FlowSort parameters elicitation based on classification examples

CoDE-SMG – Technical Report Series

Dimitri Van Assche, Yves De Smet

CoDE-SMG – Technical Report Series

Technical Report No.
TR/SMG/2014-003
June 2014

CoDE-SMG – Technical Report Series
ISSN 2030-6296

Published by:
CoDE-SMG, CP 210/01
Université Libre de Bruxelles
Bvd du Triomphe
1050 Ixelles, Belgium

Technical report number TR/SMG/2014-003

FlowSort parameters elicitation based on classification examples

CoDE-SMG – Technical Report Series

Dimitri Van Assche dimitri.van.assche@ulb.ac.be
Yves De Smet yves.de.smet@ulb.ac.be
CoDE-SMG, Université Libre de Bruxelles, Brussels, Belgium

June 2014

FlowSort parameters elicitation based on classification examples - Technical report - June 2014

Dimitri Van Assche[a], Yves De Smet[a]

[a] *Computer & Decision Engineering Department (CoDE), École polytechnique de Bruxelles, Université libre de Bruxelles, Avenue Franklin D. Roosevelt 50, CP210/01, B-1050 Brussels, Belgium*

Abstract

In multi-criteria sorting methods, it is often difficult for decision makers to precisely define their preferences. It is even harder to express them into parameters values. The idea of this work is to automatically find the parameters of a sorting model using classification examples. The sorting method we are working with is FlowSort, which is based on the PROMETHEE methodology. Starting with an evaluation table and known allocations, we propose a heuristic based on a genetic algorithm (GA) to identify the weights, indifference and preference thresholds but also profiles characterizing the categories. The parameters of the GA will be optimized using the iRace procedure. We illustrate both the performances of the algorithm and the quality of the solutions on three standard datasets.

Keywords: preference learning, PROMETHEE, FlowSort, genetic algorithm

1. Introduction

Multi-criteria decision aid (MCDA) has been an active research field for more than 40 years. In this context, possible decisions are simultaneously evaluated on multiple conflicting criteria. For instance, in the common example of buying a new car, you typically try to minimize the cost and consumption while maximizing performances, comfort, etc. Obviously no real car would be the best on all those criteria. Therefore, the notion of optimal solution is most of time replaced by the idea of compromise solution. [1]

Different approaches have been proposed to tackle this kind of problem, among which we can distinguish three main families [2]: multi-attribute utility theory (MAUT) [3], outranking methods [4], and interactive methods [5].

In this article, we decided to work in the context of outranking methods. Those are based on pairwise comparisons of the alternatives. Among the best-known we may cite ELECTRE [6] and PROMETHEE [7].

Usually, three main multi-criteria problems are considered: ranking, sorting and choosing [8]. In the ranking problematic, alternatives are ranked from the best to the

Email addresses: dvassche@ulb.ac.be (Dimitri Van Assche), yvdesmet@ulb.ac.be (Yves De Smet)

worst ones on the basis of a complete or partial order. For instance, the academic institutions' ranking[1] on the basis of different scientific indicators. The sorting problematic is dedicated to the assignment of alternatives into predefined categories. For instance, sorting countries into risk categories on the basis of economical, financial and political indicators. Finally, the choice problematic addresses situations where a subset of the best possible alternatives has to be determined, such as identifying the best candidate for a given position in a hiring process.

Let us point out that different methods have been developed to sort in a multi-criteria context. Among them, we may cite UTADIS [9], PROAFTN [10], ELECTRE TRI [11] and FlowSort [12].

The traditional approach used to deal with sorting is to directly ask the Decision Maker to give central, or limiting, profiles defining the categories, but also weights and preference parameters for each criterion. But obtaining such information is often a cognitive complex task. Most of the time, the Decision Maker does not know himself what are exactly his preferences. Furthermore, they are hard to correctly formalize into weights, thresholds and profiles.

Henceforth, we will limit ourselves to the sorting method developed within the PROMETHEE methodology: FlowSort. In this paper, we will consider the problem inside out. Based on an existing categorization of alternatives, we will try to find the parameters of the sorting model that replicate the best this information. This will be formalized as an optimization problem that will be solved using a genetic algorithm.

