Schulz, Arne

Efficient neighborhood evaluation for the maximally diverse grouping problem

Annals of Operations Research

Provided in Cooperation with:
Springer Nature

Suggested Citation: Schulz, Arne (2024) : Efficient neighborhood evaluation for the maximally diverse grouping problem, Annals of Operations Research, ISSN 1572-9338, Springer US, New York, NY, Vol. 341, Iss. 2, pp. 1247-1265, https://doi.org/10.1007/s10479-024-06217-9

Annals of Operations Research (2024) 341:1247–1265
https://doi.org/10.1007/s10479-024-06217-9

ORIGINAL-COMPARATIVE COMPUTATIONAL STUDY

Efficient neighborhood evaluation for the maximally diverse grouping problem

Arne Schulz[1,2]

Received: 23 October 2023 / Accepted: 5 August 2024 / Published online: 21 August 2024

Abstract
The Maximally Diverse Grouping Problem is one of the well-known combinatorial optimization problems with applications in the assignment of students to groups or courses. Due to its NP-hardness several (meta)heuristic solution approaches have been presented in the literature. Most of them include the insertion of an item of one group into another group and the swap of two items currently assigned to different groups as neighborhoods. The paper presents a new efficient implementation for both neighborhoods and compares it with the standard implementation, in which all inserts/swaps are evaluated, as well as the neighborhood decomposition approach. The results show that the newly presented approach is clearly superior for larger instances allowing for up to 160% more iterations in comparison to the standard implementation and up to 76% more iterations in comparison to the neighborhood decomposition approach. Moreover, the results can also be used for (meta)heuristic algorithms for other grouping or clustering problems.

Keywords Combinatorial optimization · Grouping · Local search · Computational efficiency

1 Introduction

The Maximally Diverse Grouping Problem (MDGP) is a well-known and well-investigated combinatorial optimization problem. Given a set of items $i \in I$ with a pairwise diversity $d_{ij} \geq 0$, the task is to assign the items to groups $g \in G$ such that each group g gets at least l_g and at most u_g items assigned and the within group diversity is maximized over all groups. The MDGP is an important combinatorial optimization problem for two reasons: First, it has a wide field of applications such as the assignment of students to project groups (Beheshtian-Ardekani & Mahmood, 1986) or teams (Dias & Borges, 2017), the assignment of pupils to tutor groups (Baker & Benn, 2001) or of children to equally strong sport teams (Rubin & Bai, 2015). Moreover, there are applications in final exam scheduling, VLSI design (Weitz & Lakshminarayanan, 1998), and anticlustering. Anticlustering aims like the MDGP to partition

✉ Arne Schulz
arne.schulz@uni-hamburg.de ; arne.schulz@hsu-hh.de

1 Institute of Operations Management, Universität Hamburg, Moorweidenstraße 18, 20148 Hamburg, Germany

2 Institute of Quantitative Logistics, Helmut Schmidt University, Holstenhofweg 85, 22043 Hamburg, Germany

items into disjoint groups such that groups are similar but within-group heterogeneity is high (Brusco et al., 2020; Papenberg, 2024). Applications are in the assignment of participants to groups (Batista et al., 2023) and in dividing data sets for cross validation (Papenberg & Klau, 2021). Second, the MDGP is NP-hard to solve (Feo & Khellaf, 1990) although it can be formulated as a short integer program, which can easily be linearized (compare e.g. (Gallego et al., 2013)):

$$\max \sum_{g \in G} \sum_{i \in I} \sum_{j \in I: j > i} d_{ij} x_{ig} x_{jg}$$

with the constraints (1)

$$\sum_{g \in G} x_{ig} = 1 \qquad \forall i \in I \tag{2}$$

$$l_g \leq \sum_{i \in I} x_{ig} \leq u_g \qquad \forall g \in G \tag{3}$$

$$x_{ig} \in \{0, 1\} \qquad \forall i \in I, g \in G \tag{4}$$

In the study by Gallego et al. (2013), only instances with up to 12 items could be solved to optimality. Given that d_{ij} values often have a certain structure in practice, e.g. they are the difference of attribute values (Schulz, 2021) or binary values (Mingers & O'Brien, 1995), instances of up to 30–70 items can be solved to proven optimality (Schulz, 2022). If at most two attributes are considered, Schulz (2021) proved that even large instances can be solved efficiently.

However, if we want to solve large instances or instances with general $d_{ij} \geq 0$ ($d_{ii} = 0$ for all $i \in I$), efficient heuristic solution methods are required. While earlier approaches focussed on construction and local search improvement heuristics (compare the overview by Weitz and Lakshminarayanan (1998)), later papers focussed on different metaheuristic solution approaches. In the last years, Brimberg et al. (2015) developed a skewed general variable neighborhood search, Palubeckis et al. (2015) an iterated tabu search approach, Lai and Hao (2016) an iterated maxima search heuristic, Lai et al. (2021a) a neighborhood decomposition based variable neighborhood search and tabu search, and Yang et al. (2022) a three-phase approach with a dynamic population size. These are the most recent and successful approaches. Please see Lai and Hao (2016) for a more depth review of metaheuristic solution approaches for the MDGP.

The focus of the paper at hand is not to develop a new advanced metaheuristic solution approach but to consider the evaluation of neighborhoods, especially insertions and swaps, within these approaches. We present a new method to evaluate these neighborhoods more efficiently. In doing so, the paper is inspired by the neighborhood decomposition approach by Lai et al. (2021a).

Given a feasible solution for the MDGP, i.e. an assignment of each item to exactly one group such that each group has a number of items between l_g and u_g assigned, the insertion neighborhood contains all feasible solutions such that exactly one item is assigned to a different group. Thus, given solution y, whereat y_i indicates the group of item i, the neighborhood includes all solutions $\bar{y}$ such that (2)–(4) are fulfilled and $y_i = \bar{y}_i$ for all but exactly one item $i \in I$. Correspondingly, the swap neighborhood of y includes all solutions $\bar{y}$ such that (2)–(4) are fulfilled, $y_i = \bar{y}_i$ for all items $i \in I \backslash \{j, j'\}$, $\bar{y}_{j'} = y_j$, and $\bar{y}_j = y_{j'}$. These two neighborhoods are used in most of the advanced solution methods for the MDGP, including the five advanced algorithms mentioned before as well as for example Baker and Powell (2002), Chen et al. (2011), Fan et al. (2011), Palubeckis et al. (2011), Rodriguez et al. (2013),

Urošević (2014), and Schulz (2023). In the paper at hand, we present a new efficient method enhancing the neighborhood decomposition (ND) method described in Lai et al. (2021a) to evaluate the two neighborhoods faster than in the *standard* implementation, which simply evaluates the entire neighborhood, and the *ND* implementation.

