12$, $a13$, $a15$, $a23$, $a25$, $a26$, $a34$, $a38$, $a43$, $a44$, $a45$, $a49$]
] < - 0.20]:	[$a44$]

Alternatives $a08$, $a17$, $a19$ and $a49$ are contained in two, resp. three adjacent quintile classes.

Pre-ranking the *q*-tiles sorting

- The *q*-tiles sorting usually results in *overlapping* quantile classes.
- We gather the decision alternatives by their lowest and highest quantile limits. Alternatives $a08$, $a17$ and $a19$, for instance, are contained in the quantile class $]0.40 - 0.80]$, whereas alternative $a49$ is contained in class $]0.20 - 0.80]$.
- The result gives a more or less refined partition of the potential decision alternatives.
- We rank the parts of this partition from *best to worst* by descending average of *low and high* quantile class limits. In case of a tie, we order furthermore by *descending high limit*. Class $]0.20 - 0.80]$ hence is ranked before class $]0.40 - 0.60]$

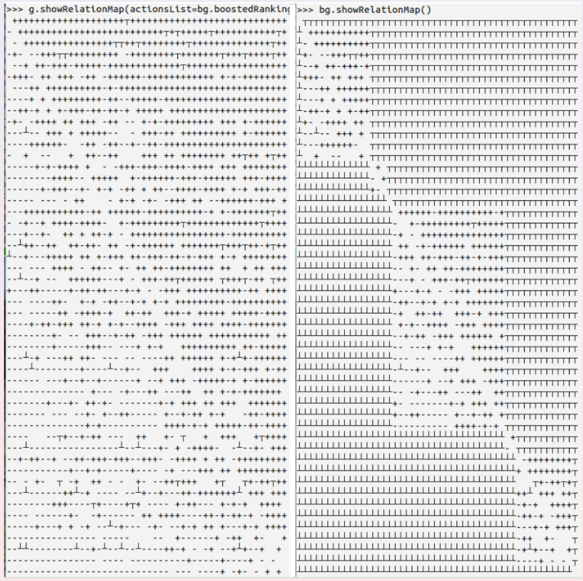
Ordering the upper-closed quintiles sorting result

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

The `PreRankedOutrankingDigraph` class

```
>>> from randomPerfTabs import *
>>> t = RandomPerformanceTableau(numberOfActions=50,seed=5)
>>> from sparseOutrankingDigraphs import\
            PreRankedOutrankingDigraph
>>> bg = PreRankedOutrankingDigraph(t,quantiles=5,\
            quantileOrderingStrateg='average')
>>> bg.showDecomposition()
*--- quantiles decomposition in decreasing order---*
c1. ]0.80-1.00] : ['a22']
c2. ]0.60-0.80] : ['a03', 'a07', 'a11', 'a14', 'a20', 'a29',
                  'a32', 'a33', 'a37', 'a39', 'a41', 'a42']
c3. ]0.40-0.80] : ['a08', 'a17', 'a19']
c4. ]0.20-0.80] : ['a49']
c5. ]0.40-0.60] : ['a01', 'a02', 'a05', 'a06', 'a09', 'a16', 'a18',
                  'a21', 'a24', 'a27', 'a28', 'a30', 'a31', 'a35',
                  'a36', 'a40', 'a46', 'a47', 'a48', 'a50']
c6. ]0.20-0.60] : ['a04', 'a43']
c7. ]0.20-0.40] : ['a10', 'a12', 'a13', 'a15', 'a23',
                  'a25', 'a26', 'a34', 'a38', 'a45']
c8. ] <  -0.40] : ['a44']
```

Standard versus sparse outranking digraph of order 50



```
>>> g.showRelationMap(actionsList=bg.boostedRanking
```

```
>>> bg.showRelationMap()
```

