Algorithmic Decision Theory

Final course evaluation project

deadline Tuesday, June 28, 09:00

Maciej Żurad
maciej.zurad@gmail.com

1 What is apparently the best decision action in your problem from the environmental point of view, from the economic point of view, from the societal point of view, and from a global multi-objectives compromise point of view ?

All operations on data were done using `Digraph3`[1] Python3 package, which implements decision aid algorithms in the context of bipolarly-valued outranking approach.

Let us first take a look, at best decision from the point of view of each category. In order to see that, we have to first make sure to neglect all other criteria. To achieve that, we must first create a partial performance tableau from the original performance tableau. Program `code/bcr_categories.py` implements all operations needed to show Best Choice Recommendation from the environmental, economical and societal point of view. The output of this program was also saved to `code/bcr_categories.out`

We first find a set of criteria that belong to a certain category (e.g Economical) and create a partial performance tableau with only these criterias. We now create a Bipolar Outranking Digraph from that performance table and use Rubis[1, 2] solver to get a potential Best Choice Recommendation, which is depicted in Figure 1.

[1] Digraph3 is hosted at `https://github.com/rbisdorff/Digraph3`

```
from perfTabs import *
from outrankingDigraphs import BipolarOutrankingDigraph
pt = XMCDA2PerformanceTableau('project_2')
criterias = [c for c, val in pt.criteria.items() if 'Eco' in val['name']]
ppt = PartialPerformanceTableau(pt,criteriaSubset=criterias)
partial_digraph = BipolarOutrankingDigraph(ppt)
partial_digraph.showRubisBestChoiceRecommendation()
```

Figure 1: Rubis Best Choice Recommendation code-snippet

Outranking Digraph is based on the idea of outranking. For two alternatives a and b, a outranks b (aSb) if there exist a significant majority of criteria supporting that a is *at least* as good as y and no considerable counter-performance (*no veto*) is observed on any discordant criterion. For each Outranking Digraph a relation table was generated and saved in directory:
`report/figures/*_bipolar_adj_matrix.html`

1.1 From the Economical point of view

The output of the Rubis Best Choice Recommendation from the Economic point of view is the following: we get three potential BCRs, an alternative **a19** and a tuple of alternatives **(a31, a48)** with the same values. All the values share the same *dominance* at 33.33 and *absorbency* at -100.00. However, **a19** is a **better** candidate, because *covering* is maxed out at 100.00% and *determinateness* is higher as well at 66.67% compared to 50.00%. Another way to confirm that **a19** is in fact a better BCR candidate is to check the Condorcet and the Weak-Condorcet winners. Figure 2 is showing us how to calculate this. The Condorcet winner from Economical point of view is as expected **a19** and the Weak-Condorcet winners are **a19, a36, a48**, which also disqualifies previously seen **a31**. Therefore, **a19** is the best choice from Economical point of view!

```
weak_condorcet_winners = partial_digraph.weakCondorcetWinners()
condorcet_winners      = partial_digraph.condorcetWinners()
```

Figure 2: Condorcet and Weak Condorcet winner code-snippet

1.2 From the Environmental point of view

The output of the Rubis Choice Recommendation from the Environmental point of view is **a28, a41, a48**, where all alternatives have exactly the same values, when it comes to *irredundancy, independence, dominance, absorbency, covering* and *determinateness*. Condorcet winner from Environmental point of view gives us the same alternatives and the Weak-Condorcet winner gives us the same together with **a35**. Which leaves us with **3** Best Choice Recommendation from Environmental point of view: **a28, a41, a48**.

1.3 From the Societal point of view

Rubis Choice Recommendation from the Societal point of view bring the same situation as from Environmental point of view (section 1.2). The Rubis solver gives us three alternatives: **a18, a38, a49**. They also share exactly the same values and the Condorcet winners are also **a18, a38, a49**. Weak-Condorcet winners are: **a18, a38, a49, a12, a33, a45**. Another interesting thing we can try is to do Strictly Best Choice by computing the **codual** of the Outranking Digraph, that is a converse of a dual and after that run this graph through Rubis solver, which is presented in Figure 3. This gives a single BCR[2] which is a list and contains all Weak-Condorcet winners. However, when we look into the characteristic vector, which represents, whether an alternative may indeed be considered a best choice, we see that only Condorcet winners are determined and that sole Weak-Condorcet winners are not relevant, because they have a 0.0 value. In the end I chose only those 3: **a18, a38, a49** to be the BCRs from Societal point of view.

```
codual_partial = ~(-partial_digraph)                    # computes codual
codual_partial.showRubisBestChoiceRecommendation()
```

