UNIVERSITÄT BASEL

Optimization of Student Time Slot Allocation using Hungarian Method and Mixed-Integer Linear Programming

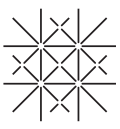
Bachelor's Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Salomé Eriksson and Claudia Grundke

Hamza Zarah
h.zarah@stud.unibas.ch
20-051-801

18.08.2024



Acknowledgments

First of all, I would like to express my gratitude to Prof. Dr. Malte Helmert for providing me the opportunity to write my Bachelor's thesis in the Artificial Intelligence Research Group and for offering me an interesting selection of thesis topics.
I am especially thankful to my supervisors, Salomé Eriksson and Claudia Grundke, for their unwavering support, patience, understanding, and guidance throughout this project.

Abstract

The assignment of students to exercise groups represents a classic assignment problem, which aims to optimize their preferences. While simple variants, where students only indicate a single preference, can be solved in polynomial time using algorithms such as weighted maximum matching, more complex variants become significantly more challenging. This thesis explores various versions of the assignment problem and applies combinatorial optimization techniques to find solutions. These include considering group and language preferences and evaluating algorithms like the Hungarian Method and an advanced Mixed Integer Linear Programming (MILP) variant.

The evaluation revealed that while the Hungarian Method approach is efficient for smaller problem sizes with limited complexity, its performance significantly degrades as the problem size and constraints, such as language and group preferences, increase. In contrast, the MILP-based SmartAlloc approach, although more resource-intensive for smaller problems, demonstrated superior flexibility and scalability in larger and more complex scenarios. These findings suggest that SmartAlloc is more suitable for real-world applications involving multiple constraints, while the Hungarian Method approach remains useful for simpler, more straightforward assignments.

Table of Contents

1 Introduction

Assigning students to exercise groups (time slots) is a typical example of an assignment problem that occurs in many academic and organizational contexts. Generally, it involves optimally assigning individuals to various groups or resources while considering their different preferences. These assignment problems are not only important in education but also in fields like logistics, human resource management and resource planning.

In its simplest form, students can specify a preference for different time slots. Such simple variants can be solved in polynomial time, for example, using weighted maximum matching algorithms. These algorithms are known for providing efficient solutions for bipartite graphs, where the goal is to find an optimal pairing between two sets[1]. The Problem becomes more complex when additional preferences and constraints are considered such as language preferences or the desire for students to be in the same group.

The aim of this thesis is to formulate various variants of the assignment problem and solve them using different combinatorial optimization techniques. A particular focus is on the investigation of the Hungarian Method also known as the Kuhn-Munkres algorithm [1], as well as advanced variants of Mixed Integer Linear Programming (MILP). The Hungarian Method is a well-known algorithm for solving the assignment problem in polynomial time.

For more complex variants, where multiple preferences and constraints must be considered, MILP is a suitable method. For instance, students may have preferences for certain timeslots, desire to be in the same group as other students or have language preferences that must be accommodated. MILP allows for the inclusion of both discrete and continuous variables in an optimization problem and is therefore particularly well-suited for problems where such multiple preferences and constraints must be handled simultaneously [2]. In this thesis, a MILP-based method called "SmartAlloc" will be implemented and studied. "SmartAlloc" focuses on student assignment by explicitly modeling specific preferences and constraints within the standard MILP framework, thereby aiming to improve the efficiency of the solution search.

[1] Brilliant.org, *Matching Algorithms (Graph Theory)*, Brilliant Math & Science Wiki, 2024. Available at: `https://brilliant.org/wiki/matching-algorithms/`. Accessed: 2024-04-30.

An important aspect of this thesis is the evaluation of the different approaches in various scenarios. For this purpose, different test scenarios will be created using a generator to assess the performance and scalability of the algorithms. The results of these tests will provide insights into how well the different approaches work in practice and what advantages and disadvantages they have in different application contexts. Although this thesis focuses on the student assignment problem, the techniques developed can be applied to other types of assignment problems as well.

2
Background

In this section, all relevant definitions and concepts are presented to provide the theoretical framework for understanding the assignment problem and the optimization techniques used in this thesis. The theoretical concepts will be directly applied in the following chapters.

2.1 Combinatorial Optimization

Combinatorial optimization is a field of optimization in mathematics studying problems which are characterized by a set of constraints and an objective function to be optimized. Examples include the traveling salesman problem, the knapsack problem and assignment problems [3]. In this thesis, we specifically address the assignment problem, where we seek to allocate students to time slots, which will be discussed in more detail later.
Combinatorial optimization problems do not involve actions or transitions as in dynamic systems; instead, we seek a configuration (or state) that minimizes cost or maximizes quality. This comprehension of combinatorial optimization is primarily derived from Professor Malte Helmert's lecture series in the "Foundations of Artificial Intelligence" course at the University of Basel[2].

Definition 1: Combinatorial Optimization Problems

A combinatorial optimization problem (COP) is defined as a problem where we seek to find an optimal object from a finite set of objects. More formally, a COP can be represented by a four tuple $\mathcal{O} = (C, S, \text{opt}, v)$, where:

- C is a finite set of solution candidates.
- $S \subseteq C$ is a finite set of feasible solutions.
- $\text{opt} \in \{\min, \max\}$ indicates whether we seek to minimize or maximize the objective function.

[2] Malte Helmert, *Foundations of Artificial Intelligence*, C1. Combinatorial Optimization: Introduction and Hill-Climbing, University of Basel, 2024.

- $v : S \to \mathbb{R}$ is the objective function that assigns a real value to each feasible solution.

In combinatorial optimization, the objective function v measures the quality of feasible solutions. The goal is to find a solution $s \in S$ that optimizes this function. The optimal solution quality v^* for a COP $\mathcal{O} = \langle C, S, \text{opt}, v \rangle$ is defined as:

$$v^* = \begin{cases} \min_{c \in S} v(c) & \text{if opt} = \min \\ \max_{c \in S} v(c) & \text{if opt} = \max \end{cases}$$

A solution $s \in S$ is considered optimal if $v(s) = v^*$.

Constraints in COPs can be categorized into **hard constraints**, which must be satisfied for a solution to be feasible and **soft constraints**, which are preferred but not necessary.
Hard constraints might include factors such as availability or capacity limits, while soft constraints are often integrated into the objective function to optimize the solution, such as minimizing penalties based on preferences.

Combinatorial optimization problems involve both a **search aspect** and an **optimization aspect**. The search aspect involves finding feasible solutions within the candidate set C. The challenge is to identify solutions that satisfy all the constraints and thus belong to S. The optimization aspect involves finding the best solution among the feasible solutions in S according to the objective function v.[3]

There are special cases where one of these aspects is trivial. In **pure search problems** all solutions are of equal quality, making the challenge solely about finding any feasible solution. In this case, the objective function v is constant and the optimization direction opt can be chosen arbitrarily. In **pure optimization problems**, all candidates are feasible solutions, making the challenge solely about finding the highest quality solutions. Here, $S = C$.

The fundamental algorithmic challenge in combinatorial optimization is to find a solution of **good**, ideally **optimal**, **quality** for a given COP $\mathcal{O}$, or to prove that no solution exists. "Good" typically means a solution close to the optimal quality v^*.

Combinatorial optimization is a core focus of operations research due to its practical importance and the inherent complexity of many such problems. Many combinatorial optimization problems, such as the traveling salesman problem and the knapsack problem, are **NP-complete** [4], making them difficult to solve efficiently. However, it is worth mentioning that the classical assignment problem may be handled efficiently using algorithms like the Hungarian algorithm [1], which run in polynomial time. Nevertheless, certain variations of the problem, such as the quadratic assignment problem, are known to be NP-complete [5].

[3] This section was entirely written by me. I used tools like ChatGPT and Deepl only for minor checks, with all content-related and stylistic revisions being carried out solely by myself.

2.2 Linear Programming and Mixed-Integer Linear Programming

Linear Programming (LP) and Mixed-Integer Linear Programming (MILP) are essential optimization methods used in diverse fields like logistics, manufacturing and telecommunications to solve practical problems. This section offers an in-depth look of Linear Programming (LP) and Mixed Integer Linear Programming (MILP).
The theories and methodologies discussed here are primarily based on the book "Theory of Linear and Integer Programming" by Alexander Schrijver [2] and the book "Optimization Over Integers" by Dimitris Bertsimas and Robert Weismantel [6].

2.2.1 Linear Programming (LP)

Linear Programming (LP) is an optimization technique used to find the optimal solution in a mathematical model that consists of linear relationships. The process involves maximising or minimising a linear objective function while adhering to a collection of linear equality and inequality constraints.

Components of Linear Programming

- **Decision Variables**: These are the variables that influence the outcome. They must satisfy the constraints of the problem and are typically non-negative.
- **Objective Function**: The function that needs to be optimized (maximized or minimized). It is mathematically expressed as:

$$\text{Maximize (or Minimize)} \quad Z = c_1x_1 + c_2x_2 + \cdots + c_nx_n,$$

  where $c_1, c_2, \ldots, c_n$ are coefficients and $x_1, x_2, \ldots, x_n$ are decision variables.
- **Constraints**: Linear inequalities that limit the values of the decision variables. These constraints can be represented as:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2, \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m, \end{aligned}$$

  where a_{ij} are coefficients and b_i are constants.
- **Non-negativity Restriction**: Ensures that decision variables cannot be negative:

$$x_i \geq 0 \quad \text{for all} \quad i.$$

To summarize, an LP problem can be formulated as:

$$\begin{aligned} \text{Maximize (or Minimize)} \quad & Z = \mathbf{c}^\top \mathbf{x}, \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \end{aligned}$$

where $\mathbf{c}$ is a vector of coefficients, $\mathbf{x}$ is a vector of decision variables, A is a matrix of coefficients and $\mathbf{b}$ is a vector of constants.

2.2.2 Mixed-Integer Linear Programming (MILP)

Mixed-Integer Linear Programming (MILP) extends LP by incorporating integer constraints on some or all decision variables. This extension is crucial for problems involving discrete entities, such as scheduling, production planning and allocation problems where fractional values are not feasible[4]. It is worth noting that MILP is a specific type of Mixed-Integer Programming (MIP), which can also include problems with non-linear constraints, such as quadratic constraints. For example, a problem with a quadratic objective function or quadratic constraints falls under the broader category of MIP but not MILP. However, in this thesis, we focus solely on linear constraints, making MILP the appropriate term.

