IMPERIAL

A fast single-language two-sided project allocation algorithm minimising perceived unfairness

Author

Rhea Khoury

CID: 01877720

Supervised by

Dr Thomas Clarke

Second Marker

Mr Sarim Baig

A Thesis submitted in fulfillment of requirements for the degree of
Master of Engineering in Electronic and Information Engineering

Department of Electrical and Electronic Engineering
Imperial College London

2024

Abstract

This report presents the design, development, and evaluation of an efficient allocation algorithm for the final year project (FYP) allocation in the Electrical and Electronic (EE) engineering department. The primary objective is to create an algorithm that equitably satisfies both students and staff.

Addressing the allocation problem involves leveraging and adapting existing algorithms to our specific needs. This report provides background research on existing assignment problem solutions, an overview of the current system used by the EE department, and a discussion on the chosen implementation language.

The core of this project focuses on capturing specific requirements , notably perceived fairness, and customizing the algorithm accordingly. We utilized the Jonker-Volgenant algorithm, an improvement over the Hungarian algorithm for dense cases, and integrated concepts from the Student-Project Allocation (SPA) algorithm to meet all requirements. This approach effectively combines the Hungarian method with stable marriage concepts.

Implemented in F#, the algorithm was rigorously evaluated and benchmarked. Preprocessing code was developed to seamlessly interface data files with the algorithm. The system's performance is critiqued, and potential areas for further improvement are discussed.

Declaration of Originality

I hereby declare that the work presented in this thesis is my own unless otherwise stated. To the best of my knowledge the work is original and ideas developed in collaboration with others have been appropriately referenced.

I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link.

I affirm that I have submitted, or will submit, an identical electronic copy of my final year project to the provided Blackboard module for Plagiarism checking.

I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me, but is represented as my own work.

I have used ChatGPT v4 as an aid in the preparation of my report. Specifically, I have utilized it to enhance the quality of my English throughout. However, all technical content and references are derived from my original text.

Copyright Declaration

Acknowledgments

I would first like to express my deepest gratitude to my project supervisor, Dr. Thomas Clarke, for his invaluable help and mentorship throughout this project. His dedication and responsiveness have been unmatched, playing a crucial role in the project's progress. His advice and support have been instrumental in the successful completion of this work, and I am very grateful to have been able to work under his guidance.

I would also like to thank Dr. Stathaki and Dr. Mikolajczyk for their willingness to meet with me and discuss this project. Their time and valuable input have been greatly appreciated.

Finally, I want to extend my heartfelt thanks to my friends and family for their unwavering support during this significant period of my life.

Contents

List of Acronyms

LAPJV Linear Assignment Problem, Jonker-Volgenant

LSAP Linear Sum Assignment Problem

EE Electrical and Electronic

FYP Final Year Project

HLP High Level Programming

SPA Student-Project allocation

HR Hospital Resident

ILS Iterated Local Search

EE Electrical and Electronic

List of Figures

List of Tables

1 Introduction

In their final year, students within the Electrical and Electronic (EE) Department at Imperial College London are required to undertake a Final Year Project (FYP) under the guidance of a faculty member. This project is a pivotal element of their academic curriculum, offering two primary pathways for engagement: students may either propose their own project, potentially in collaboration with industry, or opt to work on a project proposed by a faculty member. The principal objective of my FYP is to design and implement an efficient algorithm for matching students to projects. This algorithm aims to ensure a fair process that prevents manipulation by any party, effectively aligning student preferences with faculty assessments of suitability.

The algorithm's applicability is determined by the annual influx of approximately 200 students in the EE department, who are to be allocated across roughly 250 available or self-proposed projects. With around 50 faculty members supervising between one and eight projects each, the challenge lies in defining the requirements needed to ensure fairness and satisfaction for both parties. The next step is to develop an algorithm that addresses these requirements while operating within a reasonable timeframe.

The motivation for this FYP arises from shortcomings in the existing system, which is both time-consuming and incapable of guaranteeing an optimal match between students and projects. A previous attempt to address these issues was made last year through the implementation of a new system on a different platform, which unfortunately resulted in compatibility issues. Additionally, that project's focus was on platform development, leaving ample scope for further research into the algorithm's requirements and efficiency, as well as the rationale behind some design choices.

In this report, we will detail the process of defining the requirements, researching, selecting, and designing the algorithm to address this issue, followed by an evaluation of its effectiveness and a discussion of the results.

2

Background research & appraisal

Contents

2.1 Introduction

This section of the report is dedicated to an in-depth exploration and critical analysis aimed at understanding the intricacies of the project allocation challenge. It encompasses a thorough review of the current system and the algorithms employed by the department, cataloging their limitations and meticulously evaluating the decision-making processes and assumptions that underpin these existing solutions. By scrutinizing the effectiveness and constraints of the currently implemented algorithms, our objective is to uncover insights that will inform the development of a more sophisticated and robust solution. Subsequently, we will conduct a comprehensive literature review on various optimization algorithms to assess their suitability and efficiency in our specific context, which will facilitate the selection of the most appropriate algorithm for our needs. Moreover, choosing the right programming language is a pivotal aspect of this endeavor, requiring a detailed discussion on its merits and functionalities to effectively support our algorithmic solution. The goal is to construct a solution that not only addresses the identified gaps but is also specifically tailored to the unique needs and dynamics of our department's project allocation scenario.

2.2 The Current Algorithms

Within the realm of project allocation at our department, two main algorithms have been previously employed by students: the Steepest Descent Algorithm and the Hungarian Algorithm. While both have played pivotal roles in attempting to streamline the allocation process, they are not without their shortcomings. These implementations, though innovative at the time of their introduction, now present opportunities for enhancement. Identifying and addressing their limitations is crucial for optimizing the allocation process, with the aim of achieving the most effective match between students and projects within a constrained timeframe.

2.2.1 Steepest Descent Algorithm

Developed by Ramon Jutaviriya as part of her FYP [1], this algorithm—also known as hill climbing [2]—aims to find a good project allocation solution. Typically, it reaches a plausible solution within 30 minutes, though instances have shown enhanced outcomes after prolonged runs of up to three hours, indicating the absence of a guaranteed optimal solution point. A primary concern with hill climbing is the potential entrapment in local optima, a risk that varies based on the specific implementation nuances.

In practice, the execution of the Steepest Descent Algorithm requires iterative intervention by the FYP coordinator to hone in on a good allocation. Initially running the algorithm with the complete set of student preferences, the coordinator systematically excludes the lowest preferences in subsequent iterations. This methodical exclusion continues until a significant increase in the number of unallocated students is observed, signaling a threshold beyond which the elimination of preferences may detrimentally impact overall allocation outcomes. Throughout this iterative process, the coordinator meticulously records the dynamics of allocation—monitoring changes in student-project pairings and tracking the constancy of certain assignments. This exhaustive manual effort aims to discern an optimal solution by comparing various iterations, ultimately selecting the allocation that strikes the best balance between maximizing higher student preferences and minimizing the count of unallocated students. This approach, while thorough, underscores the algorithm's need for enhanced automation to streamline the allocation process and reduce reliance on labor-intensive manual oversight.

Moreover, the extensive runtime complicates parameter tuning, such as determining the optimal number of student preferences or the algorithm's duration. Given these challenges—ranging from uncertain optimality, susceptibility to local optima, to cumbersome manual interventions and parameterization—the Steepest Descent Algorithm, as currently implemented, falls short of efficiently meeting our allocation objectives. Therefore, there is a compelling case for its refinement to enhance both performance and automation, ensuring more effective and efficient project allocations.

2.2.2 Hungarian Algorithm

Last year, Dominic Konec [3] implemented the Hungarian Algorithm as a component of his FYP. Unlike previous projects, Konec's work predominantly focused on constructing an entire platform for managing various FYP-related steps, rather than solely concentrating on the allocation algorithm. Consequently, while the Hungarian Algorithm was a part of this larger system, there remained untapped potential for further exploration and refinement specifically within the context of project allocation.

The platform developed by Konec utilized Power Apps, which, despite its innovative approach, was ultimately considered unstable. As a result, this platform—and the allocation algorithm integrated within it—is not in use for the current FYP allocation process due to compatibility

issues. Addressing these incompatibilities is crucial for the potential future integration of this system.

Notably, the Hungarian Algorithm, known for its global optimization capabilities, addressed many of the shortcomings identified in the Steepest Descent Algorithm (refer to 2.2.1). It improved upon aspects such as runtime efficiency, solution optimality, and automation of several processes. However, there is room for further refinement to ensure the algorithm comprehensively meets our specific requirements. This report will delve into these aspects, assessing the Hungarian Algorithm in detail and exploring other potential solutions to ensure the chosen approach is the most suitable for our needs. We will also define and analyze the requirements in more detail, tailoring our algorithm to better meet these specific requirements.

It is important to note for context that the allocation algorithm in Konec's FYP was divided into two distinct stages. The first, a preprocessing stage, involved using data from students and staff to create a cost matrix. This part of the code was written in F# [4]. The second stage, executing the Hungarian Algorithm itself, was developed in C++ [5].

2.3 Implementation Language

As highlighted in the earlier discussion, the preprocessing component of the allocation algorithm was developed in F# by Dominic, while the Hungarian Algorithm itself was implemented in C++, adapted from an existing online version. In contrast to this bifurcated approach, I have opted to utilize F# exclusively for both components of the project. The rationale behind this choice, along with a detailed exploration of F#'s capabilities and advantages for our specific application, will be elaborated upon in the subsequent subsection. This will lead us into an examination of some specific features of F# that make it particularly suited to our needs.

2.3.1 Choosing F#

F#, with its functional-first paradigm [6], offers an excellent framework for data transformation and manipulation [7], making it a superior choice for the preprocessing tasks required in our project. This language distinguishes itself through a concise syntax and robust features designed to facilitate complex data processing tasks[8]. Its capability to utilize immutability and pure functions enables the expression of complex data transformations in a way that is both clear and maintainable. Furthermore, F#'s strong type inference system plays a crucial role in error detection during compile-time, significantly improving code reliability for data processing operations.

The ability of F# to interoperate seamlessly with the extensive .NET libraries [9] opens up a wealth of opportunities for incorporating sophisticated tools for statistical analysis, machine learning, and data visualisation into our project. Additionally, the language's native support for asynchronous and parallel programming, complemented by features such as pattern matching[10] and immutable collections, positions F# as an ideal candidate for handling large and varied datasets efficiently.

One potential challenge with F# is its relatively lower popularity, which can limit the availability of online resources and community support. However, this potential drawback is mitigated in our case, notably because Professor Clarke, who is not only an expert in F# but also the supervisor of this FYP, leads the High Level Programming (HLP) course using F#. This ensures a readily available source of expert guidance and support for our project. Given that a portion of the algorithm has already been implemented in F# by Dominic, leveraging F# for both preprocessing and the algorithm itself presents a logical and streamlined approach to development. Moreover, using a single language for the entire project facilitates ease of deployment and ensures a self-contained solution.

While numerous implementations of the Hungarian Algorithm are readily available in C++, the compatibility between F# and C++ initially suggested a feasible integration. However, the additional overhead of managing C++ dependencies and the gcc compiler solidified the decision to standardize on F# for the entirety of the project. To thoroughly understand the implications of this choice, it was essential to assess the performance difference between the two implementations. Consequently, I benchmarked the Hungarian Algorithm code for both languages, as shown in Table 2.1. As expected, the C++ implementation is significantly faster; however, when we include reading the cost matrix from a text file—which is necessary if we use C++ since the preprocessing is done in F#—the running time becomes closer to that of the F# implementation. In the context of our application, the difference between 305 ms for F# and 14.57 ms for C++ with matrix reading is negligible from a user's perspective. This makes our decision simpler since performance is not a critical factor. Consequently, we chose F# because it eliminates the need for additional dependencies and ensures a simple, robust, concise, and easily maintainable development environment, which is crucial for ensuring correctness in our project.

Algorithm	Average running time
F# Implementation	305.6 ms
C++ Implementation	540 ns
C++ Implementation with matrix read from text file	14571324 ns = 14.571324 ms

Table 2.1: Benchmarking results for C++ and F# algorithm implementations

F# Data Structures

The main data structures that fit our case are arrays, lists, or maps. Arrays and lists have similar built-in functions and can be interchangeable in some scenarios. After reading through the F# documentation [11]–[13], we can compare these data structures [10], [14]: the main difference is that arrays are mutable and have a fixed size, whereas lists are immutable and consist of nodes that can be easily manipulated to create new lists, while maps store key-value pairs. In F#, maps are immutable, while dictionaries are their mutable counterpart.

When comparing performance, arrays tend to be better since they have O(1) time complexity for random access, whereas lists are O(n) to access an element by index since they traverse the elements sequentially. Additionally, because arrays are mutable, we don't need to create a new instance every time we want to change an element, which makes them more memory and time efficient. Maps provide average O(1) time complexity for lookups, insertions, and deletions, making them efficient for tasks that involve key-based access. However, iterating over maps can be more complex and less efficient compared to arrays and lists, as maps are optimized for key-based access rather than sequential processing. Additionally, maps could have extra overhead since they store both keys and values.

For sparse data, both lists and arrays store all elements, including zeros, which can lead to inefficiencies. Lists have overhead for each node, which includes the element and a reference to the next node, while arrays have minimal per-element overhead but require contiguous memory allocation. Maps (or dictionaries) provide a more efficient way to handle sparse data, as they store only non-zero elements and their indices. This approach significantly reduces memory usage by avoiding the storage of zero elements.

The specific choice of data structure will depend on the algorithm and representation we select. For example, arrays might benefit graph implementations due to their mutability and efficient random access, but lists could be better for scenarios where we need to frequently prepend elements or traverse sequentially. The decision on which data structure to utilize will become clearer once we finalize our choice of algorithm, depending on whether we need to access random indices or perform sequential access. This will further be investigated and benchmarked in section 5.4.4 to justify our decision.

2.4 Optimisation Algorithms

To choose the appropriate algorithm for solving our problem, we first need to decide how to model it. Our problem can be formulated in two ways: as a one-sided matching problem or a two-sided matching problem. Each formulation has its own set of algorithms. In the following subsections, we will discuss both options and analyze which one best fits our case.

2.4.1 Two-sided matching problem

Our FYP allocation problem can be depicted as a two-sided matching problem [15] due to the pairing of agents on distinct sides. In this scenario, agents on one side express preferences over agents on the other side, and vice versa. Specifically, one side involves students, while the other encompasses projects and supervisors.

The Student-Project allocation problem (SPA)

The Student-Project allocation (SPA) [16] problem closely corresponds to our FYP allocation issue and can be categorized as a two-sided matching problem. It serves as a generalization of the Hospital Resident (HR) problem [17] or the Stable Marriage problem [18] and involves a set of students, projects, and supervisors. The primary objective is to match students with projects, taking into account various factors such as student preferences for projects, supervisor preferences for students, and the capacity constraints of both projects and supervisors.

The goal of this algorithm is to achieve a **stable matching** M. In this context, stability means that there are no **blocking pairs**. A blocking pair occurs when a student and a project, not currently matched together, would both prefer to be matched with each other over their current assignments. By eliminating blocking pairs, the algorithm guarantees a harmonious allocation where no participant has an incentive to deviate from their assigned match.

Applying the SPA to our problem involves several considerations to assess its suitability. Firstly, the algorithm requires rankings from both the students and the supervisors, meaning that professors need to rank all students who have selected their project as a preference for each project they plan to supervise. This can be particularly challenging for supervisors, especially when dealing with highly popular projects. The algorithm lacks flexibility to accommodate equal suitability rankings, as it relies on the preference lists to break ties. If two students share the same suitability ranking, the tie persists, which limits flexibility for both staff and students.

Secondly, as mentioned earlier, the algorithm outputs stable matchings. Finding stable matches can be particularly challenging, especially in scenarios involving a large number of participants with similar preferences. Consequently, this may lead to a significant number of students remaining unallocated, which is contrary to our intended outcome.

In summary, while the SPA algorithm may not be the most optimal choice for our case, the underlying concepts it employs, such as stable matchings and blocking pairs, hold significant importance. These concepts can prove valuable for later analysis of our solution. Despite its limitations, it's worth noting that the algorithm runs in polynomial time, signifying that it is relatively fast. The concepts of the SPA are further discussed in section 4.4.1, particularly when examining perceived fairness—a requirement addressed by the SPA but not by the alternative algorithms we will be considering.

2.4.2 One-sided matching problem

Our FYP allocation problem can also be conceptualized as a one-sided matching problem [19], providing us with the opportunity to explore a variety of algorithms. By combining the preferences of both staff and students, we can establish weighted connections between projects and students, transforming it into a one-sided matching problem. The most straightforward form of a one-sided problem is the Linear Sum Assignment Problem (LSAP) [20].

In an LSAP, we work with a set of agents, tasks, and associated costs, where each agent is assigned a specific task. This problem can be represented through a cost matrix or a bipartite graph. In our context, students serve as the agents, projects as the tasks, and the costs are a composite of student and staff preferences. Typically, a solution to the LSAP is an assignment that minimizes the overall cost. While our problem can be formulated as an LSAP, incorporating all the requirements that will be discussed in Chapter 3 might prove challenging. In Chapter 4 we will explore the feasibility of integrating these requirements and discuss potential strategies. If feasible, this formulation allows us to leverage various optimization algorithms designed for solving the LSAP.

In the following section, we reviewed several algorithms, notably two picked from the comprehensive list of complex bipartite Graphs algorithms provided in a paper by Ran Duan and Seth Pettie [21]. We aimed to choose simple algorithms that could best perform in our case, given that we are dealing with a rather small but dense problem. Additionally, we selected different types of algorithms to explore a range of approaches and identify the most suitable one for our needs.

The Hungarian Algorithm (1955)

The Hungarian method [22] stands as one of the earliest algorithms designed to solve the LSAP in polynomial time. Developed in 1955 by Harold Kuhn, this global optimization algorithm has earned acclaim for its efficiency and simplicity. The algorithm offers two primary formulations: it can be represented either as a matrix[23] or as a bipartite graph[24], both illustrated in Figure 2.1. It's essential to note that the Hungarian algorithm is specifically designed for solving a perfectly matched bipartite graph or a square matrix. This requirement arises from the algorithm's foundation in augmenting paths within a bipartite graph, and the inherent square structure of the graph.

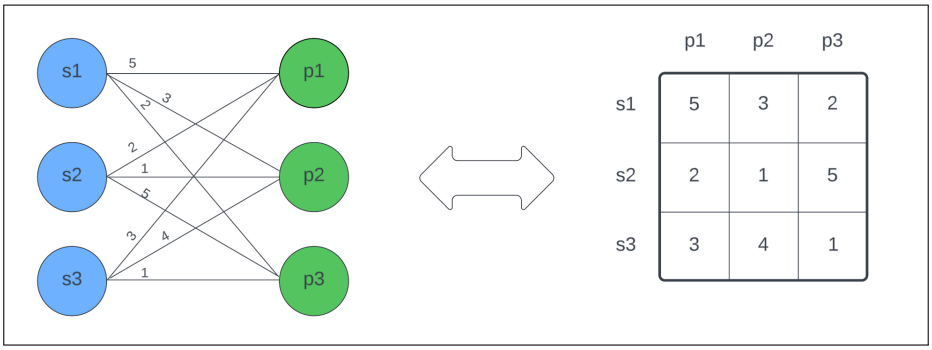


Figure 2.1: Example of a bipartite graph representation (on the left) and a cost matrix representation (on the right) of an assignment problem

The underlying logic for both formulations is fundamentally the same; however, their applications differ, as outlined in Table 2.2. Before delving into the table, it is essential to clarify some definitions to fully comprehend the concepts involved:

- **Vertex Labeling:** In a bipartite graph, when initializing the Hungarian algorithm, creating labels for vertices is a crucial step. For these labels to be feasible, the sum of the labels of two vertices connected by an edge must be greater than or equal to the weight of the edge.
- **Perfect Matching:** This represents an optimal matching where all vertices are paired. In the context of a bipartite graph, it denotes a perfect matching of tight edges, implying that for each pair of matched vertices, the sum of their labels equals the weight of the connecting edge.
- **Alternating Paths:** These are paths that alternate between vertices in the current matching and vertices not in the matching. The latter are referred to as **exposed** vertices.
- **Augmenting Path:** Augmenting paths are alternating paths that commence and conclude at an exposed vertex. This characteristic allows the augmentation of the current matching by inverting edges along this path, thereby increasing the size of the matching.
- **Alternating Tree:** An alternating tree initiates from a vertex, known as the **root**, where all paths originating from the root are alternating.

Steps	Bipartite Graph Representation	Time Complexity	Matrix Representation	Time Complexity
Initialization and Reduction	Find feasible vertex labelling and some initial matching	O(n^2)	Create the adjacency matrix. Subtract the minimum of each row from each element in that row. Do the same for columns.	O(n^2)
Augmentation Step (Main Loop)	Build alternating trees and look for augmenting paths to augment matching and update node labels. Keep iteratively augmenting and improving along with updating labels until maximum weight matching is achieved.	O(n^3)	Cover the zeroes in the matrix by the minimum number of lines. If the number of lines is not equal to the number of agents, augment the elements by subtracting all uncovered elements by the minimum uncovered element and increment covered elements by two lines by the same number. Repeat the step untill the number of lines needed is equal to the number of agents.	O(n^3)
Optimality Check and Assignment	Check for perfect matching and make assignments based on graph markings	O(n^2)	Make sure number of lines to cover all zeroes in matrix is equal to the number of agents then make assignments based on markings	O(n^2)

Table 2.2: Comparison of Hungarian Algorithm Steps: Matrix vs. Bipartite Graph Representation with Time Complexity Analysis

As illustrated in Table 2.2, both formulations exhibit the same time complexity of O(N^3), where N represents the number of agents, or in this context, students. A closer examination reveals that, for both versions, the primary bottleneck to the algorithm lies in the augmentation step, constituting the main loop.

Specifically, in the bipartite graph representation, during the process of finding augmenting paths, we iterate N times (representing each agent) and check each edge at most once, resulting in a complexity of O(NE), where E is the number of edges. This complexity arises because all vertices on each side are connected, implying that the number of edges (E) is equivalent to the product

of the number of agents and tasks . In a perfectly matched square graph, such as the one under consideration, the number of edges becomes equal to N^2 since the number of agents on each side is equal. Conversely, in the matrix representation, the complexity is $O(N^2M)$. This complexity arises from iterating N times over a matrix with dimensions N times M, where M is the number of projects. Additionally, since the matrix is square, M is equal to N.

After a meticulous and detailed comparison of both representations, the choice between them demands careful consideration. In our scenario, opting for the matrix formulation proves to be more convenient. In general, when it comes to coding, working with matrices is often more straightforward than grappling with graphs. Furthermore, a plethora of functions and data structures is available to support matrix operations, making the matrix formulation a pragmatic choice for our specific context. The steps outlined in Table 2.2 also appear to be straightforward and easy to implement, and F# provides the necessary data structures and functions to cover all requirements.

While the Hungarian Algorithm is highly efficient and quick to execute for small datasets due to its simplicity, its execution time increases significantly as the dataset size grows. Consequently, numerous models were developed after the Hungarian method to address and reduce the asymptotic time complexity. In the subsequent sections, we will explore these models and evaluate their relevance to our specific case, considering factors such as dataset size and computational efficiency.

Auction Algorithm (1979)

The auction algorithm, introduced by Dimitri Bertsekas [25], is a parallelizable combinatorial optimization algorithm capable of solving various problems, including the assignment problem. In essence, the algorithm leverages auction processes and economic concepts like prices and equilibria to determine the most profitable assignments. Each agent i is required to pay a price p_j for a specific object j. However, the agent holds a personal valuation a_{ij} of the object. For instance, if a single right shoe is auctioned, and I possess a pair with the right one missing, I may value this object more than others and potentially more than its auction price. In this scenario, the goal is to maximize the agents' happiness, quantified as $a_{ij} - p_j$.

Let's provide an overview of the algorithm's steps. Initially, all agents place bids on the objects they desire. An initial assignment is then established by matching the highest bidders with the objects. Subsequently, prices and bids are iteratively updated based on competitiveness, and the process repeats until the optimal matching is achieved. The time complexity of this algorithm is $O(NA\log(NM))$, where N is the number of agents, A is the number of edges, and M is the largest weight. Importantly, the bipartite graph need not be complete, resulting in a smaller number of edges compared to the Hungarian algorithm.

