Development and implementation of a calendar scheduling algorithm

By:

Saumya Singh

Student ID: 34302735

Supervisor:

Prof. Toni Martinez Sykora

A dissertation submitted in partial fulfillment of the degree

MSc Data and Decision Analytics

ERGO ID: 83165

Faculty of Social Sciences

School of Mathematical Sciences

University of Southampton

Word Count: 17,300

September 2023

Statement of Originality

This project is entirely the original work of student Saumya Singh with registration number 34302735 (student ID number). I declare that this dissertation is my own work, and that where material is obtained from published or unpublished works, this has been fully acknowledged in the references. This dissertation may include material of my own work from a research proposal that has been previously submitted for assessment for this programme.

Saumya Singh

Acknowledgement

I would like to take this opportunity to thank my university supervisor Prof. Toni Martinez Sykora and my industry supervisor Dr. James Flynn for their immense support and guidance throughout the research. I am truly grateful for the valuable insights and feedback that they have provided at each step of this study, propelling it to achieve its objectives. I thank Prof. Toni for introducing me to new novel methods and helping me structure my model and Dr. James for always bringing new ideas of enhancing the model and the work.

I am thankful to the University of Southampton and my industrial sponsor the Ford Motor Company for giving me the opportunity to work on this meaningful project. I learned a lot during this work and realised the importance that this study holds in the corporate world. I am glad I was able to contribute to the efforts of improving the work environment for employees.

I would also like to thank my friends and family for their constant love and support throughout this journey. This journey would have been incomplete without them. A special thanks to Priydarshi for his help and encouragement.

Sincerely,
Saumya Singh

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

This study presents a novel approach to corporate meeting scheduling, comparing an automated algorithm with manual methods. Leveraging greedy heuristics within resource-constrained environments, the algorithm surpassed manual efforts in efficiency, cost and time. It uniquely integrates individual preferences and organisational constraints, resulting in superior solution calendars developed in half the time. Cost analysis reveals significant future savings. Beyond financial benefits, this approach boosts productivity, satisfaction, and overall success. While promising, opportunities for algorithmic improvement remain, including adapting to dynamic constraints, faster scheduling, advanced AI integration, and enhanced user interfaces. Incorporating real-time data and exploring machine learning hybridisation are areas for further advancement, ensuring the algorithm stays aligned with evolving organisational needs.

Chapter 1

Introduction

In the fast-paced corporate world, creating efficient calendars for employees is not only about optimising time management but it is also a strategic imperative. Work calendars can have a significant impact on the company's productivity, employees' satisfaction and overall success. In this study, we have taken up the challenge to design and implement a new and efficient calendar scheduling approach that addresses the complexity of satisfying various organisational constraints while also accounting for employee preferences.

1.1 Scheduling Problems

Scheduling problems are the core of various real world challenges, from project management and automated manufacturing to transportation and employee timetables. The fundamental goal of these problems is to allocate limited resources such as time, personnel, machines or other facilities to meet certain goals, minimise costs, or maximise productivity. Scheduling problems have various variations, each having a unique set of constraints and objectives. Whether the task is to determine the most efficient machine job sequence in a factory, scheduling flights to reduce clashes and air traffic or organising work shifts to satisfy employee preferences, scheduling problems play a pivotal role in decision making.

Many challenges arise while handling scheduling problems. They are required to take into consideration factors like the duration of the task, availability of resources, deadlines, precedence relationships and other conflicting objectives. The process of finding the best schedule can be computationally demanding, where alternatives algorithms and heuristics are used to arrive at feasible solutions. The quest of finding optimal solutions has driven many research and advancements in computer science, operational research and artificial intelligence. Although this research topic has expanded exponentially in the past few years, we notice that some areas are still less explored.

The driving force behind our work is a recognition of a significant gap in the existing literature—a gap where meeting scheduling in a corporate setting, considering individual preferences and organisational constraints, lacked a comprehensive and practical solution. The corporate meeting scheduling is structurally different from the other variants of scheduling problems. Drawing inspiration from this void, our research endeavours set out to bridge this gap, introducing a new hybrid approach that not only caters to these essential considerations but also prioritises cost-effectiveness, ease-of-use and intuitive visualisation, within a reasonable time complexity.