Components of MILP

The components of MILP are similar to those of LP, but with the addition of integer constraints:

- **Decision Variables**: These include both continuous and integer variables. The integer variables are essential for representing discrete choices, such as the number of units to produce or the number of staff to schedule.
- **Objective Function**: This is a linear function to be optimized, represented as:

  $$\text{Maximize (or Minimize)} \quad Z = \mathbf{c}^\top \mathbf{x} + \mathbf{d}^\top \mathbf{y},$$

  where $\mathbf{c}$ and $\mathbf{d}$ are vectors of coefficients, $\mathbf{x}$ is a vector of continuous decision variables and $\mathbf{y}$ is a vector of integer decision variables.
- **Constraints**: These are linear equations or inequalities that the decision variables must satisfy:

  $$A\mathbf{x} + B\mathbf{y} \leq \mathbf{b},$$

  where A and B are matrices of coefficients and $\mathbf{b}$ is a vector of constants.
- **Integer Constraints**: Specific variables are constrained to take only integer values:

  $$\mathbf{y} \in \mathbb{Z}^n.$$

[4] Google Developers, *Solving MIP Problems*, 2023. Available at: `https://developers.google.com/optimization/mip/mip_example?hl=de`. Accessed: 2024-05-11.

- **Non-Negativity Constraints**: Continuous decision variables are typically required to be non-negative:

$$\mathbf{x} \geq 0.$$

To summarize, an MILP problem can be formulated as:

$$\begin{aligned} \text{Maximize (or Minimize)} \quad & Z = \mathbf{c}^\top \mathbf{x} + \mathbf{d}^\top \mathbf{y}, \\ \text{subject to} \quad & A\mathbf{x} + B\mathbf{y} \leq \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \quad \mathbf{y} \in \mathbb{Z}^n. \end{aligned}$$

Computational Complexity and Solution Techniques

Mixed-Integer Linear Programming (MILP) problems are more complex than Linear Programming (LP) problems because of the requirement for integer constraints. LP problems can be solved in polynomial time with algorithms like the Ellipsoid Method and Interior-Point Methods, as discussed by Schrijver [2]. LP problems fall within the class P, meaning they can be solved in polynomial time. However, MILP problems, classified as NP-hard, are at least as difficult as the most challenging problems in NP and no known polynomial-time algorithms exist for solving them. This difference implies that MILP problems typically require much more computational effort as they increase in size.[5]

To tackle the complexity of MILP problems, several advanced techniques are utilized. A widely used method is Branch and Bound, which involves recursively dividing the problem into smaller subproblems (branching) and applying bounds to eliminate subproblems that cannot yield a better solution than the current best (bounding). While this method can reduce the search space, it still has exponential worst-case complexity [7]. Another important technique is Cutting Planes, which iteratively adds linear constraints to the MILP to exclude non-integer feasible regions. These constraints help in converging to the optimal integer solution but require solving a series of LP relaxations, which can be computationally intensive [8]. Additionally, heuristic methods, such as Genetic Algorithms and Simulated Annealing, are often employed to find approximate solutions for large MILP problems where exact methods are impractical [9]. These approaches do not guarantee optimal solutions but can provide satisfactory solutions within reasonable timeframes.

2.3 Matching Algorithms

Matching algorithms play a vital role in the realms of combinatorial optimization and graph theory, addressing a wide range of real-world problems, from scheduling to resource allocation. This section delves into matching algorithms with a specific focus on bipartite graphs and the Hungarian algorithm.

The theories and approaches mentioned here are derived from multiple sources. Brilliant.org[6] provides a comprehensive understanding of matching algorithms, serving as a strong basis for the broader subject. The lecture slides provided by Zoltán Szigeti [10] in 2012 and the

[5] This section was entirely written by me. I used tools like ChatGPT and Deepl only for minor checks, with all content-related and stylistic revisions being carried out solely by myself.

[6] Brilliant.org, *Matching Algorithms (Graph Theory)*, Brilliant Math & Science Wiki, 2024. Available at: `https://brilliant.org/wiki/matching-algorithms/`. Accessed: 2024-04-30.

book "Matching Theory" by László Lovász and M. D. Plummer[7] [11] offer great insights into the formal definitions of matchings in bipartite graphs and the Hungarian method. Finally, H. W. Kuhn's seminal paper (1955) specifically addresses the Hungarian method and the assignment problem [1].

2.3.1 Bipartite Graphs

A bipartite graph G = (V,E) is a graph whose vertex set V can be divided. A bipartite graph $G = (V, E)$ is a graph whose vertex set V can be divided into two disjoint subsets U and W such that there are no connected vertices inside the same subset. Each edge in E connects a vertex in U to a vertex in W and is undirected. This structure is particularly useful in various matching problems where the objective is to optimally pair components from two distinct sets. Bipartite graphs are characterised by the absence of odd cycles, where an odd cycle is a cycle that contains an odd number of edges. This property can be used to determine whether a graph is bipartite: if a graph contains no odd cycles, it is bipartite [10].[8]

2.3.2 Matching in Bipartite Graphs

In graph theory, a matching M in a graph $G = (V, E)$ is defined as a subset of edges such that no two edges share a common vertex. The primary objective in many matching problems is to identify a maximum matching, which includes the greatest possible number of edges. A perfect matching is a specific type of matching where every vertex in the graph is matched, ensuring that each vertex is connected to exactly one edge in the matching[9] [10].[10]

2.3.3 The Hungarian Algorithm

The Hungarian Method, developed by Harold Kuhn in 1955 [1], is an efficient algorithm designed to solve assignment problems. The assignment problem involves assigning n tasks to n agents in such a way that the total cost is minimized. Each agent is assigned to exactly one task and each task is assigned to exactly one agent. This method, named in honor of Hungarian mathematicians Dénes Kőnig and Jenő Egerváry, provides a computationally efficient alternative to the simplex method used in linear programming.[11]

The Assignment Problem

The assignment problem [11] can be represented by an $n \times n$ cost matrix, where each entry c_{ij} denotes the cost of assigning agent i to task j. The goal is to find a one-to-one assignment

[7] For a better understanding, I used the handout *Bipartite Matching & the Hungarian Method* by Mordecai J. Golin, in addition to the book *Network Flows: Theory, Algorithms, and Applications*. Assignment Notes, 2006. Available at: `https://www.columbia.edu/~cs2035/courses/ieor6614.S16/GolinAssignmentNotes.pdf`. Accessed: 2024-05-10.

[8] This section was entirely written by me. I used tools like ChatGPT and Deepl only for minor checks, with all content-related and stylistic revisions being carried out solely by myself.

[9] Brilliant.org, *Matching Algorithms (Graph Theory)*, Brilliant Math & Science Wiki, 2024. Available at: `https://brilliant.org/wiki/matching-algorithms/`

[10] This section was entirely written by me. I used tools like ChatGPT and Deepl only for minor checks, with all content-related and stylistic revisions being carried out solely by myself.

[11] This section was entirely written by me. I used tools like ChatGPT and Deepl only for minor checks, with all content-related and stylistic revisions being carried out solely by myself.

that minimizes the total cost. Formally, the objective is to minimize the sum:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

subject to the constraints:

1. $\sum_{i=1}^{n} x_{ij} = 1$ for all $j \in \{1, \ldots, n\}$ (each task is assigned to exactly one agent),
2. $\sum_{j=1}^{n} x_{ij} = 1$ for all $i \in \{1, \ldots, n\}$ (each agent is assigned to exactly one task),
3. $x_{ij} \in \{0, 1\}$.

Steps of the Hungarian

The Hungarian Method involves the following steps:

1. **Subtract Row Minimums:** For each row of the cost matrix, subtract the smallest element from every element in that row.
2. **Subtract Column Minimums:** For each column of the resulting matrix, subtract the smallest element from every element in that column.
3. **Cover All Zeros:** Cover all zeros in the matrix using a minimum number of horizontal and vertical lines. If the minimum number of lines is n, an optimal assignment is possible among the zeros. If fewer than n lines are required, proceed to the next step.
4. **Adjust the Matrix:**
   (a) Find the smallest element not covered by any line. Subtract this element from all uncovered elements.
   (b) Add this element to all elements that are covered twice (i.e., by both a horizontal and a vertical line).
   (c) Return to step 3 until exactly n lines are required to cover all zeros.
5. **Make Assignments:** Select zeros in such a way that no two selected zeros are in the same row or column. These selected zeros represent the optimal assignment. Mark these positions in the matrix, ensuring each row and column has exactly one marked position.

2.3.4 Complexity and Performance

The Hungarian algorithm operates with a time complexity of $O(n^3)$, making it efficient for solving large-scale assignment problems. It is a robust method that guarantees finding the optimal solution by iteratively refining the cost matrix and adjusting the feasible labeling, which refers to the assignment of dual variables (or labels) to the nodes in the bipartite graph to maintain the feasibility of the matching during the algorithm's execution [11][1].[12]

[12] This section was entirely written by me. I used tools like ChatGPT and Deepl only for minor checks, with all content-related and stylistic revisions being carried out solely by myself.

2.4 Problem Definition

This thesis focuses on the task of allocating students to available timeslots, taking into account their individual preferences. The main components of the problem are as follows: We have a set of **students** S, each distinguished by their unique identifier and distinct preferences towards available timeslots, languages and groupings with other students. In addition, there is a set of available **timeslots** T, each identified by a unique identifier and a specific time frame, such as "Thursday 12:15-14:00". Each timeslot $t \in T$ has a **capacity** $c(t)$ describing the maximum number of students it can accommodate. Furthermore, there is a set of **languages** L, representing the different languages in which the timeslots can be conducted.

Every student has individual preferences in three specific areas: preferences for specific timeslots, preferences for specific languages and preferences for specific groups. **Timeslot preferences** require each student to assign a score to each timeslot on a scale of 0 (indicating "no"), 1 (indicating "maybe"), or 2 (indicating a strong preference). Formally, this is described by a function $p_t : S \times T \to \{0, 1, 2\}$. Likewise, each student rates their **language preferences** using the identical scale, described by a function $p_l : S \times L \to \{0, 1, 2\}$. **Group preferences** involve certain students expressing a preference to be grouped with particular fellow students, which must be taken into account when making assignments. We assume student preferences to be mutual, that is, if student s wants to group with student s', then student s' also wants to group with student s. This is described by a function $p_g : S \times S \to \{0, 1\}$, where 0 indicates no preference to be grouped with the particular student and 1 indicates a yes preference to be grouped together.

This scheduling process involves making two key assignments. Firstly, it is necessary to allocate a distinct language to each timeslot that is suitable for the students assigned to it, represented by the function $a_l : T \to L$. Secondly, it is necessary to allocate each student to precisely one timeslot in a manner that maximizes their overall satisfaction according to their preferences, represented by the function $a_t : S \to T$.