The auction algorithm exhibits numerous advantages and excels in solving certain problem types. Primarily, it is tailored for optimal performance with sparse graphs or matrices, where a significant number of entries are zero or undefined. This characteristic aligns with the algorithm's time complexity, as sparsity results in fewer edges and, consequently, leads to more efficient asymptotic behavior. Furthermore, the auction algorithm is parallelizable, making it particularly suitable for distributed systems by facilitating decentralized decision-making. Lastly, it accommodates non-integer costs, adding to its flexibility.

In contrast, the Hungarian algorithm is more suited to integer values and dense matrices. The choice between the two algorithms depends on a careful performance comparison to determine the best fit for a given scenario. Fortunately, such a comparison is available in a study [26], revealing that both algorithms deliver comparable solution quality and memory allocation requirements. However, in terms of performance, the Hungarian algorithm consistently outperforms the auction algorithm across various cases. The centralized approach of the Hungarian algorithm, where agents have a holistic view of the entire problem, contributes to its superior performance. Consequently, based on our analysis, the Auction algorithm will not be considered further in our study.

Jonker-Volgenant Algorithm (LAPJV) (1987)

The Jonker-Volgenant algorithm, also known as LAPJV [27], is designed to address the assignment problem, leveraging a shortest path augmenting approach. This method efficiently discovers augmenting paths, surpassing the Hungarian algorithm's augmenting path step in terms of efficiency. The Hungarian algorithm's most time-consuming step lies in finding augmenting paths, as discussed in 2.4.2 so an improvement in that area makes sense. LAPJV introduces a novel initialization process and incorporates Dijkstra's shortest path method [28] with certain modifications.

Steps	Algorithm Description
Initialization	In this initial phase, three crucial procedures are performed: the reduction of columns, the transfer of reduction from unassigned to assigned rows, and the augmenting reduction of unassigned rows. These operations set up the foundational structures required for subsequent steps.
Augmentation	The augmentation step involves determining the shortest augmenting path for one additional assignment. This is achieved through an adaptation of Dijkstra's algorithm, where an alternating path of minimal total reduced cost is found and used to augment the solution. This step is key to achieving an optimal assignment.
Adjustment of Solution	Following the augmentation, the adjustment of the solution takes place. This step involves updating the variables based on the current state of the assignment. If the matching is not complete, the algorithm may iterate and repeat the augmentation step. The adjustment step contributes to refining the solution.

Table 2.3: Overview of the steps of the LAPJV algorithm

Looking at Table 2.3, we observe that the initialization process in LAPJV is more extensive compared to the Hungarian algorithm's initialization. However, this elongated initialization significantly accelerates the subsequent augmentation process. The overall complexity of LAPJV is O(n^3), a result of both the initialization and augmentation steps sharing this time complexity. The initialization step complexity can be explained by the fact that augmenting reduction is at most O(n^2) steps with each step involving O(n) operations. While the theoretical complexity of the LAPJV algorithm aligns with that of the Hungarian Algorithm, real-world testing and experiments, as reported in the following paper [27], reveal that LAPJV consistently outperforms the Hungarian and other algorithms in practical scenarios.

In our context, deciding between the LAPJV and the Hungarian algorithm requires careful consideration. While the test results in the cited paper [27] demonstrate LAPJV's faster runtime in a Pascal implementation, our use of F# introduces differences. LAPJV operates exclusively on a bipartite graph representation and avoids matrix operations, meaning it would necessitate different structures than those used by the Hungarian Algorithm when implemented in F#. This includes the use of mutable variables and arrays, which might offer more efficiency but require careful handling. However, it is challenging to assess which algorithm would perform better without implementing and testing them ourselves. This evaluation will be conducted and detailed in section 5.4.3.

Scaling Algorithm (1989)

In 1989, Harold Gabow and Robert Tarjan introduced a novel scaling algorithm [29] designed to efficiently solve various network problems, including the assignment problem. This algorithm employs a bipartite graph representation and utilizes an augmenting path approach to achieve optimal matching. It has a time complexity of O($\sqrt{n}mlog(nN)$) where n is the number of vertices, m is the number of edges, and N represents the maximum cost magnitude. This algorithm's

asymptotic time complexity is lower than that of the other algorithms reviewed, a distinction that can be attributed to its innovative scaling method. The Gabow-Tarjan algorithm ingeniously merges the optimality criterion of the Hungarian Algorithm with the efficiency of the Hopcroft-Karp Algorithm [30], which seeks the shortest length augmenting paths, whereas the Hungarian algorithm selects the minimum cost augmenting paths. This combination becomes particularly effective when costs are minimized, enabling the shortest path to concurrently be the least costly. Through the application of a scaling technique that adjusts costs, the algorithm ensures a perfect matching at each of the $log(nN)$ scales.

When it comes to analysing this algorithm for our case, we have to recall that we are dealing with a rather limited dataset. Despite its superior time complexity, the Gabow-Tarjan algorithm's performance does not automatically translate to faster execution for small datasets. The implementation's scaling technique and the use of advanced data structures entail a computational overhead, the benefits of which become pronounced primarily in large and especially sparse graph instances. For small problems, this overhead may negate any theoretical time complexity advantages, potentially rendering the algorithm slower than simpler alternatives.

Given these considerations, the Gabow-Tarjan scaling algorithm was ultimately considered overly complex for our specific needs, particularly in the context of a small dataset. Furthermore, the intricacies of implementing the required sophisticated data structures in F# would likely offset the language's advantages, leading to the decision against its use. This decision takes into account not only the inherent complexity and potential performance implications but also aligns with our rationale for choosing F# as the development language, prioritizing simplicity and maintainability.

2.5 Design Desiderata

Deciding how to design solutions is always a challenging task, especially when dealing with ambiguous questions that can be interpreted and solved in many different ways. It is important to set foundations early on to have a clear path, especially when faced with a difficult problem. First, we will discuss the philosophy behind our decision-making and then address a recurrent case we will encounter when designing our algorithm and possible solutions.

2.5.1 Occam's Razor

Occam's Razor [31] is a principle attributed to the 14th-century English logician and Franciscan friar William of Ockham. It states that among competing hypotheses that predict equally well, the one with the fewest assumptions should be selected. This principle advocates for simplicity, suggesting that the simplest solution is often the best one. In the context of our project, Occam's Razor will guide our design choices, favoring straightforward and efficient solutions over more complex ones whenever possible. This approach will make decision-making more efficient and help us move on to testing faster, thereby allowing us to evaluate the hypotheses we initially made. Throughout our project, we will encounter many instances where we need to decide on the weights of different objectives, often referred to as fudge factors. Determining the best values for these can be challenging, particularly when dealing with conflicting metrics. In these cases, we will apply Occam's Razor to choose the simplest and most effective solutions.

2.5.2 Combining metrics

When designing our solution, we will frequently need to combine multiple metrics, effectively merging various cost functions [32] into a single global function. The goal is often to find a Pareto optimal [33] solution. This concept helps identify trade-offs between conflicting metrics

and ensures that improvements in one metric do not disproportionately degrade others. A Pareto optimal solution is one where improving one metric would worsen another.

The simplest method to approach this is the weighted sum approach [34]. Here, each metric is assigned a weight based on its importance. However, before applying this method, we must consider two important aspects:

- **Normalization:** The metrics must be on the same scale to ensure that one does not dominate the other if the weights are set equally. Normalizing the metrics allows for fair comparison and combination.
- **Convexity:** The objective functions should ideally be convex. When combining metrics using a linear combination, convex functions ensure that any local minimum is also a global minimum. If the objective functions are not convex, the weighted sum approach might not reach all Pareto optimal solutions because non-convex objectives can have regions that are not accessible through simple weight adjustments [35].

By considering these factors, we can develop a robust framework for combining metrics that balances multiple objectives efficiently.

2.6 Conclusion

In conclusion, the need for this FYP arises from the potential improvements over previous FYPs on this subject. The previously implemented Steepest Descent algorithm proved to be very time-consuming and did not guarantee an optimal solution, while the previously implemented Hungarian algorithm was designed around a system that is not in use, rendering it incompatible. Additionally, since the entire system was designed, there is considerable room for further exploration, such as investigating other algorithms and examining the requirements in detail to tailor our solution accordingly.

We chose F# as our implementation language due to its robustness, simplicity, maintainability and compatibility with the preprocessing part. Furthermore, we found that formulating our problem as a one-sided matching problem was the most feasible approach, as the two-sided problem required data that was impractical to obtain, such as staff rankings of students. Within the one-sided matching algorithms, we narrowed it down to two possible options: the Hungarian and LAPJV algorithms, which will be further explored to determine the best one. Other options were deemed unsuitable due to their performance and complexity.

Throughout this project, we decided to adhere to the principle of Occam's Razor, favoring straightforward and efficient solutions over more complex ones. This approach will not only guide our design decisions but also ensure that our solutions are practical and implementable within the given constraints. We have also discussed how to combine metrics and the important aspects to consider when faced with this task, ensuring a balanced and effective solution.

3 Requirement Capture

Contents

3.1 Introduction

As discussed in 2, a major reason for the need for this FYP is to closely examine the requirements to create a solution that addresses all of them. In this chapter, we will provide an overview of these requirements. These requirements were determined from prior surveys conducted by Dominic in his FYP the previous year to gain insight into the student point of view, as well as my own experience with this process. However, the staff point of view is also crucial in determining these requirements, which is why I engaged in discussions with staff members to gain a comprehensive understanding of potential improvements. Through this collaborative process, we established a set of criteria that describe what constitutes high-quality allocation, along with other necessary requirements for the algorithm. An algorithm is deemed successful if it meets all the specified requirements below.

3.2 Staff Opinions

Gathering insights from staff members was crucial for defining the algorithm's requirements. Engaging in discussions with several MSc organizers allowed me to gain a comprehensive understanding of their perspectives. Ensuring their satisfaction and willingness to utilize the developed algorithm was a key consideration during these interactions.

3.2.1 Meeting with Dr. Stathaki

The meeting with Dr. Stathaki was enlightening, revealing several insights I had not previously considered. A key realization was the dual aspect of fairness; it must be extended to both students and staff. For students, fairness entails providing equal opportunities without granting preferential treatment based on higher grades. Similarly, fairness for staff means ensuring they all have an equal opportunity to work with eager and capable students. It is inequitable for some faculty members to end up supervising students who lack interest in their project's subject area merely because they are committed to being inclusive and accommodating. Instead, there should be an equitable distribution of students, ensuring all staff members have the chance to engage with students who are genuinely interested in their projects. In conclusion, the guiding principles are fairness and inclusion, achieved by ensuring everyone has equal opportunities within the algorithm. This opinion means preventing staff and students from forming their own pairings, thereby upholding the integrity of the equitable distribution process.

3.2.2 Meeting with Dr. Mikolajczyk

The discussion with Dr. Mikolajczyk highlighted several critical insights. Primarily, it was emphasized that a student's project is a significant milestone in their university career, and thus, the matching decision should be approached with the utmost seriousness by both students and staff. Given the year-long commitment required, it was argued that students and staff should have substantial autonomy in forming their pairings, ensuring a mutually satisfactory and enjoyable match. However, this autonomy might lead to a situation where some students are left without partners, thereby being compelled to settle for projects that are less appealing.

3.2.3 Meeting with Dr. Clarke

As the coordinator of the Final Year Project (FYP), Dr. Clarke offers invaluable insights into the intricacies of FYP management. He highlighted several nuanced scenarios that could lead to perceptions of unfairness, which will be further explored in 3.4.1, as well as some insights about unallocated students, which will be explored in 3.4.3. Additionally, Dr. Clarke pointed out crucial considerations, such as supervisor workload, detailed in 3.5. An important distinction was made between a student's ability and interest. It was noted that expecting students without the necessary skills to undertake certain projects, or to enroll in related modules, is unreasonable. Conversely, it's considered reasonable to expect students who have the requisite skills but lack interest to engage in projects. This underscores the belief that students should be prepared to undertake tasks even when they fall outside their primary areas of interest.

3.2.4 Concluding Reflections on Staff Insights

Having considered a range of opinions, some of which were divergent, it became crucial for me to reconcile these perspectives to ensure satisfaction with the final outcome. Recognizing the significant stress the project selection process can impose on students, and its importance, my

objective is to synthesize these varied opinions into a balanced approach. This approach aims to facilitate a solution where both students and staff play an integral role in the decision-making process while ensuring fairness and inclusivity. One proposed idea, aiming to satisfy both Dr. Mikolajczyk's and Dr. Stathaki's viewpoints, is to allow staff to select students for a limited number of projects, striking a balance between differing perspectives. However, this does not fall within the scope of this project and should be left to the FYP coordinator to implement.

3.3 Student's opinion

Having gathered staff insights, it is equally important to consider the students' opinions, as a student's project is a significant milestone in their university journey. This will be done by examining previously conducted surveys and discussions, as well as drawing from my own and my classmates' experiences with this process.

Generally, it is better for students to list more preferences, as this makes it easier for the algorithm to allocate as many students as possible. However, students are often dissatisfied when they are allocated to lower preferences, even if they selected them. Despite having the option to rank their preferences equally, many students do not utilize this feature. This insight underpins the "no low preferences" requirement discussed in Section 3.4.2.

I also gained an understanding of what students perceive as unfair, which will be explored in Section 3.4.1. Additionally, Dominic pointed out in his report that students are smart and may attempt to game the system, which is an important consideration when designing the algorithm and will be further discussed in the section 3.6.

3.4 Quality of Allocation

Using the insights gathered from staff and students, we have established requirements that define a good allocation if the algorithm adheres to them. A high-quality solution is one that satisfies both staff and students.

3.4.1 Fairness

First, the algorithm must output a fair solution. However, fairness is a concept that varies across perspectives. After discussions with multiple professors and students, two distinct interpretations of fairness have been identified for our context.

Real Fairness

According to the Merriam-Webster dictionary [36], fairness is defined as the quality of impartial treatment, characterized by the absence of favoritism towards any side. In our context, this implies ensuring that the established rules apply uniformly to all parties, preventing the algorithm from showing favoritism towards certain students. A common scenario that challenges fairness occurs when students attempt to manipulate the system by submitting fewer preferences than required. This manipulation can lead the algorithm to allocate them higher preferences at the expense of others. An example of this can be seen in Figure 3.1, where the number of required preferences is two. As shown, s2 selected two preferences, but s1 only selected one preference. Since the algorithm tries to allocate as many students as possible, s2 did not get their first preference because s1 did not adhere to the number of required preferences. The thick lines in the figure represent the allocations.

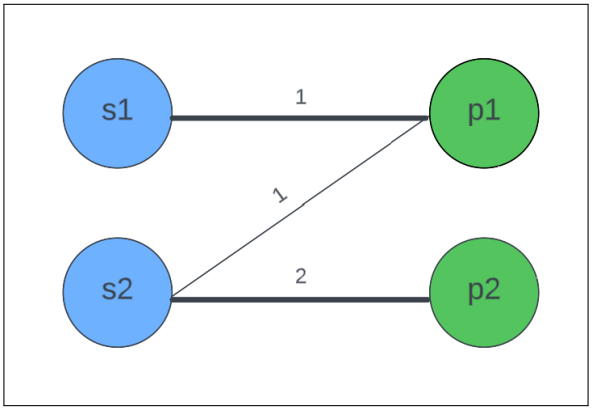


Figure 3.1: Example of real unfairness

Perceived Fairness

Upon consultation with staff and students, it became apparent that there is another aspect of fairness not covered by real fairness. From the staff's point of view, staff members typically prefer students who express high enthusiasm for a project. In other words, staff favor getting assigned students who have ranked the project as a top preference, preferably their first choice. From this perspective, it might be perceived as unfair to allocate a project to a student who selected it as their fifth choice when another student ranked it as their first choice. This potential unfairness arises because the algorithm operates as a global optimization, focusing on minimizing overall cost rather than evaluating each case individually. Similarly, from the students' perspective, it would be seen as unfair if their first choice was allocated to another student who ranked it lower. An example of this can be seen in Figure 3.2, where a project was allocated to s3 as their third choice, even though s2 ranked it as their first preference. The thick lines in the figure represent the allocations and the red line highlights the perceived unfairness.

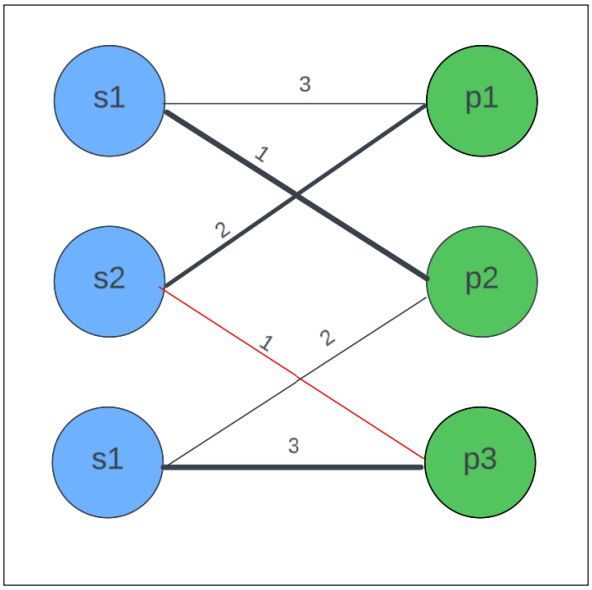


Figure 3.2: Example of perceived unfairness

3.4.2 High Preference Average / No Low Preferences

For the algorithm, it is crucial not only to aim for a high average in student preferences to ensure engagement and satisfaction with their assigned projects but also to minimize the occurrence of allocations at the lower end of their preference lists. As discussed in the student's opinion section 3.3, even though students are required to select many preferences to allow the algorithm to allocate most, if not all, students, they prefer to be allocated within their top four preferences. Achieving a solution with high preference allocations benefits both students and staff by enhancing the overall project experience and ensuring a deeper commitment to the project outcomes. Therefore, while monitoring the effectiveness of the allocation process, special attention should be paid not just to maximizing the average preference score but also to ensuring that allocations do not frequently fall into the lowest preference categories. This dual focus helps strike a balance between overall satisfaction and avoiding the disengagement that can occur when students are assigned projects that rank at the bottom of their preference list.

3.4.3 Full Allocation

The solution should aim for full allocation wherever feasible. This means the algorithm must strive to allocate as many students as possible. Typically, for unallocated students, there are three possible outcomes:

1. They may choose a project from the limited and often less desirable remaining pool.
2. A supervisor from one of their top preferences might agree to take on an additional project if their initial offerings were fully subscribed.
3. A supervisor familiar with the student's field of interest, based on their project selections, might create a new project specifically tailored for that student.

The last two scenarios are generally preferable for students; however, these opportunities are limited and uncertain. In general, having unallocated students creates more work in trying to figure out which project to allocate them to, as we also have to consider the projects that the student is capable of doing. We can't just allocate them to any random remaining project. Therefore, having unallocated students is highly undesirable and should be avoided whenever possible.

3.4.4 Concluding remarks on the requirements

When considering all the defined requirements, we realize that many of them are contradictory. For example, achieving full allocation often necessitates assigning some students to lower preference projects. Conversely, if we aim to allocate only high preferences, it may leave some students unallocated. Another example could be seen when looking at perceived fairness. It arises because the algorithm is designed to perform a global allocation that minimizes the total cost. If we modify the allocations to adhere to perceived fairness, we will likely increase the total cost, which could negatively impact the average preference score.

These examples highlight the inherent contradictions in our problem. Creating a solution that meets all these requirements will require finding a middle ground where compromises are made. Balancing these competing demands is essential for developing an effective and satisfactory allocation algorithm.

3.5 Supervisor Loading

Supervisors typically face constraints on the number of projects they can feasibly oversee, making it crucial to respect these limits. There are instances where supervisors might propose a number of projects that exceed their supervisory capacity. To address this, it is imperative for the algorithm to incorporate mechanisms that restrict matchings based on each supervisor's capacity. It's important to note that these limits are not uniform across all supervisors; they vary according to each individual's workload capacity and other commitments. This variability necessitates a flexible approach in the algorithm to ensure equitable and manageable distribution of supervisory responsibilities.

3.6 Gameability of the System

Ensuring that students cannot exploit or "game" the system is paramount to maintaining fairness for both students and staff. As seen in the section 3.3, students are smart and like to game the system if they are able to. While measures to prevent gaming were implemented in last year's project, it remains a critical concern that necessitates continued attention. Strategies such as requiring students to submit multiple preferences and restricting their access to specific scoring metrics have proven effective in reducing the system's susceptibility to manipulation. However, it is crucial to rigorously evaluate the final product against this requirement to confirm its resilience to gaming and uphold the integrity of the allocation process.

3.7 Conclusion

In conclusion, both student and staff opinions have proven to be crucial in defining what constitutes a high-quality allocation. However, some of these opinions are conflicting, which is also reflected in the opposing requirements. This highlights the primary challenge of this project: finding common ground where all requirements are addressed as much as possible. Additionally, other important factors must be considered when creating the algorithm, such as managing supervisor workloads to ensure staff can effectively supervise, and addressing the gameability of the system to ensure fairness and maintain the integrity of the process.

4 Analysis & Design

Contents

4.1 Introduction

This chapter explores the thought process behind the design of the full allocation system. It includes the preprocessing of data to ensure it is in a suitable form for the optimization algorithm, and the allocation algorithm that produces the final allocations. Our primary focus will be on analyzing and designing a solution that addresses the requirements described in Chapter 3. In this design phase, we must consider the trade-offs and compromises necessary to meet all the requirements, which will then be evaluated in the evaluation chapter 6 after implementing our design.

4.2 Preprocessing of Data

Given our decision to frame our project allocation challenge as a one-sided matching problem as described in chapter 2, a critical step involves the combination of both student and staff preferences to construct a comprehensive cost matrix. This matrix will serve as the foundation for whichever optimization algorithm we select, facilitating the determination of optimal project-student matches. It's imperative to ensure that the generated weights or costs not only support the algorithm's function but also strictly conform to our predefined requirements. Importantly, this preprocessing stage remains consistent across different optimization algorithms, underscoring its universal applicability to our chosen solution path (The Hungarian algorithm or the LAPJV).

4.2.1 Combining Staff and Student Preferences

To transform the data into a one-sided matching problem, we need to combine staff and student preferences into costs for the edges, reflecting both perspectives. This combined score serves as the cost function for the optimization algorithm, as it is these scores that the algorithm will try to minimize. First, let's look at the staff suitability rankings, detailed in Table 4.1 below. Staff assign one of three options to each student who lists their project as a preference. These options are then translated into costs. Students rank each project from 1 to 10. The two scores are combined using a weighted sum. The formula for this combination is given below:

$$C = w_{student} R_{student} + w_{staff} R_{staff} \tag{4.1}$$

where:

C = Cost of the edge between student and project
$R_{student}$ = Rank assigned by the student to the project
R_{staff} = Suitability rank assigned by staff to the student
$w_{student}$ = Weight of the student ranking in the cost calculation
w_{staff} = Weight of the staff suitability ranking in the cost calculation

The two weights are hyperparameters set by the FYP coordinator. By default, they are both set to 1, indicating equal importance in the decision-making process. In this default setting, preferences are not normalized because students tend to avoid projects where staff have given a "Maybe" ranking. Consequently, staff suitability rankings are almost always "Definite" and do not significantly impact the allocation. Additionally, the FYP coordinator can adjust the equivalent costs and weights through hyperparameters, as detailed in 4.8, making extra normalization unnecessary.

Since all the optimization algorithms we intend to use support data in an adjacency matrix format[37], we can utilize this representation. In this format, the rows represent the students, the columns represent the projects, and the indices indicate the values of the edges connecting them. An example of this is shown in Figure 2.1, which displays both the graph and the corresponding matrix representation.

Suitability Option	Meaning	Equivalent Cost
Unsuitable	The student does not have the necessary skills to undertake the project	Not applicable, as a preference with a staff suitability of unsuitable will be discarded
Maybe	The student might be able to undertake the project	2
Definite	The student possesses the necessary skills to undertake the project	0

Figure 4.1: Table of staff suitability options

4.2.2 Dealing with Unallocated Students

Given that each student currently expresses preferences for only 10 projects, scenarios may arise where a complete allocation—assigning every student to one of their preferred projects—is not feasible. This constraint could lead to some students potentially remaining unallocated based on their stated preferences alone. The framework of the allocation problem, however, assumes a fully matched bipartite graph, necessitating that each student be linked to every project to ensure a comprehensive matching.