The presented neighborhood evaluation can also be applied to other grouping or clustering problems. These include clustering problems like the capacitated clustering problem (Lai et al., 2021b) and the capacitated p-median problem (Zheng et al., 2021). It can also be applied to the ratio cut and normalized cut graph partitioning problem (Palubeckis, 2022), to vehicle routing problems (e.g. Pfeiffer and Schulz (2022) or Zhou et al. (2023)) or to parallel machine scheduling (Yalaoui & Chu, 2002).

The paper is constructed as follows: All three implementations are introduced in the following Sect. 2. Section 3 presents the general framework used in the computational study, in which all three implementations are evaluated on benchmark instances (Sect. 4). The paper closes with a conclusion (Sect. 5).

2 Implementation of neighborhoods

In this section, we present the three implementations *standard*, *ND*, and *efficient ND* to implement the insertion and the swap neighborhood.

2.1 Standard implementation

In the standard implementation, simply all solutions of the neighborhoods are evaluated. Thereby, a solution is encoded by parameters y_i and additionally by sets $I_g = \{i \in I : y_i = g\}$, $g \in G$, i.e. I_g is the set of items assigned to group g in the current solution. The pseudo-code for the insertion neighborhood can be found in Algorithm 1.

Algorithm 1 *[standard insertion]*

```
1: Let a solution y be given.
2: for all g ∈ G : l_g < |I_g| do
3:     for all g' ∈ G : g' ≠ g ∧ |I_g'| < u_g' do
4:         for all i ∈ I_g do
5:             Evaluate insert of i in group g' and save it if it is the best found so far.
6:         end for
7:     end for
8: end for
9: Realize best insert if one improving the objective value of y was found.
```

By using I_g, we only once have the check for each pair of groups whether they are identical (Line 3). Often authors replace the three for-loops starting in Lines 2–4 by a for-loop over all items, a for-loop over all groups, and an if-statement checking whether the item is in the group or not. Thus, the if-check needs to be done $|I| \cdot |G|$ times while in the above implementation the check in Line 3 is only done $|G|^2$ times, whereat $|G| << |I|$ holds typically. We use this implementation for Algorithms 1 and 2 (swap neighborhood) also to ensure that the neighborhoods are always evaluated in the exact same way in the three different implementations presented in this paper. By this, we ensure that the search is the same for all three implementations. Thus, if one implementation leads to a higher number of

operated iterations due to its more efficient implementation, the best found solution cannot be worse than the best one found with the other two implementations.

Algorithm 2 presents the pseudo-code for a full evaluation of the swap neighborhood.

Algorithm 2 *[standard swap]*
Let a solution y be given.
for all $g \in G$ ***do***
  for all $g' \in G : g' > g$ ***do***
    for all $i \in I_g$ ***do***
      for all $j \in I_{g'}$ ***do***
        Evaluate swap of items i and j, i.e. reassign item i to group g' and item j to group g and save it if it is the best found swap so far.
      end for
    end for
  end for
end for
Realize best swap if one improving the objective value of y was found.

It is well-known in the literature (see e.g. Brimberg et al. (2015)) that inserts and swaps can be evaluated effectively by using matrix $(D_{ig})_{i \in I, g \in G}$ with

$$D_{ig} = \sum_{j \in I_g} d_{ij}$$

which indicates the sum of diversities of item i with all items assigned to group g. By this, the change in the objective value due to a move of item i from group g to group g' can directly be computed as

$$D_{ig'} - D_{ig}. \tag{5}$$

For a swap of the groups of items i and j currently assigned to groups g and g' we obtain the change in the objective value by

$$D_{ig'} - D_{ig} + D_{jg} - D_{jg'} - 2 \cdot d_{ij}. \tag{6}$$

As d_{ij} is included in $D_{ig'}$ and D_{jg}, but i is removed from g and j is removed from g', we have to subtract d_{ij} twice. After realizing an insert or a swap D_{ig} needs to be updated for all items and the involved two groups g and g' by subtracting the diversity with the removed item and adding the diversity with the added item.

2.2 Neighborhood decomposition implementation

Lai et al. (2021a) recognized that it is not necessary to evaluate the entire neighborhood in every iteration. In every iteration, the assignment to only two groups is changed. Thus, if we found out that there is no promising insert of an item from group g into group g' or no promising swap between items of groups g and g', we do not need to evaluate these inserts or swaps again until at least one of the two groups is changed by removing or adding an item.

As evaluating all inserts of items of group g into group g' or all swaps between items of groups g and g' is independent of the evaluation of all inserts/swaps of all other group pairs, Lai et al. (2021a) call the part of the neighborhood containing these inserts/swaps the neighborhood block of groups g and g'. Note that the neighborhood block of g and g' is

identical to the one of g' and g for the swap neighborhood, but there is a difference for the insertion neighborhood. In both cases, all neighborhood blocks $g, g' \in G$ (for swap with $g < g'$) are disjunct and their union is the entire neighborhood.

Lai et al. (2021a) introduced two zero–one matrices $W^1 = (W^1_{gg'})_{g,g' \in G}$ and $W^2 = (W^2_{gg'})_{g,g' \in G}$ to picture whether the neighborhood block including groups g and g' needs to be evaluated. If an entry of the matrices is 1, the inserts/swaps between the corresponding groups need to be evaluated. If an entry is 0, the neighborhood block can be skipped, as we know already that it does not contain any promising insert/swap.

Note that W^2 is symmetric while W^1 is not. It might be that there is no promising insert of an element of group g into group g', but there is one in the opposite direction. If an item is added to or removed from a group g, $W^1_{gg'}$, $W^1_{g'g}$, $W^2_{gg'}$, and $W^2_{g'g}$ are set to 1 for all $g' \in G \backslash \{g\}$, i.e. they have to be re-evaluated (last line of Algorithm 3 and 4, respectively). Lai et al. (2021a) call the procedure *neighborhood decomposition*, as the neighborhoods are decomposed for each pair of groups $g, g' \in G$ into one independent block (swap) and two independent blocks (insert), respectively. We write for short *ND* instead of *neighborhood decomposition* in the following. The pseudo-code for the insertion and the swap neighborhood using the *ND* implementation are presented in Algorithms 3 and 4, respectively.

Algorithm 3 *[ND insertion]*

Let a solution y be given.
for all $g \in G : l_g < |I_g|$ ***do***
  for all $g' \in G : g' \neq g \wedge |I_{g'}| < u_{g'}$ ***do***
    if $W^1_{gg'} = 1$ ***then***
      $W^1_{gg'} = 0$
      for all $i \in I_g$ ***do***
        Evaluate insert of i in group g', save it if it is the best found so far, and set $W^1_{gg'} = 1$ if it improves the objective value of y.
      end for
    end if
  end for
end for
Realize best insert if one improving the objective value of y was found.
Update matrices W^1 and W^2 if an insert was realized.