In order to achieve this objective, several constraints must be included. Every student must be assigned to one and only one timeslot, represented by the function a_t. Each timeslot must not exceed its capacity $c(t)$. The language assigned to each timeslot, represented by the function a_l, must be appropriate for the students assigned to that timeslot. Lastly, it is important to respect group preferences, which means that students who choose to be placed together have to be assigned to the same timeslot.

To address this problem, two key assignments need to be defined. The first is **to assign timeslots to languages**. This mapping a_l will allocate a specific language to each timeslot based on the collective preferences of the students. The second is **to assign students to timeslots**. This mapping a_t will allocate each student to a timeslot, taking into account their preferences for timeslots, languages and groupings.

Summarised and formally defined, the assignment problem we study consists of:

- a set of students S
- a set of timeslots T
- a set of languages L
- a function $p_t : S \times T \to \{0, 1, 2\}$ describing timeslot preferences of students

- a function $p_l : S \times L \to \{0, 1, 2\}$ describing language preferences of students
- a function $p_g : S \times S \to \{0, 1\}$ describing group preferences of students
- a function $c : T \to \mathbb{N}_0$ describing the capacity of each timeslot

The goal is to find two functions $a_t : S \to T$ and $a_l : T \to L$ such that $p_t(s, a_t(s)) \neq 0$ and $p_l(s, a_l(a_t(s))) \neq 0$ for all $s \in S$. Additionally, the function a_t should respect the group preferences p_g and the capacity constraints c and the overall satisfaction of the students should be optimized by maximizing the following **objective function**:

$$\sum_{s \in S} (p_t(s, a_t(s)) + p_l(s, a_l(a_t(s))))$$

subject to the **constraints**:

$$\begin{aligned}
&p_t(s, a_t(s)) \neq 0 \quad \forall s \in S \\
&p_l(s, a_l(a_t(s))) \neq 0 \quad \forall s \in S \\
&\sum_{s \in S} \mathbf{1}(a_t(s) = t) \leq c(t) \quad \forall t \in T \\
&a_t(s) = a_t(s') \quad \forall (s, s') \in S \times S \text{ where } p_g(s, s') = 1
\end{aligned}$$

Here, the term $\mathbf{1}(a_t(s) = t)$ is an **indicator function** that takes the value 1 if student s is assigned to timeslot t and 0 otherwise.

3

Mixed-Integer Linear Programming (MILP)

In this chapter, we explore the application of Mixed-Integer Linear Programming (MILP) to solve the problem of assigning students to time slots, with the aim of optimizing the allocation based on various constraints and preferences.

3.1 Simplified Approach

My research began by considering a simplified version of the student time slot assignment problem. In this initial approach, each student indicates their availability for each time slot with a binary response—either "yes" (available) or "no" (not available). The objective was to find an assignment that maximizes the number of students assigned to their available time slots, without considering additional factors such as language preferences or group preferences.

This simplified approach allowed for a more straightforward application of optimization techniques, serving as a foundation for exploring more complex variants of the problem, which include additional preferences and constraints such as language assignments (p_l) and group preferences (p_g). The insights gained from this initial model informed the development of more sophisticated methods discussed later in this chapter.

To model the problem, I used the variables $X = \{x_{ij}\}$ for $i \in \{1, \ldots, n\}$ (number of students) and $j \in \{1, \ldots, m\}$ (number of time slots). Here, $x_{ij} = 1$ means that student i is assigned to time slot j and $x_{ij} = 0$ means that student i is not assigned to time slot j. Each time slot j has a fixed capacity c_j and each student i has an availability a_{ij} for each time slot j, where a_{ij} is either 1 (available) or 0 (not available).

The objective function is to maximize the sum of all x_{ij}, where i and j are the indices of the students and time slots, to optimize the assignment and maximize the number of assigned students:

$$\text{Maximize} \quad Z = \sum_{i=1}^{n} \sum_{j=1}^{m} x_{ij}$$

Multiple constraints need to be met. The first constraint is the capacity constraint, which states that the total number of assignments for each time slot j should not exceed its capacity

c_j:

$$\sum_{i=1}^{n} x_{ij} \leq c_j \quad \forall j \in \{1, \ldots, m\}$$

The second constraint is the availability constraint, which states that a student i can only be assigned to time slots that they are available for, denoted by a_{ij}:

$$x_{ij} \leq a_{ij} \quad \forall i \in \{1, \ldots, n\}, \quad \forall j \in \{1, \ldots, m\}$$

Thirdly, there is a uniqueness restriction that limits each student to being assigned to a maximum of one time slot:

$$\sum_{j=1}^{m} x_{ij} \leq 1 \quad \forall i \in \{1, \ldots, n\}$$

Given the nature of these constraints, it became clear that a simple linear programming (LP) approach would not suffice, as it does not adequately handle the integer decision variables necessary for this problem. My initial formal formulation aligned well with the MILP framework, which is better suited for incorporating these variables and accurately modeling the problem's constraints, as discussed in the background section of this thesis.

3.2 Representation

This section presents the formalisation of the Mixed-Integer Linear Programming (MILP) model that is used to solve the problem of assigning students to specific time slots, taking into account their preferences and other constraints.

3.2.1 Problem Definition

The objective of the assignment problem is to allocate each student to a time slot that maximizes their overall satisfaction, taking into account constraints such as time slot capacities and language preferences. The key elements of the problem are as follows:

- **Students**: $S = \{s_1, s_2, \ldots, s_n\}$, where s_i represents a student.
- **Time Slots**: $T = \{t_1, t_2, \ldots, t_m\}$, where t_j represents a time slot.
- **Group Preferences**: $p_g : S \times S \to \{0, 1\}$ describes group preferences of students, where $p_g(s_i, s_p) = 1$ means student s_i prefers to be in the same time slot as student s_p and $p_g(s_i, s_p) = 0$ means no such preference exists.
- **Availability**: $a_{i,j} \in \{0, 1, 2\}$ for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$, representing the availability of student s_i for time slot t_j (0: no, 1: maybe, 2: yes).
- **Language Preference**: $l_{i,j} \in \{0, 1, 2\}$ for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$, where $l_{i,j}$ indicates the preference of student s_i for the language assigned to time slot t_j. The language assignment to each time slot will be considered in the solution process by evaluating all possible combinations of language assignments.
- **Capacity**: c_j representing the maximum number of students that can be assigned to time slot t_j.

3.2.2 Decision Variables

- $x_{i,j} \in \{0, 1\}$: A binary variable that is 1 if student s_i is assigned to time slot t_j and 0 otherwise.

3.2.3 Objective Function

The objective is to minimize the total penalty for not adhering to student preferences. This is represented by:

$$\text{Minimize} \quad \sum_{i=1}^{n} \sum_{j=1}^{m} \text{Penalty}_{i,j} \cdot x_{i,j}$$

where $\text{Penalty}_{i,j}$ is defined as:

$$\text{Penalty}_{i,j} = \mathbf{1}\{a_{i,j} = 1\} + \mathbf{1}\{l_{i,j} = 1\}$$

We assume that the language assignment $l_{i,j}$ to time slot t_j is known for the purpose of this model. The evaluation considers all possible language assignments across time slots.

3.2.4 Constraints

1. **Capacity Constraint**:

$$\sum_{i=1}^{n} x_{i,j} \leq c_j \quad \forall j \in \{1, \ldots, m\}$$

   This ensures that the number of students assigned to any time slot does not exceed its capacity.

2. **Availability Constraint**:

$$x_{i,j} \leq a_{i,j} \quad \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}$$

   This ensures that students are only assigned to time slots for which they are available.

3. **Language Preference Constraint**:

$$x_{i,j} \leq l_{i,j} \quad \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}$$

   This ensures that students are only assigned to time slots with their preferred language.

4. **Group Constraint**:

$$x_{i,j} = x_{p,j} \quad \forall i, p \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\} \text{ where } p_g(s_i, s_p) = 1$$

   This ensures that students who wish to be together are assigned to the same time slot.

   Note that equality constraints in MILP can be expressed using inequalities. Specifically, the equality $a = b$ can be represented as two inequality constraints $a \leq b$ and $b \leq a$. Thus, the above constraint $x_{i,j} = x_{p,j}$ can be expressed as:

$$x_{i,j} \leq x_{p,j} \quad \text{and} \quad x_{p,j} \leq x_{i,j} \quad \forall i, p \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\} \text{ where } p_g(s_i, s_p) = 1$$

This transformation allows us to implement the constraint within the MILP framework, which typically supports inequality constraints.

5. **Unique Assignment Constraint**:

$$\sum_{j=1}^{m} x_{i,j} = 1 \quad \forall i \in \{1, \ldots, n\}$$

This ensures that each student is assigned to exactly one time slot.

3.2.5 Explanation

The MILP model integrates binary decision variables, a linear objective function and multiple linear constraints to accurately reflect the allocation problem. The decision variable $x_{i,j}$ indicates whether student s_i is assigned to time slot t_j and the collection of these variables allows us to extract the final assignment solution. Specifically, for each student s_i, the time slot t_j to which they are assigned is determined by finding the index j for which $x_{i,j} = 1$.
The objective function is designed to minimize the dissatisfaction of students based on their availability and language preferences, which are converted into penalty values. This means that the optimization process seeks to find assignments that minimize the total penalties, ensuring that the assignment is as close as possible to the students' preferences.
It is important to note that this approach differs from the initial idea outlined in the "Simplified Approach" section, where the problem was framed as a maximization problem aimed at maximizing the number of assignments. In that earlier model, the goal was to maximize the total number of students assigned to any time slot. However, with the MILP approach, we adjusted the objective function to minimize penalties associated with non-preferred assignments.
One critical adjustment was the use of an equality constraint $\sum_{j=1}^{m} x_{i,j} = 1$ instead of ≤ 1 for the unique assignment. If we had kept the ≤ 1 constraint, the solver could potentially assign no students at all (assigning all $x_{i,j} = 0$), which would result in no penalties. This scenario would not meet the requirement of assigning every student to exactly one time slot. Therefore, the constraint was changed to ensure that each student is assigned to exactly one time slot, which prevents the trivial solution where no assignments are made.
Each constraint addresses a specific aspect of the problem:

- **The capacity constraint** ensures that no time slot exceeds its maximum capacity.
- **The availability constraint** guarantees that students are only assigned to time slots they are available for.
- **The language preference constraint** ensures that students are assigned to time slots with their preferred language.
- **The group constraint** enforces that students who wish to be in the same group are assigned to the same time slot.
- **The unique assignment constraint** ensures that each student is assigned to exactly one time slot.