To reconcile this discrepancy and facilitate a full matching within the algorithm's parameters, we introduce artificial edges connecting students to projects not included in their preferences, assigning these edges significantly high weights (also refered to as $C_{unallocated}$). This approach effectively disincentivizes the algorithm from pairing students with these less desirable projects, except as a last resort. Consequently, if a student is matched to a project via one of these high-weighted connections, it is indicative of their unallocated status in practical terms, as it signifies a match outside their preferred options. This method allows us to maintain the structural requirements of a fully matched bipartite graph while acknowledging the reality of potential unallocations in a manner that is consistent with the algorithm's operational logic.

In the matrix representation of assignment problems, strategically introducing high costs between students and projects not listed as preferences does not impact the asymptotic complexity of the algorithm, since the algorithm checks all costs and its time complexity is determined solely by the dimensions of the cost matrix—the number of projects and students-, as described in section 2.4.2. However, in the graph representation, the time complexity is $O(NE)$, with E being the number of edges, meaning adding a significant number of edges might affect performance notably. For instance, if we have 200 students and 250 projects, and each student has set 10 preferences, we would have an increase of $240 \times 200 = 48,000$ edges to ensure a perfect matching. An alternative method could be to add an extra dummy project per student, connected to that student by an edge with a large cost. If a student is matched to the dummy project, they are deemed unallocated. This strategy reduces the extra number of edges to 1 per student; in the example, this would result in only 200 additional edges.

4.2.3 Dealing with an Unbalanced Problem

The optimization algorithms we are evaluating are tailored to address the allocation problem, typically modeled through a square matrix or a fully-connected bipartite graph, where the two sets of nodes (e.g., students and projects) have a one-to-one correspondence. In our scenario, however, we encounter an imbalance: there are approximately 200 students for 250 projects, indicating a surplus of projects. To rectify this imbalance and enable the use of these optimization algorithms, we introduce the concept of dummy students. These dummy students are hypothetically added to the student set to match the number of projects, ensuring that each project can potentially be assigned. Each dummy student is linked to every project with a nominal cost of 0. This approach

allows the algorithm to operate on a balanced matrix, facilitating the identification of an optimal allocation while maintaining the integrity of the actual student-project preferences and constraints.

This method makes sense when using the matrix representation, as most implementations of the Hungarian algorithm or others have built-in methods to deal with unbalanced matrices by adding these dummy agents to pad the matrix, turning it square. When using the graph implementation, this method still offers benefits. Although it involves adding nodes and edges, introducing zero-cost edges between dummy students and projects does not increase the computational burden in the same manner as additional real edges would since it will later be revealed that sparser graphs are easier to solve, as can be seen in the following table 5.1. However, if the method mentioned in 4.2.2 is employed, this implies that the number of projects has effectively increased by the number of students, potentially leading to performance issues. For example, if we have 200 students and 250 projects, employing the previous method means we would have 200 students and 450 projects. Implementing the dummy student method to make the graph square, we increase the number of edges by $250 \times 450 = 112,500$. An alternative method would be the doubling technique described in the referenced paper [38], where each side of the graph would include all projects and all students, adding an edge between each node and its corresponding node on the other side. This would increase the number of edges by $450 + 200 = 650$, a significant improvement in terms of the number of additional edges required.

4.2.4 Dealing with Supervisor Loading

As outlined in the supervisor loading requirement section 3.5, it is crucial to adhere to the constraints regarding the maximum number of projects each staff member can supervise. Addressing this constraint introduces a layer of complexity that necessitates automation to ensure fairness and efficiency in the project allocation process.

Encountering situations where a professor has proposed more projects than their supervisory capacity allows presents a unique challenge. Directly eliminating excess projects based solely on preferences is impractical, as it does not provide a clear indication of which projects would be most beneficial to retain or remove. Nonetheless, knowing the exact number of projects to be excluded provides a starting point for resolution. Suppose a staff member proposes P projects but has a supervisory limit of L; the solution involves removing $P - L$ projects from consideration.

To automate this adjustment, we introduce $P - L$ dummy students into the system for each staff member exceeding their project supervision limit. These dummy students are uniquely linked to all projects proposed by the respective professor, each with an edge weight of zero. This configuration ensures these placeholder students are assigned projects, effectively excluding $P - L$ projects from the actual allocation process. Moreover, this approach maintains the optimality of the overall solution, as these dummy allocations serve to remove projects that would otherwise increase the total cost of the solution minimally, ensuring that only the most cost-effective, real student-project matches are made.

An illustrative example of handling supervisor loading constraints is depicted in Figure 4.2 below. In this scenario, the green projects (where $P = 3$) are overseen by a single staff member whose supervision limit is $L = 2$. Given that P exceeds L, the introduction of a dummy student becomes necessary to maintain adherence to these limits. The bold lines in the figure indicate the actual allocations, with labels above each line showing the weights of these allocations. Notably, Student 3 is assigned to Project 4 rather than Project 3, despite the latter having a higher weight. This arrangement underscores the mechanism's effectiveness in preserving the optimality of the matching; allocating the dummy student to Project 2, for example, would result in a higher overall cost. Through this method, we ensure that the solution remains cost-effective while respecting staff supervision capacities.

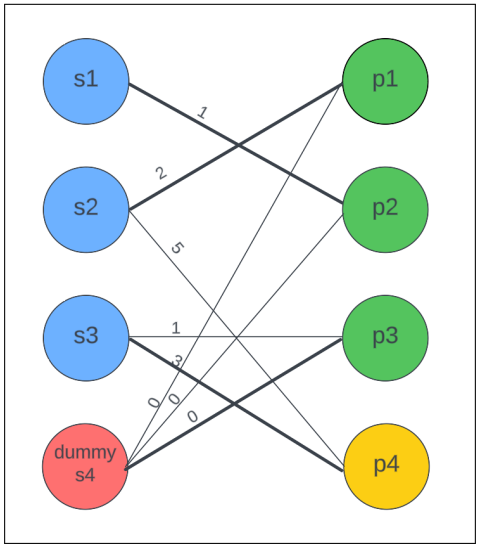


Figure 4.2: Example of the addition dummy students solving the supervisor loading problem

4.2.5 Implementing Real Fairness

As discussed in 3.4.1, our allocation system should incorporate measures to penalize students who fail to adhere to specified rules, specifically those not submitting the required number of preferences. This disciplinary action is integrated during the preprocessing phase to ensure fairness during the allocation. To effectively deprioritize these students in the matching process—thereby making it a lower priority to match them compared to their peers—we create a new multiplier that scales their preferences. This new weight acts as a deterrent, making the algorithm less likely to select these students for their preferred projects, effectively penalizing non-compliance. To determine the value of this weight, we need to consider the issue more closely.

There are different severities to this issue. For example, if the number of required preferences is 10, a student who selected only 1 preference has violated the rules much more than a student who has selected 9 preferences. Therefore, we interpolate the weight between a maximum and minimum value depending on the difference between the number of preferences given by the students and the required amount. We perform a linear interpolation [39] for a given value of x within the range $[x_{\text{Min}}, x_{\text{Max}}]$. The formula for this interpolation is given by:

$$w = w_{\text{Min}} + \frac{(x - x_{\text{Min}})(w_{\text{Max}} - w_{\text{Min}})}{x_{\text{Max}} - x_{\text{Min}}} \tag{4.2}$$

where:

w = Interpolated weight
w_{Min} = Minimum weight, corresponding to x_{Min}
w_{Max} = Maximum weight, corresponding to x_{Max}
x = The value for which we are interpolating the weight
x_{Max} = Maximum value of x
x_{Min} = Minimum value of x

In our case, x_{Min} is the required preferences minus 1, x_{Max} is 1, and w_{Min} and w_{Max} are hyperparameters set by the FYP coordinator. This approach ensures that students are fairly penalized according to the severity of their actions.

4.2.6 Concluding Remarks on Preprocessing Design

In conclusion, we have outlined the crucial adaptations required in the preprocessing phase of the algorithm, as summarized in Table 4.2 below. Notably, issues such as supervisor loading and real fairness requirements are addressed at this stage, obviating the need for their consideration in the design of the optimization algorithm itself. The preprocessing steps are designed to be general and will remain consistent regardless of the optimization algorithm employed. There are also several hyperparameters required to implement the discussed adaptations; these are set to default values but can be adjusted by the FYP coordinator. These will be described in more detail in the implementation chapter.

The preprocessing phase will output a cost matrix (also called an adjacency matrix) where the rows represent the students, the columns represent the projects, and the values represent the cost of the edges connecting them. The various cases and their corresponding costs are detailed in Table 4.1.

Case	Equivalent Cost
Student ranked the project, the staff suitability ranking of the student is not unsuitable, and the student adhered to the required number of preferences	$C = w_{student}R_{student} + w_{staff}R_{staff}$
Staff suitability ranking is unsuitable	$C = C_{unallocated}$
Student did not include the project in their preferences	$C = C_{unallocated}$
Student ranked the project, the staff suitability ranking of the student is not unsuitable, but the student did not adhere to the required number of preferences	$C = w_{penalty}w_{student}R_{student} + w_{staff}R_{staff}$. The equation for $w_{penalty}$ can be found in 4.2.

Table 4.1: Table of possible cases with respective costs

Problem	Proposed Solution	Requirement Addressed
Algorithm requires one cost for each edge	Combine both student and staff rankings using a weighted sum formula to create costs. Further explanations are in section 4.2.1	
Algorithm requires a fully connected graph	Set a large unallocation cost for all edges connecting students to projects they have not ranked. If these are assigned, they are deemed unallocated, as discussed in section 4.2.2	
Unequal number of students and projects	Create dummy projects or students to make the numbers equal. If a project or student is matched to a dummy counterpart, they are deemed unallocated, as seen in section 4.2.3	
Supervisors have a limit on the number of projects they can supervise	Add dummy students such that the system allocates them to projects, relieving the burden on supervisors, as discussed in 3.5	Supervisor loading (see 3.5)
Some students select fewer preferences than required	Implement a penalty mechanism, as seen in section 4.2.5	Real fairness (see 3.4.1)

Table 4.2: Summary of Adaptations done in preprocessing for the allocation algorithm

4.3 Algorithm Design

Once the data has undergone preprocessing, we obtain a cost matrix and are ready to apply an optimization algorithm to determine the optimal matching of students to projects. At this stage, we face strategic choices regarding the operational framework within which the optimization algorithm functions. For example, we must decide whether to conduct a single allocation process, accepting the initial solution as final, or to execute multiple allocation iterations, subsequently comparing these outcomes to select the most suitable solution. These decisions will be based on which approach best addresses all requirements.

4.3.1 Multi-allocation

These types of designs usually require multiple runs of the optimization algorithm. We will discuss two approaches: the bottom-up and the top-down approach.

Bottom up

Inspired by the process described in 2.2.1, where the FYP coordinator incrementally removes the lowest preferences to refine allocations, we explore a ”bottom-up” multi-allocation approach. This method involves systematically reducing preference lists to potentially improve the matching outcome, detailed as follows:

- **Step 1:** Conduct an initial allocation using all student preferences, recording both the total cost and the count of unallocated students.
- **Step 2:** Remove the lowest preference for all students and rerun the allocation, noting changes in total cost and unallocated students.
- **Step 3:** Continue truncating the next lowest preference and reallocating until observing an increase in the number of unallocated students.

The feasibility of adopting this approach will be assessed against our project requirements, as summarized in Table 4.3. Additionally, should this method be chosen, fine-tuning the hyper-parameter that dictates the acceptable increase in unallocated students will be necessary to optimize the loop’s termination condition.

Requirements Addressed	Requirements Ignored
No Low Preferences: This method inherently seeks to eliminate lower preferences from consideration, directly aiming to ensure higher satisfaction by avoiding least-desired allocations.	**Perceived Fairness:** The global optimization approach at each iteration does not account for individual fairness concerns, potentially overlooking specific cases highlighted in 3.4.1.
Full Allocation: Initially, the method strives for complete allocation, retreating from preference truncation if it leads to an increase in unallocated students, thereby maintaining a focus on achieving as full an allocation as possible.	

Table 4.3: Evaluation of Bottom-Up Multi-Allocation against Requirements

The main trade-off with this approach is between unallocated students and achieving higher preferences, meaning we must balance between the full-allocation requirement 3.4.3 and the no low

preference requirement 3.4.2. We must ask ourselves which is worse: having unallocated students or assigning low preferences? Is there a way to meet both requirements halfway? Unallocated students might have to choose from a limited pool of remaining worse, which seems worse than getting a low preference. However, they might get a better project if a professor increases their load or creates a new project as discussed in section 3.4.3. This complexity will be explored further in the implementation and testing section, particularly in section 6.2.2, to see the real-world implications of this trade-off.

Top Down

This method iteratively allocates projects based on student preferences, starting from the highest and progressing to the lowest. The steps are as follows:

- **Step 1:** Conduct an allocation using only the students' top preferences.
- **Step 2:** Exclude matches made in Step 1, then reallocate based solely on the students' second preferences.
- **Step 3:** Continue this process through all preference levels, reallocating at each step for the next preference down the list.

Requirements Addressed	Requirements Ignored
Perceived Fairness: This approach specifically caters to the requirement of perceived fairness. By isolating each preference level for allocation, the method transforms a global optimization challenge into a series of smaller, local optimizations. Allocating from top to bottom ensures highest preferences are given priority, addressing concerns outlined in 3.4.1.	**High Average Preference:** While aiming to fulfill higher preferences, this method does not holistically address the overall optimization. It might not achieve the highest possible average preference, and variability in allocations could result in some students receiving much lower preferences.
No Low Preferences: This method ensures that the highest preferences are allocated first, prioritizing higher preferences as much as possible before considering lower ones.	**Full Allocation:** The localized focus of this method may overlook the objective of full allocation, prioritizing the fulfillment of top preferences without ensuring every student is matched.

Table 4.4: Evaluation of Top-Down Multi-Allocation against Requirements

This method introduces several complications, such as potentially overlooking a student's preference if it has already been allocated since it was another student's higher preference. Additionally, maintaining adherence to supervisor capacity limits might become challenging, as allocations are fragmented across multiple steps rather than determined holistically.

Upon evaluating this approach against our requirements (see 4.4), it becomes evident that it fails to comprehensively address several key objectives. Moreover, employing an algorithm like the Hungarian, inherently designed for global optimization, in a fragmented manner undermines its fundamental purpose and strengths.

4.3.2 One-allocation

In a one-allocation system, the optimization algorithm is run once using the complete dataset, and the resulting matching is used as the final solution. The goal is to incorporate all relevant requirements into the cost function, ensuring the algorithm's output is both optimal and aligned

with our objectives, thus eliminating the need for further modifications or comparisons. However, this may not always be achievable, so we will analyze this method to determine its effectiveness in our case.

Requirements Addressed	Requirements Ignored
Full Allocation: By design, the algorithm strives for complete allocation, minimizing the cost function to the greatest extent possible. Any unallocated individual significantly increases the overall cost, pushing the algorithm towards finding a full matching.	**Perceived Fairness:** Given its nature as a global optimization challenge, the algorithm may not address individual fairness concerns, lacking specific mechanisms to ensure equitable treatment across all cases.
High Average: The cost function essentially reflects the average preference satisfaction. Minimizing this function directly contributes to maximizing the average preference fulfillment.	

Table 4.5: Evaluation of One-Allocation against Requirements

As indicated in Table 4.5, the primary overlooked requirement is perceived fairness (see 3.4.1), which is the most challenging requirement to address. The one-allocation approach is undoubtedly the simplest, adhering to Occam's Razor discussed in the background 2.5.1, and it is also the fastest approach since the algorithm runs only once. However, to implement this method, we must find a way to address perceived fairness, as it is completely ignored in this approach.

4.3.3 Concluding remarks on Algorithm Design

In conclusion, none of the approaches discussed fully addresses all the requirements. Each approach satisfies some requirements while prioritizing them differently, presenting various trade-offs. This highlights the complexity and ambiguity of the project. However, there are valuable elements from each approach that we can utilize. The one-allocation approach serves as the best foundation since it is the simplest, fastest, and meets most requirements. Nevertheless, perceived fairness is only addressed in the top-down approach, so we should draw inspiration from it and design an enhancement to the one-allocation method to incorporate this aspect. Additionally, the bottom-up approach underscores the trade-off between unallocated students and low preferences, which will be revisited once we create an algorithm that addresses all requirements (see 6.2.2).

4.4 Addressing Perceived Fairness

As perceived fairness is the only requirement not addressed by the foundational approach we have decided to use, it is important to create enhancements to tackle it in order to achieve a complete solution that adheres to all requirements.

4.4.1 Understanding perceived fairness

Before diving into this subject, it is important to understand what perceived fairness means in our context and how others have tackled it to develop our own solution. First, we consider the SPA algorithm 2.4.1 described in the background chapter. This algorithm is specifically designed for projects and students, addressing most considerations. However, the one-sided matching problem we are now using has not been tailored for students and projects, necessitating enhancements and additional study to meet our requirements.

In the SPA algorithm, perceived unfairness is fully addressed because the allocation consists solely of stable matchings, meaning there are no blocking pairs. Blocking pairs occur when a project and a student both prefer to be matched together but are not, which is the essence of perceived unfairness. If a blocking pair exists outside the assignment, it indicates perceived unfairness, showing that the SPA is designed to avoid this issue.

In our case, staff do not rank students but provide a suitability ranking, as described in Table 4.1. These rankings are combined into one, making perceived fairness more complex to understand and visualize, as the SPA requires mutual preference, but we only have one score. Additionally, the SPA does not consider unallocated students, focusing solely on achieving a stable matching, which helps eliminate perceived unfairness.

For our case, we define perceived unfairness as an edge with a lower cost than the assigned one, both connected to the same student. We also consider the edge allocated to the project where this perceived unfairness occurred. Unlike the SPA, where it is clearly defined, our scenario requires closer attention to understand and ultimately flag perceived unfairness. An example is given in the perceived fairness section 3.4.1 of the requirement capture chapter.

4.4.2 Visualising Perceived Unfairness

To design a solution, we must first be able to identify instances of perceived unfairness within an allocation. We propose to examine each student and detect any perceived unfairness. If unfairness is detected, we flag that index and report it to the FYP coordinator as a first step. The steps we intend to follow are outlined below:

- **Step 1:** Go through each student in the cost matrix (the rows).
- **Step 2:** Examine their first preference and check which preference was allocated. If the project was allocated a lower preference, flag it and go to step 4.
- **Step 3:** Look at the next preference and perform the same checks.
- **Step 4:** Repeat these steps for each student.

Currently, we consider all preferences of a student, but preferences beyond a certain number may become less important. For instance, if perceived unfairness arises because a student did not receive their 7th preference, it might be deemed insignificant. To address this, we could introduce a hyperparameter that allows the FYP coordinator to decide how many preferences to consider.

This concept of checking for perceived fairness will be elaborated in the implementation section 5.5.1. After implementation, we will be able to identify cases of perceived unfairness, but we also need a method to quantify it.

4.4.3 Cases of Perceived Unfairness

To quantify perceived unfairness, we first need to identify all possible cases and then create a metric that incorporates each of them. The cases, along with explanations, are shown in Table 4.6 below.

Cases of Perceived Unfairness	Comments
Case 1 : The student and project were allocated with lower preferences than the edge connecting both	This perceived unfairness can be viewed from two perspectives: the staff's perspective, where the project was allocated to a lower preference, requiring us to compare the allocated project cost with the cost connecting both; and the student's perspective, where we compare the student's allocated cost with the cost connecting both.
Case 2: The student was not allocated a project, and the project was allocated to a lower preference than the edge connecting both	The student was not allocated so we could argue it is worse than case 1.
Case 3: The project was not allocated, but the student was allocated to a lower preference than the edge connecting both	This usually indicates that the project could not be allocated due to supervisor loading limits. To determine if this is truly perceived unfairness, we must examine the projects supervised by the staff member that were allocated and compare the preferences.
Case 4: Both the project and the student were not allocated	This is typically related to supervisor loading limits. We need to examine the allocated projects of the supervisor to determine if it is truly perceived unfairness. However, if it is, it is worse than the previous case since it caused the student to remain unallocated.

Table 4.6: Cases of perceived Unfairness with some explanations

For cases 3 and 4, which are related to supervisor loading, determining perceived unfairness is more complex. We decided to assess perceived unfairness by comparing the potential cost of allocation with the average staff allocation cost. Specifically, if the staff's average allocated preferences are lower than the preference for the project in question, it would be considered unfair. Conversely, if the average allocated preferences are higher, it would not be considered unfair.

4.4.4 Quantifying Perceived Unfairness

Having identified all relevant cases, we need a method to rank each flagged index. We can create a metric to assign a severity score to each flagged index. The perceived unfairness score of a complete allocation can be determined by summing the scores of all flagged indices. This allows us to compare the scores of different allocations, providing opportunities to design improvements for perceived unfairness.

First, we need to define severity to determine the basis for our scoring. From the table of perceived unfairness cases (Table 4.6), it is evident that a larger difference between the student's allocated preference and the flagged index's preference indicates a more severe case. For example, a case where a student is allocated their 7th preference while their first preference is allocated to another student as a 5th preference is more severe than a case where a student is allocated their 2nd preference while their first preference is allocated to another student, regardless of the allocation preference of the latter. This severity is assessed from the student's perspective. In Case 2, which is considered worse than Case 1 for the student (as described in Table 4.6), the associated cost should be higher than the maximum possible difference. For Cases 3 and 4, if determined to be unfair as discussed earlier, we can calculate the score similarly to Cases 1 and 2. This is discussed in more detail when implementing it in 5.5.1.

Alternatively, we could consider the staff's perspective. Staff members usually prefer students who rank their projects higher, as they want students genuinely interested in their projects. This

preference can be translated into a metric called the staff's average allocation cost. By examining all projects supervised by a staff member and calculating the average allocation cost, this metric can provide insights to the FYP coordinator about the overall situation. Although staff preferences are typically consistent and a low average allocation is often due to students not ranking their projects highly, it is still valuable to see the average allocation cost per staff member. This information should be shown to the FYP coordinator after the allocation is complete.

Case	Perceived Fairness score
Case 1	$C_{StudentAssigned} - C_{flaggedIndex}$
Case 2	$Preference\ Cutoff + 1$
Case 3	$C_{StudentAssigned} - C_{flaggedIndex}$
Case 4	$Preference\ Cutoff + 1$

Table 4.7: Perceived Unfairness Cases and their cost calculations

4.4.5 Algorithm design of enhancement

Now that we can quantify perceived unfairness by assigning it a score, we can design an algorithm to enhance the foundation using this metric. The summary of the score calculation is provided in Table 4.7. While designing this enhancement, it is crucial to retain the positives of the foundation, such as the addressed requirements.

Given the metric, the simplest approach would be iterative, following Occam's Razor as discussed in section 2.5.1. Specifically, we can apply the Iterated Local Search (ILS) algorithm [40], which seeks better solutions by slightly modifying the current best solution. This method fits well as the metric can serve as the acceptance criterion for solutions. However, it is not enough to only improve perceived fairness; the solution must also consider other requirements embedded in the cost function of our optimization algorithm (Hungarian or LAPJV). Thus, the acceptance criterion should combine the objective of the optimisation algorithm and the perceived fairness score. The steps of the algorithm are as follows:

- **Step 1:** Start with the allocation given by the chosen optimization algorithm.
- **Step 2:** Review the allocation and flag cases of perceived unfairness.
- **Step 3:** Calculate the perceived fairness score for each flagged index and sort them in descending order.
- **Step 4:** For each index in the sorted list (from most severe to least severe), remove the corresponding row and column, excluding the student and project from the allocation. Rerun the optimization algorithm to find a new solution. Calculate the perceived fairness score for the new allocation and combine it with the total cost. Compare this combined score to the current best score. If it is superior, accept this solution and continue the search. Proceed with the new allocation and update the flagged indices accordingly (Go to step 2).
- **Step 5 :** Stop when the set number of iterations is reached or when there are no more indices to check.