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 : 8

Minimal order : 1

Maximal order : 20

Average order : 6.2

fill rate : 24.9%

correlation : +0.789

q-tiles local ranking algorithm

1. **Input**: the outranking digraph $\mathcal{G}(X, \succsim)$, a partition P of k linearly ordered decreasing parts of X obtained by the q-sorting algorithm, and an empty list $\mathcal{R}$.
2. **For each** quantile class $q_i \in P$, $i = 1, ..., k$:

```
if $\#(q_i) > 1$:
    $R_i \leftarrow$ locally rank $q_i$ in $\mathcal{G}_{|q_i}$
        (if ties, render alphabetic order of action keys)
else:
    $R_i \leftarrow q_i$
append $R_i$ to $\mathcal{R}$
```

3. **Output**: $\mathcal{R}$

q-tiles local ranking algorithm – Comments

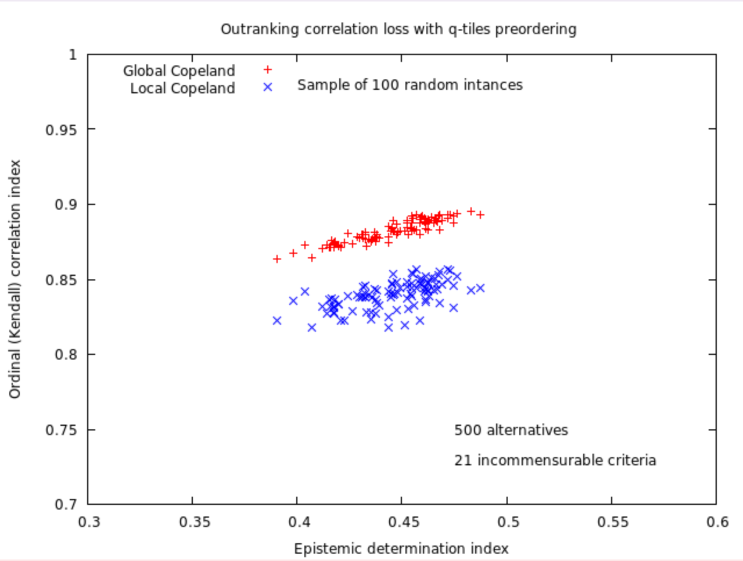
1. The complexity of the q-tiles local ranking algorithm is linear in the number k of components resulting from a q-tiles sorting which contain more than one action.
2. Two local ranking rules are scalable to big outranking digraphs: *Copeland*'s and *Net-flows*' rule; both of complexity $\mathcal{O}(\#(q_i)^2)$ on each q_i restricted outranking digraph $\mathcal{G}_{|q_i}$.
3. In case of local ties (very similar evaluations for instance), the **local ranking** procedure will render these actions in increasing alphabetic ordering of the action keys.
4. The resuting global ranking is stored in the `boostedRanking` attribute.

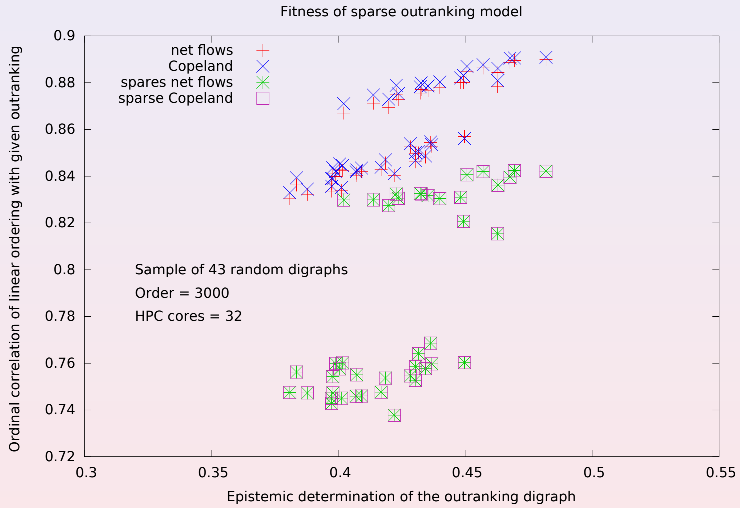
Variable access to adjacency table entries

```
def relation(self,x,y):
    """
    Functional retrieval of the outranking
    characteristic *r(x S y)*.
    """
    Min = self.valuationdomain[’min’]
    Med = self.valuationdomain[’med’]
    Max = self.valuationdomain[’max’]
    if x == y:
        return Med
    cx = self.actions[x][’component’]
    cy = self.actions[y][’component’]
    if cx == cy:
        return self.components[cx][’subGraph’].relation[x][y]
    elif self.components[cx][’rank’] \
                    < self.components[cy][’rank’]:
        return Max
    else:
        return Min
```

Only the outranking relation table of each component is stored, leading to a more or less low fillrate of the sparse outranking digraph.

