Automated Student Project Allocation

Walid LAHDADI - Majda KEMMOU - Oussama SOUIDI

June 16, 2025

Abstract

This report presents the development of an algorithm and tool for the automatic allocation of students to projects. Initially, we explored the Gale-Shapley algorithm, but due to challenges with many-to-one allocations, we shifted our focus to the Student Project Allocation (SPA) algorithm. We also developed a web interface for both students and administrators to facilitate the allocation process. The results of our implementation are discussed along with a comparison to the traditional manual allocation method.

1 Introduction

The allocation of students to projects is a critical task in educational institutions, which has an impact on both student satisfaction and project results. At EURECOM, this process was performed manually, which can be time consuming and prone to errors. The goal of this project was to develop an automated tool that utilizes algorithmic methods to streamline this process, ensuring fair and efficient assignment of students to projects.

The importance of this project lies in its ability to automate a time-consuming task while optimizing student satisfaction through more effective project allocations. By aligning students with opportunities that match their interests and skills, we aim to enhance their overall educational experience. This report outlines the methodologies employed, the algorithms developed, and the results achieved from our implementation.

2 Background

2.1 Gale-Shapley algorithm

The Gale-Shapley algorithm, introduced by David Gale and Lloyd Shapley in their seminal paper *'College Admissions and the Stability of Marriage'* [3], addresses the problem of stable marriage, which seeks to find a stable match between two sets of elements of the same size. In the context of student project allocation, one set consists of students and the other of projects.

The algorithm operates through a series of proposals and rejections, ultimately leading to a stable match where no two elements would prefer each other over their current matches. The steps of the Gale-Shapley algorithm can be summarized as follows:

1. Each student ranks the projects in preference order.

2. Each project ranks the students according to their preferences.

3. Students propose their project of top choice.

4. Each project considers the proposals it has received and tentatively accepts the best candidates according to its preference list, rejecting the others.

5. Rejected students propose their next choice, and the process repeats until no student wishes to propose.

The stability of the matches produced by the Gale-Shapley algorithm is guaranteed, as demonstrated in the original paper. However, the applicability of this algorithm to our project was constrained by the many-to-one allocation scenario, where multiple students could be assigned to a single project. To address this challenge, we explored methods to transform the many-to-one problem into a one-to-one configuration, thereby aligning more closely with the Gale-Shapley framework.

One approach we considered was inspired by the Maryland lecture [5], which suggested duplicating projects to create a one-to-one matching scenario. For instance, if a project has a quota (Q) of 2, we could represent this project as two separate entities, P_1 and P_2, each with a quota of 1:

$$P_1, P_2 \in \{P\} \quad , \quad \mathrm{Q}(P_1) = \mathrm{Q}(P_2) = 1$$

This duplication allows us to apply the Gale-Shapley algorithm more effectively. However, it is crucial to ensure that the preferences of the duplicated projects are shuffled. This shuffling is necessary to maintain the stability of the matches, as outlined in the Gale-Shapley framework. If the preferences of P_1 and P_2 are identical, it could lead to instability in the matching process.

Additionally, we explored the idea of forming sets of students, inspired by Chapter 16 of "Two-Sided Matching" by Alvin E. Roth and Marilda Sotomayor [4]. This chapter discusses the allocation of workers to jobs and suggests that grouping students could lead to more efficient outcomes. By treating combinations of students or groups as single entities in the allocation process, we could potentially streamline the matching. However, this approach posed significant computational costs, as the number of combinations increases exponentially with the number of students. If we denote the set of students as S and the number of students as n, the number of possible combinations can be represented as:

$$C(n, k) = \frac{n!}{k!(n-k)!}$$

where k is the size of the group. This combinatorial explosion could lead to impractical computation times, making it a less viable option for our implementation.

In summary, the exploration of these methods highlights the challenges and considerations involved in adapting established algorithms to fit the specific requirements of our many-to-one allocation problem, drawing on foundational concepts from both the Maryland lecture and Roth and Sotomayor's work.

2.2 Challenges with Gale-Shapley

The primary challenge encountered with the Gale-Shapley algorithm was its inability to effectively manage many-to-one allocations. In scenarios where a project could accept multiple students, the algorithm's inherent structure led to complications in ensuring that all students received their most preferred projects without creating instability. The need for a more flexible approach prompted us to explore alternative algorithms that could better accommodate the requirements of our allocation problem.

2.3 Student Project Allocation (SPA) algorithm

In response to the limitations of the Gale-Shapley algorithm, we transitioned to the Student Project Allocation algorithm. The SPA algorithm is specifically designed to handle many-to-one allocations, making it a more suitable choice for our project. The key features of the SPA algorithm include:

- It allows students to rank multiple projects, while each project can ac-

cept multiple students within defined quotas.