Algorithm 4 *[ND swap]*

Let a solution y be given.
for all $g \in G$ ***do***
  for all $g' \in G : g' > g$ ***do***
    if $W^2_{gg'} = 1$ ***then***
      $W^2_{gg'} = 0$
      for all $i \in I_g$ ***do***
        for all $j \in I_{g'}$ ***do***
          Evaluate swap of items i and j, i.e. reassign item i to group g' item j to group g, save it if it is the best found swap so far, and set $W^2_{gg'} = 1$ if it improves the objective value of y.
        end for
      end for
    end if
  end for

13: ***end for***
14: Realize best swap if one improving the objective value of y was found.
15: Update matrices W^1 *and* W^2 *if an insert was realized.*

It can clearly be seen that the neighborhoods can be evaluated more efficiently than in the *standard* implementation if the block of a group pair g and g' does not need to be evaluated (Line 4 in both algorithms). However, the benefit depends on the number of blocks which can be skipped. In contrast, there is the drawback that matrices W^1 and W^2 need to be updated after every change in the solution (Lines 13 and 15, respectively) although this requires only a linear effort ($W^1_{g\bar{g}}$, $W^1_{\bar{g}g}$, $W^2_{g\bar{g}}$, $W^2_{\bar{g}g}$, $W^1_{g'\bar{g}}$, $W^1_{\bar{g}g'}$, $W^2_{g'\bar{g}}$, and $W^2_{\bar{g}g'}$ are set to 1 for all $\bar{g} \in G\backslash\{g, g'\}$ whereat g and g' are the two groups with removed/added items).

Moreover, the approach still has the disadvantage that a block is evaluated in an iteration and again in the next iteration if there is a promising insert/swap, but another one comprising two other groups was realized.

2.3 Efficient neighborhood decomposition implementation

We present now an improved version to overcome this drawback. Therefore, we replace matrices W^1 and W^2 by new three-dimensional matrices $M^1 = (M^1_{gg'h})_{g,g' \in G, h=1,2}$ and $M^2 = (M^2_{gg'h})_{g,g' \in G, h=1,2,3}$, respectively. For each pair of groups g and g' we evaluate all inserts of an item of group g into group g' (analogously to conduct Lines 6–8 of Algorithm 3). If there is a promising one, we save the change in the objective function for the best one if the insert would be conducted in $M^1_{gg'1}$ (value of (5)) and the corresponding item number i in $M^1_{gg'2}$. If there is no promising insert, we simply set $M^1_{gg'1} = M^1_{gg'2} = 0$. Thus, M^1 gives us the best insert of an item of group g into group g' if there is one such that the neighborhood evaluation reduces as can be seen in Lines 8–10 of Algorithm 5.

Algorithm 5 *[efficient ND insertion]*
1: Let a solution y be given.
2: $max = 0$.
3: ***for all*** $g \in G : l_g < |I_g|$ ***do***
4: ***for all*** $g' \in G : g' \neq g \wedge |I_{g'}| < u_{g'}$ ***do***
5: ***if*** $M^1_{gg'1} < 0$ ***then***
6: *Evaluate all inserts of items of group g into group* g' *and save the best found in* $M^1_{gg'1}$ *and* $M^1_{gg'2}$.
7: ***end if***
8: ***if*** $M^1_{gg'1} > max$ ***then***
9: $max = M^1_{gg'1}$ *and* $i = M^1_{gg'2}$.
10: ***end if***
11: ***end for***
12: ***end for***
13: Realize best insert if $max > 0$*, i.e. insert item i in group* g'.
14: Update matrices M^1 *and* M^2 *if an insert was realized.*

Whenever a change occurs in groups g or g', we need to update $M^1_{gg'h}$ and $M^1_{g'gh}$ analogously to the *ND* method. We simply set $M^1_{gg'1} = M^1_{g'g1} = -1$ indicating that the neighborhood block containing groups g and g' needs to be re-evaluated (Line 14 of Algorithm 5). We cannot set $M^1_{gg'1}$ and $M^1_{g'g1}$ to 0, as 0 indicates that the neighborhood block

has been evaluated, but no promising insert was found. Matrix M^2 is updated analogously. If $M^1_{gg'1} = -1$, all inserts of items of group g into group g' are evaluated and the best promising one is again saved in $M^1_{gg'1}$, $M^1_{gg'2}$, and $M^1_{gg'3}$ (Lines 5–7).

As we can replace Lines 4–9 of Algorithm 3 by Lines 5–10 of Algorithm 5, evaluating the neighborhood is more efficient now. We only need to evaluate neighborhood blocks which have not been evaluated since the last change in their group assignment (Line 5), but these neighborhood blocks would also be evaluated in the *ND* implementation. Additionally, further promising neighborhood blocks might be re-evaluated in the *ND* implementation but are not in the *efficient ND* implementation.

For the swap neighborhood we analogously save the value the objective function changes in entry $M^2_{gg'1}$ (value of (6)) and the item i removed from group g in entry $M^2_{gg'2}$. Additionally, we save the item j removed from group g' in entry $M^2_{gg'3}$. Again, all three entries are 0 if there is no promising swap between groups g and g' and $M^2_{gg'1} = -1$ if the neighborhood block has not been evaluated since the last change in the assigned items to group g or g'. The swap neighborhood can then be evaluated by Algorithm 6.

Algorithm 6 *[efficient ND swap]*

1: *Let a solution y be given.*
2: $max = 0$.
3: ***for all*** $g \in G$ ***do***
4: ***for all*** $g' \in G : g' > g$ ***do***
5: ***if*** $M^2_{gg'1} < 0$ ***then***
6: *Evaluate all pairwise swaps of items of group g with items of group g' and save the best found in $M^2_{gg'1}$, $M^2_{gg'2}$, and $M^2_{gg'3}$.*
7: ***end if***
8: ***if*** $M^2_{gg'1} > max$ ***then***
9: *$max = M^2_{gg'1}$, $i = M^2_{gg'2}$, and $j = M^2_{gg'3}$.*
10: ***end if***
11: ***end for***
12: ***end for***
13: *Realize best swap if $max > 0$, i.e. insert item i in group g' and item j in group g.*
14: *Update matrices M^1 and M^2 if a swap was realized.*

2.4 Comparison of the three implementations

Comparing the three implementations, the *standard* implementation fully evaluates the neighborhoods in every iteration. The *ND* implementation saves some computations by evaluating only those parts which might be promising but always evaluates a neighborhood block if anything has changed in the assignment of one of the two involved groups. The *efficient ND* implementation saves the best insert/swap which can be realized between two groups such that the neighborhood block only needs to be re-evaluated if anything changes in one of the two groups. In other words, every part of the neighborhood which needs to be evaluated in the *efficient ND* implementation needs also to be evaluated in the *ND* implementation and every part of the neighborhood which needs to be evaluated in the *ND* implementation needs to be evaluated in the *standard* implementation. Thus, we have a clear hierarchy in the efficiency of the implementations.