Combining the two metrics correctly is crucial, as discussed in section 2.5.2. To reach a Pareto optimal solution, both objectives need to be convex. The optimization algorithm solves an LSAP, which is linear and therefore convex [41]. However, while the operations for the perceived fairness metric are linear (as it involves a difference of preferences), the metric itself is not convex in determining perceived fairness cases. This does not preclude combining the metrics; it simply makes achieving a Pareto optimal solution more complex. In our case, we aim for a sufficiently good solution, and achieving a Pareto optimal solution, though desirable, is not necessary. Combining

the two metrics correctly is crucial, as discussed in section 2.5.2. To reach a Pareto optimal solution, both objectives need to be convex. The optimization algorithm solves an LSAP, which is linear and therefore convex [41]. However, while the operations for the perceived fairness metric are linear (as it involves a difference of preferences), the metric itself is not convex in determining perceived fairness cases. This does not preclude combining the metrics; it simply makes achieving a Pareto optimal solution more complex. In our case, we aim for a sufficiently good solution, and achieving a Pareto optimal solution, though desirable, is not necessary. We will use a sum formulation to combine the metrics as shown below:

$$S = S_{TC} + w_{ILS} S_{PF} \tag{4.3}$$

where:

S = Combined ILS metric
S_{TC} = Total cost of the allocation output by the chosen optimization algorithm (LAPJV or Hungarian)
S_{PF} = Perceived fairness score of the allocation, calculated by adding the scores of perceived fairness flagged cases
w_{ILS} = Weight given to the perceived fairness metric to scale it relative to the total cost metric (hyperparameter)

There are a few considerations to keep in mind. Firstly, for the total cost metric (S_{TC}), we could consider excluding students being allocated their first preference, as this cost is already minimized. Including it skews the cost metric since getting a first preference is optimal. Additionally, most allocations see the majority of students receiving their top preference, which can distort the cost metric.

The second consideration involves the weight hyperparameter. By default, it is set to 1, following Occam's razor for simplicity. This can also be justified by considering that one student allocated a low preference is equivalent to one case of perceived fairness. Determining these weights effectively answers the question: How many students being allocated low preferences are equivalent to a case of perceived unfairness? This hyperparameter can be set by the FYP coordinator.

The third consideration is unallocated students. They could be integrated into the total cost metric (S_{TC}); however, if the unallocation cost is very large (as it is by default), it might completely dominate the metric. To address this, we could introduce a hyperparameter that allows the FYP coordinator finer control over the number of unallocated students within the ILS. This would enable the coordinator to decide how many additional unallocated students the ILS is allowed to allocate, thus eliminating the need to consider unallocated students in the total cost metric. Alternatively, if the coordinator prefers not to use this method, we could provide a boolean option to switch between the two approaches. This is discussed further in 5.5.4 since it was discovered after testing.

It's crucial to understand that our algorithm utilizes a local search strategy, which inherently risks becoming trapped in a local minimum. To mitigate this risk, techniques such as perturbations are typically employed. In our approach, however, we initiate the search from an optimal cost solution—an integral part of our ILS metric. This starting point generally increases the likelihood of progressing towards a superior solution within the same number of iterations compared to beginning at a random solution. Nonetheless, this remains a hypothesis that will be rigorously tested in the evaluation section of our study 6.3.The algorithm is designed to allow for changes within the neighborhood once a better solution is identified, ensuring ensuring it is not completely restricted but tailored to our needs. Furthermore, our implementation of ILS is guided; it strategically navigates through indices based on their rankings. We hypothesize that addressing the indices with the highest perceived fairness score first will more likely lead to an improved solution. This assumption will also be verified in the evaluation section 6.3.3.

This algorithm was inspired by the top-down multi-allocation approach 4.3.1 discussed in the algorithm design section, as it was the only approach addressing perceived fairness. Both methods exclude preset matches from the allocation and rerun the optimization algorithm. However, our approach is less radical, allowing us to address all requirements at least partially, without focusing solely on perceived fairness. The initial allocation results from a one-allocation algorithm, which covers the no low preference 3.4.2 and full allocation 3.4.3 requirements. The ILS then incorporates perceived fairness by merging both objective functions, creating a metric that balances all three requirements and finds solutions in the middle ground.

4.4.6 Concluding remarks on Addressing Perceived Fairness

In conclusion, we explored how others have tackled perceived fairness and related it to our more complex and ambiguous case. We designed a function to flag perceived unfairness given a complete allocation, allowing us to visualize the problem and create a solution. We categorized flagged indices into cases (Table 4.6) to fully understand and quantify them. This led to a scoring system to rank flagged indices, detailed in Table 4.7. Combining this score with the cost of the allocation, we created an ILS algorithm to extend the initial foundation, addressing all requirements and finding a solution that considers the trade-offs in the middle ground.

4.5 Hyperparameters

The allocation process can be optimized by adjusting various hyperparameters, which enable the FYP Coordinator to tailor the student-to-project assignment process. Most of these have already been discussed; however, a comprehensive table with the full list and definitions is provided below (see Table 4.8). By fine-tuning these hyperparameters, the FYP Coordinator can shape the algorithm's behavior to achieve specific allocation goals and ensure that the process aligns with the desired educational and administrative outcomes.

Hyperparameter	Explanation	Referenced Section
Unallocation Cost ($C_{\text{Unallocated}}$)	Adjusts the cost of leaving a student unallocated. A lower unallocation cost implies a smaller penalty for having unallocated students, whereas a higher value increases the penalty.	4.2.2
Preference Multiplier (w_{student})	Adjusts the weight of student preferences in the cost calculation.	4.2.1
Suitability Multiplier (w_{staff})	Adjusts the weight of staff suitability rankings in the cost calculation.	4.2.1
Suitability Rankings	Defines the cost associated with each suitability ranking.	4.1
Preference Cutoff	Discards preferences beyond a certain number to minimize the likelihood of students who submit more than the required number of preferences gaining an unfair advantage.	4.3.1
Required Preferences	The minimum number of preferences required from each student.	4.2.1
Max Preference Multiplier (w_{Max})	The largest weight, indicating the maximum penalty a student could face for not submitting the required number of preferences.	4.2.5
Min Preference Multiplier (w_{Min})	The smallest weight, indicating the minimum penalty a student could face for not submitting the required number of preferences.	4.2.5
Weight of the perceived fairness metric (w_{ILS})	The weight assigned to the perceived fairness metric when combining it with the total cost to obtain the combined ILS metric. This acts as a scalar to ensure both metrics are on the same scale or to allow one to have more influence than the other.	4.4.5
Ignore number of unallocated students	Boolean that allows the FYP coordinator to decide whether to set the additional number of unallocated students allowed by the ILS. If true, the coordinator does not set this number.	4.4.5
Number of unallocated students	If the previous hyperparameter is set to false, the FYP coordinator can set the additional number of unallocated students allowed by the ILS for finer control.	4.4.5
Number of preferences checked by the ILS	Sets the cost threshold beyond which the ILS algorithm will not check for perceived unfairness.	4.4.2

Table 4.8: Table of hyperparameters and their definitions

The default values of these hyperparameters are discussed in the implementation section 5.2.1.

4.6 Conclusion

In conclusion, we have defined the major adaptations required for the preprocessing part of the code and summarized them in Table 4.2. The preprocessing code will output an adjacency matrix containing the costs of each combination, with the calculation of these costs shown in Table 4.1. We explored multiple strategies for designing the optimization algorithm and decided to use a one-allocation algorithm as a foundation. However, since the perceived fairness requirement was initially ignored, we analyzed, visualized, and quantified it to create an algorithm to address all requirements, as discussed in 4.4.6. Finally, we summarized all hyperparameters needed to allow the FYP coordinator to control the allocation process, as shown in Table 4.8.

5

5 Implementation

Contents

5.1 Introduction

This chapter is dedicated to bringing to life everything we discussed in the design section. It details the technical implementation of the preprocessing, optimization algorithm, and ILS algorithm. We will include testing results to verify their functionality and benchmarking results to ensure they run within the intended timeframe. Additionally, we will discuss the challenges faced during

implementation and how they were overcome. This comprehensive overview will provide insights into the practical aspects of our project, highlighting both successes and difficulties encountered along the way.

5.2 Shared Modules Overview

First, we will discuss the universal modules that are utilized by all parts of the code.

5.2.1 Hyperparameter Module

The hyperparameter module contains all the hyperparameters that can be modified by the FYP coordinator. A comprehensive table of these hyperparameters and their definitions was presented in the design chapter 4.8. In this section, we will discuss the default settings for each parameter and provide justifications for these values.

- **Unallocation Cost** (`UNALLOCATION_COST`): Set to 100, reflecting the priority of achieving a full allocation, as this is significantly higher than any allocated costs. Prioritising a full allocation is extremely important and is one of our primary requirements 3.4.3.
- **Student Suitability Multiplier** & **Staff Preference Multiplier** (`SUITABILITY_MULTIPLIER` & `PREFERENCE_MULTIPLIER`): Both are set to one to balance the weight equally between staff and student preferences as discussed in 4.2.1.
- **Preference Cutoff** (`PREFERENCE_CUTOFF`): Established at 6 to align with the preferences set for the 2024 MSc allocation.
- **Required Preferences** (`REQUIRED_PREFS`): Also set to 6, reflecting the 2024 MSc allocation guidelines.
- **Minimum Preference Multiplier** (`MIN_PREFERENCE_MULTIPLIER_PENALTY`): Set at 1.5, sufficiently above 1 to ensure it penalizes students, yet not excessively high as the minimum multiplier is applied when a student has submitted one fewer preference than required.
- **Maximum Preference Multiplier** (`MAX_PREFERENCE_MULTIPLIER_PENALTY`): Set at 4.0 to be significantly higher than the minimum multiplier, reflecting the severity of submitting only one preference, which is considered highly unfair.
- **Weight of perceived fairness metric** (`ILS_WEIGHT`): Set to 1, as discussed in section 4.4.5, because it is the simplest approach. This setting means we equate one student allocated to a low preference with one case of perceived unfairness.
- **Ignore number of unallocated students** (`IGNORE_NUMBER_OF_UNALLOCATED_STUDENTS`): Set to false since the algorithm performs better when the number of unallocated students does not dominate the metric, as discussed in 4.4.5 especially since the `ILS_WEIGHT` is set to 1.
- **Number of unallocated students** (`NUMBER_OF_UNALLOCATED_STUDENTS`): Set to 0 to strive for full allocation and meet the full allocation requirement 3.4.3. However, if many low preferences are observed, this value should be adjusted.
- **Number of preferences checked by the ILS** (`ILS_PREFERENCE_CUTOFF`): Set to 5 as an arbitrary number. After discussing with my peers, we concluded that 5 would be a good default value since it seemed to be the average point after which students' preferences become less significant.

The default settings for suitability rankings are 0 for ”Definite” and 2 for ”Maybe,” indicating that a ”Maybe” ranking effectively increases the preference value by 2. It is important to note that these values are adjustable to best fit the specific needs of the allocation process in any given year.

5.2.2 Common Types Module

This module encompasses all the record definitions essential for the codebase. Records were chosen for their ability to organize data efficiently and enhance code readability. Below is a detailed breakdown of the records and their components.

- `Student`
  - `studentId : string` Typically, the student ID is their EEID. This record helps differentiate between IDs, as they are all strings.
- `Project`
  - `projectId : string`

  Project IDs are assigned to track each project individually. This record also helps differentiate between various IDs since they are formatted as strings.
- `Staff`
  - `staffId : string`

  Similar to students, the staff ID is generally their EEID, enabling distinction between different IDs.
- `MatrixEntry`
  - `studIndex : int`
  - `projIndex : int`

  Identifies a specific entry in the cost matrix, where `studIndex` is the row index and `projIndex` is the column index.
- `FlagIndexScore`
  - `Index : matrixEntry`
  - `score : float`

  Represents an index where perceived unfairness has occurred. The score quantifies the perceived fairness for this particular index, as discussed in 4.4.4.
- `PerceivedFairness`
  - `studentAssigned : list<flagIndexScore>`
  - `studentUnassigned : list<flagIndexScore>`
  - `supLimitAssigned : list<flagIndexScore>`
  - `supLimitUnassigned : list<flagIndexScore>`

  These lists categorize flagged indices into four cases of perceived fairness as detailed in Table 4.6.

- `AssignmentMaps`
  - `studAssignmentM : Map<Student, Project*float>`
  - `projAssignmentM : Map<Project, Student*float>`

  Contains maps of assignments where the key is the student or the project, and the value is the assigned project or student with the associated cost.
- `StaffMaps`
  - `projStaffM : Map<Project, Staff>`
  - `staffAverageM : Map<Staff, float>`

  Includes maps related to staff allocations, where the first map links projects to supervising staff members and the second map tracks the average allocation cost per staff member.
- `Assignment`
  - `Student : Student`
  - `Project : Project`
  - `Cost : float`

  Represents a match where a student is allocated to a project at a specified cost.
- `IdLists`
  - `studIds : Student list`
  - `projIds : Project list`

  Contains lists of student and project IDs corresponding to the rows and columns of the cost matrix, respectively.

5.3 Preprocessing

The preprocessing module encompasses all functions necessary to convert the data provided by the FYP Coordinator, from text file format into a comprehensive cost matrix. This transformation incorporates all relevant data for subsequent processing.

5.3.1 Text Files

The initial step involves reading various text files, parsing them, and organizing the data into appropriate structures. Below, we describe each of the provided text files and outline the information they contain:

- **projects.txt**: Comprising three columns, this file includes the *projectId*, *staffId* (the project supervisor), and a column for *studentId*. If the *studentId* column is not empty, it indicates that the project has already been allocated, typically to a self-proposed project.
- **exported-stu-prefs.txt**: This file documents student preferences. It features three columns: *studentId*, *projectId*, and the preference ranking that the student has assigned to the project.
- **exported-staff-prefs.txt**: Contains staff suitability rankings in a three-column format: *studentId*, *staffId*, and the ranking.

- **suplimits.txt**: Details the supervisory load limits, presented in two columns: one for *staffId* and another for the corresponding load limit.

Now that we know what's in the files let's dive into how we read the files and how we store the data. The helper functions used to do these tasks are the following:

- `getPathTofile` : Given the path to the file directory and the file name, it combines them to return the full path.
- `readFile` : Reads all lines of a text file and splits it by tabs. It returns the data in a string array array containing all lines.
- `createIdList` : Takes an array and returns a list of distinct elements. (removes duplicates)
- `createMapOfArrays` : Takes in two arrays, zips them and creates a map where the fist array contents are the keys and the second are the values.
- `createEeIdProjectsMap` : Takes two arrays one of eeId and one of projectIds and creates a map where the eeIds are the keys and the projects are the values. The returned map is of type Id,list<projectId> since a student has multiple preferences or a member of staff supervises multiple projects and so it combines all projectIds with the same eeId.

5.3.2 Data Organization

Using helper functions, we now establish the following variables to store and organize the data extracted from the text files:

- `staffProjectsM: Map<Staff, List<Project>` A map where the key is the `staffId` and the value is a list of all projects they supervise. This helps easily check the number and projectIds of projects under a specific staff member.
- `listId: IdLists` This includes lists of unique `studentId`s and `projectId`s. These lists are crucial as they serve as the blueprint for the cost matrix, where the `studentId` represents the rows and the `projectId` represents the columns. For instance, the first student in the `studentId` list corresponds to the first row of the cost matrix, and similarly, the first project in the `projectId` list corresponds to the first column.
- `studentsProjectM: Map<Student, List<Project>` A map where the key is the `studentId` and the value is a list of all projects they have listed as preferences. This mapping is essential for tracking the number and projectIds of student preferences.
- `projStafftM: Map<Project, Staff>` A map where the key is the `projectId` and the value is the staff member supervising the project. This facilitates easy access to the supervisor details for each project.
- `studentPreferences: Student array * Project array * string array` This structure maps each student's preferences, where each index corresponds to a preference set. For example, the first element in the student array chose the first project in the project array, with the preference level specified in the string array.
- `staffRankings: Student array * Project array * string array` Similar to preferences, each index here represents a staff's suitability ranking for a student and a project, linking each student in the student array to a project in the project array with a ranking specified in the string array.

- `staffLimits: Staff array * string array`: This holds the load limits for each staff, where the `staffId` from the staff array corresponds to their respective limit in the string array, outlining how many projects a staff member can supervise.

These structures are designed to systematically manage the vast array of data needed for efficient preprocessing and subsequent operations within the cost matrix framework.

5.3.3 Creating the Cost Matrix

This section details the functions involved in manipulating data to construct a cost matrix, which is then utilized by the optimization algorithm. Below is a description of these functions:

- `getSelfAllocated`: Returns a dictionary of type <Student, Project> containing all self-proposed projects. It identifies these by checking the `studentPreferences` against the `projectId` list from `idLists`. A project not listed in `projectId` is considered pre-allocated.
- `removeSelfAllocated`: Removes all projects and students already allocated from `idLists`.
- `addStudentPreferences`: Processes `studentPreferences` to add preference scores to the matrix. The scores are derived using the `penaliseStudents` function, which calculates penalties as detailed in 4.2.5.
- `penaliseStudents`: Determines the student preference score. It assesses the need for penalties based on the number of preferences each student has selected using `studentsProjectM` and applies a calculation that results in either a `PreferenceValue` or its scaled version if penalized, multiplied by `PREFERENCE_MULTIPLIER`.
- `calculateMultiplier`: Calculates the multiplier for penalties using the formula specified in 4.2.
- `addStaffPreferences`: Integrates staff preferences into the cost matrix by converting `staffRankings` into costs, scaling these by `SUITABILITY_MULTIPLIER`, and placing them appropriately in the matrix.
- `insertUnallocationCost`: Substitutes all zeros in the cost matrix with `UNALLOCATION_COST`, implying that any unpreferred student-project pairing incurs a default unallocation cost (see 4.2.2).
- `limitSupervisorLoading`: Adjusts for supervisor overloading by processing `staffLimits` and using the function `CheckSupervisorLimit` to compare against the project counts from `staffProjectsM`. Exceeding supervisors incur dummy student rows in the matrix, as described in 4.2.4. These dummy entries have zero cost for supervised projects and the unallocation cost for others.
- `SquareCostMatrix`: Ensures the matrix is square by adding necessary rows or columns with zero cost to balance the dimensions, updating the `studentId` or `projectId` lists accordingly (see 4.2.3).

5.3.4 Top Level Preprocessing Function

The overarching preprocessing function, `getCostMatrix`, orchestrates the transformation from text files to a fully prepared square cost matrix for the allocation algorithm. The process unfolds as follows:

- Initializes data structures described in 5.3.2 from the input text files by calling function `getData`.
- Retrieves and removes self-allocated students from the lists using `getSelfAllocated` and `removeSelfAllocated`.
- Constructs an initial 2D array for the cost matrix, sized according to `studentId` and `projectId`, initialized to zeros. Arrays are chosen for their mutability, facilitating subsequent updates.
- Populates the matrix with student preferences via `addStudentPreferences` and staff preferences via `addStaffPreferences`.
- Integrates unallocation costs for non-preferred student-project combinations using `insertUnallocationCost`.
- Manages supervisor loading limits by introducing dummy students with `limitSupervisorLoading`.
- Balances the matrix dimensions with `SquareCostMatrix` to ensure it is square.

5.4 Optimization Allocation Algorithm

The optimization algorithm processes the cost matrix to produce the assignments. As discussed in the background section, two algorithms appeared feasible based on our review of the literature: the LAPJV algorithm and the Hungarian algorithm (see 2.4.2 and 2.4.2). Both algorithms will be implemented, tested, and benchmarked to determine the fastest one for our purposes.

5.4.1 Hungarian Algorithm

For the F# implementation of the Hungarian Algorithm, I referenced existing code in other languages to guide both coding and subsequent testing. Initially, I adapted a Python implementation found online [42], which was thoroughly explained and served as a useful blueprint. For this specific implementation, I opted to use lists instead of arrays since a lot of the matrix operations are on subsequent elements. This meant the code was written in an immutable style, employing recursion to replace the commonly used for loops. Focusing on a cost matrix representation, the implementation includes several crucial functions:

- `transformToBoolean`: Converts a float matrix into a boolean matrix, marking only the zero elements as true, and others as false.
- `markZeroes`: Identifies and returns the indices of the marked zeroes (true values) in the matrix.
- `markMatrix`: Determines the minimum number of rows and columns needed to cover all zeroes in the matrix, also returning the indices of these zeroes.
- `adjustMatrix`: Modifies the matrix by decreasing the unmarked elements by the minimum of these elements and increasing the intersecting elements of marked rows and columns by this minimum value.

- `HungarianAlgorithm` : This top-level function prepares the matrix by subtracting each row by its minimum element and does the same for columns. It then utilizes the other functions to ultimately return the indices of the zeroes, representing the assignments.
- `getAnswer` : Calculates the total cost of the assignment based on the zero indices (the matchings) and constructs a final matrix displaying these matchings.

Testing revealed limitations in this implementation, particularly for larger test cases. It was found to be incapable of consistently returning optimal solutions for these scenarios, a challenge that will be further explored in the testing section 5.4.1.

The subsequent implementation of the Hungarian Algorithm, which proved more robust, was adapted from a widely used C++ version [43]. For this implementation, arrays were utilized due to their suitability for our specific needs, as using immutable structures was found to be less efficient due to the necessity of recreating instances after each modification. Mutable variables were used sparingly and only in contexts where their usage was safe. The key functions of this implementation are as follows:

- `step1` : Subtracts the smallest element in each row and then each column from all elements in that respective row or column.
- `step2` : Identifies zeros in the matrix and stars them if they are not in a starred column or row, covering those rows and columns.
- `step3` : Covers columns containing starred zeros. If all columns are covered, the solution is found; otherwise, it proceeds to the next step.
- `step4` : Finds and primes an uncovered zero, making further adjustments if a starred zero is in the same row. It repeats until there are no more uncovered zeros, then moves to step 6.
- `step5` : Constructs a series of alternating primed and starred zeros and augments the path, returning to step 3.
- `step6` : Adjusts the matrix based on the smallest uncovered value and returns to step 4 for further processing.
- `output_solution` : Calculates the optimal assignment cost based on the final masked matrix.
- `hungarian` : The main function that orchestrates the steps and computes the optimal assignment.

Testing

Testing is a critical component of ensuring that our code not only functions as intended but also adheres to the expected standards of accuracy and efficiency. This was particularly evident in our project, where the initial implementation, though seemingly correct, was found to be flawed upon rigorous testing.

To effectively evaluate the algorithms, I created a diverse set of test cases, varying both in size and sparsity, to assess whether the code could consistently produce the optimal solution. Each function within the algorithm was also tested individually to verify its correct operation. Additionally, it was crucial to compare our results not only with the blueprint implementation but also with other implementations to identify any potential errors in the original code.

During the testing phase, discrepancies were noted in the outcomes of the first implementation. Specifically, the cost figures reported were lower than expected, which raised concerns about

the accuracy of the calculation. Upon closer inspection, especially in the matching results, it became apparent that certain matches were missing. This led to the identification of a critical flaw in the `markMatrix` function, which failed to cover the zeroes accurately and used more rows and columns than necessary. The realization that the online reference implementation also produced incorrect results confirmed these suspicions. Further scrutiny of the individual functions revealed additional discrepancies. A notable issue was identified in the `markZeroes` function. This function's selection criterion for zeroes was found to be inadequate, as it focused solely on the first zero found in rows with the minimum number of zeroes, without considering the distribution of zeroes in the corresponding columns. The ideal approach should involve selecting zeroes based on both their row and column distributions. By neglecting this dual consideration, the algorithm risked choosing suboptimal matches, leading to the observed discrepancies and lower than expected cost figures.

Subsequently, extensive testing was carried out on the second implementation. This version successfully passed all test cases, confirming the robustness and accuracy of our F# implementation of the Hungarian Algorithm.

5.4.2 Jonker-Volgenant Algorithm (LAPJV)

For the F# implementation of the Jonker-Volgenant Algorithm, I adapted a reference C implementation [44]. This algorithm, which uses the graph representation of the assignment problem, lends itself well to an implementation using mutable variables and arrays. The key functions and steps, as extracted from the provided C code, are outlined as follows:

- `lap`: This is the main function that orchestrates the Jonker-Volgenant algorithm, managing the assignment process, and updating dual variables. It returns the cost of the assignment and the matching.
- **Column Reduction**: Step in which we find the minimum cost in each column, adjusts column values, and makes preliminary assignments.
- **Reduction Transfer**: Step in which we transfer reduction from rows with single assignments to other rows, aiding in creating more zeros.
- **Augmenting Row Reduction**: Step in which we iteratively scan free rows for further reduction and assignment, refining the solution.
- **Augment Solution**: Step in which we augment the solution by finding the shortest augmenting path for unassigned rows, effectively improving the assignment.
- **Modify Costs**: Step in which we adjust the cost matrix based on the uncovered elements, crucial for finding the smallest uncovered value in the matrix.