Figure 3: Codual of the Outranking Digraph and Strict Best Choice

1.4 From the global multi-objectives compromise point of view

We follow like in the previous examples, except that now we don't have first create Partial Performance Tableau like shown in Figure 1 and we directly call Rubis solver with the the Performance Tableau received when loading the data. We receive following: **a45, a20** and a tuple **(a38, a48)**. The Condorcet winners are also **a45, a20** and as we would expect the Weak-Condorcet winners are **a45, a20, a38, a48**. Figure 4 presents the values for alternatives **a20, a45** and figure 5 presents for **(a38, a48)**. We can see that **a45** has the best values, not only it has the highest *determinateness* but also *dominance*, which makes it the `Best Choice Recommendation` from the global multi-objectives compromise point of view.

```
* choice               : ['a20']
  +-irredundancy       : 100.00
  independence         : 100.00
  dominance            : 11.11
  absorbency           : -100.00
  covering (%)         : 100.00
  determinateness (%)  : 55.56
```

(a) Alternative 20

```
* choice               : ['a45']
  +-irredundancy       : 100.00
  independence         : 100.00
  dominance            : 15.87
  absorbency           : -100.00
  covering (%)         : 100.00
  determinateness (%)  : 57.94
```

(b) Alternative 45

Figure 4: BCR candidates - **a20** and **a45**

[2]BCR stands for Best Choice Recommendation

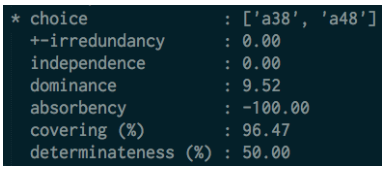

Figure 5: BCR candidates - **a38** and **a48**

All the output to calculate BCR from the global multi-objective compromise point of view was saved `code/bcr_multi_objective.out` and can be generated again using `code/bcr_multi_objective.py`.

2 What are the five potential best compromise choice candidates in your problem ?

Section 1.4 described BCR from the multi-objective point of view. We saw there, 4 Best Choice Recommendations that Rubis solver offered, from which I chose a single one (**a45**). To choose 5, we can now simply choose those 4 and we will be missing only last. To find the 5^{th}, we will use a performance heatmap, presented in figure 7. We can generate this performance heatmap very easily. Instead of showing it in browser, I directly save it on a disk using this code snippet depicted in Figure 6. This heatmap is by default ranked with a Copeland ranking rule, which will be described later on (section 3.1). If we took 4 BCRs that we defined previously from the Heatmap, then we would take from row 1,2,3 and 6. A common sense would tell us to take the next best in this ranking, which would be in row 4 (**a35**). We will see later that a different ranking rule would give us a different result. Therefore, the five potential best compromise choice candidates are **a45, a20, a38, a48** and **a35**.

```
def fwrite(string, file):
  with open(file, 'w') as f:
    f.write(string)

make_path = lambda x: os.path.join('..', 'report', 'figures', x.lower())

pt = XMCDA2PerformanceTableau('project_2')
html_heatmap = pt.htmlPerformanceHeatmap()
fwrite(html_heatmap, make_path("full_heatmap.html"))
```

Figure 6: Performance Heatmap generation and saving

3 How would you rank your set of alternatives?

Ranking is a very tricky process. That is because an Outranking Digraph does not usually present a linear ordering. Pairwise relations do not posses transitive