Concretely, we need to evaluate $|I| \cdot (|G| - 1) = (|I_1| + \ldots |I_{|G|}|) \cdot (|G| - 1)$ inserts in the *standard* implementation while we only need to evaluate up to $(|I_{g_1}| + |I_{g_2}|) \cdot (|G| - 1)$ inserts

in the *efficient ND* implementation whereat g_1 and g_2 are the two groups which were changed by an insert/swap in the previous iteration. All items of these two groups ($|I_{g_1}| + |I_{g_2}|$) need to be reinserted into all remaining $|G| - 1$ groups. It can clearly be seen that the *efficient ND* implementation requires fewer evaluated inserts than the *standard* implementation if $|G| > 2$ while the difference is the larger the larger $|G|$ is. The *ND* implementation is in the best case as efficient as the *efficient ND* implementation, but in the worst case as inefficient as the *standard* implementation.

If the swap neighborhood is considered, we need to evaluate $\sum_{g \in G} |I_g| \cdot (|I| \backslash |I_g|)/2$ swaps in the *standard* implementation, i.e. for every item of a group ($|I_g|$) the swap with any item assigned to another group ($|I| \backslash |I_g|$). As the swap of two items only needs to be evaluated once, we divide the result by two. In the *efficient ND* implementation, we only need to update swaps of the items assigned to a group g with the items assigned to all other groups if group g was changed in the previous iteration. As in one iteration at most two groups g_1 and g_2 are changed, we need to evaluate at most $\sum_{g \in \{g_1, g_2\}} |I_g| \cdot (|I| \backslash |I_g|) - |I_{g_1}| \cdot |I_{g_2}|$ swaps whereat $|I_{g_1}| \cdot |I_{g_2}|$ subtracts the swaps between the two groups g_1 and g_2 which are otherwise counted twice. Again, the *efficient ND* implementation is more efficient than the *standard* implementation if $|G| > 2$ and the difference is the larger the larger $|G|$ is. Moreover, the *ND* implementation is again in the best case as efficient as the *efficient ND* implementation, but in the worst case as inefficient as the *standard* implementation.

The difference is even clearer if one of the neighbourhoods is called without having any change in the group assignment since the last call of the algorithm. Then, matrices M^1 and M^2, respectively, are up to date such that the *efficient ND* implementation does not need to re-evaluate any part of the neighborhood. It only has an effort of up to $|G| \cdot (|G| - 1)$ to find the best neighbouring solution. This is clearly less effort than in the *standard* implementation requiring $|I| \cdot (|G| - 1)$ for the insertion neighborhood and $|I|^2$ for the swap neighborhood. We are certainly in this situation if we are in a local optimum for both neighborhoods. Then, we need to re-evaluate both neighborhoods before we know that we are in a local optimum, but for the one leading to the local optimal solution nothing hast changed since the last call.

Of course we do not want to stay in a local optimum. To not stick there, we introduce a variable neighborhood search (VNS) based framework with perturbation in the next section which is used to evaluate the introduced neighborhood implementations in the computational study.

3 Variable neighborhood search framework

Before we introduce the overall framework we first introduce the used perturbation methods to avoid sticking in local optimal solutions. We use the weak and strong perturbation method presented in Lai and Hao (2016). They are presented in Algorithms 7 and 8. As in Lai and Hao (2016) we set $\eta_w = 3$ and $\eta_s = \Theta \cdot |I|/|G|$, whereat $\Theta = 1$ if $|I| \leq 400$ and 1.5 otherwise. In Algorithm 7, we determine 5 solutions in the for-loop starting in Line 4. Lai and Hao (2016) determined $|I|$ solutions in the for-loop. However, in our preliminary evaluations this resulted in a very significant time spent for the weak perturbation. Therefore, we decreased the value such that the algorithm spends much more time for the neighborhood evaluation and we can perform clearly more iterations.

Algorithm 7 *[Weak perturbation]*
1: *Let a solution y be given.*
2: ***for all*** $n = 1$ *to* η_w ***do***

Randomly pick a neighbour solution $\bar{y}$ of y (probability 0.5 for an insertion and 0.5 for a swap, uniformly within the neighborhood).
for all $i = 1$ *to* 5 ***do***
Randomly pick a neighbour solution $\hat{y}$ of y (probability 0.5 for an insertion and 0.5 for a swap, uniformly within the neighborhood).
if *objective value of $\hat{y}$ is higher than of $\bar{y}$* ***then***
Replace $\bar{y}$ by $\hat{y}$.
end if
end for
Replace y by $\bar{y}$.
end for
Return perturbed solution y.

Algorithm 8 *[Strong perturbation]*
Let a solution y be given.
for all $n = 1$ *to* η_s ***do***
Randomly pick a neighbour solution $\bar{y}$ of y (probability 0.5 for an insertion and 0.5 for a swap, uniformly within the neighborhood).
Replace y by $\bar{y}$.
end for
Return perturbed solution y.

With the weak and the strong perturbation we have all components for the variable neighborhood search framework in Algorithm 9.

Algorithm 9 *[VNS framework]*
Find an initial solution and save it as best found solution so far y_{best}.
Set best_objective_value to the objective value of y_{best}.
Set neighborhood = 1 and last_improvement = 0.
while *time limit is not reached* ***do***
Set objective_value_weak = 0 and counter = 0.
while *counter $< \alpha$ and time limit is not reached* ***do***
Set objetive_value_old = current_objective_value.
if *neighborhood = 1* ***then***
Call insertion procedure (Algorithm 1, 3, and 5, respectively). Update current_objective_value.
else if *neighborhood = 2* ***then***
Call swap procedure (Algorithm 2, 4, and 6, respectively).
Update current_objective_value.
end if
if *current_objective_value $\leq$ objective_value_old* ***then***
last_improvement += 1.
else *last_improvement = 0.*
end if
if *current_objective_value $>$ best_objective_value* ***then***
Update best_objective_value and y_{best}.
end if
if *last_improvement = 2* ***then***
if *current_objective_value $>$ objective_value_weak* ***then***
counter = 0 and objective_value_weak = current_objective_value.

```
24:          else counter += 1.
25:          end if
26:          Perform weak perturbation (Algorithm 7).
27:       end if
28:    end while
29:    Perform strong perturbation (Algorithm 8).
30: end while
31: Return y_best.
```

Algorithm 9 summarizes the entire procedure. The algorithm starts with an initial solution. In line with the literature (Lai & Hao, 2016), we add uniformly l_g items to all groups g in the first step before the remaining items are uniformly distributed over all groups such that no group exceeds its capacity u_g. Then, we improve this solution with the swap neighborhood (Algorithm 6) until no improvement is found any more. We do this ten times, i.e. generate ten initial solutions. The best one of them is used in Line 1 of Algorithm 9 as initial solution.