Testing

Testing the implementation of the LAPJV Algorithm was streamlined due to the availability of a functional Hungarian Algorithm implementation in F#. Given that both algorithms aim to return the optimal assignment for the same problem, I could directly compare their outputs to validate correctness. The primary difference between the two lies in their respective methodologies and performance efficiencies.

In conducting the tests, I subjected the LAPJV implementation to a variety of test cases, ensuring a comprehensive assessment of its reliability across different scenarios. The algorithm proved robust, consistently outputting the correct optimal solution for all test cases presented.

5.4.3 Comparing Algorithms

After verifying the accuracy of both the LAPJV and Hungarian algorithm implementations in F#, a comparative analysis was conducted to evaluate their performance. This comparison, focusing on computational efficiency and scalability, is essential for determining the most suitable algorithm for our application.

To facilitate this comparison, I utilized the BenchmarkDotNet library [45], which efficiently benchmarks code by running it multiple times. This library not only generates histograms of running times but also compiles a detailed table that includes the average running time, standard deviation, and error for each test. To generate the diverse range of test cases required for this comprehensive analysis, I developed a function named `generateSparseMatrix`. This function, detailed in Figure 5.1 below, is specifically designed to create test matrices of varying sizes, densities, and cost ranges. Testing both algorithms across these various parameters was crucial to understanding their performance under different conditions. The collected data is summarized in Table 5.1.


```
///<summary>Generates a random sparse matrix</summary>
/// <param name="size">The size of the matrix</param>
/// <param name="density">The density of the matrix between 0 and 1</param>
/// <param name="maxCost">The maximum cost of an edge</param>
0 References
let generateSparseMatrix (size: int) (density: float) (maxCost: int) : list<list<int>> = // int -> float -> int -> list<list<int>>
    List.init length = size (fun _ ->
        List.init length = size (fun _ ->
            if rand.NextDouble() < density then
                rand.Next(minValue = 1, maxValue = maxCost + 1)
            else
                0))
```

Figure 5.1: Screenshot of function that generates random test cases

Density	Cost Range	Size	Hungarian			LAPJV		
			Mean	Std Dev	Error	Mean	Std Dev	Error
5%	1-100	100x100	5.261 ms	1.027 ms	0.3484 ms	5.789 ms	1.027 ms	0.3484 ms
		200x200	45.02 ms	5.946 ms	2.061 ms	43.96 ms	8.137 ms	2.760 ms
		400x400	424.3 ms	96.49 ms	32.73 ms	493.0 ms	135.6 ms	47.27 ms
	1-1000	100x100	5.619 ms	0.8105 ms	0.2764 ms	5.839 ms	1.065 ms	0.3614 ms
		200x200	40.98 ms	7.190 ms	2.438 ms	53.86 ms	8.704 ms	3.000 ms
		400x400	382.0 ms	62.97 ms	21.36 ms	517.4 ms	107.0 ms	36.66 ms
20%	1-100	100x100	4.977 ms	0.6303 ms	0.2161 ms	6.838 ms	0.9564 ms	0.3244 ms
		200x200	45.03 ms	7.819 ms	2.652 ms	52.84 ms	8.530 ms	2.956 ms
		400x400	374.0 ms	47.58 ms	16.14 ms	497.0 ms	127.2 ms	43.15 ms
	1-1000	100x100	4.886 ms	0.7816 ms	0.2665 ms	6.954 ms	1.357 ms	0.4601 ms
		200x200	43.13 ms	4.597 ms	1.559 ms	51.89 ms	5.908 ms	2.004 ms
		400x400	328.5 ms	46.25 ms	16.21 ms	453.3 ms	87.08 ms	29.85 ms
50%	1-100	100x100	73.66 ms	11.11 ms	3.767 ms	5.892 ms	1.354 ms	0.4642 ms
		200x200	1.478 s	0.0790 s	0.2266 s	47.75 ms	6.193 ms	2.135 ms
		400x400	22.12 s	2.014 s	0.718 s	410.9 ms	26.31 ms	26.31 ms
	1-1000	100x100	6.887 ms	0.8709 ms	0.2970 ms	7.841 ms	0.7639 ms	0.2633 ms
		200x200	1.765 s	0.1524 s	0.0528 s	54.13 ms	4.728 ms	1.657 ms
		400x400	24.74 s	0.885 s	0.438 s	434.8 ms	29.36 ms	10.12 ms

Table 5.1: Comparison of Hungarian and LAPJV Algorithms

From Table 5.1, it is evident that for both algorithms, the running time escalates as the size of the cost matrix increases. At lower densities (5% and 20%), the algorithms perform comparably,

with the Hungarian algorithm showing a slight edge in most cases. However, a significant divergence in performance is observed with denser matrices (50%). The Hungarian algorithm's running time becomes considerably variable, reaching up to 24 seconds for a 400x400 matrix, whereas the LAPJV algorithm maintains a much faster runtime, peaking at around 434 ms for the same size. This discrepancy is attributed to the LAPJV algorithm's optimizations for dense matrices, allowing it to adapt more effectively than the Hungarian algorithm, which lacks such improvements. Nevertheless, it is noteworthy that a 24-second runtime, while longer, is still relatively efficient.

In conclusion, the LAPJV algorithm demonstrates superior performance, particularly with dense matrices, and proves to be a more reliable choice. However, the Hungarian algorithm still offers competitive performance, especially in less dense scenarios. For this project, I will primarily employ the LAPJV algorithm due to its robustness, while also retaining the Hungarian algorithm implementation. This approach allows users the flexibility to choose between the two, depending on their specific requirements.

5

5.4.4 Data Structures Used

Reflecting on our initial discussion about optimal data structures, which began in the background section 2.3.1, we originally opted to utilize a list structure for the cost matrix due to its efficient sequential index access.

However, subsequent testing with the LAPJV algorithm revealed significant slowdowns in specific scenarios. These scenarios shared similar densities and only minor variations, making the root cause of the slowdowns difficult to pinpoint. Initially, I suspected the issue might stem from the algorithm itself—perhaps a minor change was undermining a critical aspect of the algorithm, thereby escalating the problem's complexity. However, upon reviewing the code, I realized that the performance issue was due to random accesses on lists, which have a time complexity of O(n) and are notoriously slow.

After converting the data structure from lists to arrays, the performance discrepancies disappeared. The initial hypothesis was not entirely off the mark—a slight modification indeed had a disproportionately large impact on the problem's difficulty. However, the LAPJV algorithm, which is typically robust for dense problems, should not have been significantly affected by these changes. It turned out that the minor modification had led to an increased need for random index accesses within the list, which was the primary cause of the slowdown.

The empirical data from our benchmarking, which is summarized in Table 5.2, clearly illustrates the performance enhancement achieved by switching to arrays. As shown, the mean execution time for arrays was significantly lower than for lists, with a considerable reduction in both error margin and standard deviation, indicating a more consistent performance.

Ultimately, this experience led us to conclude that using arrays is the most effective approach for our specific requirements. Arrays eliminate the performance degradation associated with random access, confirming that they are the optimal data structure for our implementation of the LAPJV algorithm.

Data structure used	Mean	Error	Standard deviation
List	24.87 s	0.485 s	0.430 s
Array	349.5 ms	23.23 ms	317.6 ms

Figure 5.2: Benchmarking results for the LAPJV algorithm using different data structures

5.5 Iterated Local Search

This section details the F# implementation of the ILS algorithm, which is tailored specifically to address and enhance perceived fairness within the allocation process. As initially described in 4.4.5, the approach begins by developing methods to identify and score potentially unfair indices as established in 4.4.4. This scoring informs the subsequent application of the ILS.

5.5.1 Flagging Perceived Fairness Indices

The process to flag perceived unfairness follows the initial assignments made by the LAPJV algorithm. It utilizes two primary functions: `findAndFlagIndex` and `flagPerceivedFairness`, which systematically assess and record instances of unfairness, as conceptualized in 4.4.2.

Function: findAndFlagIndex

This recursive function critically assesses each assignment for a student within the cost matrix, sorting and checking each potential cost from the most to least preferred:

- The function terminates if it reaches the end of the sorted costs, returning `None`.
- For each cost:
  - If both the project and student are currently assigned, it checks if the actual assignment cost of each is higher than the potential cost. If so, it flags this as potentially unfair.
  - If only the project is assigned, the potential cost is compared to the project's assigned cost. If the potential cost is lower, it is flagged as unfair.
  - If only the student is assigned, the potential cost is compared to the staff's average assigned cost. If it is lower, it is flagged as unfair.
  - If neither the project nor the student is assigned, the potential cost is directly compared to the staff's average.
- The function recurses to evaluate the next cost or terminates with a flagged index based on these assessments.

Function: flagPerceivedFairness

This function coordinates the overall evaluation process and manages the input to `findAndFlagIndex`:

- Iterates through each row of the matrix, representing each student's possible assignments.
- Filters out zero costs (typically associated with 'fake' projects) and ignores 'fake' students, which are used for load balancing.
- Each valid row is processed to sort the potential assignments by cost, subsequently invoking `findAndFlagIndex` to scrutinize each cost for fairness.
- Indices flagged by `findAndFlagIndex` as representing an unfair cost are compiled.
- These flagged indices are collected into a comprehensive list, representing all detected instances of perceived unfairness across the matrix.

Summary: Utilizing the combined efforts of `findAndFlagIndex` and `flagPerceivedFairness`, the assignment matrix is subjected to a comprehensive examination. This ensures that all local scenarios are meticulously assessed, complementing the LAPJV's global optimization approach.

5.5.2 Scoring Flagged Indices

As outlined in 4.4.4, scoring each flagged index is crucial for comparing the severity of cases and assessing the overall fairness of allocations. We have developed a function named `addScoresToIndicesCombined` to calculate and append scores to each index. The procedural steps of this function are detailed below:

- **Index Evaluation:** Each flagged index is processed individually.
- **Scoring for Assigned Students:** If a student was assigned to a project as per the LAPJV algorithm's results, the perceived fairness score for the student is calculated as the difference between the assigned cost and the potential lower cost identified.
- **Scoring for Unassigned Students:** If a student was not assigned, the perceived fairness score is set to `PREFERENCE_CUTOFF` + 1, ensuring that this score is higher than any other assigned case.
- **Ordering of Indices:** Indices are then ordered by their scores in descending order, from the most severe to the least severe cases.

This function effectively translates the criteria from Table 4.1 into operational code, assigning perceived fairness scores.

Following the scoring process, the indices along with their scores can be presented to the FYP coordinator for detailed review. For this purpose, we have implemented another function, `printSortedFlaggedIndices`, which organizes indices into the four predefined categories as listed in Table 4.6. Each category's details and the reasons for perceived unfairness are articulated to provide the FYP coordinator with a comprehensive understanding of each case.

Finally, the overall score for the assignment is computed by summing all individual scores. The function `getTotalPerceivedFairnessScoreCombined` is responsible for this calculation, providing a quantitative measure of the allocation's perceived fairness by summing the scores across all flagged indices.

5.5.3 ILS Main Function

The main function of the ILS, named `ILS_PerceivedFairness`, is a recursive function designed to operate over a user-specified number of iterations. The steps involved in the function are detailed below:

- **Priority Sorting:** The function begins by processing a list of flagged indices sorted by severity of perceived unfairness.
- **Matrix Reduction:** For the current index in the sorted list, the corresponding row and column are removed from the cost matrix. This reduction simulates the exclusion of a particular assignment.
- **Re-assignment:** The LAPJV algorithm is executed on the newly reduced cost matrix to obtain fresh assignments.
- **Fairness Evaluation:** Perceived fairness indices are flagged anew by invoking `flagPerceivedFairness`. Subsequently, perceived fairness scores are recalculated using `addScoresToIndicesCombined` and `getTotalPerceivedFairnessScoreCombined`.
- **Score Comparison and Update:** The new perceived fairness score is combined with the total allocation cost. This combined score is then compared with the previous score:

  - **Score Improvement:** If the new score is better, the reduced cost matrix is adopted for further iterations, and the process continues with the new list of flagged indices. The current index is then considered a 'set match.'
  - **No Improvement:** If there is no improvement, the function continues with the next index using the original matrix.

- **Termination:** The function concludes either after completing the predetermined number of iterations or when there are no more indices to evaluate.

This structured approach enables the ILS to iteratively refine the allocation process by actively seeking to minimize perceived unfairness while maintaining the overall cost effectiveness of the allocation. This balance is achieved by concurrently considering both fairness and cost metrics. The algorithm embodies the strategies discussed in the design section 4.4.5, effectively translating theoretical concepts into practical application.

5.5.4 Modifications Made While Testing

Testing the ILS involved a detailed evaluation of each component within the iteration process, followed by an assessment of the entire iteration cycle. This testing phase often generated new ideas for refining the algorithm, leading to several key modifications outlined below:

Introduction of Small Tolerance

During the testing phase, particularly when flagging indices, it became apparent that a small tolerance, denoted as ϵ, was necessary when comparing costs for equality. This tolerance compensates for the minor, random noise introduced to the costs, a common practice designed to ensure consistency across different optimization algorithms like Hungarian and LAPJV. Implementing this tolerance proved crucial, as even negligible discrepancies could otherwise skew cost comparisons, falsely indicating slight inequalities.

Expanding the Perceived Fairness Evaluation

A significant insight was gained regarding the algorithm's approach to perceived fairness evaluation. Initially, the algorithm only considered the top preference for each student, overlooking all other preferences. This limitation became evident when, despite nearly identical allocation costs, the unfairness score registered as zero, indicating a narrow evaluation scope. To address this, the evaluation process was expanded to evaluate every preference starting from the top to the bottom stopping where we find a case of perceived unfairness. This allows the algorithm to fairly look at all students making sure they are all considered. However, as discussed in 4.4.5 sometimes not all preferences are relevant which is why we added a hyperparameter to limit the preferences checked (`ILS_PREF_CUTOFF`).

Handling Equal Scores

Further testing revealed that many scores were identical, yet the algorithm was initially designed to handle one index at a time. This approach was problematic because while addressing the first index might improve the solution, other indices with equal scores could potentially offer even greater enhancements. To resolve this, I introduced functionality to evaluate all indices sharing the same score simultaneously, then select the solution with the lowest overall score. If no improvement is observed, the algorithm progresses to the next set of indices.

Unallocated Students

The trade-off between the number of unallocated students and the allocation of low preferences has been previously discussed in 3.4.4 and will be further explored in the evaluation section 6.2.2. This trade-off prompted the development of additional functionality, giving more control to the FYP coordinator.

During the testing of how the perceived fairness and cost metrics were normalized and integrated to formulate ILS metric, we encountered discrepancies. Initially, the total allocation cost component of this combined metric only included allocated projects, inadvertently ignoring unallocated students. This oversight sometimes resulted in the metric improving while the number of unallocated students increased—an outcome we aimed to avoid. To address this, we proposed two solutions: The first involved incorporating the unallocated students into the total allocation cost, assigning them a value via the `UNALLOCATION_COST` hyperparameter. While this method successfully integrated the costs of unallocated students into the overall calculation, it lacked the flexibility to control their numbers.

To resolve this, a second solution was implemented, allowing the FYP coordinator to specify an acceptable number of unallocated students using the new `NUMBER_OF_UNALLOCATED_STUDENTS` hyperparameter. This approach allows the ILS to approve solutions that comply with this stipulated limit, thus excluding unallocated students from the average cost metric but still keeping track of their numbers. The choice between these methods is governed by another hyperparameter, `IGNORE_NUMBER_OF_UNALLOCATED_STUDENTS`, which determines the preferred operational mode for the ILS based on the coordinator's decision.

It is crucial to note that the `NUMBER_OF_UNALLOCATED_STUDENTS` parameter is in addition to any unallocated students from the initial allocation provided to the ILS, typically sourced from the LAPJV allocation. Given that the default value for `UNALLOCATION_COST` is set high, the LAPJV algorithm is configured to prioritize allocating as many students as possible. Consequently, the initial allocation delivered to the ILS generally features the minimum feasible number of unallocated students, setting a baseline for further optimization.

5.5.5 Testing the ILS

Following the individual testing and integration of each component of the ILS, the complete system was ready for a comprehensive functional test. This phase involved validating the iterations to ensure that only correct solutions were accepted. To facilitate this, numerous print statements were introduced to monitor score calculations, the exclusion of matrix rows and columns, and the tracking of indices and all passed arguments.

Since each element had been separately tested beforehand, testing the entire system was relatively straightforward. The primary focus was to ensure accurate data transfer between functions and that updates were executed correctly. During this process, we rectified a few minor errors, such as instances where a previous version of the matrix was inadvertently passed along. This error led to incorrect exclusions of rows and columns in subsequent iterations, disrupting the allocation process.

Additional notable adjustments included the correct calculation of total allocation costs. Specifically, it became necessary to include the costs of the excluded pairs—now considered self-allocated—to ensure the total cost was accurate. Furthermore, a new tolerance, ϵ, was added when reassessing indices with the same score to maintain consistency in evaluations, similar to what was discussed in 5.5.4.

5.6 Top Level Function

After assembling all components of the algorithm's design, the next step is to integrate them, output the results, and conduct tests to ensure everything operates as expected. This integration and testing occur within the `main` function, which follows these steps:

- Retrieve the cost matrix by invoking `getCostMatrix`.
- Obtain the initial assignments using the LAPJV algorithm.
- Output the self-proposed assignments.
- Generate perceived fairness data by flagging indices and calculating their scores.
- Apply the ILS to derive new assignments and finalize the set assignments.
- Display the assignment score and perceived fairness score alongside the final assignments.

5.6.1 Testing

With all components integrated, our focus shifted to ensuring the system's correctness and functionality.

Firstly, we validated the preprocessing stage by reviewing the assignments in relation to the costs, student preferences, and staff limits. This involved verifying that:

- No staff member was assigned more students than their limit, adhering to the predefined supervisor limits.
- Every student was accounted for, with each being appropriately assigned, left unallocated, or recognized as self-proposed.

This validation confirms that the preprocessing tasks were successfully executed, demonstrating the accuracy of the cost matrix and the effective implementation of supervisor limits.

Having confirmed the system's correctness, we next assessed the effectiveness of our implementation. Defining what constitutes a "good" allocation involves considering multiple factors. How can we measure and affirm the quality of our allocation process?

5.7 Conclusion

In conclusion, this chapter has detailed the development of a fully functional allocation system. Utilizing text files that contain data on staff, students, and projects, the system is capable of producing a comprehensive list of allocations. We successfully implemented preprocessing mechanisms that not only ensure compliance with staff limits but also uphold principles of real fairness. This preprocessing phase effectively translates the provided data into a square cost matrix representing the allocation costs. The optimization algorithm selected for generating the initial set of allocations was the LAPJV. This choice was based on its proficiency in handling the density of the cost matrix and its demonstrable speed, which proved superior to that of the Hungarian Algorithm. Finally, the ILS algorithm was incorporated to address perceived fairness, ensuring that the final solution meets all specified requirements comprehensively.

6 Evaluation

6

Contents

6.1 Introduction

In this chapter, we undertake a detailed analysis of our solutions, focusing on how effectively they fulfill the established requirements. Since these requirements often involve inherent compromises, it is essential to understand and manage the trade-offs that arise in practical scenarios. Our goal is to not only achieve a balanced solution but also to clearly demonstrate where and how this balance is achieved. This analysis will address several critical questions: Is our solution effective? Does it meet all the requirements? How are trade-offs manifested within the solution?

For our graphs and analysis, we will use the 2024 MSc project allocation as a case study. This initial allocation was challenging but was made manageable by introducing some adjustments and additional projects, resulting in a more straightforward allocation. To rigorously analyze trade-offs,

we will simulate a difficult scenario by reducing supervisor limits, thereby creating a challenging allocation to study in greater depth.

6.2 Evaluating the LAPJV

The LAPJV algorithm guarantees an optimal solution by outputting the allocation with the lowest possible cost. However, to fully understand what 'optimal' entails in this context, we will delve deeper into the objective function of the algorithm.

6.2.1 Objective Function

The objective function of the LAPJV algorithm is defined as follows:

$$\begin{aligned} \min_{\{x_{i,j}\}} \quad & z = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{i,j} x_{i,j} \\ \text{s.t.} \quad & \sum_{i=1}^{n} x_{i,j} = 1 \quad \text{for all } j \\ & \sum_{j=1}^{n} x_{i,j} = 1 \quad \text{for all } i \\ & x_{i,j} \in \{0, 1\} \end{aligned}$$

where:

$c_{i,j}$ = Cost associated with assigning student i to project j, influenced by student and staff preferences or by the unallocation cost.
$x_{i,j}$ = Decision variable that is 1 if student i is assigned to project j, and 0 otherwise.
Z = Total cost of the assignment, which the algorithm seeks to minimize.

This formulation clearly sets out the optimization challenge tackled by the LAPJV, which aims to minimize the total cost of assignments while ensuring each student and project is uniquely matched.

However, our specific scenario deviates slightly because we do not have equal numbers of students and projects, students do not rank all projects, and we may have unallocated students, a situation not typically accounted for in the standard LAPJV model. To accommodate our needs, we adapted the algorithm as detailed in the adaptations table 4.2. Unallocated students are identified post-optimization: students assigned to 'dummy' projects or to projects at the `UNALLOCATION_COST` are considered unallocated. The `UNALLOCATION_COST`, a crucial hyperparameter outlined in the table 4.8 and discussed in the section 5.2.1, is set by the FYP coordinator and is instrumental in influencing the algorithm's bias towards maximizing student allocations.

By implementing these specific adjustments, we effectively customize the LAPJV algorithm's objective function for our context. It is important to recognize how these modifications alter both the problem and its solutions to accurately evaluate how well the adjusted algorithm addresses the required objectives and outputs.

6.2.2 Requirements Addressed and Analysis

The optimization problem we previously described is effectively tackled by the LAPJV algorithm. As outlined in the design section 4.3.2, this algorithm meets key requirements such as "No Low Preferences" 3.4.2 and "Full Allocation" 3.4.3. However, as discussed in section 3.4.4, these requirements can sometimes be at odds. Prioritizing high-preference allocations may result in some

students remaining unallocated, while striving for full allocation might necessitate accepting some lower preferences.

Varying the Unallocation Cost (`UNALLOCATION_COST`)

To better understand how the LAPJV algorithm manages this trade-off, it is essential to examine its objective function (see 6.2.1). The primary goal of the LAPJV is to minimize the total cost of allocation. The way this cost is defined plays a crucial role in determining which requirements the algorithm prioritizes. Typically, this cost is the sum of preferences indicated by both staff and students, or it is represented by the `UNALLOCATION_COST` when a student has not ranked a project. If we assign a significantly higher `UNALLOCATION_COST` compared to other costs, the algorithm will naturally prioritize achieving a full allocation, as any unallocated student would substantially increase the total cost.

To comprehensively analyze how the `UNALLOCATION_COST` impacts this trade-off, we have varied this parameter and monitored both the number of unallocated students (addressing the Full Allocation requirement) and the average cost of allocations (addressing the No Low Preferences requirement). This approach allows us to observe how the algorithm balances these competing requirements under different settings.