- It supports lower and upper quotas, making it suitable for projects with flexible capacity.
- It aims to produce stable matchings that respect student preferences.

In the following section, we present our proposed optimization-based approach tailored to address this problem effectively.

3 Proposed algorithm and mathematical model

3.1 MILP-Based Allocation Model

To solve the student-project allocation problem, we formulate it as a Mixed Integer Linear Programming (MILP) problem. The objective is to assign each student to one of their ranked projects while ensuring that each project's minimum and maximum capacity constraints are respected. Moreover, the model seeks to enhance fairness and satisfaction by minimizing the worst rank assigned. Our approach is inspired by the MILP formulation proposed in the study *"Handling Preferences in Student-Project Allocation"* by Chiarandini et al. [2].

We introduce the following decision variables:

- $x_{s,p} \in \{0,1\}$: equals 1 if student s is assigned to project p, 0 otherwise.
- $y_p \in \{0,1\}$: equals 1 if project p is active (i.e., has students assigned), 0 otherwise.
- $z \in \mathbb{Z}^+$: the worst rank assigned across all students.

The model includes the following constraints:

1. Unique assignment constraint: Each student must be assigned to exactly one project among their ranked choices:

$$\sum_{p \in \text{Choices}(s)} x_{s,p} = 1 \quad \forall s$$

2. Project capacity constraints: The number of students assigned to a project must respect both the minimum and maximum quota:

$$\sum_{s: p \in \text{Choices}(s)} x_{s,p} \geq \text{min\_quota}(p) \cdot y_p \quad \forall p$$

$$\sum_{s: p \in \text{Choices}(s)} x_{s,p} \leq \text{max\_quota}(p) \cdot y_p \quad \forall p$$

3. Worst rank tracking constraint: The variable z captures the worst rank assigned among all students:

$$z \geq \sum_{p \in \text{Choices}(s)} \text{Rank}(s,p) \cdot x_{s,p} \quad \forall s$$

Here, $\text{Rank}(s,p) \in \{1,2,3\}$ depending on whether p is the student's first, second, or third choice.

Objective function: The objective is to balance fairness and satisfaction. First, we minimize the worst rank z, then we minimize the total dissatisfaction (sum of ranks). The combined objective function is:

$$\min \left(1000 \cdot z + \sum_{s} \sum_{p \in \text{Choices}(s)} \text{Rank}(s,p) \cdot x_{s,p} \right)$$

The large coefficient on z prioritizes fairness (minimizing the worst rank) over total satisfaction.

Optimization steps:

1. Parse student preferences and project quotas.
2. Create binary decision variables for all feasible student-project assignments.
3. Add constraints to ensure unique assignment and capacity compliance.
4. Include the stability constraint using variable z.
5. Define and solve the MILP using the PuLP CBC solver.
6. Post-process the solution to generate statistics and assignment results.

This model guarantees that all students are matched to projects fairly and efficiently while maximizing satisfaction across the cohort.

4 Implementation

4.1 Web interface development

To enhance user experience, we developed a web interface consisting of two parts: one for students and one for administrators. The student interface allows users to view available projects, their quotas, and rank their top three choices. The administrator interface enables the admin to run the matching algorithm and generate an Excel file with the final allocations.

The web interface was designed with usability in mind, ensuring that students could easily navigate the system and submit their preferences. The administrator interface was developed to facilitate the efficient execution of the matching algorithm, allowing for quick adjustments and real-time updates.

4.2 Web interface screenshots

To illustrate the functionality of our web interfaces, we present screenshots of both the student and administrator interfaces below.

Figure 1: Student interface: project visualisation

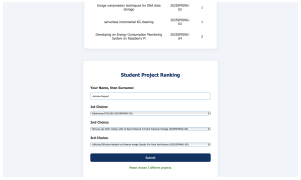

Figure 2: Student interface: project ranking

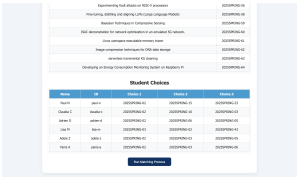

Figure 3: Administrator interface: student choices visualisation

Figure 4: Administrator interface: allocation management

4.3 Algorithm execution

The matching algorithm was implemented on a backend server hosted on Render. Upon the deadline for student preferences, the administrator can execute the SPA algorithm, which processes the submitted rankings and generates the final allocations.

The output is an Excel file containing the matched students and projects, providing a clear overview of the allocation results.