Let us point out that similar works have been proposed in the context of ELECTRE TRI [13], [14] and [15]. We may also quote MR-Sort from Sobrie [16] which aims to find the parameters of a simplified version of ELECTRE TRI. More recently, another contribution from Zheng considers the problem of learning the weights of the criteria [17]. To the best of our knowledge, no article has already addressed this question for FlowSort.

In section 2, we introduce PROMETHEE and FlowSort. In the third section, we describe the genetic algorithm we will use to solve the optimization problem. Then, in the forth section, we apply the algorithm on different standard datasets and compare the results with other methods. Finally, we investigate the performances of the algorithm with respect to the parameters tuning.

2. Promethee and FlowSort

In this section, we present the main steps of PROMETHEE[2] I and II as well as FlowSort. We refer the interested reader to [18] for a detailed description of these approaches.

Let $A = \{a_1, a_2 ..., a_n\}$ be the set of n alternatives and let $F = \{f_1, f_2 ..., f_q\}$ be the set of q criteria. The evaluation of alternative a_i for criterion l will be denoted by a real value $f_l(a_i)$.

For each pair of alternatives, let's compute $d_l(a_i, a_j)$, the difference of a_i over a_j on criterion l.

$$d_l(a_i, a_j) = f_l(a_i) - f_l(a_j) \tag{1}$$

[1]http://www.shanghairanking.com/

[2]Preference Ranking Organization METHod for Enrichment of Evaluations

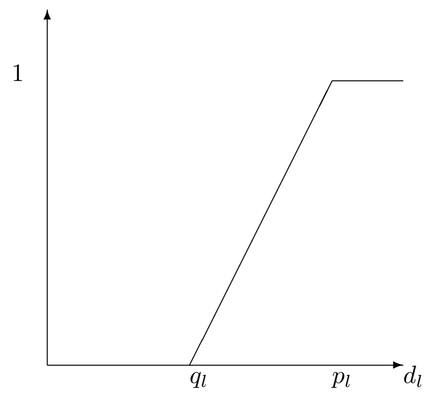

Figure 1: Linear preference function

A preference function, denoted P_l, is associated to each criterion l. This function transforms the difference of alternatives' evaluations $d_l(a_i, a_j)$ into a preference degree of the first alternative over the second one for criterion h. Without loss of generality, we consider that the criteria have to be maximized. P_l is defined as follow:

$$P_l : \mathbb{R} \rightarrow [0,1] : x \rightarrow P_l(x) \tag{2}$$

such that:

- $\forall x \in \mathbb{R}^- : P_l(x) = 0$,
- $\forall x, y \in \mathbb{R}_0^+ : x \leq y \implies P_l(x) \leq P_l(y)$

There are different kinds of preference functions. In this article, we consider only the linear preference function (see figure 1) which is characterized by two parameters: an indifference and a preference threshold: q_l, p_l.

$$\pi_l(a_i, a_j) = P_l[d_l(a_i, a_j)] = \begin{cases} 0 & \text{if } d_l(a_i, a_j) \leq q_l \\ \frac{d_l(a_i,a_j) - q_l}{p_l - q_l} & \text{if } q_l < d_l(a_i, a_j) \leq p_l \\ 1 & \text{if } p_l < d_l(a_i, a_j) \end{cases} \tag{3}$$

Once $\pi_l(a_i, a_j)$ has been computed for all pairs of alternatives, we may define the aggregated preference degree of alternative a_i over a_j using the weights w_l associated to each criterion l. Weights are assumed to be positive and normalized.

$$\pi(a_i, a_j) = \sum_{l=1}^{q} w_l . \pi_l(a_i, a_j) \tag{4}$$

The last step consists in calculating the positive flow score denoted $\phi_A^+(a_i)$ and the negative flow score denoted $\phi_A^-(a_i)$:

$$\phi_A^+(a_i) = \frac{1}{n-1} \sum_{x \in A} \pi(a_i, x) \tag{5}$$

$$\phi_A^-(a_i) = \frac{1}{n-1} \sum_{x \in A} \pi(x, a_i) \tag{6}$$

We define the net flow score of a_i as the difference between the positive flow and negative flows of a_i:

$$\phi_A(a_i) = \phi_A^+(a_i) - \phi_A^-(a_i) \tag{7}$$

The PROMETHEE I ranking is obtained as the intersection of the rankings induced by ϕ^+ and ϕ^-. For an interpretation of the net flow scores, the interested reader is referred to [19]. Finally, a complete order, called PROMETHEE II, can be derived from the order of the net flow scores.