The rest of the algorithm is also closely oriented at the procedure by Lai and Hao (2016). The core of the algorithm comprises two while-loops. The outer one (Lines 4–31) steers the strong perturbation (Line 30). Strong perturbation is performed if for α iterations of the inner while-loop no improved solution was found (objective_value_weak; Lines 23–26). The value of α is set to 5 if $|I| \leq 400$ or $|I|/|G| \leq 10$. Otherwise, it is set to 3 (compare Lai and Hao (2016)). In the inner while-loop, we call the insertion and the swap neighborhood alternately (Lines 8–14). If the solution has not improved after calling both (Lines 7, 15–17), i.e. if we are in a local optimum of both neighborhoods, we perform a weak perturbation (Lines 22–28). To avoid sticking in local optima, we first perform a weak perturbation to still evaluate the nearer environment of the current solution (intensification). If we could not find a better solution for α runs of the weak perturbation procedure, the strong perturbation procedure is executed to reach different areas of the solution space (diversification). Whenever a new best solution is found, it is saved as y_{best} in Lines 19–21. When the algorithm terminates after the time limit is reached, the best solution y_{best} is returned (Line 32).

In line with the aim of the paper to evaluate the different implementations of the insertion and the swap neighborhood, Algorithm 9 is a basic variable neighborhood search evaluating both neighborhoods alternately like it is done in Lai and Hao (2016). In difference to their paper, we fully evaluate the neighborhoods and except the best solution found instead of accepting every improvement. If we would accept every improved solution, matrices W^1, W^2, M^1, and M^2 would be updated more often meaning that the *ND* and *efficient ND* implementations cannot fully use their advantage. Thus, we would not be able to evaluate their full potential.

4 Computational study

In this section, we evaluate the different implementations of the insertion and the swap neighborhood. All algorithms were implemented in C++. The code was executed on a single AMD EPYC 7542 32 core with 2.90GHz. All three implementations (call Algorithms 1, 3, and 5 in Line 9 and Algorithms 2, 4, and 6 in Line 12 of Algorithm 9, respectively) were evaluated on the three standard benchmark sets Geo, RanInt, and RanReal all containing 160 instances where half of them have equal-sized (ss) groups and half of them not (ds). The number following the letter n in the instance's name indicates the number of items considered. In the Geo set, diversities are Euclidean distances between pairs of points with

random coordinates in the interval [0,10]. In the RanReal set, diversities are real uniformly generated numbers in (0,100). In the RealInt set, diversities are integer uniformly generated in the interval [0,100]. The instances are available under the following link: https://grafo.etsii.urjc.es/optsicom/mdgp.html. For a single run the computation time was set to 3s for n smaller or equal to 120, 20s for $n = 240$, 120s for $n = 480$, and 600s for $n = 960$. We conducted 20 runs with different seeds for all three implementations. The code for the *efficient ND* implementation is available under the following link: http://doi.org/10.25592/uhhfdm.14613. The results presented in the following are average values over all 20 runs.

The aim of the computational study is to evaluate the different implementations of the insertion and the swap neighborhood. As the focus of our algorithm is on the evaluation of the neighborhood implementations, it is not that advanced as other approaches in the literature (compare Yang et al. (2022)). Moreover, the different implementations were always started with the same seed such that all of them conduct the same search. This means that an implementation leading to a larger number of executed iterations cannot lead to a worse objective value. Together, objective values are not that interesting for this study. We therefore do not report them in detail. However, note that the best objective value we found for an instance is on average 0.3% worse than the best found in the study by Yang et al. (2022). The maximum difference was 0.85%.

As already said, given a seed, the search is the same for all three implementations. Thus, a higher number of iterations cannot lead to a worse objective value but gives the chance to evaluate new and possibly better solutions. Hence, it is desirable to increase the number of performed iterations. Table 1 presents the average number of iterations over all 20 runs and all 10 instances of the instance type performed by the three implementations as well as the percentaged increase if *ND* was superior in comparison to *standard* and if *efficient ND* was superior in comparison to one of the other two. The results show that the *standard* implementation is superior for smaller instance sizes of up to 60 items while the *ND* implementation is superior for medium-sized instances with unequal group sizes. Finally, the *efficient ND* implementation is clearly superior for the large instances.

Reasons for the good performance of the *standard* implementation on small instances are that a smaller number of items goes along with a smaller number of groups such that the other two implementations cannot use the full potential of their advantage. As an example consider the smallest instances with ten items. They only have two groups. Thus, any insert or swap always changes both groups and all groups have to be evaluated in all three implementations. Moreover, matrices W^1, W^2, M^1, M^2 need to be updated which leads to a higher demand for memory access for the *ND* and the *efficient ND* implementation. A higher demand for memory access is also a reason for the better performance of *ND* in comparison to *efficient ND* for the smaller instances. For larger instances with a higher number of groups the advantage of fewer parts of the neighborhoods which need to be evaluated exceeds the disadvantage of a higher memory access. Thus, *ND* and *efficient ND* clearly outperform *standard*. Moreover, the advantage of *efficient ND* to only evaluate a neighborhood block once until a change is done becomes relevant such that *efficient ND* clearly outperforms *ND*. This results in an increase of performed iterations of up to 170% in comparison to *standard* and up to 76% in comparison to *ND*.

The extent of skipped neighborhood blocks is shown in Table 2. For smaller instances only a small number of neighborhood blocks can be skipped without evaluation due to the small number of groups. If we consider the example of the smallest instances with ten items and two groups again, we can only skip them in an iteration where no improvement was found, i.e. if we are in a local optimum. In contrast, for the instances with 960 items assigned to 24 groups only 45 of the 276 neighborhood blocks need to be evaluated after a change.