In this model, the capacity, availability, language preference, group and unique assignment constraints are considered **hard constraints**, meaning they must be strictly satisfied. The

penalties associated with non-preferred assignments are considered **soft constraints**, as the objective function aims to minimize these penalties. Specifically, the goal is to minimize the number of assignments that fall under "maybe" preferences (i.e., a preference score of 1). These "maybe-preferences" are not strictly avoided, but their occurrence is minimized to improve overall student satisfaction.

3.3 Algorithm

This section discusses the implementation of the Mixed-Integer Linear Programming (MILP) approach called as the "SmartAlloc" variation in my project. This method utilizes Google OR-Tools[13], particularly its capability to handle Mixed-Integer Linear Programming (MILP) problems. The primary objective was to accurately represent the problem to enable the solver to effectively determine the optimal solution.
The "SmartAlloc" system starts by loading and preparing the problem description, which is provided as a JSON file. Preferences are represented by integer values, specifically 0, 1, or 2. Of particular importance, if a student expresses a preference of 0 for all time slots, these preferences are modified to 2 to guarantee that they can still be assigned. This adjustment is crucial because the hard constraint "Availability Constraint":

$$x_{i,j} \leq a_{i,j} \quad \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}$$

would otherwise prevent such students from being assigned to any time slot. This ensures that all students can be considered in the assignment process.
To determine the optimal assignment, we considered all possible combinations of languages since the time slots do not initially have any language specifications. In the representation above, we assumed that the language assigned to each time slot is known, which is why we explore all language combinations. This is achieved by using the `itertools.product` operation, which generates all feasible combinations of languages (English and German) over the given time slots.
However, it is important to note that this approach does not scale well when the number of language options or time slots increases. The combinatorial explosion in the number of language assignments means that the number of MILP problems to be solved grows exponentially. This can lead to significant increases in computation time. For scenarios with more language options or a large number of time slots, this approach may become impractical, as it could exceed the available computational resources or time limits, leading to situations where no solution is found within the allowed time.
The function `is_combination_feasible` checks the feasibility of each language combination for all students. It ensures that, for each student, there is at least one time slot with a language they prefer (i.e., they have given a preference score of either 1 or 2). If a combination does not meet this criterion, it is rejected. The function `adjust_language_preferences` then adjusts the language constraints and the penalty function according to the current language combination. This means that, based on the current language assignment, the constraints regarding language preferences and the penalty function are updated accordingly.

[13] Google Developers, *Solving MIP Problems*, 2023. Available at: `https://developers.google.com/optimization/mip/mip_example?hl=de`. Accessed: 2024-05-11.

For each feasible language combination, an optimization problem is created using the SCIP solver. Variables, constraints and objectives are defined to find the best assignment of students to time slots, respecting preferences and minimizing penalties. The solution for each feasible language combination is evaluated and the combination with the lowest total cost (penalties) is selected as the best language combination.
In addition, I have developed a modified version of "SmartAlloc" that does not include group preferences. The justification for this variation will be explained in the following chapter on the Hungarian Method.

4

Hungarian Method

The Hungarian Method is a well-established approach for solving assignment problems efficiently. In this chapter, we explore its application to the problem of assigning students to time slots, highlighting its advantages and limitations within the specific context of this problem.

4.1 Motivation

The decision to use the Hungarian Method for assigning students to time slots is based on its proven efficiency in solving classical assignment problems in polynomial time $O(n^3)$. In this thesis, a bipartite graph is employed, representing a set of students and a set of available time slot places. Each time slot is divided into as many places as the slot's capacity allows, ensuring that all available places are considered in the assignment process.[14]
A key advantage of the Hungarian Method is its ability to efficiently handle perfect matchings in bipartite graphs. In our case, where each student must be assigned exactly one place in one of the available time slots, this method ensures that the assignment is optimal while respecting the capacities and preferences. This efficiency makes the Hungarian Method particularly attractive for solving the basic version of the student time slot allocation problem.
However, it is important to note that while the Hungarian Method is efficient, it has limitations in terms of flexibility. Specifically, the method cannot easily accommodate certain types of preferences, such as group preferences, which were addressed in the MILP approach. This limitation underscores why the Hungarian Method, despite its efficiency, may not be suitable for addressing all the complexities of the student time slot allocation problem.
This chapter will explore the application of the Hungarian Method in detail, focusing on its implementation and the specific challenges encountered when applying it to the student time slot allocation problem.

[14] This section was entirely written by me. I used tools like ChatGPT and Deepl only for minor checks, with all content-related and stylistic revisions being carried out solely by myself.

4.2 Representation

In this section, we formally define the bipartite graph representation used in the Hungarian Method to solve the student time slot allocation problem. The goal is to find a perfect matching that optimally assigns students to subslots within time slots while considering their preferences and constraints.

4.2.1 Bipartite Graph Definition

We model the problem as a bipartite graph $G = (V, E)$, where the vertex set V consists of two disjoint subsets S and T':

- **Students**: $S = \{s_1, s_2, \ldots, s_n\}$ represents the collection of students.
- **Time Slots**: $T = \{t_1, t_2, \ldots, t_m\}$ represents the collection of time slots.
- **Expanded Time Slots**: T' is an extended set formed by expanding each time slot t_j into multiple sub-slots $t_j^{(k)}$ to ensure that the total number of nodes in T' matches the number of students. Specifically,

  $$T' = \bigcup_{j=1}^{m} \{t_j^{(1)}, t_j^{(2)}, \ldots, t_j^{(c_j)}\},$$

  where c_j represents the number of sub-slots for each time slot t_j and k is an index ranging from 1 to c_j. The value of c_j is determined by:

  $$c_j = \left\lfloor \frac{|S|}{|T|} \right\rfloor + \delta_j,$$

  and δ_j handles any remaining students, distributing them across the initial time slots. This ensures that each student is assigned exactly one sub-slot.
- **Connecting Edges**: E denotes the set of edges connecting each student s_i in S to every sub-slot $t_j^{(k)}$ in T'. Therefore, $E = S \times T'$, forming a complete bipartite graph. The costs associated with these edges, which represent the preferences and constraints, will be discussed in the following subsection on the cost matrix.

4.2.2 Perfect Matching

The goal is to find a perfect matching $M \subseteq E$, where each student s_i is linked to precisely one sub-slot $t_j^{(k)}$. A perfect matching in this context means that every student is assigned to exactly one sub-slot and no sub-slot is assigned to more than one student. However, among all possible perfect matchings, we aim to find the one that minimizes the total cost associated with the assignment. The costs reflect the preferences and constraints as defined earlier. The goal is to optimize the matching to achieve the lowest possible overall cost. The specifics of how these costs are constructed and minimized will be detailed in the following subsection on cost matrix construction.

4.2.3 Cost Matrix Construction

The cost matrix C plays a crucial role in determining the optimal matching by reflecting student preferences for both time slots and languages. The construction of this matrix involves:

1. **Time Slot Preferences**: Each student s_i has a preference $\text{pref}_{i,j}$ for each original time slot t_j, expressed as an integer value between 0 and 2. The initial cost matrix C is filled based on these preferences:

$$C_{i,j^{(k)}} = \begin{cases} 100 \times n, & \text{if } \text{pref}_{i,j} = 0, \\ 1, & \text{if } \text{pref}_{i,j} = 1, \\ 0, & \text{if } \text{pref}_{i,j} = 2, \end{cases}$$

   for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$ and for all $k \in \{1, \ldots, c_j\}$.

2. **Language Preferences**: Since time slots t_j initially lack language assignments, all possible language combinations are considered. Let $\mathcal{L}$ denote the set of all possible language assignments $\mathbf{L} = (L_1, L_2, \ldots, L_m)$, where L_j is the language assigned to time slot t_j. For each $\mathbf{L} \in \mathcal{L}$, the cost matrix $C^{\mathbf{L}}$ is updated according to the student's language preference l_{i,L_j}:

$$C^{\mathbf{L}}_{i,j^{(k)}} = C_{i,j^{(k)}} + \begin{cases} 100 \times n, & \text{if } l_{i,L_j} = 0, \\ 1, & \text{if } l_{i,L_j} = 1, \\ 0, & \text{if } l_{i,L_j} = 2, \end{cases}$$

   for all $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, m\}$ and for all $k \in \{1, \ldots, c_j\}$.

   The optimal language assignment $\mathbf{L}^*$ is selected by evaluating all language combinations and choosing the one that minimizes the overall cost:

$$\mathbf{L}^* = \arg\min_{\mathbf{L} \in \mathcal{L}} \sum_{(s_i, t_j^{(k)}) \in M} C^{\mathbf{L}}_{i,j^{(k)}},$$

   where M is the perfect matching in the bipartite graph.

4.2.4 Constraints Representation

The Hungarian Method handles constraints implicitly through the cost matrix C and the requirement for a perfect matching. The key constraints are:

- **Availability Constraint**: Student availability for time slots is built into the cost matrix. If a student s_i cannot attend a time slot t_j (i.e., $a_{i,j} = 0$), this is represented by assigning a high cost $C_{i,j^{(k)}} = 100 \times n$, effectively preventing the student from being assigned to that time slot.

  The rationale behind this cost is as follows: The worst-case scenario for a valid assignment occurs when all students have a "maybe" preference (i.e., $\text{pref}_{i,j} = 1$ and $l_{i,L_j} = 1$) for their assigned time slots, leading to a total cost of $2 \times n$. By setting the cost of an unavailable time slot to $100 \times n$, we ensure that any valid assignment,

even with maximum penalties, will always have a lower total cost than any assignment involving an unavailable time slot. Formally, if a valid assignment exists, it will have a cost $\leq 2 \times n$, while any assignment with an unavailable slot will incur a cost of at least $100 \times n$, ensuring that the algorithm prioritizes valid assignments.

- **Language Preference Constraint**: Language preferences are reflected in the cost matrix adjustments for each possible language assignment. For each possible language combination $\mathbf{L}$, the cost matrix $C^{\mathbf{L}}$ is updated and the algorithm selects the language combination that minimizes the total cost, ensuring that language preferences are respected as much as possible.

- **Capacity Constraint**: The capacity constraint is enforced by the construction of sub-slots T', ensuring that the total number of assignments does not exceed the available slots. This is managed mathematically by the number of sub-slots c_j assigned to each time slot t_j. The number of sub-slots corresponds to the capacity of each time slot, preventing more students from being assigned than the slot can accommodate.

- **Unique Assignment Constraint**: The unique assignment of students to sub-slots is ensured by the structure of the bipartite graph and the cost matrix. With $|S| = |T'|$, each student is guaranteed to be assigned to exactly one sub-slot, satisfying the perfect matching requirement.