criteria	g01	g02	g04	g05	g07	g09	g03	g06	g08	g10	g11	g12	g13
weights	21.00	21.00	21.00	21.00	21.00	21.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00
a20	49.98	56.25	84.01	77.02	49.95	80.55	77.24	51.48	79.86	70.88	91.16	69.39	67.93
a45	28.16	65.83	62.87	73.96	81.50	82.44	81.84	65.39	90.43	65.86	90.53	77.64	70.18
a38	64.50	48.54	55.66	47.01	87.57	76.02	83.05	80.28	92.21	80.37	57.63	62.43	NA
a35	74.94	68.82	78.26	81.87	89.04	62.60	47.68	62.06	25.36	34.36	54.17	61.27	32.56
a41	57.59	58.44	72.70	54.99	92.47	93.40	57.97	28.40	42.10	19.26	75.18	73.48	48.81
a48	87.25	87.64	61.90	59.66	86.83	93.45	37.03	21.15	25.32	35.85	33.83	20.05	28.40
a02	41.09	59.55	71.44	45.56	73.27	47.94	76.65	70.50	71.87	63.57	57.38	64.27	81.27
a14	60.99	38.41	30.15	63.39	69.94	27.16	61.18	67.29	66.15	65.58	78.81	90.62	82.89
a29	46.55	31.63	95.54	40.78	52.92	49.93	57.38	74.71	43.88	83.70	89.09	44.83	75.26
a28	60.72	17.36	49.73	89.32	68.29	96.26	61.15	27.93	76.87	48.10	51.86	18.21	25.21
a27	50.30	78.66	62.12	15.92	41.87	NA	44.56	76.59	59.57	78.32	80.35	59.00	86.50
a49	66.75	48.80	60.15	22.64	30.18	39.78	NA	78.75	54.97	83.62	78.29	78.42	85.93
a01	65.59	72.95	33.97	48.56	52.09	82.27	34.78	26.29	70.55	55.17	55.41	44.04	49.58
a31	57.53	NA	72.87	35.39	49.79	62.29	55.57	64.86	24.59	47.56	61.96	67.86	38.67
a34	28.10	54.84	31.00	77.99	83.20	44.58	62.32	81.05	57.60	29.89	54.22	65.46	65.32
a06	36.54	NA	58.08	47.23	48.79	63.38	54.19	55.66	67.72	64.56	75.88	55.64	66.42
a08	77.57	NA	48.17	72.18	58.99	66.63	50.11	48.69	52.07	50.89	11.90	45.62	55.07
a12	87.87	17.47	67.73	47.30	20.50	25.30	81.10	82.29	79.37	51.32	NA	89.22	56.22
a33	52.72	39.82	43.47	62.66	35.25	55.70	74.85	77.11	73.83	NA	33.28	76.58	72.84
a13	49.75	64.46	43.78	60.42	58.93	66.79	33.89	47.00	41.19	84.27	30.31	43.21	75.96
a50	64.62	59.55	62.70	54.35	56.64	64.41	52.14	62.68	56.23	10.67	25.35	45.51	31.87
a11	71.57	NA	39.76	NA	46.83	51.93	78.99	46.65	54.56	51.42	NA	37.92	76.79
a09	15.99	8.36	81.56	69.01	47.73	11.31	86.82	66.22	69.62	61.77	89.16	69.38	64.10
a17	13.91	17.10	40.85	72.51	74.04	72.91	69.20	22.24	NA	50.97	54.05	39.15	48.48
a22	82.22	60.50	59.23	26.79	55.15	19.62	75.97	57.86	55.91	30.71	67.96	37.85	70.71
a18	NA	37.42	6.53	38.04	66.76	50.62	57.09	79.48	NA	73.50	82.26	84.54	71.99
a10	70.91	27.40	14.13	31.29	90.23	62.97	50.80	52.38	47.49	62.04	31.00	56.30	64.00
a32	47.00	20.73	44.75	74.28	50.26	69.60	55.42	43.05	53.25	NA	65.54	30.35	51.05
a05	46.31	34.98	49.37	57.62	55.33	54.23	33.66	35.33	57.48	45.56	58.53	54.86	NA
a25	51.47	71.20	52.64	NA	77.33	61.51	NA	14.08	10.89	31.14	35.82	17.02	26.40
a47	50.88	35.10	57.57	50.99	65.47	70.62	21.28	18.41	35.08	39.16	64.20	53.05	26.47
a43	39.11	38.70	35.99	55.65	79.15	63.03	22.53	37.02	33.21	42.58	64.49	19.53	70.03
a36	71.41	86.49	82.84	47.53	9.74	16.23	44.14	70.74	46.78	33.12	47.35	62.04	44.92
a42	25.66	51.34	17.15	45.69	44.11	39.29	47.71	42.56	59.66	67.02	77.38	82.61	74.05
a26	17.73	NA	25.03	55.83	48.11	85.80	36.45	38.71	28.81	51.19	40.57	NA	30.84
a37	36.96	NA	15.52	57.49	53.98	47.85	57.05	50.02	59.72	30.58	16.97	76.32	60.10
a04	51.84	36.84	60.96	44.19	59.56	48.10	28.64	11.92	22.85	28.38	44.18	31.09	52.34
a19	89.57	68.03	92.47	44.06	11.38	25.88	31.41	20.06	6.09	32.24	29.36	32.58	22.93
a46	61.54	40.73	45.38	59.16	54.01	39.81	32.61	42.26	11.34	15.17	53.58	13.22	21.17
a40	14.88	35.67	21.85	47.83	84.44	38.05	31.30	49.82	32.20	22.43	31.41	43.51	23.13
a07	37.42	NA	58.02	28.02	30.64	61.10	28.67	39.21	40.04	55.00	28.92	17.39	48.14
a39	18.69	16.11	39.46	45.45	48.30	26.28	70.84	43.43	44.74	60.50	40.44	46.06	40.83
a16	60.14	71.34	59.23	57.02	12.40	39.87	27.09	24.35	21.77	29.69	7.54	10.45	27.62
a44	34.64	44.98	49.46	44.85	33.20	12.90	63.20	43.63	33.00	38.82	62.59	29.23	29.22
a21	39.81	7.71	61.27	18.76	56.35	5.45	37.78	51.30	48.94	39.35	34.58	47.94	47.81
a24	40.64	37.92	23.18	5.67	56.64	30.40	55.82	53.16	52.37	30.53	25.48	15.26	31.94
a15	21.02	65.76	62.99	46.69	16.52	36.38	40.11	29.04	36.69	21.08	48.75	8.80	6.77
a30	20.21	14.97	59.23	21.20	33.01	36.56	20.30	39.03	30.57	23.90	34.07	33.45	22.68
a03	37.67	24.27	9.45	31.37	14.61	38.12	38.99	28.28	44.19	12.90	33.95	19.16	25.86
a23	NA	16.96	46.08	4.57	21.38	17.18	19.28	25.40	10.69	38.67	47.81	29.46	24.99