Table 1 Comparison of number of iterations

Instance type	Number of iterations Standard	ND	Efficient ND	ND > Standard [%]	Efficient ND > Standard [%]	Efficient ND > ND [%]
Geo_n010_ds	**1328760.4**	1253917.3	1164172.3	–	–	–
Geo_n010_ss	**912585.8**	878742.0	807444.0	–	–	–
Geo_n012_ds	**1397135.4**	1303226.5	1148173.0	–	–	–
Geo_n012_ss	**930057.0**	883817.3	757774.0	–	–	–
Geo_n030_ds	**1088662.3**	962470.9	860666.3	–	–	–
Geo_n030_ss	**695985.0**	642700.0	568757.2	–	–	–
Geo_n060_ds	**522577.8**	465137.7	426260.2	–	–	–
Geo_n060_ss	**305949.8**	290242.7	272054.7	–	–	–
Geo_n120_ds	168159.8	**183216.9**	178128.0	8.95	5.93	–
Geo_n120_ss	94272.6	110849.4	**116201.3**	17.58	23.26	4.83
Geo_n240_ds	346026.2	414477.1	**460535.0**	19.78	33.09	11.11
Geo_n240_ss	178800.2	218776.1	**277723.2**	22.36	55.33	26.94
Geo_n480_ds	495384.1	860330.6	**1051359.6**	73.67	112.23	22.20
Geo_n480_ss	253788.5	413658.4	**633616.3**	62.99	149.66	53.17
Geo_n960_ds	522887.5	922581.0	**1215263.1**	76.44	132.41	31.72
Geo_n960_ss	265269.5	421354.7	**718006.1**	58.84	170.67	70.40
RanInt_n010_ds	**1270304.7**	1211082.2	1127572.6	–	–	–
RanInt_n010_ss	**944238.6**	904255.5	835903.7	–	–	–
RanInt_n012_ds	**1234995.3**	1158369.6	1025393.1	–	–	–
RanInt_n012_ss	**909333.9**	867243.0	743689.3	–	–	–
RanInt_n030_ds	**940709.9**	841926.9	745116.0	–	–	–
RanInt_n030_ss	**660305.6**	615683.0	544153.1	–	–	–
RanInt_n060_ds	**475582.2**	424603.8	377443.1	–	–	–

Table 1 continued

Instance type	Number of iterations Standard	ND	Efficient ND	ND > Standard [%]	Efficient ND > Standard [%]	Efficient ND > ND [%]
RanInt_n060_ss	**299598.9**	287712.8	267480.4	–	–	–
RanInt_n120_ds	157541.7	**169680.9**	152840.9	7.71	–	–
RanInt_n120_ss	90505.1	109180.8	**109519.9**	20.63	21.01	0.31
RanInt_n240_ds	333962.9	**393649.1**	373233.3	17.87	11.76	–
RanInt_n240_ss	176223.6	226023.4	**266766.3**	28.26	51.38	18.03
RanInt_n480_ds	490897.1	815954.1	**827602.0**	66.22	68.59	1.43
RanInt_n480_ss	250140.0	412641.6	**590938.2**	64.96	136.24	43.21
RanInt_n960_ds	508146.2	857579.6	**915346.5**	68.77	80.13	6.74
RanInt_n960_ss	254502.4	378363.3	**665010.9**	48.67	161.30	75.76
RanReal_n010_ds	**1232724.4**	1176374.7	1095791.2	–	–	–
RanReal_n010_ss	**927778.8**	888113.7	820266.1	–	–	–
RanReal_n012_ds	**1251148.4**	1173539.6	1037247.8	–	–	–
RanReal_n012_ss	**875899.6**	835546.1	718689.0	–	–	–
RanReal_n030_ds	**908523.6**	815644.6	724942.5	–	–	–
RanReal_n030_ss	**657360.8**	613679.1	542723.4	–	–	–
RanReal_n060_ds	**475260.9**	424645.5	377241.2	–	–	–
RanReal_n060_ss	**299921.9**	287680.1	267570.4	–	–	–
RanReal_n120_ds	157667.6	**170007.8**	153022.0	7.83	–	–
RanReal_n120_ss	90672.7	109491.0	**109782.7**	20.75	21.08	0.27
RanReal_n240_ds	334165.0	**393128.0**	372990.4	17.64	11.62	–
RanReal_n240_ss	175451.4	225713.6	**266263.8**	28.65	51.76	17.97
RanReal_n480_ds	489814.2	813709.1	**824149.1**	66.13	68.26	1.28
RanReal_n480_ss	249559.2	411274.7	**588523.1**	64.80	135.83	43.10
RanReal_n960_ds	499633.3	846739.4	**906903.0**	69.47	81.51	7.11
RanReal_n960_ss	248328.1	370002.0	**652705.5**	49.00	162.84	76.41

Table 2 Evaluation of neighborhood evaluations

Instance type	Skipped block evaluations [%] W^1	W^2	M^1	M^2
Geo_n010_ds	2.07	2.07	2.07	2.07
Geo_n010_ss	–	30.62	–	30.63
Geo_n012_ds	7.44	9.83	7.45	12.02
Geo_n012_ss	–	28.53	–	30.93
Geo_n030_ds	16.16	18.69	16.23	24.26
Geo_n030_ss	–	29.90	–	35.91
Geo_n060_ds	25.25	25.09	25.45	32.36
Geo_n060_ss	-	34.55	–	43.21
Geo_n120_ds	46.95	44.58	47.58	54.00
Geo_n120_ss	–	49.23	–	61.77
Geo_n240_ds	55.69	47.76	56.65	61.86
Geo_n240_ss	–	48.99	–	67.41
Geo_n480_ds	71.07	62.40	72.49	76.07
Geo_n480_ss	–	59.52	–	79.49
Geo_n960_ds	76.81	62.77	78.34	80.68
Geo_n960_ss	–	58.94	–	82.81
RanInt_n010_ds	2.59	2.75	2.59	2.76
RanInt_n010_ss	-	29.86	–	29.86
RanInt_n012_ds	6.21	9.46	6.25	10.72
RanInt_n012_ss	–	29.82	–	31.81
RanInt_n030_ds	10.20	16.88	10.49	19.81
RanInt_n030_ss	–	31.09	–	36.51
RanInt_n060_ds	18.54	22.70	19.22	26.55
RanInt_n060_ss	–	34.82	–	43.51
RanInt_n120_ds	38.79	42.44	40.39	47.46
RanInt_n120_ss	–	50.50	–	61.68
RanInt_n240_ds	43.79	45.74	46.86	53.41
RanInt_n240_ss	–	50.42	–	67.14
RanInt_n480_ds	59.26	59.92	63.91	69.49
RanInt_n480_ss	–	59.19	–	78.80
RanInt_n960_ds	64.15	60.59	70.65	74.71
RanInt_n960_ss	–	55.05	–	82.39
RanReal_n010_ds	2.60	2.76	2.60	2.76
RanReal_n010_ss	–	30.25	–	30.26
RanReal_n012_ds	6.68	9.85	6.71	11.22
RanReal_n012_ss	–	30.09	–	32.04
RanReal_n030_ds	10.19	16.88	10.47	19.77
RanReal_n030_ss	–	31.13	–	36.52
RanReal_n060_ds	18.73	22.79	19.40	26.69
RanReal_n060_ss	–	34.80	–	43.51

Table 2 continued

Instance type	Skipped block evaluations [%] W^1	W^2	M^1	M^2
RanReal_n120_ds	38.75	42.40	40.35	47.41
RanReal_n120_ss	–	50.50	–	61.67
RanReal_n240_ds	43.89	45.80	46.94	53.49
RanReal_n240_ss	–	50.41	–	67.14
RanReal_n480_ds	59.33	59.95	63.95	69.52
RanReal_n480_ss	–	59.17	–	78.80
RanReal_n960_ds	64.23	60.64	70.69	74.76
RanReal_n960_ss	–	55.05	–	82.39

If we use the *ND* implementation, we moreover need to evaluate all of the others if there is a promising insert/swap. Consequently, the share of skipped block evaluations increases to more than 82% for the swap neighborhood and over 70% for the insertion neighborhood if we use the *efficient ND* implementation.