- **Group Constraint**: Group constraints, which would assign specific students to the same time slot, are not directly implementable in the Hungarian Method without modifying the algorithm. Such modifications would disrupt the perfect matching structure and could prevent the algorithm from running efficiently. Therefore, group constraints are excluded from this model.

Using this bipartite graph and the refined cost matrix, the Hungarian algorithm identifies the optimal perfect matching that minimizes total costs while adhering to the outlined constraints, effectively addressing the student time slot allocation problem.

4.3 Algorithm

The Hungarian Method's implementation for student time slot allocation is notably simpler compared to the "SmartAlloc" variant. The main task involved accurately representing the problem within a cost matrix that the solver could effectively process. This required setting costs in a way that they appropriately reflected the variations in the problem of assigning students to time slots, with costs being tied to the number of students. For instance, a cost of $100 \times$ num_students was applied whenever a student's preference was 0, ensuring that the cost was sufficiently high across different scenarios. This approach guarantees that undesirable assignments are heavily penalized, making them impossible as long as a valid assignment exists. This is supported by the earlier proof, which shows that valid assignments are always preferred unless no valid assignment is available.
In a manner similar to the "SmartAlloc" variant, when a student assigns a preference of 0 to all available time slots, these preferences are adjusted to 2. However, unlike in "SmartAlloc", this adjustment in the Hungarian Method is aimed at reducing the overall cost, as the method inherently ensures that all students are assigned through perfect matching.

Implementing the language preference constraint followed the same strategy as in the "SmartAlloc" variant, where all possible language combinations were considered. For each language combination, a corresponding cost matrix was generated and the Hungarian Method was then used to find the perfect matching with the minimum cost. After iterating through all possible language combinations, the one with the lowest overall cost was selected as the optimal solution.

The Hungarian Method was implemented using the `linear_sum_assignment` function from the Scipy library[15]. While this function doesn't directly mirror the traditional Hungarian Method, it serves as an optimized solver for the linear sum assignment problem, designed for both speed and efficiency.

The traditional Hungarian Method, or Kuhn-Munkres algorithm, methodically identifies the optimal assignment by iteratively enhancing the solution via augmenting paths and adjusting dual variables. This algorithm guarantees a minimum cost perfect matching in a bipartite graph within polynomial time.

On the other hand, the `scipy.optimize.linear_sum_assignment` function leverages an improved version of the Hungarian Method, known as the Jonker-Volgenant algorithm. This algorithm enhances the classic method by reducing the number of operations required to reach an optimal solution, thereby increasing its speed. The key difference lies in how the algorithm approaches the assignment problem: instead of solely relying on the original iterative method, it integrates more sophisticated data structures and heuristics to streamline the process.

The `linear_sum_assignment` function benefits significantly from its implementation in C, allowing it to perform much faster than other Python-based solvers, such as `munkres.Munkres`[16]. The latter is a straightforward Python implementation of the Hungarian Method, which, while simpler, is probably slower and less efficient due to Python's inherent overhead and the absence of optimizations found in lower-level languages like C.

In most applications, the SciPy implementation is preferred because it combines the reliability of the Hungarian Method with the enhanced efficiency provided by the Jonker-Volgenant improvements[12], making it well-suited for quickly and accurately solving large-scale linear sum assignment problems.

It is also important to mention that the group preference constraint could not be applied within the Hungarian Method variant. Including this constraint would violate the fundamental definition of perfect matching by introducing edges within the student set S. Moreover, any modifications to the algorithm to accommodate this constraint would risk losing the polynomial-time guarantee of the method. As a result, a secondary version of the "SmartAlloc" variant was created without the group preference constraint, facilitating a direct comparison between the Hungarian Method and the "SmartAlloc" variant.

[15] SciPy, *scipy.optimize.linear_sum_assignment*, 2024. Available at: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linear_sum_assignment.html`. Accessed: 2024-07-09.

[16] Software Clapper, *Munkres (Hungarian) Algorithm*, Available at: `https://software.clapper.org/munkres/`. Accessed: 2024-07-09.

5 Evaluation

For the evaluation of the methods implemented in this thesis for optimizing student time slot allocation, three approaches are compared: the Hungarian Method, which does not consider group preferences and two variations of Mixed-Integer Linear Programming (MILP), namely "SmartAlloc" with and without group preferences. To assess the performance of these algorithms, their efficiency and effectiveness are analyzed in terms of solution quality and computation time.

5.1 Data Sources and Experimental Methodology

To evaluate the methods implemented in this thesis for optimizing student time slot allocation, a benchmark of 510 problem instances was generated using a script provided by one of my supervisors. The problem instances are characterized by two main parameters: the number of students (n) and the preference type (p). The number of students varies from smaller sets, such as 50 students, to larger sets, up to 2000 students. This range ensures that the scalability of the algorithms can be evaluated across different problem sizes.
The preference type (p) introduces different levels of complexity into the problem instances:

- **p0:** All time slots have equal preference levels. Approximately 30% of the students are unable to attend a given time slot, 40% are indifferent (denoted as *maybe*) and 30% strongly prefer that slot. This configuration represents a balanced scenario with no inherent preference for any particular time slot.

- **p1:** One time slot is favored, while another is less preferred. In this scenario, only 10% of the students are unable to attend the preferred slot, 30% are indifferent and 60% have a strong preference for it. Conversely, for the less preferred slot, 60% of the students are unable to attend, 30% are indifferent and 10% have a strong preference for it. The remaining time slots have preferences distributed as in p0, where 30% of the students are unable to attend, 40% are indifferent and 30% strongly prefer the slot. This setup mirrors real-world situations where certain time slots are more desirable due to factors like convenience or popularity.

- **p2:** An extreme case where one time slot is highly preferred and another is strongly disliked. Here, 5% of the students are unable to attend the highly preferred slot, 15% are indifferent and 80% have a strong preference for it. Conversely, for the strongly disliked slot, 80% of the students are unable to attend, 15% are indifferent and 5% have a strong preference for it. The remaining time slots follow the distribution in p0. This creates a challenging scenario for the algorithms to find feasible solutions.

For each combination of n and p, 10 distinct problem instances were generated, resulting in a total of 510 problems. The values of n were chosen to cover a wide range of problem sizes, specifically: 50, 60, 70, 80, 90, 100, 125, 150, 175, 200, 300, 400, 500, 750, 1000, 1500 and 2000 students. Randomization was used in the generation process to ensure variability, so even within the same configuration (e.g., 200 students with $p2$ preferences), each problem instance remains unique.
All experiments were conducted in a controlled grid computing environment using the sciCORE cluster[17] at the University of Basel to maintain consistency in the evaluation process. Each algorithm was executed under identical conditions, with a fixed time limit of 30 minutes per problem instance and a memory limit of 4 GB. These constraints were chosen to simulate realistic operational environments and evaluate the performance of the algorithms under typical resource limitations. Additionally, the Lab [13] software was employed to facilitate the execution and management of the experiments, ensuring that the processes were automated and reproducible.
The evaluation began with the Hungarian Method approach, which was tested for its ability to solve the assignment problem without considering group preferences. Following this, the "SmartAlloc" approach was evaluated in two variants: one without group preferences and one with group preferences. The inclusion of group preferences in the second variant of "SmartAlloc" added complexity to the problem, allowing for an assessment of how well the algorithm could manage additional constraints while optimizing the overall allocation.
Key metrics such as the quality of the solutions, measured by the number of students successfully assigned to their preferred time slots and the computation times were recorded for each algorithm. Additionally, any instances where the algorithms failed to find a solution within the given constraints were documented, providing insights into the limits of each method.
The collected data was then analyzed to identify trends and performance differences between the algorithms. This analysis aimed to determine not only which algorithm performed best under specific conditions but also how each method scaled with increasing problem complexity and varying preference distributions. The results of this analysis are discussed in the following sections, where the efficiency and effectiveness of the Hungarian Method and the two "SmartAlloc" variants in optimizing student time slot allocations are compared.

[17] `https://scicore.unibas.ch/`

5.2 Time Performance for Finding Assignments

The time performance of the Hungarian Method was compared with two variations of the "SmartAlloc" approach (with and without group preference) across all preference types, as well as separately for the preference types p0, p1 and p2. The scatterplots provided illustrate the runtime performance of these algorithms.
In the plots, the x-axis represents the runtime of the Hungarian Method in seconds, while the y-axis represents the runtime of "SmartAlloc" (or "SmartAlloc" without group preference) also in seconds. Both axes are logarithmically scaled, making differences in performance between the algorithms more apparent. Points lying on the diagonal indicate equal runtimes between the Hungarian Method and the corresponding "SmartAlloc" variant. Points below the diagonal suggest that the "SmartAlloc" variant is faster, as the runtime of the "SmartAlloc" variant is shorter than that of the Hungarian Method. Conversely, points above the diagonal indicate that the Hungarian Method outperforms the "SmartAlloc" variant. Dashed lines represent differences in runtime by a factor of 10. For instance, a point on the line $y = 10 * x$ indicates that "SmartAlloc" took ten times longer than the Hungarian Method.
Figures 5.1, 5.3, 5.5 and 5.7 include all points from solvable problems, points from unsolvable problems that were detected as such and points for failures where the algorithms hit resource limits (e.g., time or memory). In contrast, Figures 5.2, 5.4, 5.6 and 5.8 focus only on solvable problems, ignoring failed or unsolvable instances.

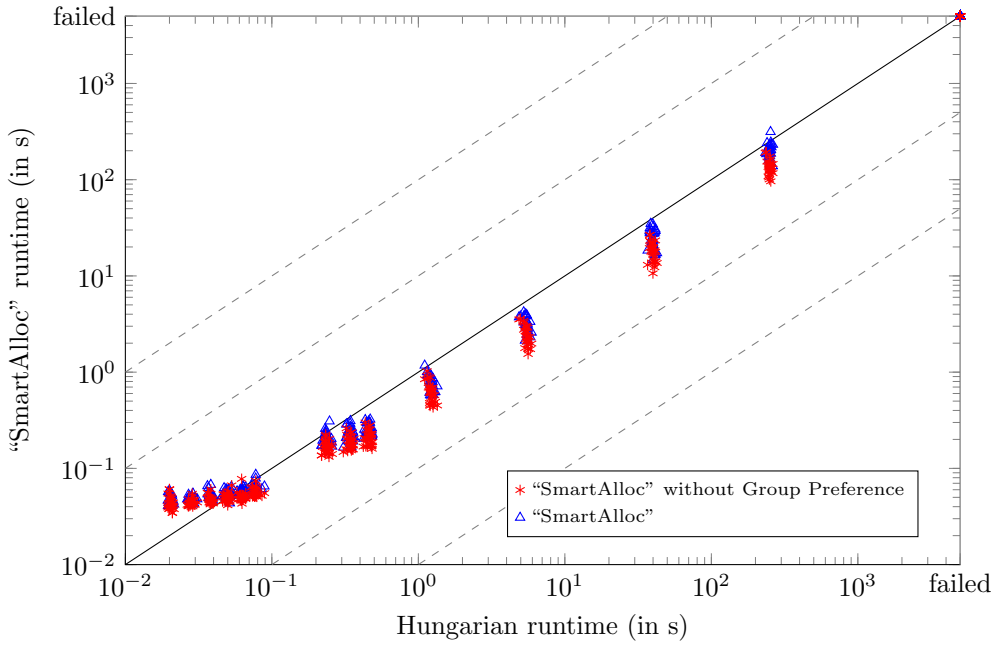


Figure 5.1: Comparison of runtime between the Hungarian Method and "SmartAlloc" variations for all preference types, including failures and unsolvable problems.