Color legend:

quantile	0.14%	0.29%	0.43%	0.57%	0.71%	0.86%	1.00%

Figure 7: Performance Heatmap with Copeland ranking rule

relation. That's because an alternative a can outrank alternative b and also b can outrank c. However, that does not imply that an alternative a would **outrank** c. To see, how much of a transitivity is preserved in an outranking digraph, we can calculate Transitivity Degree metric, which is the ratio of outranking arcs over the number of transitive closure arcs in digraph g. We can see that if this value is less than 1.0, than we have a digraph that does have some non-transitive relations, which will not be a rare case.

Another problem to correctly rank is the existence of cycles inside a digraph. For instance, if alternative a outranks b and consequently b outranks c and then c outranks a, then we have a cycle. Any potential linear ordering of these 3 alternatives will contradict itself. Figure 8 presents a code-snippet to calculate Transitivity Degree as well as number of cycles. In my case, the Transitivity Degree was **0.6012** and **33** cycles were detected.

The python script with all the code used in this section is called `code/ranking_and_sorting.py` and the output was saved to `code/ranking_and_sorting.out`

In order to deal with these problems, several heuristic have been developed and we will take a look at them.

```
from perfTabs import *
from outrankingDigraphs import BipolarOutrankingDigraph
pt = XMCDA2PerformanceTableau('project_2')
full_digraph = BipolarOutrankingDigraph(pt)
full_digraph.computeTransitivityDegree()
full_digraph.computeChordlessCircuits()
full_digraph.showChordlessCircuits()
```

Figure 8: Computing Transitivity Degree and # of cycles

3.1 Copeland ranking

Copeland ranking is very intuitive and computes for each alternative a score resulting from difference between its out-degree and in-degree. Figure 9 shows how to compute ranking using this rule.

```
from linearOrders import CopelandOrder
cop = CopelandOrder(full_digraph)
cop.showRanking()
cop_corr = full_digraph.computeOrdinalCorrelation(cop)
print("Fitness of Copeland's ranking: %.3f" % cop_corr['correlation'])
```