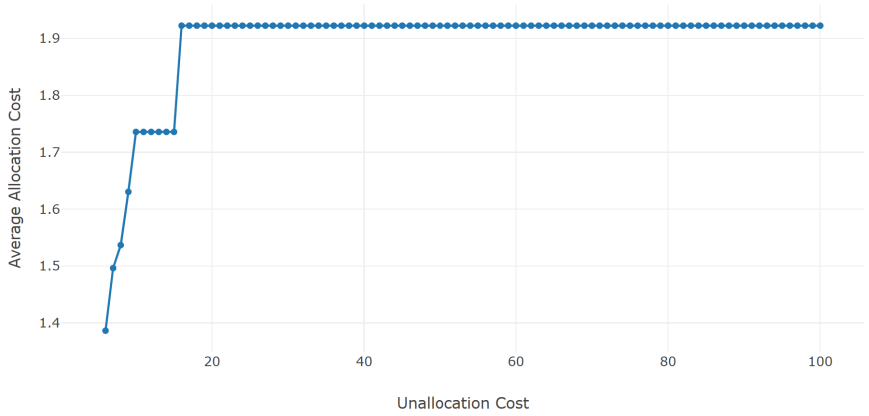


Figure 6.1: Average Allocation Cost vs Unallocation Cost

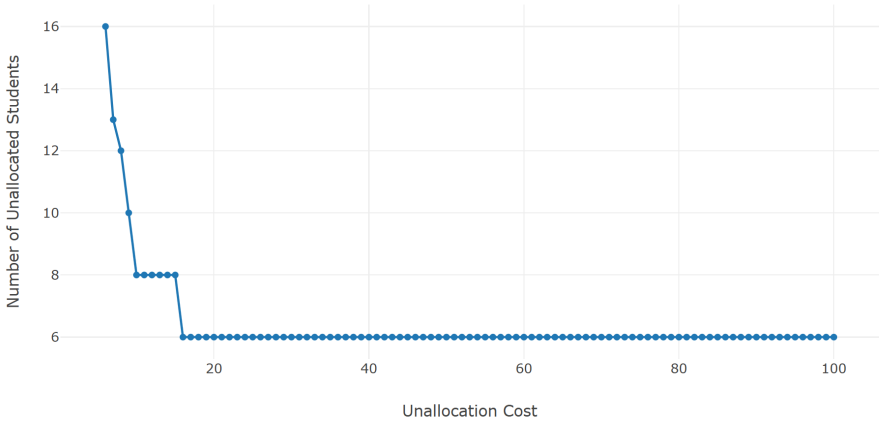


Figure 6.2: Number of Unallocated students vs Unallocation Cost

Examining both graphs, 6.2 and 6.1, we observe a clear trade-off between the two requirements. The trends in these graphs are diametrically opposed, indicating that as the number of unallocated

students increases, the average allocation cost decreases, suggesting that students are allocated to their higher preferences. This raises a critical question: Which requirement should we prioritize? Is it more detrimental to have unallocated students or students placed into low-preference assignments? Would a student prefer not to be allocated at all, or to be allocated to a less preferred project?

These questions are complex, but analyzing them helps us understand the implications of each scenario. Typically, there are three outcomes for unallocated students, as described in the requirement capture (see 3.4.3). There are both favorable and unfavorable outcomes for unallocated students, making it difficult to determine which is better. To address this, we have provided the FYP coordinator with hyperparameters to adjust, allowing for a tailored middle-ground solution. Additionally, we will provide extensive graphs and data to enable the FYP coordinator to make informed decisions. These outputs are detailed in section 8.

Upon examining graph 6.1, we observe that the average allocation cost remains relatively stable, which is anticipated given that it is an average over a large number of students. Consequently, significant changes in this metric are not easily noticeable. Nevertheless, the average allocation cost fluctuates mildly from 2.4 to 2.9, consistently staying between 2 and 3. This indicates that the majority of students are still being allocated to their higher preferences, demonstrating the effectiveness of the LAPJV algorithm.

Controling number of unallocated students

Upon closer examination of graph 6.2, we observe that there is a threshold beyond which the number of unallocated students does not decrease further. Setting the unallocation cost to 20 or any higher value results in a consistent outcome of six unallocated students. This stability reflects the inherent challenges of the problem where it is not feasible to assign every student due to limitations in project availability and preferences. Nonetheless, it would be insightful to explore unallocation costs below 20 to determine whether we can manipulate the number of unallocated students by adjusting this parameter.

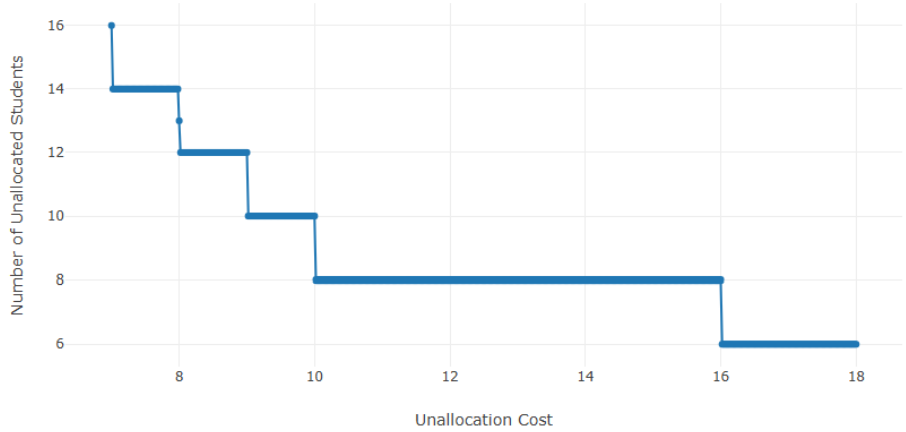
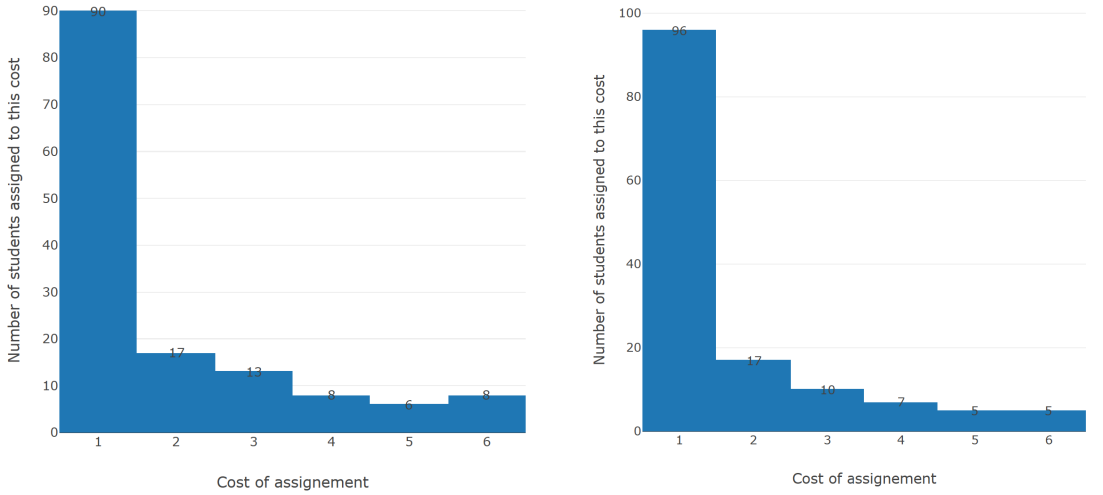


Figure 6.3: Number of Unallocated Students vs Unallocation Cost

The graph in 6.3 illustrates that the most significant increase in the number of unallocated students at any step is two. Despite this, it remains challenging to precisely control the count to a single unallocated student. This limitation is largely attributed to the complexities of the allocation process, exacerbated by our decision to reduce supervisor load limits, thereby restricting the number of projects available for allocation.

Manual vs. Algorithmic Unallocation of Students

An interesting aspect to explore is the difference between manually unallocating students and allowing the algorithm to handle unallocations. Conventionally, one might choose to manually unallocate students who received the lowest preferences after an initial allocation. However, an intriguing question arises: how does the algorithm perform if it is instructed to unallocate a certain number of students and then rerun? This scenario can be effectively visualized by comparing the preference distributions before and after algorithmic intervention.



(a) Preference distribution with 6 unallocated students

(b) Preference distribution with 8 unallocated students

Figure 6.4: Histograms comparing the distributions of costs allocated by the LAPJV algorithm before and after adjusting the number of unallocated students

The histograms in Figure 6.4 demonstrate that when the algorithm is tasked with unallocating two additional students, it does not simply remove students from the lowest cost categories as one might manually. Instead, the LAPJV algorithm strategically selects students to unallocate, resulting in an improved overall allocation. Notably, the number of students allocated at the lowest cost, which is a cost of 1, increased by six. Meanwhile, the allocations of students at higher costs, specifically costs 3, 4, and 6, decreased. This adjustment highlights the LAPJV's capability to optimize the allocation globally, taking into account all costs collectively rather than focusing narrowly on just the lowest preferences.

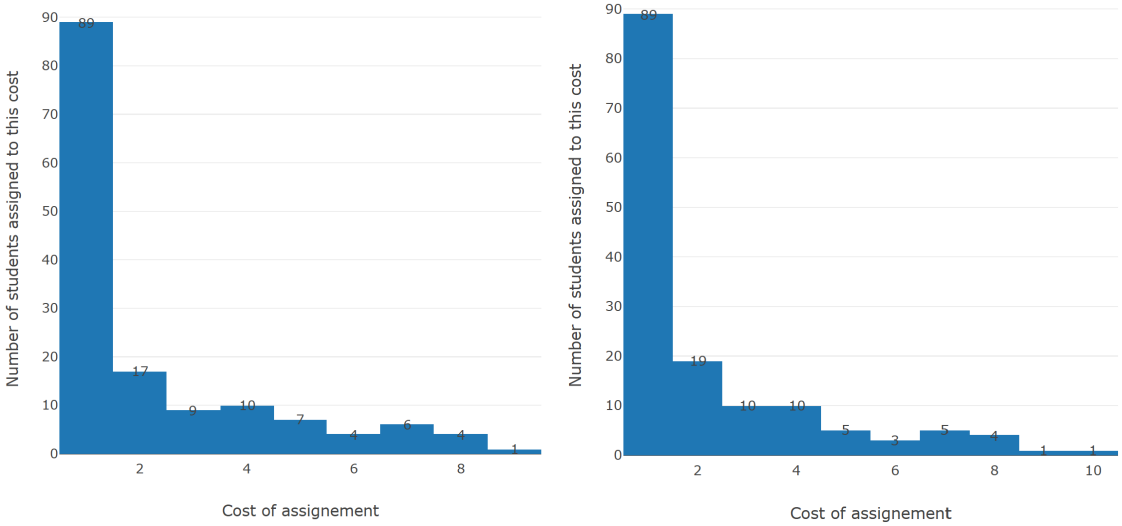
This evidence highlights the superiority of the LAPJV algorithm over manual interventions. It also justifies its use in the ILS process, where slight algorithmic adjustments, informed by a comprehensive understanding of the entire allocation landscape, can lead to significantly better outcomes than instinctive manual adjustments. The ability of the LAPJV to adaptively redistribute allocations, enhancing overall fairness and efficiency, underscores its value in complex decision-making scenarios.

Exploring the Impact of Preference Cutoff (`PREFERENCE_CUTOFF`) on Allocation Quality

In our ongoing analysis of allocation strategies, we have explored the effects of modifying the `UNALLOCATION_COST` as discussed in section 6.2.2. Similarly, we can adjust another hyperparameter, `PREFERENCE_CUTOFF`, as outlined in table 4.8 and discussed in section 5.2.1. This hyperparameter determines the preference threshold beyond which all lower preferences are discarded.

Adjusting it provides another mechanism to influence the allocation process by forcing the algorithm to allocate higher preferences. Intuitively, pruning lower preferences might lead to an increase in unallocated students due to the reduction in feasible allocation options. This adjustment revisits the trade-off between having unallocated students and securing high-preference assignments.

An intriguing scenario to consider is one where reducing the preference cutoff does not affect the number of unallocated students. This might occur if lower preferences are typically leveraged to enable a greater number of students to secure their top choices. In this situation, some preferences may serve a strategic purpose, enhancing overall allocation results. Therefore, pruning these lower preferences could shift more students toward mid-range options, rather than allowing them to achieve the highest or lowest preferences. This phenomenon is illustrated in the histograms below (see 6.5), prompting us to consider: Is it more beneficial to maintain a distribution of both lower and higher preferences, or to focus on a concentration around median preferences?



(a) Preference distribution with cutoff set to 9 (b) Preference distribution with cutoff set to 10

Figure 6.5: Comparative histograms of allocation costs with different preference cutoffs

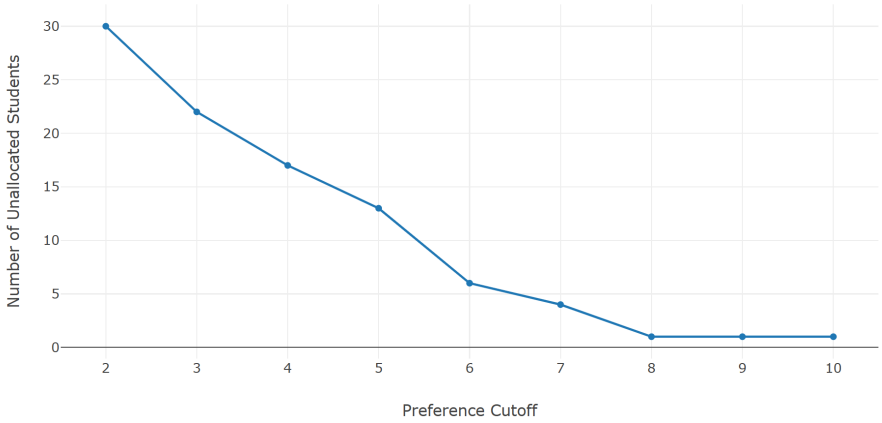


Figure 6.6: Unallocated students versus preference cutoff

It is important to note that the costs represented in these histograms 6.5 are the combined results of both student and staff preferences. The preference cutoff, however, exclusively affects students' selections. As illustrated in the graph 6.6, the number of unallocated students remains unchanged when altering the preference cutoff from 9 to 10. The histograms in 6.5 further reveal that the change in cutoff from 10 to 9 decreases the allocation at cost 2 by two and similarly

reduces the number of allocations at cost 3, while the lower preferences see an expected increase.

This evidence supports our hypothesis that permitting lower preferences enables higher preferences to be allocated more effectively. Since the LAPJV algorithm optimizes globally, it does not consider students individually but rather the overall cost of the problem.

Determining whether to prune preferences is complex since it is challenging to definitively prefer one scenario over another. However, our analysis highlights the need for mechanisms like the ILS that consider individual cases alongside the global optimization strategy of the LAPJV, providing a more nuanced approach to managing trade-offs in student project allocations.

6.2.3 Concluding Remarks

In conclusion, this analysis has thoroughly evaluated the LAPJV algorithm by exploring its objective function and the impact of varying critical parameters. Our focus has been on understanding how different settings influence the algorithm's outcomes, particularly how it balances the competing requirements of "No Low Preferences" and "Full Allocation."

Through our investigations, particularly by adjusting the `UNALLOCATION_COST` hyperparameter, we have illuminated the inherent trade-offs between these requirements. This adjustment has demonstrated the flexibility provided to the FYP coordinator, enabling precise control over the number of unallocated students. This level of control is crucial for tailoring the allocation process to specific academic or administrative goals.

Furthermore, we have assessed the effectiveness of the LAPJV by comparing outcomes derived from manual interventions versus those generated algorithmically. The results confirm that the LAPJV excels in strategic and adaptive optimization, consistently delivering optimal solutions under varied conditions. Nonetheless, its strength in global optimization sometimes leads to the oversight of local specifics, particularly evident when we modified the `PREFERENCE_CUTOFF` hyperparameter. This modification highlighted not only the changes in the number of unallocated students but also the shifts in the cost allocation distribution.

These observations underscore the necessity of integrating the ILS alongside the LAPJV. The ILS provides a valuable mechanism for addressing individual discrepancies and refining the overall allocation process, ensuring that both global efficiency and local fairness are achieved.

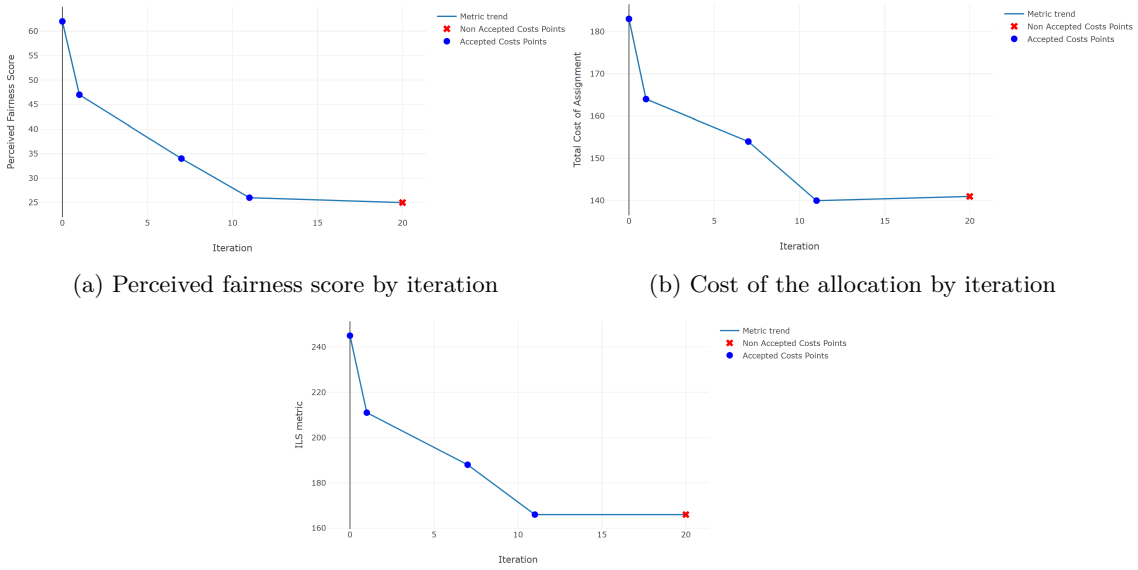
6.3 Evaluating the ILS Algorithm

First, we need to examine the iterations of the ILS algorithm to understand what it is trying to achieve and how it does so. By understanding and confirming that the algorithm's design is being implemented as intended, we can then proceed to evaluate the quality of the solutions it produces. To effectively evaluate our ILS algorithm, we must compare it with alternative versions to determine if our solution is indeed the best.

6.3.1 Overview of the ILS iterations

By examining the iterations of the ILS algorithm, we can gain insights into its process for refining allocations. The ILS employs a combined metric, as described in 4.4.5, which incorporates the total allocation cost and the perceived fairness score. A solution is accepted if it improves according to this combined metric.

The set of graphs in Figure 6.9 illustrates each component of this metric, showcasing how both the individual and combined metrics influence the decision to accept or reject a particular allocation.



(a) Perceived fairness score by iteration

(b) Cost of the allocation by iteration

(c) Combined metric by iteration

Figure 6.7: Graphs showing different metrics by iteration of the ILS algorithm

6.3.2 Analysis of results

In the first iteration, the most severe perceived unfairness case is removed, and the LAPJV algorithm is rerun. By removing the largest contributor to the perceived fairness score, we generally expect a reduction in the overall score, which is confirmed in most cases, as shown in Figure 6.7a. However, this is not always the case; occasionally, removing one significant issue can lead to the emergence of other problematic cases, potentially worsening the overall situation.

Regarding the total cost metric, as depicted in Figure 6.7b, we calculate the sum of costs across all allocated students. This value tends to decrease, even though the LAPJV algorithm provides an optimal solution, because we are only considering the allocated students without looking at the unallocated ones. However, as discussed in subsection 5.5.4, we also monitor the number of unallocated students in relation to a new hyperparameter, `NUMBER_OF_UNALLOCATED_STUDENTS` set by the FYP coordinator. If we choose not to specify this number, we can deactivate the feature, allowing the number of unallocated students to be included in the total cost metric. In such scenarios, further improvements in the total allocation cost are not possible since the initial allocation determined by the LAPJV is already optimized in this regard.

As anticipated, the graphs in Figure 6.7 demonstrate that only solutions with a reduced combined metric are accepted. The final accepted solution achieves a combined metric of 166, marking a significant improvement from the initial metric of 245.

To evaluate the effectiveness of the ILS algorithm against the perceived fairness requirement, we examine the cases of perceived unfairness before (see Table 6.1) and after applying the ILS (see Table 6.2). The results show a significant improvement, with the number of cases decreasing from 22 to 12. Additionally, the number of cases with very high scores has decreased the most, which is precisely the desired outcome. The cases with the highest scores represent the most severe instances of unfairness, so reducing these is particularly beneficial. It is preferable to have more cases with minor differences, such as a single preference difference, rather than a few cases where a student is treated significantly more unfairly than others. Thus, we can see the ILS's effectiveness in addressing the perceived fairness requirement (see 3.4.1) by focusing on local cases, which was the primary reason for its design.

Student	Project	Cost	Student Assigned Cost	Project Assigned Cost	Perceived Fairness Score
90498	1802914	2	7	6	5
90228	1803130	2	7	7	5
90208	1802971	2	7	4	5
90124	1803048	2	7	5	5
90227	1802961	2	7	3	5
90191	1802914	2	7	3	5
90242	1802946	3	7	4	4
90251	1803048	2	6	5	4
90213	1803009	2	6	NA	4
91767	1802884	2	5	7	3
89067	1802971	2	4	4	2
90479	1802914	2	4	3	2
90279	1802998	3	5	NA	2
91800	1802945	2	4	NA	2
90235	1803056	4	6	5	2
94187	1803056	4	5	5	1
65416	1802914	2	3	3	1
90498	1803140	2	3	6	1
90487	1803122	2	3	3	1
90471	1802929	2	3	NA	1
90230	1803036	2	3	3	1
90205	1802884	3	4	7	1

Table 6.1: Perceived Fairness Cases before ILS

Student	Project	Cost	Student Assigned Cost	Project Assigned Cost	Perceived Fairness Score
90208	1802971	2	7	4	5
90213	1803009	2	6	NA	4
90251	1802971	3	6	4	3
91767	1802884	2	5	7	3
89067	1802971	2	4	4	2
91800	1802945	2	4	NA	2
90235	1803056	4	6	5	2
94187	1803056	4	5	5	1
90498	1803140	2	3	6	1
90487	1803122	2	3	3	1
90471	1802929	2	3	NA	1
90205	1802884	3	4	7	1

Table 6.2: Perceived Fairness Cases after ILS

We have now seen that the ILS algorithm is effective in addressing the perceived fairness requirement. However, we still need to evaluate the overall solution it produces. To determine if it is the best solution, we must compare it against different algorithms.

6.3.3 Comparison with Alternative Approaches

Assessing key decisions in our algorithm's design, including the choice of the starting point and our strategies for exploring new solutions in subsequent iterations, is critical for evaluating its effectiveness. Many of these decisions were based on hypotheses and assumptions, which alone may not sufficiently justify their use.

The primary assumption was that our approach would be less likely to encounter local minima, and if it did, these minima would be favorable. This hypothesis is grounded in the fact that our algorithm's starting point is the initial allocation determined by the LAPJV, which is considered optimal in terms of cost. This cost component is one of the two metrics comprising the ILS metric that evaluates our solutions. The second component is the perceived fairness score, elaborated in Section 4.4.5.

Our exploration strategy also plays a critical role in this assumption. Starting from the initial allocation, we rank all indices flagged for perceived unfairness according to their fairness score. Our exploration then proceeds from the highest to the lowest scores. We hypothesized that addressing the indices with the largest scores first would more effectively improve the overall score, given their significant impact on the perceived fairness. This method assumes that tackling the most significant contributors to perceived unfairness first would yield the most substantial improvements in the overall metric.

As previously mentioned in 4.4.5, escaping local minima typically involves adding perturbation mechanisms. It's crucial to evaluate our ILS algorithm against this backdrop to verify its capability to escape local minima as assumed. To design this alternative approach, we introduced randomness into our ILS design, starting with the initial assignment (the starting point). Instead of beginning with the LAPJV solution, we experimented with randomizing the starting point. However, the comparison is not straightforward. Merely changing the starting point may not significantly impact the overall performance, since the ILS iterations themselves rely on the LAPJV to generate new allocations after excluding the index with the highest perceived fairness score. This approach does not introduce significant randomness into the evaluation. As discussed in subsection 6.2.2, when addressing minor adjustments, the algorithmic approach is preferable because it strategically assigns or unassigns specific students, which is more effective than manual adjustments. To enhance the randomness and effectiveness of these tests, we have also introduced noise into the cost matrix during the iterations. This addition allows for a more meaningful comparison by simulating varying conditions that the algorithm might encounter in practical scenarios.

Regarding the strategy used to create a new solution, it is deterministic, as it proceeds from the largest perceived fairness score to the lowest. To infuse more randomness, we considered randomizing the order in which flagged indices are checked.