Based on PROMETHEE, FlowSort has been developed to address sorting problems [20]. Let $C = \{c_1, c_2..., c_k\}$ be a set of k ordered categories. We assume that $c_i \succ c_{i+1}$: c_i is preferred to c_{i+1}. Therefore C_1 is the best category and C_k the worst one.

The categories are assumed to be represented by limit or central profiles. On the one hand, the idea of the limiting profiles is to define couples of values for each criterion, defining the lower and upper bound for the considered category. Let us note that the profile defining the upper bound of category c_i is the same as the one defining the lower bound of category c_{i+1}. On the other hand, central profiles are defined using a single value for each criterion. This value represents a kind of mean profile for the category. A common property is that the profiles of each category must dominate the profiles of the ones they are preferred to. In this work, we have chosen to work with central profiles.

Let's define $R = \{r_1, r_2..., r_k\}$, the set of central profiles representing the k categories. To identify the category of an alternative a_i, we define the subset $R_i = R \cup \{a_i\}$. Then, for each element x in the subset R_i, we compute its net flow score $\phi_{R_i}(x)$.

As in the nearest neighbor procedure, the category of alternative a_i is the one such that the profile has its net flow score the closest to the net flow score of a_i. More formally:

$$l^*(a_i) = \underset{l=1,2...,k}{argmin} |\phi_{R_i}(a_i) - \phi_{R_i}(r_l)| \tag{8}$$

For a detailed description of FlowSort, we refer the interested reader to [20].

Let us note that $(3 + k).q$ parameters have to be identified in order to instantiate FlowSort:

- $k.q$ values for the central profiles;
- $3.q$ values for the weights, indifference and preference thresholds.

3. Algorithm

As a first approach, we propose to use a genetic algorithm (GA) to address the elicitation problem. The use of a metaheuristic is motivated by the non linear nature of preference functions. In this work, we have implemented the algorithms using jMetal; a Java Framework that contains already a lot of well-known optimization algorithms [21, 22]. From our point of view, the added value of jMetal is the design of the optimization problem. It allows to be very flexible with the algorithm. It is indeed easy to compare different mutation and crossover operators, etc. The pseudo code of the algorithms described in this section is available in the appendices of this article.

Additionally, we used iRace[3] which is a tool that implements the iterated racing procedure, in order to fine tune the parameters of the GA [23] [24].

Each solution contains, for each criterion l:

- a weight w_l;
- an indifference threshold q_l;
- a preference threshold p_l;
- the k values r_{hl}, $h \in \{1, ..., k\}$ defining the categories profile for each criterion.

For convenience, we assume that all the evaluations of the alternatives have been normalized on a scale $[0, 1]$.

The structure of our genetic algorithm is quite common (see algorithm 1). We begin by generating an initial population. We set a limit on the number of evaluations denoted *max_eval*. This will be used as a stopping criterion. Obviously, the algorithm also stops when a perfect solution is found. A new population is created by selecting parents (with a selection operator based on quality) and creating offspring with particular crossover and mutation operators. Let us note that, for each generation, the two best solutions are kept in the new population.

A major step is to define an indicator that will quantify the quality of a given solution. Therefore, we have chosen to use the L_1 distance between the categorization given by FlowSort with the actual parametrization and the categorization given as input:

$$f(s) = \sum_{a \in A} |l_s^*(a) - l(a)| \tag{9}$$

where $l(a)$ represents the known categorization of the alternative a and $l_s^*(a)$ the categorization using the parameters of solution s.

In order to drive the search process to interesting solutions we define six indicators for each category. Let us denote them U_l, W_l and O_l for each category l:

- W_l represents the percentage of alternatives from the category l that are sorted in the same category using FlowSort;
- U_l represents the percentage of alternatives from the category l that are sorted in a lower category using FlowSort;
- O_l represents the percentage of alternatives from category l that are sorted in an upper category using FlowSort.

In a similar way, we define U_l^*, W_l^* and O_l^* but this time using the categorization using FlowSort as the benchmark:

- W_l^* will represent the percentage of alternatives sorted in l that are really categorized in l;

[3]Iterated Race for Automatic Algorithm Configuration

- U_l^* represents the percentage of alternatives sorted in l that are undercategorized, meaning their real category is higher than l;

- O_l^* represents the percentage of alternatives sorted in l that are overcategorized, meaning their real category is lower than l.