Figure 9: Computing Copeland ranking and its fitness

I received this ranking from Copeland ranking:

[a20, a45, a38, a35, a41, a48, a02, a14, a29, a28, a27, a49, a01, a31, a34, a06, a08, a12, a33, a13, a50, a11, a09, a17, a22, a18, a10, a32, a05, a25, a47, a43, a36, a42, a26, a37, a04, a19, a46, a40, a07, a39,

a16, a44, a21, a24, a15, a30, a03, a23]

and the following fitness value: **0.868**

3.2 Net-Flows ranking

Net-Flows heuristic is using the values given directly by the outranking digraph g. For every alternative a, we iterate over all other alternatives b and compute sum of differences between the outranking characteristic $r(aSb)$ and $r(bSa)$.

$$netFlow(x) = \sum_{y \in Y} (r(x\ S\ y) - r(y\ S\ x))$$

```
from linearOrders import NetFlowsOrder
nf = NetFlowsOrder(full_digraph)
print('Net flow values for each alternative')

for n,va in enumerate(nf.netFlows):
        print("{0} {2} -> {1:.2f}".format(n+1, *va))

nf.showRanking()
nf_corr = full_digraph.computeOrdinalCorrelation(nf)
print("Fitness of Net-flows ranking: %.3f" % nf_corr['correlation'])
```

Figure 10: Computing Net-Flows ranking and its fitness

n	alternative	netFlow
1	a45	7141.27
2	a20	6511.11
3	a38	5971.43
4	a41	5415.87
5	a35	4933.33
6	a02	3650.79
7	a48	3604.76
8	a14	3509.52
9	a29	3246.03
10	a27	3068.25
...	...	...

Table 1: NetFlows ranking with the netFlow values for the first top 10 alternatives

I got the following ranking using Net-Flows heuristic:

[a45, a20, a38, a41, a35, a02, a48, a14, a29, a27, a49, a28, a34, a33, a09, a01, a06, a12, a13, a31, a18, a11, a50, a08, a17, a10, a22, a32, a05, a47, a42, a43, a36, a25, a26, a04, a37, a40, a46, a07, a19, a39,

a44, a21, a16, a24, a30, a15, a23, a03]

and the following fitness value: **0.877**

3.3 Kohler's ranking

Kohler's ranking is also known as **ranking-by-choosing**. The algorithm is fairly easy and can be described in few steps. Step 1, we take the Relation Table from the outranking graph (basically the previously described adjacency matrix). Step 2, for each row, we find it's minimum and we select a row, where this minimum is maximum among other rows. Step 3, we place this alternative at rank highest possible rank. Step 4, we delete corresponding row and column of the last alternative and we go to Step 1 unless table is empty.

```
from linearOrders import KohlerOrder
ko = KohlerOrder(full_digraph)
ko.showRanking()
ko_corr = full_digraph.computeOrdinalCorrelation(ko)
print("Fitness of Kohler's ranking: %.3f" % ko_corr['correlation'])
```

Figure 11: Computing Kohler's ranking and its fitness

I received this ranking from Kohler's ranking:

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

and the following fitness value: **0.863**

3.4 Tideman's Ranked-Pairs

Ranked-Pairs is based on a construction, which incrementally constructs a linear order simultaneously avoiding any cycles on the fly.

```
from linearOrders import RankedPairsOrder
rp = RankedPairsOrder(full_digraph)
rp.showRanking()
rp_corr = full_digraph.computeOrdinalCorrelation(rp)
print("Fitness of Tideman's Ranked—Pairs rule ranking: %.3f" % rp_corr['correlation'])
```

Figure 12: Computing Tideman's Ranked-Pairs ranking and its fitness

I received the following ranking from Tideman's Ranked-Pairs:

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

and the following fitness value: **0.893**

3.5 Conclusions on Ranking

As we can see different Ranking methods give us quite different results. They also have higher or lower fitness values. Highest fitness values was achieved by Tideman's Ranked-Pairs ranking with value 0.893, shortly after that we have Net-Flows ranking with value 0.877 and at the end we have Copeland's with 0.868 and Kohler's with 0.863.