It is crucial to recognize that the ILS algorithm we designed is deterministic, typically producing the same solution each time it is executed. With this new, non-deterministic approach, however, it is necessary to repeat the process multiple times to ensure a meaningful comparison.

Statistic	Value
Minimum	161
First Quartile (Q1)	183
Median (Q2)	196
Third Quartile (Q3)	215
Maximum	1000

Table 6.3: Summary Statistics of Final Combined Metric for Random ILS Over 50 Trials

As shown in Table 6.3, the final combined metric of the random ILS over 50 trials ranges from 161 to 1000. The value of 1000 represents cases where the initial starting point did not yield any feasible allocation, leaving all students unallocated. Since all students are unallocated in these cases, there are no perceived unfairness cases, causing the algorithm to instantly exit. In the previous section 6.3.2, our algorithm achieved a result of 168, indicating that while it substantially reduced the score, it was not able to completely escape local minima. This suggests that while our initial assumptions were reasonable but not fully correct, introducing perturbations might further enhance the ILS performance.

We could consider a hybrid approach that combines the deterministic strategy of reducing the largest cases first with randomized starting points. Since we are already using the LAPJV within

the ILS, adding noise to the cost matrix might also help achieve better results.

The main challenge with the fully random method is its slower performance due to the need to repeat the ILS multiple times, which involves multiple iterations. Opting for a hybrid approach might make it faster since we can track if all indices were checked. However, if the checking is not deterministic, we might end up checking the same index repeatedly, wasting time.

Statistic	Value
Minimum	146
First Quartile (Q1)	157
Median (Q2)	189
Third Quartile (Q3)	190
Maximum	1000

Table 6.4: Summary Statistics of Final Combined Metric for Hybrid ILS Over 50 Trials

Examining the summary statistics for the hybrid ILS in Table 6.4, we observe that the values range from 149 to 1000. The hybrid method is effective, as it achieves better results in over 25% of the trials compared to the deterministic ILS. By introducing perturbations without overwhelming the algorithm with randomness, we were able to steer towards better solutions. However, further testing is necessary to evaluate performance and determine the optimal number of runs (see Section 6.5). We also need to assess the appropriate amount of noise to add to the cost matrix (see 6.4.1).

6.3.4 Additional Comparisons

An important comparison to make is with a pure ILS approach that does not incorporate the LAPJV within the ILS iterations. We anticipate that the pure ILS will be slower since it will attempt many more operations than just removing the perceived fairness indices, including all possible swaps. Pure ILS usually converges to a local minimum near the starting point, so it will be interesting to compare it with our algorithm. We hypothesize that the pure ILS will not perform better than our algorithm, as it lacks guidance. In contrast, our algorithm intentionally reduces both components of the ILS metric by running LAPJV for cost and removing the largest perceived unfairness case for the perceived fairness score. I plan to implement this new algorithm to test my hypotheses and hope to have results by the presentation.

6.4 Evaluating the hybrid ILS

We have determined that the hybrid ILS is more effective than the previously designed deterministic ILS. Consequently, we can now perform additional analyses and evaluations on this new version, which were not conducted on the older version.

6.4.1 Optimizing Noise Levels in the Cost Matrix

As discussed above, when dealing with the hybrid ILS, we need to optimize the noise levels added to the cost matrix. To do this, we will conduct a series of tests to find the balance between introducing sufficient randomness to avoid local minima and maintaining enough structure to keep the algorithm on track. The hypothesis is that adding too much noise could mislead the algorithm, while too little noise might not provide the necessary perturbations to escape local minima. To achieve this, we will test multiple values for the maximum noise added to each entry in the matrix and analyze the results.

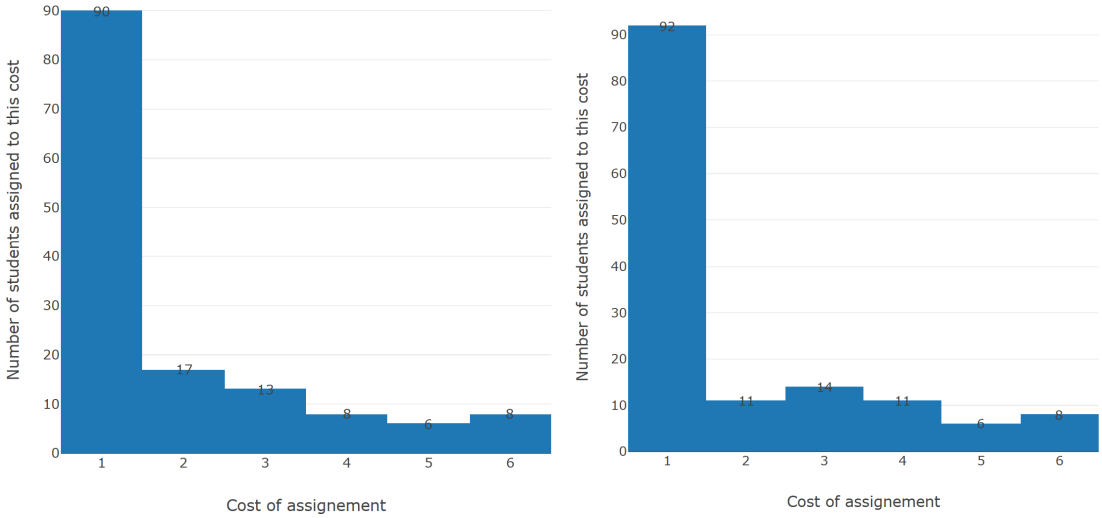
Statistic	Max noise						
	0.1	0.2	0.5	0.7	1	2	5
Minimum	154	157	146	146	148	150	152
First Quartile (Q1)	160	160	157	158	159	165	172
Median (Q2)	164	166	173	161	174	172	183
Third Quartile (Q3)	177	170	189	181	1000	189	201
Maximum	1000	1000	1000	1000	1000	1000	1000

Table 6.5: Summary Statistics of Final Combined Metric for Hybrid ILS with Different Maximum Noise Added per Matrix Entry

As shown in Table 6.5, the best solutions were obtained with maximum noise levels set to 0.5 or 0.7. This aligns with our hypothesis, as very small or very large noise values were less effective in achieving good solutions. While the inherent randomness in these processes makes it challenging to determine the optimal noise level with absolute certainty, we can confidently state that moderate noise levels are more likely to yield better results.

6.4.2 Evaluating the hybrid ILS iterations

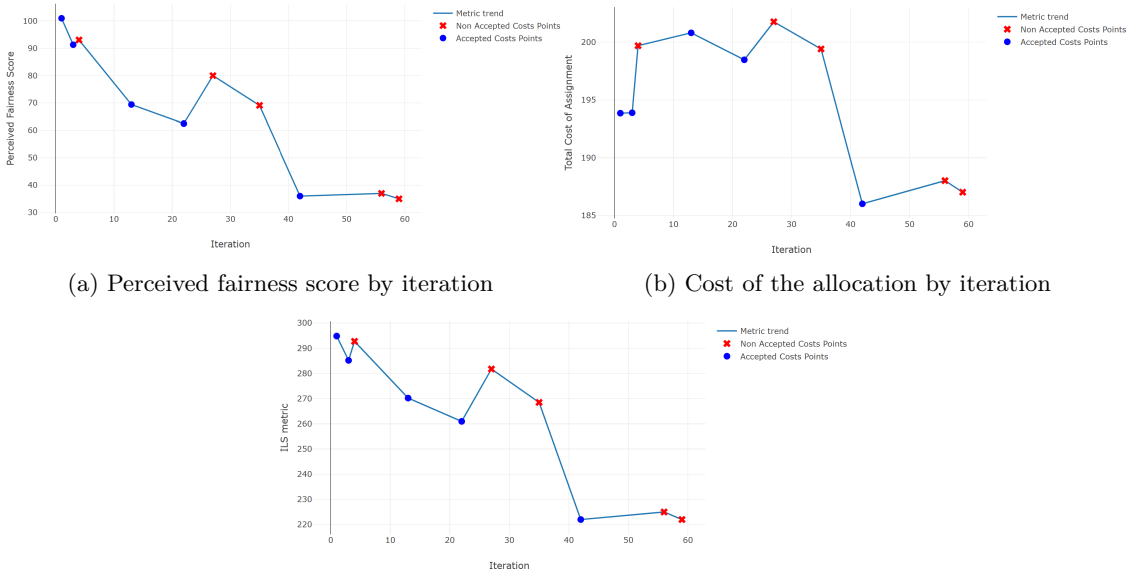
One aspect we have not yet considered, which can now be explored with the new hybrid approach, is what happens if we set `NUMBER_OF_UNALLOCATED_STUDENTS` to 0. This means the ILS would not be allowed to unallocate any additional students and must maintain the same number of allocated students as the LAPJV. Specifically, we aim to determine how much the average allocation cost or the total allocation cost would increase from the optimal one provided by the LAPJV. Additionally, it would be interesting to observe the ILS's behavior in this situation by examining the histogram of preferences allocated before and after the hybrid ILS, and analyzing the ILS metrics during the iterations.



(a) Preference distribution after LAPJV (b) Preference distribution after hybrid ILS

Figure 6.8: Comparative histograms of allocation costs after hybrid ILS and after a single LAPJV allocation

Looking at the histograms in Figure 6.8, we can see that the ILS tries its best to increase the first preference allocation without significantly increasing very low preference allocations. However, just examining the histograms is not enough; we need to look at the perceived fairness score. After the LAPJV allocation, the perceived fairness score is 64, as shown in Figure 6.7a. The ILS was able



(a) Perceived fairness score by iteration

(b) Cost of the allocation by iteration

(c) Combined metric by iteration

Figure 6.9: Graphs showing different metrics by iteration of the hybrid ILS algorithm

to reduce this score to 38, as seen in Figure 6.9a, which is a significant improvement. Additionally, the total cost of the allocation after the LAPJV is 183 (Figure 6.7b), and our hybrid ILS solution results in a cost of 187 (Figure 6.9b), indicating that the total cost is not significantly affected. The average cost varies from 1.92 to 1.95, showcasing the ILS's success in improving perceived fairness while almost maintaining the cost metric.

In conclusion, this analysis demonstrates the hybrid ILS's success in balancing both objectives and optimizing them simultaneously, thereby addressing all requirements.

6.5 Evaluating performance

Given the objective of this project to create a fast two-sided project allocation system, evaluating performance is essential. To assess performance, I first benchmarked the original deterministic ILS to assess how fast it for different numbers of iterations to ensure that the running time remains acceptable regardless of the number of iterations.

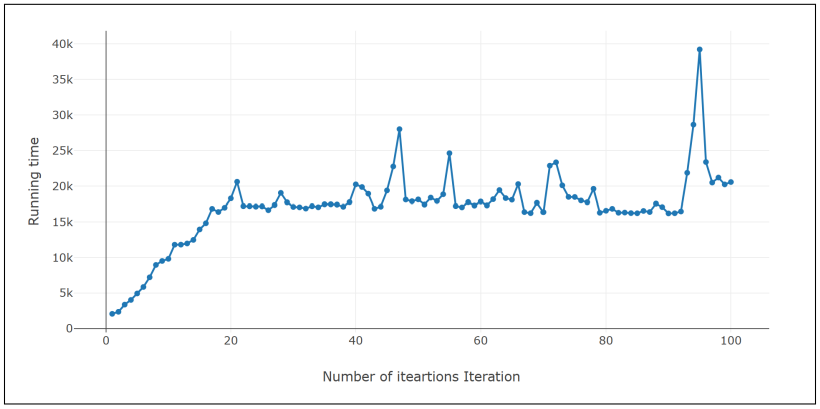


Figure 6.10: Running time (ms) vs Number of iterations for the deterministic ILS

Figure 6.10 shows the running time of the full system for different numbers of iterations of the ILS algorithm. This measurement includes preprocessing as well as the initial LAPJV allocation. The graph illustrates that while the running time increases with the number of iterations, it eventually plateaus. This plateau occurs because the ILS algorithm exits if there are no more indices to explore.

The average running time is around 20,000 milliseconds (20 seconds), which is a significant improvement from the current 30 minutes without a guarantee of an optimal solution. This demonstrates the success of our system in achieving efficient performance.

Now it is important to evaluate the performance of the new hybrid ILS design discussed in section 6.3.3, as its primary drawback could be its performance due to the need for multiple iterations because of its randomness. To analyze this, I set the ILS iterations to 50 and varied the number of times the ILS was repeated. It is essential to ensure that enough iterations are performed to achieve a good solution, which is why I also plotted the best solutions obtained across these iterations.

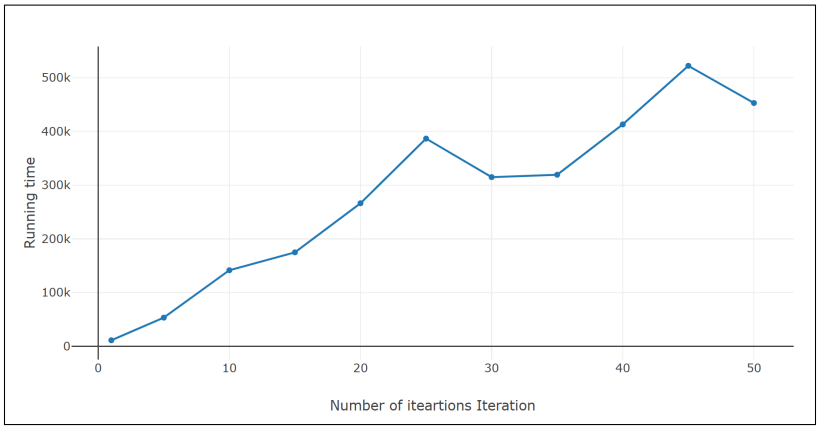


Figure 6.11: Running time (ms) vs Number of runs of the hybrid ILS

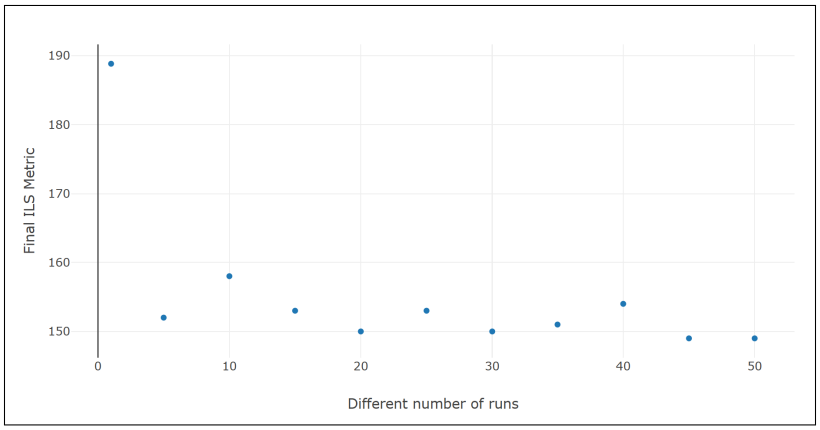


Figure 6.12: Final ILS metric value vs Number of runs of the hybrid ILS

As shown in Figure 6.11, the running time increases almost linearly with the number of iterations, which is expected. The deviations from the linear increase are due to the inherent randomness; occasionally, an iteration might take less time if the starting point is unfeasible or if it takes a shorter route, while other times it might take longer. Therefore, the optimal approach would be to use the fewest iterations necessary to achieve a good solution. However, due to this randomness, we cannot guarantee the best solution every time, but we can significantly increase

the likelihood of obtaining it. For 50 iterations, the running time is around 500,000 ms, which is approximately 8 minutes—considerably longer than the deterministic approach.

Looking at the results in Figure 6.12, we observe that regardless of the number of iterations, achieving the global minimum is a matter of chance. However, the local minima reached are consistently good, with all being below 160, which is an improvement over our deterministic approach. As previously discussed, fewer runs are preferable for speed. Around 30 runs seem sufficient, resulting in a running time of about 300,000 ms, or approximately 5 minutes, which is acceptable. However, this remains a hyperparameter and can be modified by the FYP coordinator.

In conclusion, the hybrid solution is more time-consuming but yields better results. It should be used if time permits, while the deterministic approach, being much faster, can still improve results if time is a concern, though not to the same extent as the hybrid approach. However, it is important to note that the hybrid method takes approximately 5 minutes, which is still a significant improvement compared to the current system that takes 30 minutes.

6.6 Evaluating stability

Before evaluating the stability of our system, it is important to define what stability means in this context. For the SPA algorithm (see section 2.4.1), stability is defined as an allocation with no blocking pairs. However, our case differs because we combine scores (see section 4.4.1) and may encounter equal suitability rankings. This means we cannot definitively determine which student a project would prefer most when rankings are equal.

Therefore, we define stability differently. In our context, a stable solution is one that withstands small changes. This definition is relevant because it helps the FYP coordinator understand how resilient the solution is to minor adjustments and how such changes might affect the overall allocation. Studying and displaying this stability information is crucial for the coordinator to make informed decisions.

6.6.1 Steps to Obtain Stability Data

To evaluate stability, we follow these steps:

- **Add Noise:** Introduce a small, predefined amount of noise to the cost matrix.
- **Re-run Allocation:** Execute the allocation algorithm with the modified cost matrix.
- **Count Differences**: Count the number of allocations that differ from the original solution
- **Vary Noise Levels:** Repeat the process with varying noise levels, from a 0.1 maximum addition to an entry up to 0.5.
- **Calculate Stability Metrics:** Determine the average number of differences (or percentage), the variance, and the standard deviation.

These stability metrics provide valuable insights for the user, primarily the FYP coordinator, but they do not directly impact student satisfaction. Instead, they offer an understanding of how robust the allocation solution is to minor changes.

This method was designed before deciding to proceed with the hybrid ILS, and made more sense at that time. However, since the new ILS method inherently adds noise to the matrix, testing stability against this version may not be relevant. It can still be tested against the LAPJV solution, but it would be less significant.

6.7 Gameability

Gameability of the system is a major concern and is one of our requirements. We have addressed this requirement by ensuring real fairness. Specifically, we penalize students who attempt to game the system by not adhering to the minimum number of required selections. This discourages attempts to manipulate the allocation process.

However, there are additional measures that could further mitigate gameability, although they fall outside the scope of this project. For instance, controlling the information released to students can be effective. By not disclosing specifics about how many students have selected each project, as Dominic has discussed in his report which are also stated the requirements section 3.6, we can reduce the potential for strategic behavior. These additional measures would enhance the robustness of the system against gaming but are not implemented in the current project scope.

6.8 Conclusion

In conclusion, we have thoroughly evaluated our system to ensure it meets all the requirements defined in Chapter 3. We analyzed our solutions to understand their effectiveness and to determine if they meet our standards.

We examined the trade-offs between our requirements and how they manifest in our solutions. We also discussed how these trade-offs can be managed through hyperparameters, a responsibility left to the FYP coordinator, while we provide all necessary information on the allocation for informed decision-making. Additionally, we demonstrated the effectiveness and intelligence of the LAPJV algorithm by comparing its automated adjustments against manual changes, proving it is well-suited for our case and justifying its use in our ILS algorithm.

The ILS algorithm successfully addresses perceived fairness while maintaining all previous considerations. We analyzed the outputted solution and observed a reduction in perceived unfairness. However, we also discovered that our approach sometimes gets stuck in local minima. To address this, we introduced perturbation mechanisms by randomizing the starting point and adding noise to the cost matrix, thus diversifying the algorithm's path. We also evaluated the new approach, demonstrating its ability to optimize the combined objective functions and meet all requirements.

We demonstrated that our system runs efficiently, with an average runtime of 20 seconds for the deterministic approach and 5 minutes for the hybrid approach, a significant improvement over the current system's 30-minute runtime without a guarantee of an optimal solution.

We also discussed the concept of stability in our context, explaining how stability metrics could be valuable for the FYP coordinator in managing the allocation process in some cases.

Finally, we addressed gameability by explaining how our algorithm penalizes students who attempt to game the system and discussed additional mechanisms, outside the scope of this project, that could further mitigate gaming behavior.

Overall, our system meets the defined requirements, addresses key concerns such as fairness and stability, and operates efficiently, making it a substantial improvement over existing systems and a success.

7

Conclusions

Contents

In this chapter, I will reflect on the work completed, summarizing key achievements and reviewing some important aspects and reasoning behind them. This will be followed by a discussion on future work and potential improvements.

7.1 Overview of Achievements

The achievements of this project can be divided into two main areas: a novel formulation of the problem based on requirement capture, discussions with professors, and a literature survey; and the creation of an efficient method to optimize this problem for typical use cases. A key decision made before diving into this work was to implement the entire project in F#, ensuring a simple, self-contained solution that is easy to deploy and maintain [46]. The complete reasoning behind this choice is detailed in section 2.3.

7.1.1 Novel Formulation of the Problem

In the background chapter 2, I reviewed several existing algorithms that address similar problems to the one we were facing to assess their usefulness in our case. This review helped identify the need for a unique formulation to meet our specific requirements.

After consultations with professors and discussions with classmates, I defined the necessary requirements to satisfy all stakeholders. The novelty of this project lies in addressing all these

requirements simultaneously, as no existing algorithm does so. Many of these requirements were contradictory and ambiguous, making it challenging to create a comprehensive solution.

The main challenge was the requirement of perceived fairness (see 3.4.1) because it is a local, case-by-case requirement, while the others are global (”No Low Preferences” 3.4.2, ”Full Allocation” 3.4.3 and Real Fairness 3.4.1). What struck me was the link I made between perceived fairness and the SPA (also called stable marriage) (see 2.4.1). I realized that the SPA was designed around perceived unfairness, and I could draw inspiration from its concepts to design a solution. Conversely, the global requirements were tackled by algorithms solving the LSAP (see 2.4.2). Both types of algorithms are well-documented, but they have never been used together to address all requirements simultaneously, which is the unique aspect of this project. The innovation lies in combining concepts from these two different types of algorithms, the LSAP algorithms and the SPA, each with their own strengths and weaknesses, to comprehensively address all defined requirements.

7.1.2 Optimisation algorithm

I decided to first address the global requirements using algorithms that solve the LSAP problem. This decision was made because these algorithms closely matched our problem, and transforming our data into the necessary format was feasible, as discussed in the background section.

After reviewing several options, two prime candidates emerged: the Hungarian algorithm and the LAPJV. To use these algorithms, I had to transform our problem into a one-sided allocation problem. This involved developing preprocessing code to convert our data into suitable formats while incorporating some requirements through the cost function and mechanisms. The adaptations are detailed in Table 4.2.

Next, I needed to choose between the two candidates. I implemented F# versions of both algorithms. Initially, I used existing libraries and simple online versions. However, exhaustive testing revealed inaccuracies with large test cases, making web implementations unreliable (see 5.4.1). Consequently, I developed a new version based on a widely used C++ implementation of the Hungarian algorithm. Once thoroughly tested, creating the LAPJV implementation was straightforward, as I had a correct solution for comparison.

I benchmarked both versions (see 5.4.3) across different matrix sizes and densities, proving the LAPJV to be the superior choice.

I also explored whether small modifications within these algorithms could address all requirements by examining multi-allocation and single-allocation versions, as seen in 4.3. However, we found that minor algorithmic tweaks could not address all requirements. Therefore, we decided to use the single allocation as a first step since it addressed both global requirements.

To address perceived fairness without disregarding the other requirements already addressed, I designed a solution around the ILS algorithm, creating a metric to score the severity of perceived fairness and defining which cases constituted perceived unfairness. This metric was then combined with the LAPJV objective to form a complete objective function addressing all requirements. The full algorithm design of the ILS can be seen in section 4.4.5. The ILS algorithm was chosen for its simplicity and iterative approach, which can optimize any problem with an objective function.

An algorithm is never complete without proper evaluation. I assessed the algorithm for correctness, quality, and performance, and investigated the novel objective function to visualize trade-offs and complexities. This thorough evaluation revealed some deficiencies in initial assumptions, allowing adjustments to the ILS implementation for the best solution. This process highlights the importance of proper evaluation and the value of the engineering journey and Occam’s razor 2.5.1. Starting with simple assumptions enabled a swift completion of the algorithm, facilitating comprehensive evaluation and improvement.

The new improved version was evaluated, demonstrating success in optimizing both objective functions and addressing all requirements. Performance testing showed that our solution is much faster than any previously implemented solutions.

It is important to note that hyperparameters were designed to give the FYP coordinator more control over the allocation algorithm, as a lot depends on the specific allocation at hand. These hyperparameters are described in Table 4.8, and default values are explained in Section 5.2.1. Additionally, a significant amount of information is provided to help the coordinator make informed decisions, some of which are described in section 8.4.