The initialization of the algorithm, see algorithm 2, is set up as follow:

- weights w_l are generated by using uniform random values between 0 and 1;

- indifference thresholds q_l are generated by using random values between 0 and 0.5;

- preference thresholds p_l are generated by using random values between q_l and 1;

- for the categories' profiles, the range is divided in *nb_categories* equal parts. The value of a profile l on a criterion j, denoted r_lj, is chosen randomly between the value of the previous profile, or 0 if there is no, and the upper bound of the range.

The implementation follows a traditional GA scheme. Parents are selected using a binary roulette wheel selection based on the quality indicator. Depending on a crossover probability, two children are built based on linear combinations (each component depending on gene crossover probability).

The mutation step is based on the six indicators defined before. First, if the assignment correctness of a category is high, the mutation probability of its profile will be low (and reversely). Secondly, in case of unbalanced assignment mistakes, the profiles will be mutated:

- in ascending order if alternatives are too much overcategorized;

- in descending order if alternatives are too much undercategorized.

When a new solution is built, a correction is applied to ensure that profiles respect the dominance condition.

We refer the interested reader to the appendices for a complete presentation of the algorithms.

4. Results

In order to investigate the performances of our algorithm we have chosen to test it on three different benchmark datasets. The datasets come from the website of Marburg University[4]. These originally come from the UCI repository[5] and the WEKA machine learning toolbox[6]. The alternatives with missing values have been removed, ordinal values are transformed to numeric one, and the data have been normalized between 0 and 1. They are monotone learning datasets, each criteria has to be maximized. You can find the properties of the datasets in table 1. We have chosen the following three datasets: CPU, BC and CEV.

[4]http://www.uni-marburg.de/fb12/kebi/research/repository/monodata
[5]http://archive.ics.uci.edu/ml/
[6]http://www.cs.waikato.ac.nz/ml/weka/datasets.html

dataset	#inst.	#crit.	#cat.
CPU	209	6	4
BC	278	7	2
CEV	1728	6	4

Table 1: Datasets

learning set's size	dataset	r_init.	p_init.	p_rand_par.	p_iRace	r_iRace
5%	CPU	0.326 ± 0.076	0.677 ± 0.074	0.701 ± 0.083	0.681 ± 0.073	0.712 ± 0.087
	BC	0.699 ± 0.036	0.687 ± 0.064	0.646 ± 0.079	0.694 ± 0.042	0.699 ± 0.060
	CEV	0.700 ± 0.003	0.697 ± 0.018	0.723 ± 0.079	0.736 ± 0.043	0.777 ± 0.025
20%	CPU	0.336 ± 0.080	0.738 ± 0.064	0.852 ± 0.042	0.862 ± 0.048	0.876 ± 0.043
	BC	0.716 ± 0.025	0.711 ± 0.032	0.720 ± 0.031	0.716 ± 0.027	0.716 ± 0.027
	CEV	0.700 ± 0.006	0.696 ± 0.016	0.701 ± 0.099	0.772 ± 0.027	0.805 ± 0.013
35%	CPU	0.336 ± 0.068	0.763 ± 0.059	0.877 ± 0.088	0.920 ± 0.030	0.916 ± 0.029
	BC	0.721 ± 0.020	0.714 ± 0.027	0.731 ± 0.027	0.738 ± 0.031	0.722 ± 0.021
	CEV	0.703 ± 0.009	0.696 ± 0.016	0.705 ± 0.097	0.780 ± 0.040	0.812 ± 0.010
50%	CPU	0.350 ± 0.087	0.788 ± 0.063	0.897 ± 0.043	0.941 ± 0.020	0.943 ± 0.027
	BC	0.727 ± 0.028	0.700 ± 0.029	0.723 ± 0.035	0.723 ± 0.033	0.736 ± 0.025
	CEV	0.706 ± 0.011	0.700 ± 0.017	0.708 ± 0.094	0.781 ± 0.036	0.812 ± 0.015

Table 2: Results of the algorithm - correctness

For the fine tuning of the parameters, we have used iRace[7] [23]. In what follows, we will quantify the improvement gained with the use of iRace. The 5 parameters that have been optimized are: population size, mutation probability, gene mutation probability, crossover probability, gene crossover probability.