The values are smaller if the swap as well as the insertion neighborhood are used, i.e. for the instances with unequal-sized groups. One reason is that both neighborhoods are evaluated alternately. Thus, up to four groups were changed instead of up to two before the same neighborhood is evaluated next (can be more if perturbation is executed meanwhile).

Table 3 shows the success rates of the two neighborhoods as well as the average number of iterations between two consecutive calls of the perturbation algorithms 7 and 8, respectively. The success rate is the percentage of calls of the neighborhood in which an improved solution was found. It is not surprising that more iterations were performed before the algorithm reaches a local optimum, i.e. perturbation is required, if the instance size is larger. Thus, the success rates of both neighborhoods increase with the instance size. Especially the swap neighborhood with a success rate of up to 91% is very effective.

The insertion neighborhood reaches a success rate of only about 50%. Thus, there are more iterations without any change in the current solution which explains why the *ND* implementation is more effective for instances with unequal group sizes. If an iteration is unsuccessful, no promising neighborhood block exists. This also means that in the next call of the insertion neighborhood only those neighbourhood blocks need to be evaluated which were changed due to a swap in the swap neighborhood which was called in the meantime. Thus, *ND* is as effective as *efficient ND* and clearly superior to *standard* in this case. Our implementation with the alternate calls of both neighborhoods follows Lai and Hao (2016). The small success rate of the insertion neighborhood, however, might be an argument to follow another policy in future approaches.

5 Conclusion

The paper compares the three implementations called *standard*, *ND*, and *efficient ND* of the insertion and the swap neighborhood for the MDGP. The *efficient ND* implementation is newly introduced and based on the *ND* implementation. Both implementations use the idea that the neighborhoods can be divided into independent blocks containing the inserts/swaps between two groups. While the *ND* implementation evaluates a block if there is a promising,

Table 3 Evaluation of algorithm design

Instance type	Success rate Insertion [%]	Success rate Swap [%]	Perturbation (no. it.) Weak	Strong
Geo_n010_ds	3.41	60.42	5.1	32.9
Geo_n010_ss	–	41.77	3.4	22.6
Geo_n012_ds	19.29	70.48	6.9	51.2
Geo_n012_ss	–	58.42	4.8	32.8
Geo_n030_ds	13.53	80.92	10.5	95.0
Geo_n030_ss	–	72.06	7.2	69.7
Geo_n060_ds	19.86	82.75	11.7	101.4
Geo_n060_ss	–	74.49	7.8	75.4
Geo_n120_ds	21.25	85.59	14.0	104.0
Geo_n120_ss	–	78.81	9.4	77.9
Geo_n240_ds	18.32	89.83	19.8	143.6
Geo_n240_ss	–	83.96	12.5	89.9
Geo_n480_ds	17.61	93.61	31.4	134.4
Geo_n480_ss	–	90.43	20.9	86.2
Geo_n960_ds	13.43	96.77	62.0	276.7
Geo_n960_ss	–	95.08	40.7	170.9
RanInt_n010_ds	18.88	54.95	4.8	31.4
RanInt_n010_ss	–	43.59	3.5	21.7
RanInt_n012_ds	32.06	63.80	6.1	47.5
RanInt_n012_ss	–	56.00	4.5	30.8
RanInt_n030_ds	38.54	74.29	8.5	72.0
RanInt_n030_ss	–	69.37	6.5	57.1
RanInt_n060_ds	45.67	76.91	9.3	74.6
RanInt_n060_ss	–	72.29	7.2	55.9
RanInt_n120_ds	46.63	78.78	10.1	74.4
RanInt_n120_ss	–	74.29	7.8	51.5
RanInt_n240_ds	52.38	81.73	11.8	87.1
RanInt_n240_ss	–	78.84	9.5	59.4
RanInt_n480_ds	51.82	87.45	17.0	76.2
RanInt_n480_ss	–	84.14	12.6	51.0
RanInt_n960_ds	51.09	91.39	24.5	110.6
RanInt_n960_ss	–	88.78	17.8	71.8
RanReal_n010_ds	17.83	53.61	4.7	31.2
RanReal_n010_ss	–	42.79	3.5	21.8
RanReal_n012_ds	27.78	65.44	6.1	48.4
RanReal_n012_ss	–	55.33	4.5	30.6
RanReal_n030_ds	39.00	74.17	8.5	71.1

Table 3 continued

Instance type	Success rate Insertion [%]	Success rate Swap [%]	Perturbation (no. it.) Weak	Strong
RanReal_n030_ss	–	69.35	6.5	56.4
RanReal_n060_ds	45.06	77.13	9.4	75.0
RanReal_n060_ss	–	72.34	7.2	56.0
RanReal_n120_ds	46.86	78.76	10.1	74.5
RanReal_n120_ss	–	74.30	7.8	51.4
RanReal_n240_ds	52.06	81.82	11.9	87.2
RanReal_n240_ss	–	78.87	9.5	59.5
RanReal_n480_ds	51.68	87.49	17.0	76.4
RanReal_n480_ss	–	84.14	12.6	51.0
RanReal_n960_ds	50.78	91.43	24.6	111.0
RanReal_n960_ss	–	88.79	17.8	71.8

i.e. an improving insert/swap, the *efficient ND* implementation evaluates each block only if the assignment to at least one of its groups has changed.

All three neighborhood implementations were compared in an extensive computational study on the classical benchmark sets. The results show that the *standard* implementation is superior for small instance sizes while the *efficient ND* implementation is superior for large instance sizes. The *ND* implementation performed best for medium sized instances with unequal group sizes. A reason for it is that the insertion neighborhood found only in around half of the cases an improved solution. For the large instances the *efficient ND* implementation could perform up to 160% more iterations in comparison to *standard* and up to 76% more iterations in comparison to *ND*.