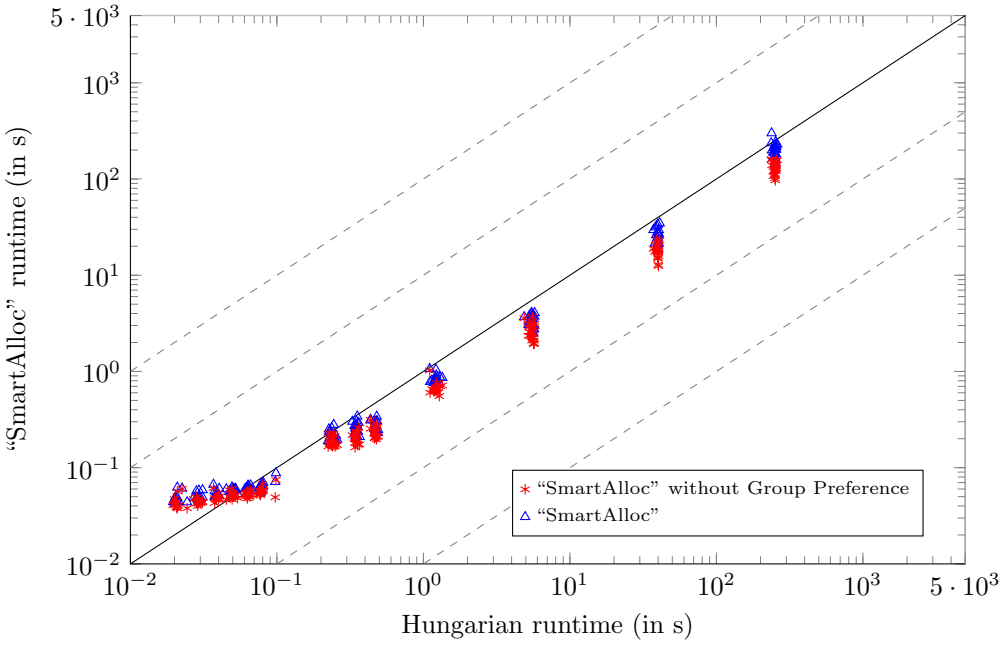


Figure 5.2: Comparison of runtime between the Hungarian Method and "SmartAlloc" variations for only solvable problems across all preference types.

As shown in Figures 5.1 and 5.2, for all preference types combined, most points are located below the diagonal, indicating that the "SmartAlloc" variants generally perform faster than the Hungarian Method. This trend is particularly noticeable for larger problem sizes, where the complexity increases due to the higher number of timeslots and therefore a higher number of language combinations. While both methods experience a combinatorial explosion regarding the language combinations, the difference in performance may be attributed to how each approach handles these complexities. The "SmartAlloc" variants, which rely on MILP, are better equipped to manage the increased problem size efficiently, possibly due to more sophisticated optimization strategies within the MILP solver. On the other hand, the Hungarian Method, although efficient in solving the assignment problem, may struggle with overhead and less effective handling of larger, more complex scenarios. However, for smaller problem sizes, where these factors are less impactful, the Hungarian Method exhibits comparable or even better performance.

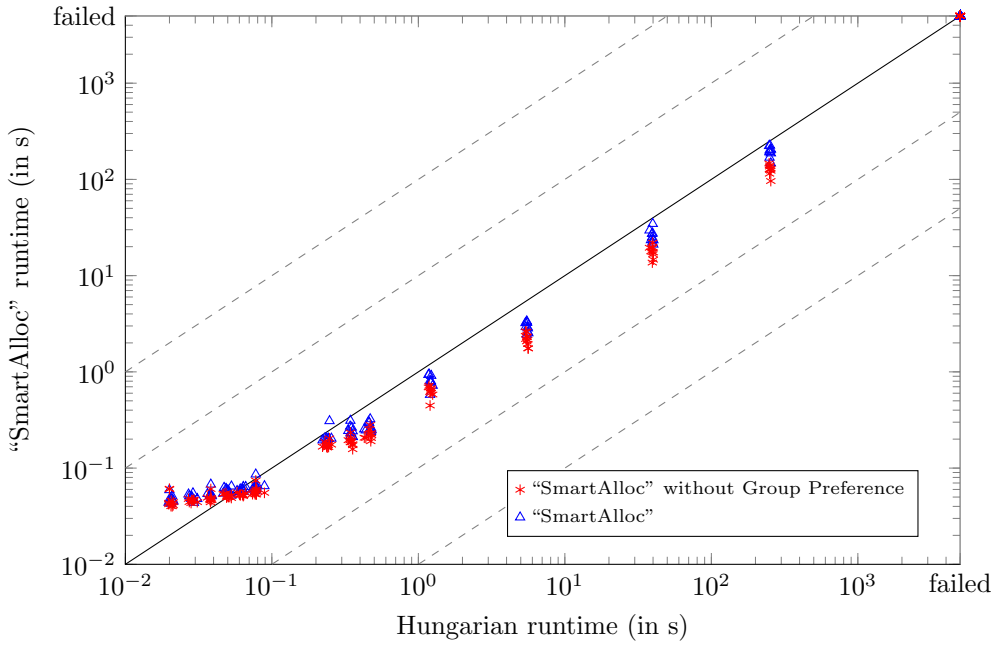


Figure 5.3: Comparison of runtime between the Hungarian Method and "SmartAlloc" variations for preference type p0, including failures and unsolvable problems.

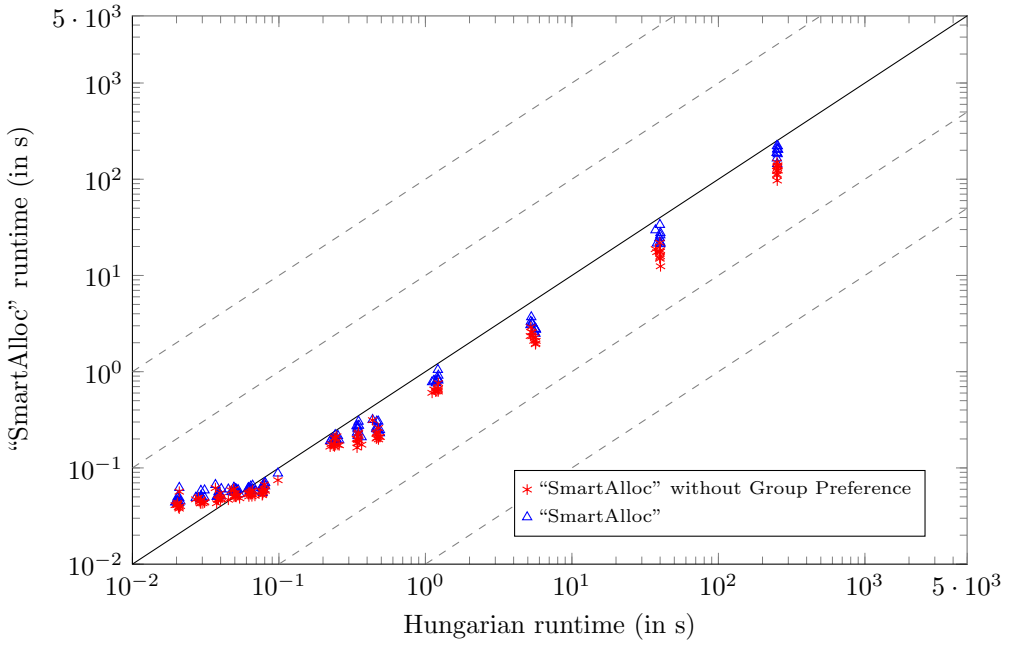


Figure 5.4: Comparison of runtime between the Hungarian Method and "SmartAlloc" variations for only solvable problems with preference type p0.

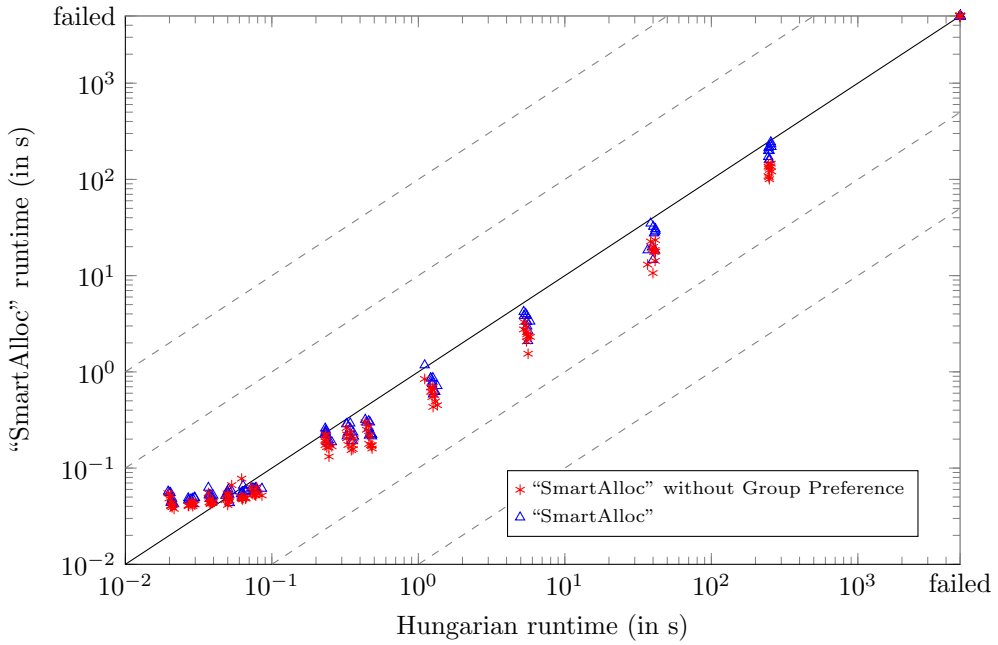


Figure 5.5: Comparison of runtime between the Hungarian Method and "SmartAlloc" variations for preference type p1, including failures and unsolvable problems.