Standard versus 50-tiled sparse outranking digraphs





Both, the *NetFlows* and *Copeland*'s, ranking rules are equally efficient on the sparse outranking digraph. The quality of the sparse model based linear ranking is depending on the model of the random performance tableaux, but **not** on its actual order.

Contents

1. Pre-ranking a *q*-tiled performance tableau
   How to rank a big performance tableau ?
   q-tiling a performance tableau
   Properties of the *q*-tiles sorting
2. Sparse outranking digraphs
   Pre-ranking the *q*-tiles sorting result
   The `PreRankedOutrankingDigraph` class
   The q-tiles local ranking algorithm
3. HPC-ranking a big performance tableau
   Multithreading the sorting&ranking procedure
   Ranking Performance using the UL HPC clusters
   Profiling the HPC sorting&ranking procedure

Multithreading the *q*-tiles sorting & ranking procedure

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 hence be safely processed in parallel threads on each restricted outranking digraph $\mathcal{G}_{|q_i}$.

Generic algorithm design for parallel processing

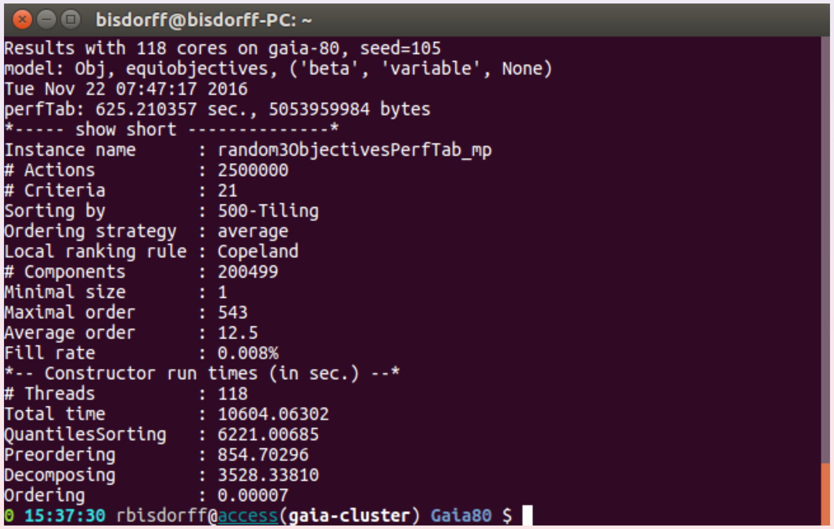
```
from multiprocessing import Process, active_children
    class myThread(Process):
        def __init__(self, threadID, ...)
                    Process.__init__(self)
                    self.threadID = threadID
                    ...
        def run(self):
            ... task description
            ...

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
```

Choosing the right HPC granularity ?

With k single threaded CPUs, is it more efficient:

- to run k simple jobs in parallel ?
- to run in parallel a smaller number of complex jobs ?
- to align the numbers of parallel jobs and tasks to k ?
- to start more parallel threads than available cores ?
- to feed k parallel workers with a shared tasks queue ?
- to split the HPC program in several separate run executables ?