5 Results

The implementation of the SPA algorithm and the web interface has yielded promising results. We conducted a performance evaluation of our algorithm by comparing the results of the automated allocation with previous manual allocations. The evaluation metrics included time efficiency and satisfaction rate.

5.1 Performance evaluation

We applied our method to the student-project allocation for the second semester at EURECOM, involving 72 students. The results demonstrate the effectiveness of the proposed approach: 59 students were assigned to their first-choice project, 8 to their second choice, and 5 to their third choice. This means that over 81% of students received their top preference, and all students were successfully matched within the defined project quotas.

To quantify the overall quality of the allocation, we computed a satisfaction rate based on a comparison between the achieved assignment scores and the best- and worst-case scenarios. The resulting satisfaction score was approximately 92%, indicating a high level of global satisfaction and fairness, with minimal deviation from students' stated preferences.

6 SPA algorithm implementation

In parallel, we implemented another student-oriented algorithm based on the SPA-student variant described by Abraham et al. [1], adapted to handle real-world constraints such as partial preferences and project quotas.

6.1 Problem overview

Traditional stable matching algorithms like Gale-Shapley assume complete preference lists and one-to-one relationships. However, our context includes:

- Students ranking only a subset of projects.
- Projects (teachers) ranking students.
- Capacity limits on both projects and supervising staff.

Our goal is to ensure a **stable matching**, meaning that no student and project would both prefer to be assigned to each other over their current match.

6.2 Algorithm description

- Students apply to projects in order of preference.
- Projects tentatively accept students up to their quota, based on their preferences.
- If a project exceeds its quota, the least preferred student is rejected.
- Rejected students apply to their next choice until all are either matched or have no remaining options.

6.3 Features

- Handles partial preferences and missing ranks.
- Respects both project and teacher quotas.
- Ensures stable matchings under capacity constraints.

6.4 Issues and advantages

Challenge: Some students may remain unmatched if their ranked projects are full and reject them due to teacher preferences. **Advantage:** The algorithm ensures no blocking pairs exist, and student proposals ensure student-optimal matchings within constraints.

6.5 SPA matching algorithm – Pseudocode

```
Input:
    student_prefs: student -> list of
    preferred projects
    project_quotas: project -> capacity
    project_prefs: project -> list of
    preferred students

Initialize:
    unmatched_students <- list of all
    students
    assignments <- empty map

While unmatched_students is not empty:
    student <- remove one student from
    unmatched_students
    if student has no remaining
    preferences:
        continue
    project <- student's first preferred
    project
    Tentatively assign student to project

    if number of assigned students to
    project > project_quotas[project]:
        Remove least preferred student
    from project
        Add them back to
    unmatched_students

    if project is full:
        Remove from project_prefs all
    students less preferred
        than the current least preferred
    assigned student

Return assignments
```

In the end we have chosen the algotithm described in "3 Proposed Algorithm and Mathematical Model", since it is an algorithm more likely to generate a full matching so no non-allocated students by trying to minimize the z value described in whereas the advantage of the algorithm above described in "6 SPA Algorithm Implementation" is to take into account the specificity of the project, (ie. a cybersecurity project would prefer a student in the cybersecurity track).

7 Comparison with manual allocation

We compared the results of our proposed automatic allocation method with the manual allocation previously carried out by Eurecom. The objective of the comparison is to evaluate how effectively each method satisfies student preferences.

- **Total number of students:** 72

Proposed method (automated allocation):

- Rank 1: 61 students (84.7%)
- Rank 2: 7 students (9.7%)
- Rank 3: 4 students (5.6%)
- No Match: 0 students

Manual allocation (Eurecom):

- Rank 1: 59 students (81.9%)
- Rank 2: 8 students (11.1%)
- Rank 3: 4 students (5.6%)
- No Match: 1 student (Bahaeddine Melki was assigned a project not included in his top three preferences)

From the comparison, our automated method slightly outperforms the manual allocation, assigning more students to their first choice and ensuring that all students receive one of their top three preferences. The manual method, while close in performance, resulted in one student receiving a project outside their preference list.

This suggests that the automated approach not only improves fairness but also ensures stricter adherence to student preferences, reducing dissatisfaction and potential conflicts.

8 Conclusion

In this project, we developed an automated student project allocation system addressing the many-to-one matching problem common in academia. While classical algorithms like Gale-Shapley proved insufficient for handling real-world constraints such as quotas and partial preferences, we formulated a Mixed Integer Linear Programming model that ensures fairness and minimizes dissatisfaction.

We also implemented a variant of the Student Project Allocation algorithm that incorporates both student and project preferences, offering a more realistic approach. The automated system improved allocation outcomes compared to manual methods, achieving full student assignments while increasing satisfaction. A user-friendly web interface further streamlined the process for both students and administrators.