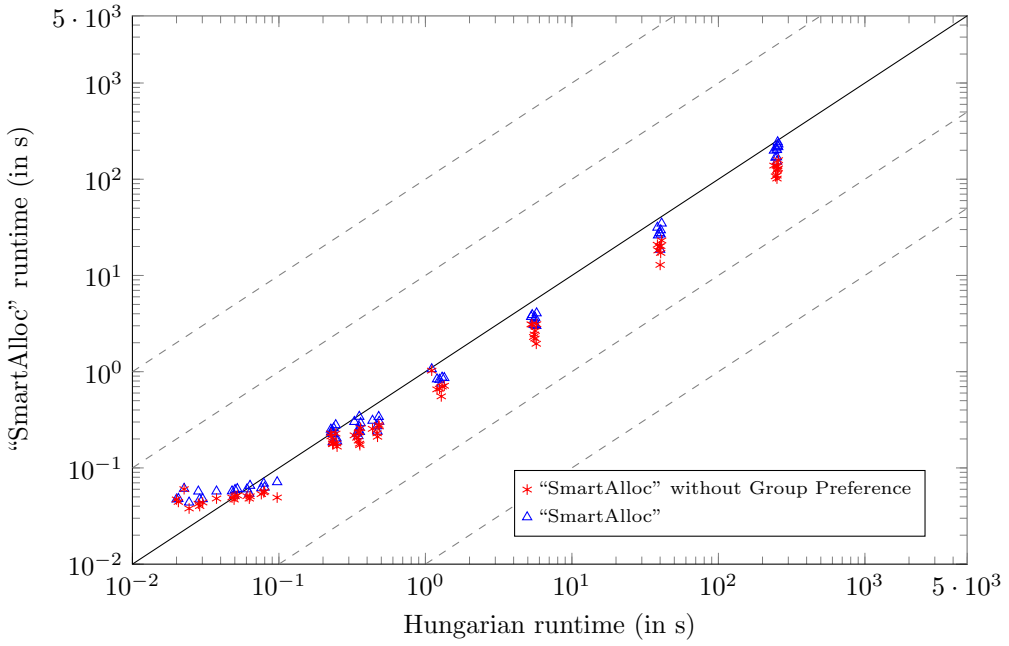


Figure 5.6: Comparison of runtime between the Hungarian Method and "SmartAlloc" variations for only solvable problems with preference type p1.

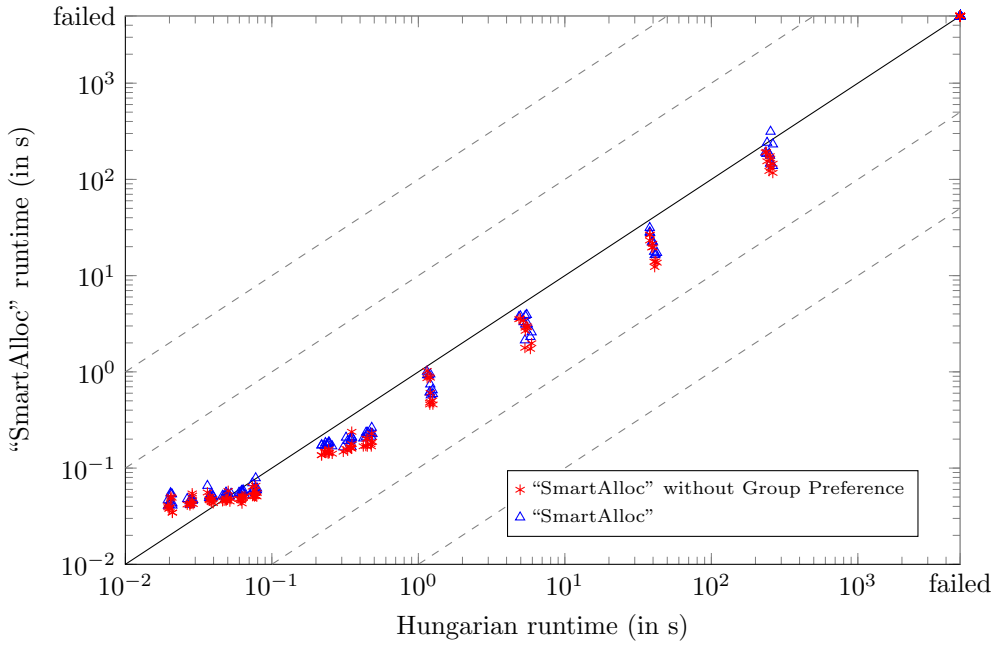


Figure 5.7: Comparison of runtime between the Hungarian Method and "SmartAlloc" variations for preference type p2, including failures and unsolvable problems.

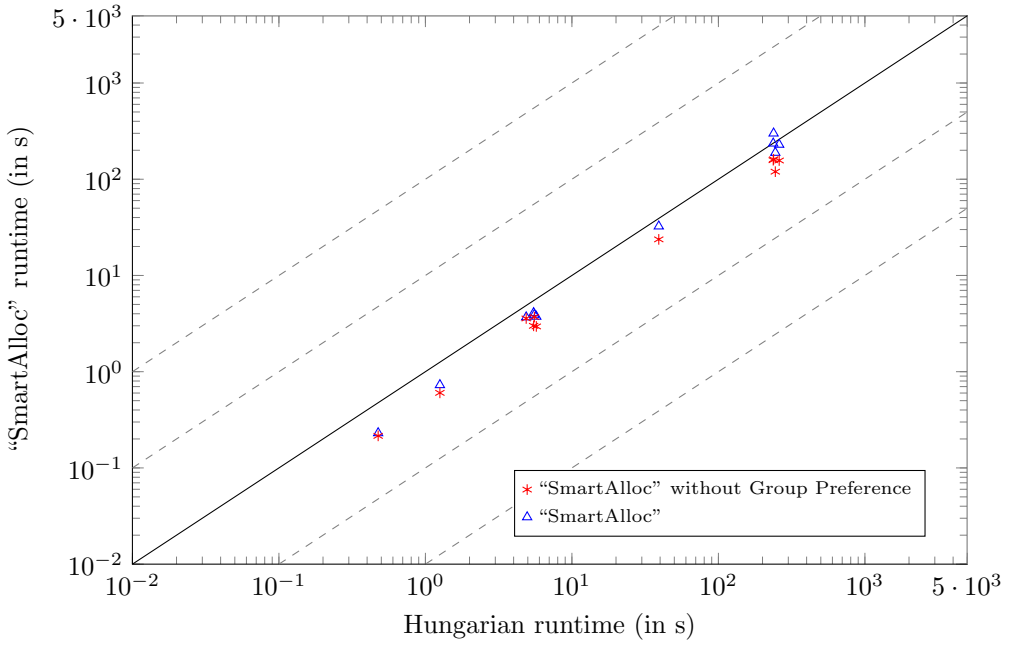


Figure 5.8: Comparison of runtime between the Hungarian Method and "SmartAlloc" variations for only solvable problems with preference type p2.

When examining individual preference types (Figures 5.3, 5.4, 5.5, 5.6, 5.7 and 5.8), similar trends are observed. In all three cases, the "SmartAlloc" variants tend to perform faster than the Hungarian Method, with the "SmartAlloc" variant without group preference generally being faster than the one with group preference. The scattering of points in Figures 5.4, 5.6 and 5.8 reflects the varying problem sizes, with preference types p0 and p1 exhibiting

more points than p2. This indicates that fewer problems were successfully solved under p2, not because of reaching memory or time limits, but because they were unsolvable due to the stringent constraints imposed by this preference type. Notably, as the problem size increases (i.e., more students), a greater number of p2 problems are solved, likely due to statistical reasons.

For scenarios involving 750 students or more, none of the algorithms, including the Hungarian Method, were able to complete the assignment within the specified time limit due to reaching resource limits, such as time or memory (visible in Figures 5.3, 5.5 and 5.7 by the points that fall into the "Failed" category). The complexity arises from the design of the generator, where each exercise group consists of 25 to 60 students. As the system scales, the number of time slots also increases, leading to a combinatorial explosion in determining the language assignments. For example, with 10 time slots, 2^{10} different language combinations need to be considered. To provide evidence that the number of timeslots is the bottleneck, some local tests with 2000 students and only 5 timeslots were conducted. These tests could indeed be solved, as shown in Table 5.1.

Problem	**Hungarian Method**	**"SmartAlloc"**	**"SmartAlloc" w/o group pref.**
n2000-p0-t5-0	241.65	20.02	15.16
n2000-p0-t5-1	241.63	18.22	14.49
n2000-p0-t5-2	235.21	18.32	14.00
n2000-p0-t5-3	236.64	16.25	12.37
n2000-p0-t5-4	242.62	17.73	13.23
n2000-p0-t5-5	243.95	16.40	11.55
n2000-p0-t5-6	241.13	15.81	12.61
n2000-p0-t5-7	234.16	17.57	12.55
n2000-p0-t5-8	239.21	18.17	13.14
n2000-p0-t5-9	234.20	16.51	12.36
n2000-p1-t5-0	251.59	22.02	15.42
n2000-p1-t5-1	241.44	20.30	14.78
n2000-p1-t5-2	251.03	18.13	13.76
n2000-p1-t5-3	249.61	21.61	14.78
n2000-p1-t5-4	239.88	19.53	14.40
n2000-p1-t5-5	257.73	19.97	13.18
n2000-p1-t5-6	255.08	22.24	16.11
n2000-p1-t5-7	257.11	23.68	12.71
n2000-p1-t5-8	243.74	18.54	14.26
n2000-p1-t5-9	254.99	22.77	16.64
n2000-p2-t5-0	254.95	24.76	16.28
n2000-p2-t5-1	259.27	21.48	21.95
n2000-p2-t5-2	256.31	22.57	16.05
n2000-p2-t5-3	262.58	18.23	14.01
n2000-p2-t5-4	265.06	23.09	22.06
n2000-p2-t5-5	262.96	32.34	44.80
n2000-p2-t5-6	263.98	24.04	20.96
n2000-p2-t5-7	260.47	17.34	13.85
n2000-p2-t5-8	262.58	20.39	22.56
n2000-p2-t5-9	267.31	25.98	26.58

Table 5.1: Solve Times (in s) From Local Tests with 2000 Students and 5 Timeslots

The use of `itertools` to generate language combinations was identified as a significant memory consumer, especially in scenarios involving 1500 students or more. The issue became particularly evident with the Hungarian Method when dealing with 1500 students. At this problem size onwards the Hungarian Method started encountering a memory error, as indicated by the appearance of unexplained errors in the report in addition to the -24 exit codes. This issue was mitigated by introducing a Try-Catch block, which resolved the memory error and the -24 exit codes but resulted in a new exit code 1 being displayed instead.