We have tested the algorithms as follow: each dataset has been divided in a learning set and a test set. Different size of learning set have been considered. Alternatives in the learning set have been randomly selected, but we have forced the algorithm to at least randomly select one alternative from each category. For each learning and test set, the algorithm has been executed on the learning set to elicit the parameters. Then the parameters found have been evaluated on the test set. This operation has been executed 16 times for each learning set. For robustness' sake, the whole operation has been executed 10 times for each learning set size.

A summary of the main results of our testing is available in table 2. The number represents the mean correctness of the prediction rate for the test set and the standard deviation on this value. Here is a description of the different algorithm configurations (see figure 3):

- r_init. is a test based on the random generation of initial solutions (algorithm 2), we have generated 10000 initial solutions and selected the best one. No optimization is performed at this stage;

- p_init. is a random generation based on the random selection of an alternative for each category, the values are then sorted regarding each criterion to respect the dominance between the profiles. It is a generation of solutions using the information of the learning set. We have also generated 10000 solutions. No optimization is performed at this stage;

[7]http://iridia.ulb.ac.be/irace/

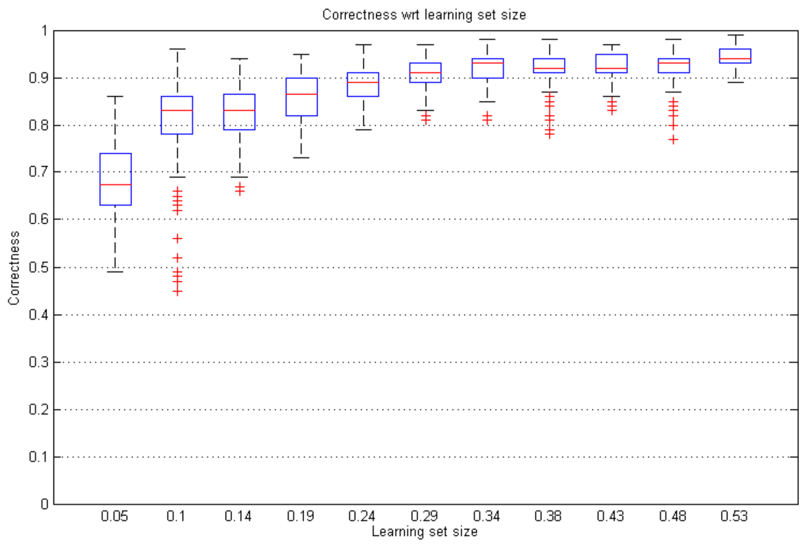

Figure 2: Correctness with respect to the learning set size. Using the algorithm with parameters optimized with iRace, 250000 evaluations per learning set.

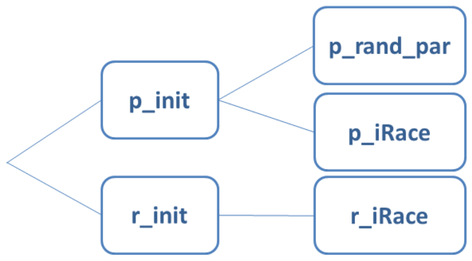

Figure 3: Testing scheme

- p_rand_par. the initial solutions are generated the same way as p_init., but for the optimization we have randomly instantiated the parameters of the GA. We have limited the number of evaluations to 250000;
- p_iRace the initialization algorithm is the same as p_init. In this case, we have optimized the parameters of the GA using iRace (with 6000 evaluations). We have limited the number of evaluations to 250000;
- r_iRace is exactely the same as p_iRace but with the initial generation of solutions being random, like in r_init.

The results show a good prediction degree for the dataset CPU. We can also note

learning set's size	dataset	META MR-SORT	LP UTADIS	r_iRace
20%	CPU	0.7443 ± 0.0559	0.8679 ± 0.0488	0.876 ± 0.043
	BC	0.7196 ± 0.0302	0.7085 ± 0.0307	0.699 ± 0.060
	CEV	0.7761 ± 0.0183	0.7719 ± 0.0153	0.777 ± 0.025
50%	CPU	0.8052 ± 0.0361	0.9340 ± 0.0266	0.943 ± 0.027
	BC	0.7246 ± 0.0446	0.7146 ± 0.0246	0.736 ± 0.025
	CEV	0.7833 ± 0.0180	0.7714 ± 0.0158	0.812 ± 0.015

Table 3: Results of MR-SORT and LP UTADIS from [25] wrt r_iRace- correctness

an improvement on the solution quality for CEV. The gain on the correctness is clear with the parameters optimized using iRace instead of a random instantiation. Due to the nature of the BC dataset, no improvement is observed in table 2. The average performance of the proposed algorithm is as good as the one proposed for other methods , like MR-Sort for ELECTRE TRI or LP UTADIS (see table 3) [25]. These statistics have to be considered cautiously since experiments have not been executed with the same settings. Nevertheless, they allow to show that we reach similar performances.