Finally, we extended the framework to solve a timetabling problem using iterative forward search and genetic algorithms, which further shows the versatility of algorithmic solutions for academic resource management.

Appendix: Class timetabling using iterative forward search

As a related problem, we explored the room-time allocation challenge in university timetabling, where classes must be assigned to rooms and time slots while avoiding conflicts and minimizing penalties related to preferences.

Problem definition

Each class has:

- A list of preferred time slots and rooms.
- A designated instructor (who must not have time conflicts).

The goal is to:

- Avoid room/time/instructor conflicts (hard constraints).
- Minimize total dissatisfaction based on preferences (soft constraint).

Iterative forward search algorithm

We implemented Müller's Iterative Forward Search method to solve this problem.

```
Procedure IterativeForwardSearch(classes,
    time_slots, rooms):
    best_penalty <- infinity
    best_assignments <- empty

    Repeat 1000 times:
        Shuffle classes
        For each class:
            Generate valid (time, room)
    pairs
            Sort by penalty
            Assign first conflict-free
    pair

        Compute total penalty
        If better than best_penalty:
            Update best_assignments

Return best_assignments
```

Appendix 2: Second method proposed for class timetabling

.1 Overview

To build a valid class timetable, we used a Genetic Algorithm (GA). This is a method inspired by natural selection, where we try different timetables, keep the best ones, and

slightly change them over time to improve the result. In the end, we also use a small optimization step to polish the timetable.

.2 Data description

Our algorithm works with the following input data:

- **Courses:** Each course has a name, a teacher, number of students, and a duration (either short or long). A **short course** takes place during half of the semester, while a **long course** takes place during the whole semester.
- **Rooms:** Each room has a name, a type (e.g., classroom or amphitheater), and a maximum number of seats.
- **Timeslots:** These are combinations of days and times (like Monday morning or Wednesday afternoon). We used 10 timeslots in total and excluded Thursday evening.

We also consider the teacher's preferences and unavailability for certain times.

.3 Timetable representation

Each possible timetable is called an **individual**. It assigns each course to:

- A timeslot (when the course happens),
- A room (where it happens),
- A semester part: `first`, `second`, or `whole`. Short courses are assigned to either the first or second half of the semester. Long courses are assigned to the full semester.

.4 Genetic algorithm

The algorithm works in the following steps:

- **Start:** Create a random set of timetables.
- **Evaluate:** Give a score to each timetable. Good ones have no conflicts and respect preferences.
- **Selection:** Keep the better ones.
- **Crossover:** Mix parts of two timetables to create a new one.
- **Mutation:** Randomly change some parts of the timetable (like the timeslot or room of a course).
- **Repeat:** Do this for many rounds to improve the solution step by step.

.5 Fitness function

We measure how good a timetable is using a scoring system:

- **Hard constraints (must be respected):**
  - A teacher cannot teach two classes at the same time.
  - A room cannot host two courses at the same time.
  - The room must have enough seats.
- **Soft constraints (preferences):**
  - Try to place courses in preferred times for teachers.
  - Avoid placing courses when the teacher is unavailable.

Hard constraint violations are heavily penalized. Soft constraint violations have smaller penalties.

.6 Improvement step (local search)

After the Genetic Algorithm gives us a good timetable, we apply a small improvement step:

- Try moving a course to a better timeslot or room.
- Try switching the semester part (only for short courses).
- Only keep changes that make the timetable better.

.7 Final result format

The final timetable is saved in a simple format. Each line looks like this:

```
(Course, Room, Timeslot, Semester
                Part)
```

Example:

```
("Machine Learning", "Room B",
    "Tuesday_AM", "second")
```

This means the course "Machine Learning" will be held in Room B, on Tuesday morning, during the second half of the semester.

This setup gave us a valid timetable with no major conflicts and very few preference issues.

References

[1] David J Abraham, Robert W Irving, and David F Manlove. Two algorithms for the student-project allocation problem. *Journal of Discrete Algorithms*, 5(1):79–91, 2007.

[2] Marco Chiarandini, Rolf Fagerberg, and Stefano Gualandi. Handling preferences in student-project allocation. *Annals of Operations Research*, 2019.

[3] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[4] Alvin E. Roth and Marilda A. Oliveira Sotomayor. *Two-Sided Matching: A Study in Game-Theoretic Modeling and Analysis*, volume 18 of *Econometric Society Monographs*. Cambridge University Press, Cambridge, 1990.

[5] Daniel R. Vincent. Introduction to matching problems: Set-up and definitions. Lecture Notes, HONR259L, University of Maryland, December 2018.