Interestingly, the exit-code -24 persisted for the Hungarian Method in scenarios with 750 to 1000 students and similarly for both "SmartAlloc" variants from 750 up to 2000 students.

This suggests that from a problem size of 750 onwards, the algorithms are likely hitting a time limit due to the non-scalable nature of handling language combinations. Beyond this threshold, the precise cause of the error remains unclear.
The transition from `exit_code 0` to `exit_code -24` for for all algorithms starting from a problem size of 750 students and the transition from `exit_code -24` to `exit_code 1` for the Hungarian Method at the 1500-student problem size can be seen in table 5.2.

Prob.	**Hungarian Method**	**"SmartAlloc"**	**"SmartAlloc" w/o group pref.**
n50-p0-0	0	0	0
n50-p0-1	0	0	0
n50-p0-2	0	0	0
...	...	...	...
n500-p2-7	0	0	0
n500-p2-8	0	0	0
n500-p2-9	0	0	0
n750-p0-0	-24	-24	-24
n750-p0-1	-24	-24	-24
n750-p0-2	-24	-24	-24
...	...	...	...
n1000-p2-7	-24	-24	-24
n1000-p2-8	-24	-24	-24
n1000-p2-9	-24	-24	-24
n1500-p0-0	1	-24	-24
n1500-p0-1	1	-24	-24
n1500-p0-2	1	-24	-24
...	...	...	...
n2000-p2-7	1	-24	-24
n2000-p2-8	1	-24	-24
n2000-p2-9	1	-24	-24

Table 5.2: Exit Codes Across Different Problem Sizes After Introducing a Try-Catch Block

The findings in this section suggest that, while solving a single perfect matching with the Hungarian Method is theoretically polynomial, the overall runtime behavior in practice is heavily influenced by the exponential increase in the number of language combinations. As the number of timeslots increases, the number of possible language combinations grows exponentially, leading to an overall exponential runtime when considering all combinations. This effect is particularly pronounced in the larger problem instances, resulting in significant deviations from the expected polynomial performance when only a single matching is considered.

5.3 Solution Quality and Cost Analysis

This section analyzes the solution quality and associated costs of the different algorithms, focusing on the ability of the Hungarian Method and the "SmartAlloc" variants (with and without group preference) to find valid assignments within the given time constraints and the costs associated with these assignments.
The Hungarian Method consistently produced solutions; however, many of these solutions violated hard constraints and were associated with significantly higher costs. This suggests that while the Hungarian Method may find assignments that technically cover all students, those with higher costs should be interpreted as indicators that hard constraints have been violated, rendering the solution invalid. In contrast, the "SmartAlloc" variants signal the absence of a valid solution by returning None when no valid assignment exists, especially in cases where the problem size is within solvable limits.

Algorithm	**Solved (Valid)**	**Found Unsolveable**	**Not Solved**
Hungarian Method	212	178	120
"SmartAlloc"	212	178	120
"SmartAlloc" w/o group pref.	212	178	120

Table 5.3: Summary of Valid Solutions, Found Unsolvable and Not Solved Problems

As shown in Table 5.3, the summary indicates that out of 510 problem cases, the Hungarian Method and both "SmartAlloc" variants found valid solutions in 212 cases. In contrast, for 178 problems, the Hungarian Method returned solutions that violated hard constraints, effectively signaling that no valid assignment existed—similar to the "Found Unsolvable" category in the "SmartAlloc" variants. Analyzing the solvability rates for different preference types, it was found that for problems involving up to 500 students, the p0 preference type achieved a 100% rate of valid solutions, while p1 and p2 had valid solution rates of 40% and 10%, respectively.
Looking at the problems where all algorithms found a valid solution, all algorithms found the same cost. This consistency is not surprising when comparing the Hungarian Method to the "SmartAlloc" variant without group preferences, as both are designed to solve the same problem optimally. The more interesting comparison arises when considering the "SmartAlloc" variant with group preferences. Despite the additional complexity introduced by group preferences the overall solution quality as reflected in the total costs, remained consistent. This suggests that the inclusion of group preferences did not lead to overall worse satisfaction, indicating that the preferences of students to be grouped together were naturally aligned with other preferences, such as timeslot and language preferences.
The analysis also revealed that the preference of students to be grouped with others had no significant impact on the overall solution costs. This outcome is logical, as students who prefer to be grouped together often share similar preferences in other aspects, leading to consistent cost outcomes. The difference in the number of solutions found by the Hungarian Method compared to the "SmartAlloc" variants primarily reflects the instances where constraint violations occurred. These violations, particularly in the Hungarian Method, resulted in solutions that were technically feasible but associated with higher costs due to the breach of hard constraints. When focusing solely on the valid solutions—those that adhered to all

constraints—it becomes evident that the actual number of feasible and effective solutions was consistent across all algorithms. Thus, the observed variations in the total number of solutions are more indicative of differences in how each algorithm handles and discards invalid solutions, rather than a fundamental difference in their capacity to find valid and optimal assignments.

6 Conclusion

This thesis presented a comparative evaluation of three methods for optimizing student time slot allocation: the Hungarian Method, "SmartAlloc" without group preferences and "SmartAlloc" with group preferences. The evaluation primarily focused on the performance of these algorithms in terms of solution quality and computation time, particularly as the complexity of the problem increased.

The Hungarian Method, known for its polynomial time efficiency in solving assignment problems, performed adequately in scenarios with smaller problem sizes where the number of language combinations was limited. However, as the problem size increased, the method's performance degraded notably. While this decline could be attributed to the exponential growth in the number of language combinations, it is important to note that this exponential growth affects both the Hungarian Method and "SmartAlloc". The key difference lies in how each algorithm handles this complexity. "SmartAlloc" was able to manage the combinatorial explosion more effectively due to its more robust optimization framework. In smaller problem instances, the overhead associated with the MILP solver in "SmartAlloc" made it less efficient compared to the Hungarian Method, which has a simpler, more streamlined process. However, as the problem size grew, the additional capabilities of "SmartAlloc" allowed it to scale better, handling the increased complexity more efficiently than the Hungarian Method. In contrast, the "SmartAlloc" approach, based on Mixed-Integer Linear Programming (MILP), demonstrated greater flexibility and scalability. The extensive optimization capabilities of the MILP solver enabled "SmartAlloc" to efficiently handle larger problem instances and various preference types. The ability to incorporate group preferences into the allocation process highlighted the practical advantages of "SmartAlloc", making it more applicable to complex, real-world scenarios. While "SmartAlloc" required more computational resources for smaller problem instances, its performance benefits became increasingly apparent as problem size and complexity grew. This is largely because "SmartAlloc" is better equipped to manage the exponential growth in the number of language combinations, allowing it to scale more effectively than the Hungarian Method.

The evaluation also identified potential areas for future work. A significant challenge was managing the various language combinations, which, under the current methodology, required generating and testing all possible combinations. This approach was found to be

not only memory-intensive but also time-consuming and less feasible for larger problem instances due to the exponential explosion in complexity. Future research could explore integrating these language combinations directly into the MILP model as constraints, potentially reducing both memory consumption and the time complexity, thereby improving overall efficiency.

Additionally, alternative methods that generate language combinations incrementally, rather than storing them all at once, could enhance the performance of both "SmartAlloc" and the Hungarian Method by mitigating the exponential explosion of combinations and making the solution process more scalable.

All in all, while the Hungarian Method remains suitable for simpler assignment problems, the MILP-based "SmartAlloc" approaches offered enhanced flexibility and scalability, particularly in scenarios involving multiple preferences and constraints. Although the performance difference between "SmartAlloc" and the Hungarian Method was not always substantial in smaller instances, "SmartAlloc" consistently managed the combinatorial explosion of languages in larger, more complex problem instances. Future work could focus on further optimizing these methods, particularly in handling language combinations, to improve their applicability to a broader range of real-world scheduling problems.

Bibliography

[1] Kuhn, H. W. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, pages 83–97 (1955).

[2] Schrijver, A. *Theory of Linear and Integer Programming*. John Wiley & Sons (1998).

[3] Sarker, R. A. and Newton, C. S. *Optimization Modelling*. CRC Press (2007).

[4] Karp, R. M. Reducibility Among Combinatorial Problems. In Miller, R. E. and Thatcher, J. W., editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York (1972).

[5] Cela, E. *The Quadratic Assignment Problem*. Springer Science & Business Media (1998).

[6] Dimitris Bertsimas, R. W. *Optimization Over Integers*. Dynamic Ideas (2005).

[7] Shoja, S. *On Complexity Certification of Branch-and-Bound Methods for MILP and MIQP with Applications to Hybrid MPC*. Phd thesis, Linköping University, Linköping, Sweden (2023). URL `https://www.diva-portal.org/smash/record.jsf?pid=diva2:1639268`.

[8] Deza, A. and Khalil, E. B. Machine Learning for Cutting Planes in Integer Programming: A Survey. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, IJCAI-2023, page 6592–6600. International Joint Conferences on Artificial Intelligence Organization (2023). URL `http://dx.doi.org/10.24963/IJCAI.2023/739`.

[9] Silva, W. A., Bobbio, F., Caye, F., Liu, D., Pepin, J., Perreault-Lafleur, C., and St-Arnaud, W. Design and Implementation of an Heuristic-Enhanced Branch-and-Bound Solver for MILP (2022). URL `https://arxiv.org/abs/2206.01857`.

[10] Szigeti, Z. Matchings in bipartite graphs. *Combinatorial Optimization and Graph Theory ORCO*, pages 1–34 (2012). URL `https://pagesperso.g-scop.grenoble-inp.fr/~szigetiz/OCG/bipartite_matching_CM.pdf`. Accessed: 2024-05-09.

[11] Ravindra K. Ahuja, J. B. O., Thomas L. Magnanti. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall (1993).

[12] Crouse, D. F. On implementing 2D rectangular assignment algorithms. *IEEE Transactions on Aerospace and Electronic Systems*, pages 1679–1696 (2016). URL `https://doi.org/10.1109/taes.2016.140952`.

[13] Seipp, J., Pommerening, F., Sievers, S., and Helmert, M. Downward Lab. `https://doi.org/10.5281/zenodo.790461` (2017).