Another important thing is to remember that Copeland and Net-Flows are very fast algorithms, where Kohler's and Tideman's are slower and do not scale so well. There also exist optimal ranking algorithms such as Kemeny and Slater. However, these two algorithms are exponential in complexity and running them on a graph bigger than 15 seems infeasible.

In the end my final ranking is Tideman's ranking, because it exhibited the highest fitness value.

4 How would you sort your alternatives into performance deciles ?

In order to sort our alternatives into performance deciles, we have to use Sorting Digraphs. QuantilesSortingDigraphs are subclass of SortingDigraphs, that work well for large graphs. Figure 13 shows code-snippet, which allows to compute that. Figure 14 presents our graph. The graph shows only 8 rows, because 2 deciles are empty. There is no alternative, which is in the top and bottom deciles ([0.9 - 1.0] and [0.0 - 0.1]). Output of the program described in figure 13 was saved as well to `code/ranking_and_sorting.out`. From the graph, we can clearly recognize our BCRs candidates and Condorcet winners, which as we would expect are in the top row, together with the 5^{th} BCR that we added in section 2

```
from sortingDigraphs import QuantilesSortingDigraph
qs = QuantilesSortingDigraph(pt,limitingQuantiles=10)
qs.showSorting()
qs.showQuantileOrdering()
qs.exportGraphViz(make_path("sorted_deciles_graph"))
```

Figure 13: Computing Quantile Ordering for 10 quantiles (deciles)

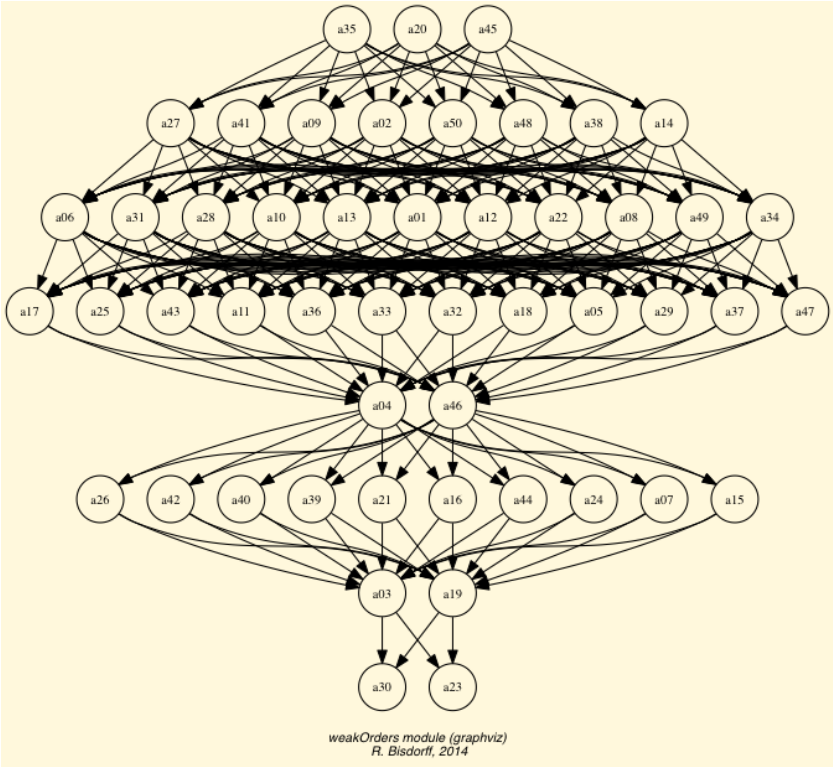

Figure 14: Decile Sorting Digraph

References

[1] Raymond Bisdorff, Patrick Meyer, and Marc Roubens. "R UBIS: a bipolar-valued outranking method for the choice problem". In: *4OR* 6.2 (2008), pp. 143–165.

[2] Raymond Bisdorff, Marc Pirlot, and Marc Roubens. "Choices and kernels in bipolar valued digraphs". In: *European Journal of Operational Research* 175.1 (2006), pp. 155–170.