Gaia-80 November 2016 ranking record



```
Results with 118 cores on gaia-80, seed=105
model: Obj, equiobjectives, ('beta', 'variable', None)
Tue Nov 22 07:47:17 2016
perfTab: 625.210357 sec., 5053959984 bytes
*----- show short --------------*
Instance name       : random3ObjectivesPerfTab_mp
# Actions           : 2500000
# Criteria          : 21
Sorting by          : 500-Tiling
Ordering strategy   : average
Local ranking rule : Copeland
# Components        : 200499
Minimal size        : 1
Maximal order       : 543
Average order       : 12.5
Fill rate           : 0.008%
*-- Constructor run times (in sec.) --*
# Threads           : 118
Total time          : 10604.06302
QuantilesSorting    : 6221.00685
Preordering         : 854.70296
Decomposing         : 3528.33810
Ordering            : 0.00007
0 15:37:30 rbisdorff@access(gaia-cluster) Gaia80 $
```

10604 sec. = 176 min. 44 sec. = 2h. 56 min. 44 sec. ≈ 3h.

HPC performance measurements HPC shool 2017

digraph	standard model			sparse model		
order	#c.	t_g sec.	τ_g	#c.	t_{bg}	τ_{bg}
1 000	118	6"	+0.88	8	4"	+0.83
2 000	118	15"	+0.88	8	9"	+0.83
2 500	118	27"	+0.88	8	14"	+0.83
10 000				118	13"	
15 000				118	22"	
25 000				118	39"	
50 000				118	2'	
100 000	(size	=	10^{10})	118	5'	(fill rate = 0.223%)
1 000 000	(size	=	10^{12})	118	1h17'	(fill rate = 0.049%)
1 732 051	(size	=	3×10^{12})	118	3h09'	(fill rate = 0.038%)
2 236 068	(size	=	5×10^{12})	118	4h50'	(fill rate = 0.032%)

Legend:

- #c. = number of cores;
- g: standard outranking digraph, bg: the sparse outranking digraph;
- t_g, resp. t_{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.

New performance measurements Spring 2018

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

These run times are achieved both:

- on the Iris -skylake nodes with 28 cores,
- on the 3TB -bigmem Gaia-183 node with 64 cores, and
- running cythonized python modules in an Intel compiled virtual Python 3.6.5 environment [GCC Intel(R) 17.0.1 –enable-optimizations c++ 6.3 mode] on Debian 8 linux.

Successful actions for enhancing the performances - 1

Algorithmic refinements: The pre-ranking quantiles sorting algorithm may be further optimized, reducing considerably the fill rate of the sparse outranking digraphs.

```
>>> # same performance tableau t
>>> bg = PreRankedOutrankingDigraph(t,\
         quantiles=5,LowerClosed=False,\
         quantilesOrderingStrategy='optimal')
>>> bg
 *----- Object instance description ------*
  Instance class : PreRankedOutrankingDigraph
  Instance name  : randomperftab_pr
  # Actions      : 50
  # Criteria     : 7
  Sorting by        : 5-Tiling
  Ordering strategy : optimal
  Ranking rule      : Copeland
  # Components      : 37              <<=====
  Minimal order     : 1
  Maximal order     : 4               <<=====
  Average order     : 1.4             <<=====
  fill rate         : 1.633%          <<=====
  Correlation       : +0.706
```

Optimal quantiles ordering criteria when x sorted into quantile classes $]q^{k-1}, q^{k+r}]$, where $r = 0, 1, ..$:

1) *average of low and high* limits: $q^{k-1} + q^{k+r}$,

2) *high* quantile limit: q^{k+r},

3) average outranking low and high limits: $r(q^{k-1} \prec x) + r(q^{k+r} \prec x)$, and

4) outranking high limit: $r(q^{k+r} \prec x)$.

Optimal quantiles ordering with cPython

```
>>> tp1 = Random3ObjectivesPerformanceTableau(\
          numberOfActions=5000,numberOfCriteria=21)
>>> tp2 = tp1.convert2BigData()
>>> from cSparseIntegerOutrankingDigraphs import *
>>> qr = cQuantilesRankingDigraph(tp2,5,Threading=True,nbrOfCPUs=8)
>>> qr
*----- Object instance description --------------*
Instance class     : cQuantilesRankingDigraph
Instance name      : bgd_random3ObjectivesPerfTab_mp
# Actions          : 5000  # Criteria        : 21
Sorting by         : 5-Tiling
Ordering strategy  : optimal           <<=====
Ranking rule       : Copeland
# Components       : 4632              <<=====
Minimal order      : 1
Maximal order      : 9
Average order      : 1.1
fill rate          : 0.004%            <<=====
----  Constructor run times (in sec.) ----
# Threads          : 8
Total time         : 1.03086           <<=====
QuantilesSorting   : 0.69641
q-tiles ordering   : 0.04400
local ranking      : 0.29038
```