Our results lead to several directions for future research. First, the new neighborhood implementation can be used for other grouping or clustering problems. These include as mentioned in the introduction problems like the capacitated clustering problem or the capacitated p-median problem as well as the ratio cut and normalized cut graph partitioning problem, vehicle routing problems, or parallel machine scheduling (Yalaoui & Chu, 2002). Second, we found that the insertion neighborhood has a success rate of only about 50% if both neighborhoods, insertion and swap, are called alternately. Future research could evaluate other policies to increase the neighborhoods' success rates. Finally, future research can extend our approach by using the matrices M^1 and M^2 to determine more than one promising insert/swap per iteration. The matrices indicate the best insert/swap between two groups. As long as group pairs are disjunct one could also realize additional neighborhood moves in the same iteration. The best selection can be determined by a maximum weighted matching.

Funding Open Access funding enabled and organized by Projekt DEAL. No funding was received for conducting this study.

Declaration

Conflict of interest The author has no Conflict of interest to declare that are relevant to the content of this article.

References

Baker, B. M., & Benn, C. (2001). Assigning pupils to tutor groups in a comprehensive school. *Journal of the Operational Research Society, 52*(6), 623–629.

Baker, K. R., & Powell, S. G. (2002). Methods for assigning students to groups: A study of alternative objective functions. *Journal of the Operational Research Society, 53*, 397–404.

Batista, R. M., Mao, E., & Sussman, A. B. (2023). Keeping cash and revolving debt: Examining co-holding in the field and in the lab. Available at SSRN 4558490

Beheshtian-Ardekani, M., & Mahmood, M. A. (1986). Education development and validation of a tool for assigning students to groups for class projects. *Decision Sciences, 17*(1), 92–113.

Brimberg, J., Mladenović, N., & Urošević, D. (2015). Solving the maximally diverse grouping problem by skewed general variable neighborhood search. *Information Sciences, 295*, 650–675.

Brusco, M. J., Cradit, J. D., & Steinley, D. (2020). Combining diversity and dispersion criteria for anticlustering: A bicriterion approach. *British Journal of Mathematical and Statistical Psychology, 73*(3), 375–396.

Chen, Y., Fan, Z. P., Ma, J., & Zeng, S. (2011). A hybrid grouping genetic algorithm for reviewer group construction problem. *Expert Systems with Applications, 38*(3), 2401–2411.

Dias, T. G., & Borges, J. (2017). A new algorithm to create balanced teams promoting more diversity. *European Journal of Engineering Education, 42*(6), 1365–1377.

Fan, Z. P., Chen, Y., Ma, J., & Zeng, S. (2011). Erratum: A hybrid genetic algorithmic approach to the maximally diverse grouping problem. *Journal of the Operational Research Society, 62*(7), 1423–1430.

Feo, T. A., & Khellaf, M. (1990). A class of bounded approximation algorithms for graph partitioning. *Networks, 20*(2), 181–195.

Gallego, M., Laguna, M., Martí, R., & Duarte, A. (2013). Tabu search with strategic oscillation for the maximally diverse grouping problem. *Journal of the Operational Research Society, 64*(5), 724–734.

Lai, X., & Hao, J. K. (2016). Iterated maxima search for the maximally diverse grouping problem. *European Journal of Operational Research, 254*(3), 780–800.

Lai, X., Hao, J. K., Fu, Z. H., & Yue, D. (2021). Neighborhood decomposition based variable neighborhood search and tabu search for maximally diverse grouping. *European Journal of Operational Research, 289*(3), 1067–1086.

Lai, X., Hao, J. K., Fu, Z. H., & Yue, D. (2021). Neighborhood decomposition-driven variable neighborhood search for capacitated clustering. *Computers & Operations Research, 134*, 105362.

Mingers, J., & O'Brien, F. A. (1995). Creating student groups with similar characteristics: A heuristic approach. *Omega, 23*(3), 313–321.

Palubeckis, G. (2022). Metaheuristic approaches for ratio cut and normalized cut graph partitioning. *Memetic Computing, 14*(3), 253–285.

Palubeckis, G., Karčiauskas, E., & Riškus, A. (2011). Comparative performance of three metaheuristic approaches for the maximally diverse grouping problem. *Information Technology and Control, 40*(4), 277–285.

Palubeckis, G., Ostreika, A., & Rubliauskas, D. (2015). Maximally diverse grouping: An iterated tabu search approach. *Journal of the Operational Research Society, 66*(4), 579–592.

Papenberg, M. (2024). K-plus anticlustering: An improved k-means criterion for maximizing between-group similarity. *British Journal of Mathematical and Statistical Psychology, 77*(1), 80–102.

Papenberg, M., & Klau, G. W. (2021). Using anticlustering to partition data sets into equivalent parts. *Psychological Methods, 26*(2), 161.

Pfeiffer, C., & Schulz, A. (2022). An alns algorithm for the static dial-a-ride problem with ride and waiting time minimization. *OR Spectrum, 44*(1), 87–119.

Rodriguez, F. J., Lozano, M., García-Martínez, C., & González-Barrera, J. D. (2013). An artificial bee colony algorithm for the maximally diverse grouping problem. *Information Sciences, 230*, 183–196.

Rubin, P. A., & Bai, L. (2015). Forming competitively balanced teams. *IIE Transactions, 47*(6), 620–633.

Schulz, A. (2021). The balanced maximally diverse grouping problem with block constraints. *European Journal of Operational Research, 294*(1), 42–53.
Schulz, A. (2022). A new mixed-integer programming formulation for the maximally diverse grouping problem with attribute values. *Annals of Operations Research, 318*(1), 501–530.
Schulz, A. (2023). The balanced maximally diverse grouping problem with attribute values. *Discrete Applied Mathematics, 335*, 82–103.
Urošević, D. (2014). Variable neighborhood search for maximum diverse grouping problem. *Yugoslav Journal of Operations Research, 24*(1), 21–33.
Weitz, R. R., & Lakshminarayanan, S. (1998). An empirical comparison of heuristic methods for creating maximally diverse groups. *Journal of the Operational Research Society, 49*(6), 635–646.
Yalaoui, F., & Chu, C. (2002). Parallel machine scheduling to minimize total tardiness. *International Journal of Production Economics, 76*(3), 265–279.
Yang, X., Cai, Z., Jin, T., Tang, Z., & Gao, S. (2022). A three-phase search approach with dynamic population size for solving the maximally diverse grouping problem. *European Journal of Operational Research, 302*(3), 925–953.
Zheng, S., Lai, X., & Gong, W. (2021). Neighborhood decomposition based variable neighborhood search for the capacitated p-median problem. In: 2021 China Automation Congress (CAC), IEEE, pp. 1101–1105
Zhou, Y., Kou, Y., & Zhou, M. (2023). Bilevel memetic search approach to the soft-clustered vehicle routing problem. *Transportation Science, 57*(3), 701–716.