The execution time is rather small, for instance for the CPU dataset the algorithm runs in about 18 seconds on a Intel i7-2640m with 8go RAM, running under Windows 8.1 with an implementation of the algorithm in Java 8.

In figure 2, you can see a boxplot representing the evolution of the correctness of the prediction rate in function of the size of the learning set. It is interesting to observe that we already reach a good prediction rate with low size of the learning set.

5. Conclusion

In this paper, we have proposed a first method for the parameters elicitation of FlowSort. The method is based on categorization given by the decision maker. Our approach is based on a genetic algorithm. The first results obtained are encouraging. For instance, the run time is below 30 seconds for the CPU dataset containing 209 alternatives, 6 criteria and 4 categories. The performances of the algorithm on this instance is quite good, and we reach a good prediction rate of around 94%.

In further research, we may investigate the possibility of developing an exact method by using a simplified version of FlowSort. Another interesting question is to improve the performances of the algorithm for small learning sets. In this context, a lot of very different solutions can lead to an exact prediction on the learning set but not on the test set. The combination of different instantiations of FlowSort can be investigated in order to improve the robustness of the allocation.

Finally, a more general question about the representability of FlowSort has to be deepened, which allocations can, or not, be reproduced by using such a model? This is also related to the identification of possible inconsistencies in the datasets.

Acknowledgements

We thank Olivier Sobrie for the constructive discussion in the beginning of this work.

Appendix A. Algorithms

```
population = generate_initial_solutions();
while max_eval not reached do
    new_population = select_2_best_solutions(population);
    while size(new_population) lesser than size(population) do
        parents = selection_operator(population);
        new_solutions = crossover_operator(parents);
        mutation_operator(new_solutions);
        evaluate(new_solutions);
        new_population.add(new_solutions);
    end
    population = new_population;
    if population contains perfect solution then
        break;
    end
end
```

Algorithm 1: Genetic Algorithm structure

```
new_sol = create_sol_structure();
for $j = 1$ to nb_criteria do
    new_sol.w[j] = rand(0, 1);
    new_sol.q[j] = rand(0, 0.5);
    new_sol.p[j] = rand(new_sol.q[j], 1);
    new_sol.profile[1][j] = rand(0, 1/nb_categories);
    for $l = 2$ to nb_categories do
        new_sol.profile[l][j] = rand(new_sol.profile[l-1][j], l/nb_categories);
    end
end
```

Algorithm 2: Initialization operator

References

[1] B. Roy, P. Vincke, Multicriteria analysis: survey and new directions, European Journal of Operational Research 8 (3) (1981) 207–218.

[2] B. Roy, Paradigms and challenges, in: Multiple criteria decision analysis: State of the art surveys, Springer, 2005, pp. 3–24.

[3] J. Dyer, Maut — multiattribute utility theory, in: Multiple Criteria Decision Analysis: State of the Art Surveys, Vol. 78 of International Series in Operations Research & Management Science, Springer New York, 2005, pp. 265–292.

[4] J. Siskos, G. Wäscher, H.-M. Winkels, Outranking approaches versus maut in mcdm, European Journal of Operational Research 16 (2) (1984) 270–271.

[5] P. Korhonen, Interactive methods, in: Multiple Criteria Decision Analysis: State of the Art Surveys, Vol. 78 of International Series in Operations Research & Management Science, Springer New York, 2005, pp. 641–661.

```
input  : the solution to mutate; mutation probability
rand_nbr = rand(0, 1);
if rand_nbr < mutation_probability then
    rand_nbr = rand(0, 1);
    if rand_nbr < 0.4 then
        profiles_mutation_operator(solution);
    else
        if rand_nbr < 0.7 then
            weights_mutation_operator(solution);
        else
            thresholds_mutation_operator(solution);
        end
    end
end
```

Algorithm 3: Mutation operator

[6] J. Figueira, V. Mousseau, B. Roy, Electre methods, in: Multiple criteria decision analysis: State of the art surveys, Springer, 2005, pp. 133–153.