Successful actions for enhancing the performances - 2

- **Algorithmic refinements**: The pre-ranking quantiles sorting algorithm was further optimized, reducing considerably the fill rate of the sparse outranking digraphs;
- **Reducing the size of python data objects**: A special bigData performance tableau model with integer dictionary keys and float evaluations is used for optimized Cython and C compiler variable typing;

Reducing the size of python data objects

tp1 Standard Random 3 Objectives performance tableau instance with 5000 decision actions and 21 performance criteria: $size(tp1) = 3\,602\,132$ Bytes.

tp2 Same BigData Random 3 Objectives performance tableau instance: $size(tp2) = 1\,398\,365$ Bytes.

bg1 Standard pre-ranked outranking digraph instance generated from tp1: $size(bg1) = 9\,471\,896$ Bytes.

bg2 BigData pre-ranked outranking digraph instance generated from tp2: $size(bg2) = 1\,791\,755$ Bytes.

Efficient Cython inline function declaration with variable typing

```
cdef inline int _localConcordance(float d, float ind, float wp, float p):
    """ None = -1.0 """
    if p > -1.0:
        if   d <= -p:
            return -1
        elif ind > -1.0:
            if d >= -ind:
                return 1
            else:
                return 0
        elif wp > -1.0:
            if d > -wp:
                return 1
            else:
                return 0
        else:
            if d < 0.0:
                return -1
            else:
                return 1
    else:
```

Successful actions for enhancing the performances - 3

- **Algorithmic refinements**: The pre-ranking quantiles sorting algorithm was further optimized, reducing considerably the fill rate of the sparse outranking digraphs;
- **Reducing the size of python data objects**: A special bigData performance tableau model with integer dictionary keys and float evaluations is used for optimized Cython and C compiler variable typing;
- **Efficient sharing of static data**: Global python variables allow to efficiently communicate static data objects to parallel threads when using -bigmem nodes;

Successful actions for enhancing the performances - 4

- **Algorithmic refinements**: The pre-ranking quantiles sorting algorithm was further optimized, reducing considerably the fill rate of the sparse outranking digraphs;
- **Reducing the size of python data objects**: A special bigData performance tableau model with integer dictionary keys and float evaluations is used for optimized Cython and C compiler variable typing;
- **Efficient sharing of static data**: Global python variables allow to efficiently communicate static object data to parallel threads when using -bigmem nodes;
- **Using a multiprocessing tasks queue**: Sorting tasks in decreasing durations and using an automatic multithreading mechanism (see the *multiprocessing python3 documentation*)

Using a multiprocessing tasks queue

```
with TemporaryDirectory(dir=tempDir) as tempDirName:
    ## tasks queue and workers launching
    NUMBER_OF_WORKERS = nbrOfCPUs
    tasksIndex = [(i,len(decomposition[i][1])) for i in range(nc)]
    tasksIndex.sort(key=lambda pos: pos[1],reverse=True)
    TASKS = [(Comments,(pos[0],nc,tempDirName)) for pos in tasksIndex]
    task_queue = Queue()
    for task in TASKS:
        task_queue.put(task)
    for i in range(NUMBER_OF_WORKERS):
        Process(target=_worker,args=(task_queue,)).start()
    if Comments:
        print('started')
    for i in range(NUMBER_OF_WORKERS):
        task_queue.put('STOP')

    while active_children() != []:
        pass
    if Comments:
        print('Exit %d threads' % NUMBER_OF_WORKERS)
```

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.

Further documentation resources

The cythonized Python HPC modules are freely available under the cython directory in a Digraph3 working copy.
Tutorials and technical documentation + source code listings may be consulted on:

- https://digraph3.readthedocs.io/en/latest/