7.1.3 Concluding Remarks

In conclusion, I thoroughly enjoyed every aspect of this project, from research and analysis to coding, testing, and evaluation. It has been incredibly gratifying to see my designs and work in action, functioning successfully. At the start, the project was ambiguously defined, but as time went became progressively clearer over time. This journey of turning a vague idea into a well-defined, successful project has been truly rewarding.

7.2 Further Work

There are always improvements that could enhance the project. However, most are not directly relevant to my deliverables and fall outside the project's scope. These improvements can be divided into two categories: further research and implementations to enhance user experience.

7.2.1 Enhancing User Experience

To utilize this algorithm effectively, it needs to be linked to the current allocation algorithm. Additionally, some improvements could make it easier and more enjoyable for the user to use.

- **Database Integration:** Implementing a database to store project and student information, as well as hyperparameters, would streamline access and management for the FYP coordinator. Having hyperparameters separate from deployment in a database would be cleaner.
- **User Interface and Dashboards:** Creating a user interface and dashboards to display allocation results more effectively would make the system more user-friendly and visually appealing.
- **System Integration:** Building a fully integrated system with the current preference submission platform would automate the process, eliminating the need for manual text file creation.

7.2.2 Further Research

There are also some areas for further research that, while not essential, could be beneficial.

- **Advanced Algorithms:** Exploring other algorithms, such as Tabu search or simulated annealing instead of the ILS. We primarily focused on the ILS for its simplicity and efficiency, following Occam's Razor, and because it produced great results. Thus, we did not initially consider other algorithms.

- **Dynamic Weighting:** Investigating dynamic weights for student and staff preferences based on project or supervisor popularity could enhance the algorithm's adaptability. However, since staff submit suitability rankings rather than actual rankings, this is less relevant.

By addressing these areas, the system could be further enhanced to create a more complete and visually appealing solution.

8

User & Deployment Guide

Contents

8.1 Introduction

In this chapter, we will guide you through the process of cloning the code repository, acquiring all necessary dependencies, and running the allocation algorithm. We will also explain how to modify the hyperparameters and specify where to change the text file paths to ensure the code runs successfully. The deployment and user guide are combined into one since we are using F# for the entire codebase, making it very simple to deploy and maintain.

8.2 Running the code

To run the code, follow these steps:

1. Clone the GitHub Repository:
   - URL: `https://github.com/rkhoury18/FYP`
2. Download and Install Necessary Dependencies:
   - .NET 7 - Available at: `https://dotnet.microsoft.com/en-us/download/dotnet/7.0`
   - In the main directory (`/FYP-Allocation-code`), run the following command to restore dependencies: `dotnet restore`
3. Update File Paths:

- Ensure that the paths to the text files described in 5.3.1 are updated in the `PreProcessing.fs` file.

4. Run the Code:
   - To run the application, execute: `dotnet run`.
5. Run Benchmarks:
   - To run benchmarks, you need to run the code in release mode. Uncomment the benchmarks in the `Program.fs` file and then execute: `dotnet run -c Release`

8.3 Changing the hyperparameters

The hyperparameters discussed in 4.8 and 5.2.1 can be found in the `Hyperparameters.fs` file. The default values are currently set, but they can be easily changed by the user as needed. Adjust the values directly in the file to modify the behavior of the algorithm.

8.4 Output Generated by the Code

After running the code, various graphs and information are outputted to the user. This section details the outputs provided:

- **Final Allocation:** The final allocation is displayed, including self-proposed students, unallocated students, and the assignments made by the algorithm, indicating the student, project, and associated cost.
- **Histogram of Final Allocation:** A histogram of the final allocation is shown for the FYP coordinator.
- **Perceived Fairness Cases:** The perceived fairness cases before and after the ILS are displayed, including details about the specific cases 4.6 they fall into and the scores for each flagged index.
- **Metric Data:** All metrics are displayed, including average allocation cost, total allocation cost, ILS metric, and perceived fairness score.

This section may be updated until the presentation to finalize what should be shown. Additionally, the code is currently untidy due to ongoing testing and result gathering, which will hopefully be improved by the time of the presentation.

A

Data Visualisation and Analysis

To gain deeper insights into our solutions and connect various concepts, I explored different visualizations to identify any discernible patterns or conclusions.

To achieve this, I developed code that introduces noise into our cost matrix, reruns the algorithm, and plots the results in different formats. The `generateRandomNoise` function modifies the cost matrix by adding a random value (noise) between 0 and a specified maximum to each element. Another function, `getMoreSolutions`, repeatedly invokes `generateRandomNoise` and reruns the optimization algorithm to find new solutions. Python was chosen for visualization due to its simplicity and extensive libraries available for various types of visualizations. The following subsections detail the different visualizations explored and the insights they provided.

A.0.1 Heatmap Visualisation

My initial approach was to understand the extent of variation in solutions introduced by the noise. A heatmap was created to visualize the differences between solutions, with similarity indices where 0 indicates no differences and higher values indicate more differences. Matplotlib [47] was used for this purpose, as illustrated in Figure A.1.

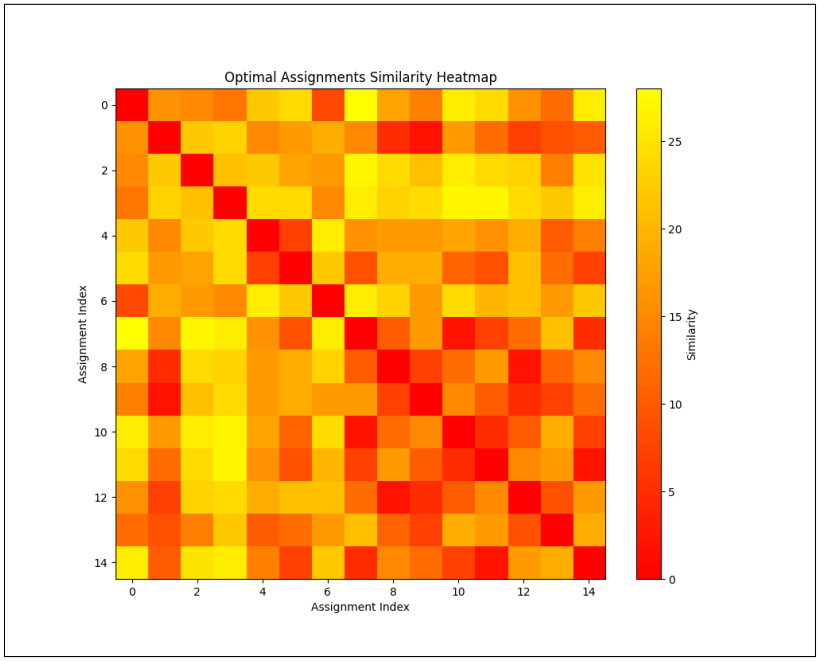


Figure A.1: Heatmap of Optimal Assignment's Similarity

However, interpreting the heatmap (Figure A.1) was challenging, leading me to explore alternative visualisation methods.

A.0.2 Graphical Visulisation

The next step was to visualise the proximity of solutions using a graph, where nodes represent solutions, and edge lengths correspond to the differences between solutions. For this, I utilized Graphviz [48] for visualisation and NetworkX [49] for complex network analysis. The resulting graphs, using different layout programs, are shown in Figures A.2 and A.3.

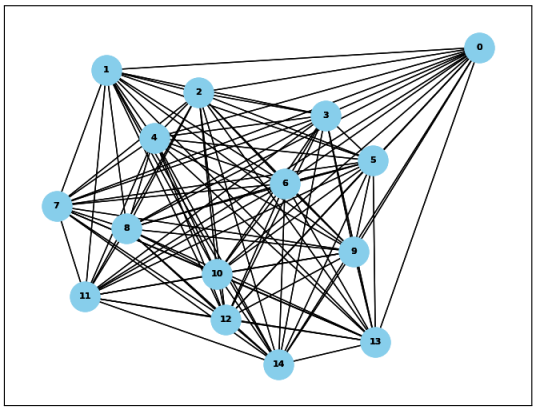


Figure A.2: Graphical Representation of the Optimal Solutions Using 'dot' Program

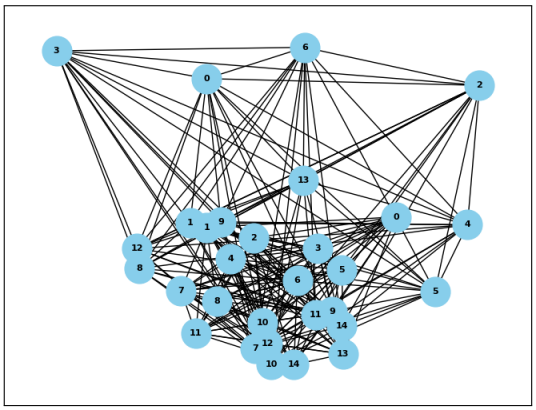


Figure A.3: Graphical Representation of the Optimal Solutions Using 'neato' Program

Figure A.4: Graphs of the Optimal Solutions Using Different Layout Programs

The 'neato' layout (Figure A.3) focuses on maintaining the structural integrity of the graph,

while the 'dot' layout (Figure A.2) emphasizes readability. When we see a graph like this a logical next step is to cluster them for better visualisation.

A.0.3 Dendogram Visualisation

Hierarchical clustering [50] was then applied to group these solutions, visualised using a dendrogram [51]. Dendrograms show relationships with horizontal lines, where the vertical lines indicate the closeness between clusters. This clustering, performed using the scipy.cluster.hierarchy library in Python, is depicted in Figure A.5.

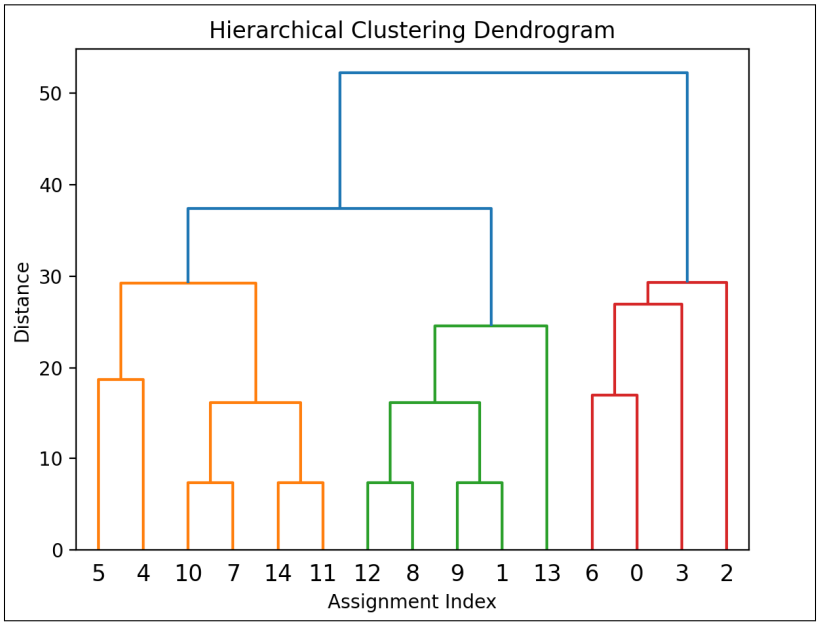


Figure A.5: Dendrogram of Hierarchical Clustering

A.0.4 Plot Visualisation

Finally, I plotted the relationship between the amount of noise and the differences in solutions to assess the impact of noise on solution stability.

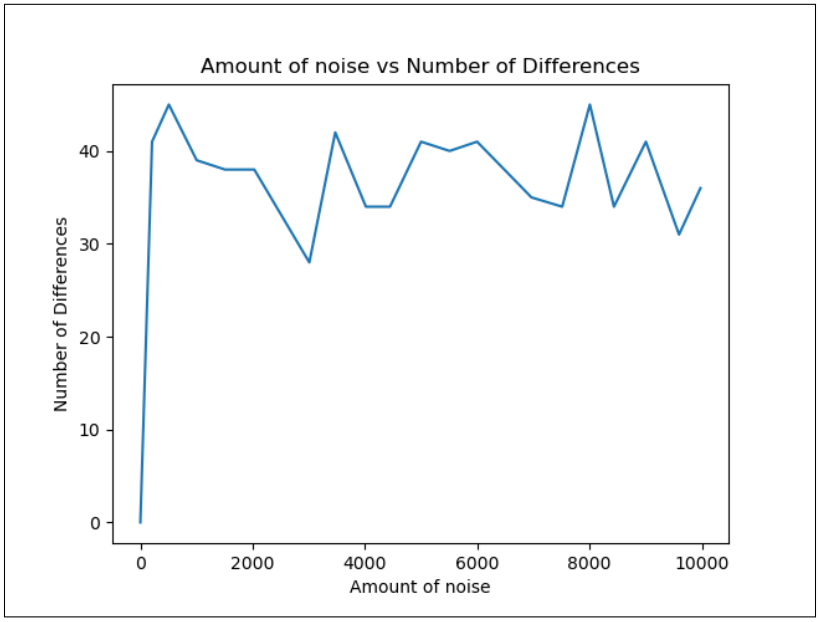


Figure A.6: Number of Differences vs. Amount of Noise

A.0.5 Insights on Visualisation

These visualisations led to several conclusions. Using a 100x100 cost matrix with density 1 and a maximum cost of 100, the heatmap (Figure A.1) indicates that the largest difference is 26, suggesting that approximately 75% of the allocation remains stable despite the added noise. The graph visualisations (Figure A.4) reveal that many solutions are closely grouped, forming three major clusters, which is also evident in the dendrogram (Figure A.5). The `getMoreSolutions` function was executed 21 times, but only 14 unique solutions were identified, indicating the robustness of the allocation process.In addition, the graph in Figure A.6 illustrates that despite varying levels of noise, the number of differences remains within a specific range, highlighting the significant stability in other parts of the solutions.

Further analysis on different types of cost matrices showed that sparser matrices (with more zeros) are more affected by noise, leading to greater differences between solutions and a greater number of solutions. Conversely, matrices with closely clustered values also showed higher sensitivity to noise.

Bibliography

[1] R. Jutaviriya., "A perfect project selection and allocation system for imperial college," 2021.

[2] U. Rawat, *Introduction to hill climbing in artificial intelligence*, Geeks for Geeks, 2023. [Online]. Available: https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/.

[3] D. J. Konec, "Building an accessible project selection system and improving project allocation," 2023.

[4] Microsoft, *What is f#*, 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fsharp/what-is-fsharp.

[5] W. Schools, *C++ introduction*, Refsnes Data. [Online]. Available: https://www.w3schools.com/cpp/cpp_intro.asp#:~:text=C%2B%2B%20is%20one%20of%20the,be%20reused%2C%20lowering%20development%20costs..

[6] Microsoft, *Functional programming concepts*, 2024. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fsharp/tutorials/functional-programming-concepts.

[7] S. Wlaschin, *Why use f#*, 2024. [Online]. Available: https://fsharpforfunandprofit.com/series/why-use-fsharp/.

[8] S. Wlaschin, *Why f#?* FPbridge, 2014. [Online]. Available: https://fpbridge.co.uk/why-fsharp.html.

[9] F. S. Foundation, *Data science with f#*, 2024. [Online]. Available: https://fsharp.org/guides/data-science/.

[10] Microsoft, *Pattern matching*, 2021. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/pattern-matching.

[11] I. Nota, *F#: Seq, list, array, map, set. which one to use?* 2020. [Online]. Available: https://www.itnota.com/fsharp-seq-list-array-map-set-which-one-to-use/.

[12] Microsoft, *Arrays (f#)*, 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/arrays.

[13] Microsoft, *F# collection types*, 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/fsharp-collection-types.

[14] S. Wlaschin, *Choosing between collection functions*, 2015. [Online]. Available: https://fsharpforfunandprofit.com/posts/list-module-functions/s.

[15] A. E. Roth and M. Sotomayor, "Chapter 16 two-sided matching," in ser. Handbook of Game Theory with Economic Applications, vol. 1, Elsevier, 1992, pp. 485–541. DOI: https://doi.org/10.1016/S1574-0005(05)80019-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574000505800190.

[16] D. J. Abraham, R. W. Irving, and D. F. Manlove, "Two algorithms for the student-project allocation problem," *Journal of Discrete Algorithms*, vol. 5, no. 1, pp. 73–90, 2007, ISSN: 1570-8667. DOI: https://doi.org/10.1016/j.jda.2006.03.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570866706000207.

[17] D. F. Manlove, "Hospitals/residents problem," in *Encyclopedia of Algorithms*, M.-Y. Kao, Ed. Boston, MA: Springer US, 2008, pp. 390–394, ISBN: 978-0-387-30162-4. DOI: 10.1007/978-0-387-30162-4_180. [Online]. Available: https://doi.org/10.1007/978-0-387-30162-4_180.

[18] D. Gale and L. S. Shapley, "College admissions and the stability of marriage," *The American Mathematical Monthly*, vol. 69, no. 1, pp. 9–15, 1962, ISSN: 00029890, 19300972. [Online]. Available: http://www.jstor.org/stable/2312726 (visited on 05/18/2024).

[19] "One-sided matching markets," in *Online and Matching-Based Market Design*. Cambridge University Press, 2023, pp. 37–65.

[20] R. E. Burkard and U. Derigs, "The linear sum assignment problem," in *Assignment and Matching Problems: Solution Methods with FORTRAN-Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980, pp. 1–15, ISBN: 978-3-642-51576-7. DOI: 10.1007/978-3-642-51576-7_1. [Online]. Available: https://doi.org/10.1007/978-3-642-51576-7_1.

[21] R. Duan and S. Pettie, "Linear-time approximation for maximum weight matching," *J. ACM*, vol. 61, 1:1–1:23, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:207208641.

[22] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955. DOI: https://doi.org/10.1002/nav.3800020109. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109.

[23] Z. Takhirov, *Hungarian algorithm*, z-scale, 2017. [Online]. Available: http://zafar.cc/2017/7/19/hungarian-algorithm/.

[24] topcoder, *Assignement problem and hungarian algorithm*, 2018. [Online]. Available: https://www.topcoder.com/thrive/articles/Assignment%5C%20Problem%5C%20and%5C%20Hungarian%5C%20Algorithm.

[25] D. P. Bertsekas, "The auction algorithm: A distributed relaxation method for the assignment problem," *Annals of Operations Research*, vol. 14, pp. 105–123, 1988. [Online]. Available: https://api.semanticscholar.org/CorpusID:16476944.

[26] S. Mongkolsr, M. Meyyappan, B. B. Nagarathnam, and A. Narayanan, "Experimental comparison of hungarian and auction algorithms to solve the assignment problem," Texas A&M University, Tech. Rep., 2019.

[27] R. Jonker and T. Volgenant, "A shortest augmenting path algorithm for dense and sparse linear assignment problems," *Computing*, vol. 38, pp. 325–340, 1987. [Online]. Available: https://api.semanticscholar.org/CorpusID:7806079.

[28] M. A. Khan, "A comprehensive study of dijkstra's algorithm," Apr. 2020, Available at SSRN: https://ssrn.com/abstract=4559304 or http://dx.doi.org/10.2139/ssrn.4559304. [Online]. Available: https://ssrn.com/abstract=4559304.

[29] H. N. Gabow and R. E. Tarjan, "Faster scaling algorithms for general graph matching problems," *J. ACM*, vol. 38, no. 4, pp. 815–853, Oct. 1991, ISSN: 0004-5411. DOI: 10.1145/115234.115366. [Online]. Available: https://doi.org/10.1145/115234.115366.

[30] J. E. Hopcroft and R. M. Karp, "A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM J. Comput.*, vol. 2, pp. 225–231, 1971. [Online]. Available: https://api.semanticscholar.org/CorpusID:16337582.

[31] B. Duignan, "Occam's razor," *Encyclopedia Britannica*, Apr. 2024, Accessed 19 May 2024. [Online]. Available: https://www.britannica.com/topic/Occams-razor.

[32] Q. Liu, X. Li, H. Liu, and Z. Guo, "Multi-objective metaheuristics for discrete optimization problems: A review of the state-of-the-art," *Applied Soft Computing*, vol. 93, p. 106 382, 2020, ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2020.106382. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494620303227.

[33] S. Suffian, R. Dzombak, and K. Mehta, "3 - future directions for nonconventional and vernacular material research and applications," in *Nonconventional and Vernacular Construction Materials (Second Edition)*, ser. Woodhead Publishing Series in Civil and Structural Engineering, K. A. Harries and B. Sharma, Eds., Second Edition, Woodhead Publishing, 2020, pp. 63–80, ISBN: 978-0-08-102704-2. DOI: https://doi.org/10.1016/B978-0-08-102704-2.00003-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780081027042000032.

[34] P. Greenstreet, *Weighted sum approach*, https://www.lancaster.ac.uk/stor-i-student-sites/peter-greenstreet/2020/04/24/weighted-sum-approach/, Accessed: 2024-05-22, 2020.

[35] P. Greenstreet, *Issues with weighted sum approach for non-convex sets*, https://www.lancaster.ac.uk/stor-i-student-sites/peter-greenstreet/2020/04/26/issues-with-weighted-sum-approach-for-non-convex-sets/, Accessed: 2024-05-22, 2020.

[36] Merriam-Webster, *Fairness*, Accessed on the 19th of january 2024, 2024. [Online]. Available: https://www.merriam-webster.com/dictionary/fairness.

[37] L. Sauras-Altuzarra and E. W. Weisstein, *Adjacency matrix*, From MathWorld–A Wolfram Web Resource, 2024. [Online]. Available: https://mathworld.wolfram.com/AdjacencyMatrix.html.

[38] L. Ramshaw and R. E. Tarjan, "On minimum-cost assignments in unbalanced bipartite graphs," 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:6964149.

[39] StudySmarter, *Linear interpolation*, 2024. [Online]. Available: https://www.studysmarter.co.uk/explanations/math/statistics/linear-interpolation/.

[40] H. R. Lourenço, O. Martin, and T. Stützle, "Iterated local search: Framework and applications," in *Handbook of Metaheuristics*, ser. International Series in Operations Research & Management Science, M. Gendreau and J.-Y. Potvin, Eds., vol. 146, Kluwer Academic Publishers, 2010, pp. 363–397, ISBN: 978-1-4419-1663-1. DOI: 10.1007/978-1-4419-1665-5_12. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.187.2089.

[41] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, UK: Cambridge University Press, 2004. [Online]. Available: https://web.stanford.edu/class/ee364a/lectures.html.

[42] Eason, *Hungarian algorithm introduction & python implementation*, 2021. [Online]. Available: https://python.plainenglish.io/hungarian-algorithm-introduction-python-implementation-93e7c0890e15.

[43] F. B. Giannasi, *An implementation of the hungarian algorithm in c++*, https://github.com/phoemur/hungarian_algorithm, GitHub repository, GitHub, 2018.

[44] Y. Yang, *Jonker-volgenant /lapjv algorithm for the linear assignment problem, in c language*, https://github.com/yongyanghz/LAPJV-algorithm-c/tree/master, GitHub repository, GitHub, 2016.

[45] I. Abraham, *Benchmarking f# code*, https://www.compositional-it.com/news-blog/benchmarking-f-code/, Compositional IT, 2021.

[46] F. S. Foundation, *Enterprise applications with f#*, 2024. [Online]. Available: https://fsharp.org/guides/enterprise/.

[47] Vanshgaur, *How to draw 2d heatmap using matplotlib in python?* https://www.geeksforgeeks.org/how-to-draw-2d-heatmap-using-matplotlib-in-python/, Geeks For Geeks, 2023.

[48] S. Bank, *Graphviz : User guide*, https://graphviz.readthedocs.io/en/stable/manual.html, 2022.

[49] R. M. Perera, *Networkx visualization with graphviz (example)*, https://www.malinga.me/networkx-visualization-with-graphviz-example/, 2020.

[50] F. Karabiber, *Hierarchical clustering*, LearnDataSci. [Online]. Available: https://www.learndatasci.com/glossary/hierarchical-clustering/#:~:text=a%20real%20dataset-,What%20is%20Hierarchical%20Clustering%3F,hierarchical%20tree%20called%20a%20dendrogram..

[51] T. Bock, *What is a dendrogram?* https://www.displayr.com/what-is-dendrogram/, LearnDataSci.