[7] M. Behzadian, R. B. Kazemzadeh, A. Albadvi, M. Aghdasi, Promethee: A comprehensive literature review on methodologies and applications, European Journal of Operational Research 200 (1) (2010) 198–215.

[8] P. Vincke, Multicriteria Decision-Aid, J. Wiley, New York, 1992.

[9] C. Zopounidis, M. Doumpos, Business failure prediction using the utadis multicriteria analysis method, Journal of the Operational Research Society (1999) 1138–1148.

[10] N. Belacel, Multicriteria assignment method¡ i¿ proaftn¡/i¿: Methodology and medical application, European Journal of Operational Research 125 (1) (2000) 175–183.

[11] W. Yu, Aide multicritère à la décision dans le cadre de la problématique du tri: concepts, méthodes et applications, Ph.D. thesis, Paris 9 (1992).

[12] P. Nemery, C. Lamboray, Flowsort: a flow-based sorting method with limiting or central profiles, Top 16 (1) (2008) 90–113.

[13] L. C. Dias, V. Mousseau, Inferring electre's veto-related parameters from outranking examples, European Journal of Operational Research 170 (1) (2006) 172 – 191.

[14] M. Doumpos, Y. Marinakis, M. Marinaki, C. Zopounidis, An evolutionary approach to construction of outranking models for multicriteria classification: The case of the {ELECTRE} {TRI} method, European Journal of Operational Research 199 (2) (2009) 496 – 505.

[15] O. Cailloux, P. Meyer, V. Mousseau, Eliciting electre tri category limits for a group of decision makers, European Journal of Operational Research 223 (1) (2012) 133 – 140.

[16] O. Sobrie, V. Mousseau, M. Pirlot, Learning a majority rule model from large sets of assignment examples, in: Algorithmic Decision Theory, Springer, 2013, pp. 336–350.

[17] J. Zheng, S. A. Metchebon Takougang, V. Mousseau, M. Pirlot, Learning criteria weights of an optimistic electre tri sorting rule, Computers & Operations Research 49 (2014) 28–40.

[18] J.-P. Brans, B. Mareschal, Promethee methods, in: Multiple Criteria Decision Analysis: State of the Art Surveys, Vol. 78 of International Series in Operations Research & Management Science, Springer New York, 2005, pp. 163–186.

[19] B. Mareschal, Y. De Smet, P. Nemery, Rank reversal in the promethee ii method: some new results, in: Industrial Engineering and Engineering Management, 2008. IEEM 2008. IEEE International Conference on, IEEE, 2008, pp. 959–963.

[20] P. Nemery de Bellevaux, P. Vincke, On the use of multicriteria ranking methods in sorting problems/utilisation des méthodes de rangement multicritères dans les problèmes de tri.

[21] J. J. Durillo, A. J. Nebro, jmetal: A java framework for multi-objective optimization, Advances in Engineering Software 42 (2011) 760–771.

[22] J. Durillo, A. Nebro, E. Alba, The jmetal framework for multi-objective optimization: Design and architecture, in: CEC 2010, Barcelona, Spain, 2010, pp. 4138–4325.

```
input  : the solution to mutate; gene mutation probability ($gmp$); $U_l$, $W_l$, $O_l$, $U_l^*$,
         $W_l^*$, $O_l^*$ for each profile $l$
For simplifications purposes, we consider $\forall j$ solution.profile[l][j] = 0 for $l < 1$ and
solution.profile[l][j] = 1 for $l > nb\_categories$.
for $l = 1$ to nb_categories do
    if rand(0, 2) > ($W_l + W_l^*$) then
        if $U_l/(U_l + O_l) > 0.8$ and $W_l < 0.6$ then
            for $j = 1$ to nb_criteria do
                solution.profile[l][j] = (solution.profile[l][j] + solution.profile[l-1][j]) /
                2;
            end
        else
            if $O_l/(U_l + O_l) > 0.8$ and $W_l < 0.6$ then
                for $j = 1$ to nb_criteria do
                    solution.profile[l][j] = (solution.profile[l][j] +
                    solution.profile[l+1][j]) / 2;
                end
            else
                for $j = 1$ to nb_criteria do
                    if rand(0, 1) < gmp then
                        if rand(0, 1) < $U_l + U_l^*/(U_l + O_l + U_l^* + O_l^*)$ then
                            solution.profile[l][j] = solution.profile[l][j] + $(U_l + U_l^*)/2$ *
                            rand(solution.profile[l-1][j] - solution.profile[l][j], 0);
                        else
                            solution.profile[l][j] = solution.profile[l][j] + $(O_l + O_l^*)/2$
                            * rand(0, solution.profile[l+1][j] - solution.profile[l][j]);
                        end
                    end
                end
            end
        end
    end
end
```

