Group Allocation System

Overview

The Group Allocation System is a Python-based application designed to allocate students into groups based to maximise the in-group differences between group members based on certain criteria e.g. gender, age, qualifications and to minimise violations. Its core algorithm is based on the one described in MCADSS however with some modifications. The application is run using a GUI (Graphical user Interface) to allow users to select an Excel file containing student data, configure weights, and run the allocation algorithm.

Inputs

The program takes an excel file as an input which must be provided before running. The format of the excel file is in ***this section***. Other inputs which are loaded with a default value:

- `Students per Group`: The expected number of people per group Denoted by m. Default value: 5
- `Number of Semesters`: Number of repetitions of semesters (or how many group sets should be created with same students and same number of students per group). Denoted by s. Default value: 2 semesters
- `Number of Trials`: How many times the program shoud run to allocate groupings and choose the best group set for each semester. Denoted by t. Default value: 100 trials
- `Weights` (more on them in later sections)

Let the total students in the provided excel file be n, then the number of groups g that will be made will be the integer part of $\frac{n}{m}$. It will be written using *floor function* from now on as $\lfloor \frac{n}{m} \rfloor$. It returns the integer part of the division e.g. $\lfloor \frac{73}{5} \rfloor = 14$

Hence, the remaining extra students - lets denote with r - will be the remainder part of $\frac{n}{m}$. It will be written using *mod function* as $n \mod m$. It returns the integer part of the division e.g. $73 \mod 5 = 3$. Therefore exactly 3 groups will each have one extra student.

Difference with MCADSS

The algorithm is similar in MCADSS, which initially chooses a student at random, allocates it to the first group, and then keep choosing the next most similar student based on *difference_weights* and places it in the next group. Therefore, the final outcome becomes highly dependent on the first randomly chosen student.

However, this implementation expands on the parts that were not mentioned or not explained in depth in the publication.

Even distribution

The first priority in allocating students is ensuring that all groups have an even distribution of students. Taking the previous example of $n = 73$ and $m = 5$:

- There should be 14 Groups in each group set ($g = \lfloor \frac{n}{m} \rfloor = 14$)
- 3 Groups should each have an extra student - total of 6 students ($r = n \mod m = 3$)
- Hence followed from previous point, the algorithm ensures that 11 groups have 5 students and 3 groups have 6 students.

Following the core MCADSS algorithm ensures maximum in-group diversity however groups will often be made with variable sizes. To balance out group sizes after all students are allocated, a `redistribution()` function is called that reassigns students and balances the group sizes out. Note: In the case where $r = 0$, the algorithm will ensure that each group has strictly `m` students.

Constraint violations

MCADSS states that if placing a student in a group violates any constraint, the student should be placed in the next group. However, it is often the case that placing the student in the next group also results in violation(s), or that placing the student in any group will lead to some violation(s). To address this:

- A general list of violations are described, where each violation has some weight related to it. These can be configured in the interface.
- For each group, the sum of the weights for all violated constraints is calculated when a student is added to it.
- If all the groups have violations, then the student is placed in the group with the minimum violation. If two or more groups have the same sum of violation(s), then the one closer to the starting group is chosen.

Additional Features

Triplets repetition

To ensure that no three or more students are placed in the same group across different semesters (or groupsets), the algorithm checks for triplets in past groupings before assigning a student to a group. If placing a student in a group would result in a triplet that already existed in a previous groupset, that group is skipped, regardless having any or no violation.

Trials and scoring

Since the algorithm's outcome is significantly influenced by the initial student selection, running it only once may yield suboptimal results. To mitigate this, the algorithm performs t trials for each group set allocation. A score is then calculated for each trial, considering the following factors:

- `avg_diff`: The average difference among students across all groups.
- `min_diff`: The smallest difference found in any group.
- `max_diff`: The largest difference found in any group.
- `avg_viol`: The average violation score across all groups.
- `min_viol`: The smallest violation score found in any group.
- `max_viol`: The largest violation score found in any group.

The goal is to maximize the first three factors and minimize the last three. A simple intuitive formula which can be used to achieve this is:

$$\frac{avg\_diff + min\_diff + max\_diff}{avg\_viol + min\_viol + max\_viol}$$

However, different scenarios and requirements may arise depending on the use case. Therefore, weights can be applied to each factor:

$$\frac{AVG\_DIFF\_WEIGHT \cdot avg\_diff + MIN\_DIFF\_WEIGHT \cdot min\_diff + MAX\_DIFF\_WEIGHT \cdot max\_diff}{AVG\_VIOL\_WEIGHT \cdot avg\_viol + MIN\_VIOL\_WEIGHT \cdot min\_viol + MAX\_VIOL\_WEIGHT \cdot max\_viol}$$

These weights can be configured in the Weights section in the interface.

Constraints

The exact constraints are ommitted in this documenatation for confidential reasons. However, to get an idea, they are constraints aimed to balance out different criteria e.g. even distribution of students' gender, scholarship students v.s. those with none etc.

Excel Format

Omitted

Additional Notes

1. In some scenarios, if the value of n is not significantly larger than m, the chosen `Students per Group` might not be suitable. For example, consider a scenario where $n = 19$ and $m = 5$. The number of groups g would be 3, and the number of extra students to be distributed among these groups would be 4. However, since $3 < 4$, there can be an uneven distribution of students hence the `Students per Group` should be increased to 6 to avoid it.

Generally, this constraint is violated if:

$$m^2 > n \text{ and } r > g$$

The second condition can be also be expressed as:

$$n \mod m > \left\lfloor \frac{n}{m} \right\rfloor$$

Hence, for good practice, choose a value for m such that:

$$m^2 \leq n$$

2. Since there are n possible starting students, there will be n possible outcomes for groupings for the first semester. Each successive semester will have $n-1, n-2...$ and so on possible outcomes since the starting student(s) cannot be chosen again as starting. This can continue until possibilities left are 0, meaning all n possible are exhausted for all semesters s. Hence ensure that:

   $$s \leq n$$

   If you input any larger value, the program will set s to be n.

3. Similarly, with n potential starting students, the first semester grouping can perform up to n trials at max. If the input value for trials t exceeds n, the program defaults to n trials. For each subsequent grouping, the trial count decreases by one since a starting student is no longer available. Additionally, if the remaining number of students is fewer than the specified trials, the program automatically adjusts the trial count to match the available students. This adjustment is to account for s; if successive groupings leave fewer than t students remaining, the program sets the number of trials to the number of remaining students.

4. There is obviously an upper bound limit of how many semesters s (or group sets) can be formed given n students and forming g groups, each with atleast m students due to triplets repeating. A formula is yet to be found, however this number increases somewhat with increasing number of students n, but decreases significantly with increasing number of (minimum) people in a group m. The program can go into an infinite loop if m is too large.

5. Due to the randomized nature of the algorithm and the triplets constraint, assigning a student to the best candidate group might be skipped because of triplet repetitions in previous group sets. This probability increases with each new group set formed, making it likely that subsequent semesters or group sets are less ideal, with decreased quality and diversity within groups. Therefore, the first group set always has the best chance at optimal groupings.

6. Time Complexity: If the number of semesters s is 1, the runtime is $O(min(t, n) \cdot n^2)$.
Otherwise, it's $O(min(s, n) \cdot max(min(t, n) \cdot n^2, n \cdot m^2))$ because the algorithm stores triplet IDs after each semester's grouping which costs $n \cdot m^2$.