Algorithm 4: Mutation operator for profiles

```
input  : the solution to mutate; correctness of the solution; gene mutation
         probability ($gmp$)
for $j = 1$ to nb_criteria do
    if rand(0, 1) < gmp then
        solution.w[j] = solution.w[j] + (1 - correctness) * rand(-solution.w[j], 1 -
        solution.w[j]);
    end
end
```

Algorithm 5: Mutation operator for weights

```
input  : the solution to mutate; correctness of the solution; gene mutation
         probability (gmp)
for j = 1 to nb_criteria do
    if rand(0, 1) < gmp then
        solution.q[j] = solution.q[j] + (1 - correctness) * rand(-solution.q[j],
        solution.p[j] - solution.q[j]);
    end
    if rand(0, 1) < gmp then
        solution.p[j] = solution.p[j] + (1 - correctness) * rand(solution.q[j] -
        solution.p[j], 1 - solution.p[j]);
    end
end
```

Algorithm 6: Mutation operator for preference thresholds

```
input  : the solutions to crossover: parent_1, parent_2; crossover probability;
         correctness of the solution; gene crossover probability (gcp)
if rand(0, 1) < crossover_probability then
    lambda = rand(0.1, 0.9);
    for j = 1 to nb_criteria do
        for l = 1 to nb_categories do
            if rand(0, 1) < gcp then
                new_sol_1.profile[j] = lambda * parent_1.profile[j] + (1 - lambda) *
                parent_2.profile[j];
                new_sol_2.profile[j] = (1 - lambda) * parent_1.profile[j] + lambda *
                parent_2.profile[j];
            end
        end
        if rand(0, 1) < gcp then
            new_sol_1.w[j] = lambda * parent_1.w[j] + (1 - lambda) * parent_2.w[j];
            new_sol_2.w[j] = (1 - lambda) * parent_1.w[j] + lambda * parent_2.w[j];
        end
        if rand(0, 1) < gcp then
            new_sol_1.q[j] = lambda * parent_1.q[j] + (1 - lambda) * parent_2.q[j];
            new_sol_2.q[j] = (1 - lambda) * parent_1.q[j] + lambda * parent_2.q[j];
        end
        if rand(0, 1) < gcp then
            new_sol_1.p[j] = lambda * parent_1.p[j] + (1 - lambda) * parent_2.p[j];
            new_sol_2.p[j] = (1 - lambda) * parent_1.p[j] + lambda * parent_2.p[j];
        end
    end
    correct_solution(new_sol_1);
    correct_solution(new_sol_2);
end
```

Algorithm 7: Crossover operator

```
chosen_solution = 0;
solution_1, solution_2 = randomly_chose_2_sol(population);
if rand(0, 1) < solution_1.objective/(solution_1.objective + solution_2.objective)
then
    chosen_solution = solution_2;
else
    chosen_solution = solution_1;
end
```

Algorithm 8: Selection operator

[23] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, M. Birattari, The irace package, iterated race for automatic algorithm configuration, Tech. Rep. TR/IRIDIA/2011-004, IRIDIA, Université libre de Bruxelles, Belgium (2011).
URL `http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2011-004.pdf`

[24] M. Birattari, Z. Yuan, P. Balaprakash, T. Stützle, F-race and iterated f-race: An overview, in: Experimental methods for the analysis of optimization algorithms, Springer, 2010, pp. 311–336.

[25] O. Sobrie, V. Mousseau, M. Pirlot, Learning the parameters of a multiple criteria sorting method from large sets of assignment examples, in: DA2PL'2012 From Multiple Criteria Decision Aid to Preference Learning, UMONS (Université de Mons), 2012.