1.2 Corporate Meeting Scheduling

For a multinational company, with teams and departments spread across different time zones and locations, the task of efficiently scheduling meetings becomes increasingly complex. Scheduling methods are particularly vital in large organisations where it can be challenging to coordinate between numerous employees with varying schedules and priorities. The focus is not only on scheduling meetings to achieve a organisational aims but also on personalisation, dynamic scheduling and creating user-friendly interfaces as briefed below:

1. **Personalisation** refers to considering individual employee preferences and constraints, such as preferred meeting times or the need for breaks between meetings.

2. **Dynamic Scheduling** refers to creating algorithms that can adapt to real time changes, accommodating any last-minute meeting requests or cancellations.

3. **User-friendly Interface** to create a user-centric intuitive interface and allow any employees from non-technical background to easily input their availability and preferences.

Corporate meeting scheduling juggles between multiple variables such as the availability of employees, priority of meetings, and incorporating various constraints like different time zones, meeting breaks, and focus times. We delve into the evolution of traditional manual scheduling methods to the modern algorithmic solutions, showcasing innovative approaches that have revolutionised how businesses organise and manage their meetings. By understanding these methods, organisations can better navigate through the corporate meeting scheduling task, promoting effective collaboration and driving overall success.

1.3 Thesis Structure

The thesis is structured as follows: In chapter 2, we introduce the problem, giving a more detailed overview of the challenges faced while scheduling meetings. A literature review of relevant works is also presented with the summary of key findings as well as how this influences our chosen approach. In chapter 3, we describe the problem in more detail while also identifying the various constraints and preferences of this specific study. Chapter 4 presents the existing standard methods that are frequently used to solve other variants of the calendar scheduling problem and how we draw inspiration from it. In chapter 5, we detail the design and working of the final method to solve CSP, which is supported by probabilistic reasoning. Chapter 6 deals with the descriptive statistics and analysis of the real world data that was provided by the Ford Motor Company, while in chapter 7 we implement the designed algorithm on the real data. The results from manual method and automated method are scored and compared, along with the final results. A cost effective analysis is also presented at the end of chapter 7. Finally, in chapter 8, we discuss the concluding remarks and the scope for the future.

Chapter 2

Background and Literature Review

In our day to day life, we often go about setting up a meeting by following a few simple steps: creating a meeting as per our calendar preference, sending out the invites to the attendees and finally waiting for them to accept the invite. In many cases, the attendees would request to move the meeting to another slot, in which case we reschedule the meeting to our next best choice of time slot. We keep repeating this process until everyone is satisfied. Let us imagine a case where we have to schedule a hundred meetings for more than two hundred people. The invitation system is going to be troublesome. In that case, iterating over the meeting time slots by asking everyone to accept or reject an invitation is going to be very inefficient. This gives rise to a new branch of scheduling problems known as the Calendar Scheduling Problem (CSP).

2.1 Calendar Scheduling Problem

The Calendar Scheduling Problem is based on scheduling a large number of meetings for a large group of employees, to maximise the number of meetings scheduled while also considering individual preferences. A simpler method than the invitation system is to go through the attendees' calendars, find a common free slot and schedule a meeting assuming that the attendees would not have major subjective preference issues. While this may sound doable, it has some major drawbacks. For instance, not deciding the time of meeting optimally may lead to a lot of clashes or unscheduled meetings. It is also difficult to incorporate personal preferences of employees.

The Ford Motor Company has been facing various similar issues in handling and scheduling meetings owing to a large number of meetings that need to be scheduled, a huge workforce of employees who need to attend different meetings, and the necessity to create a balanced calendar which allows for ideal breaks and focused work times. The CSP becomes more complex and detailed when we try to allow greater flexibility such as the different time zones of employees all over the world, priorities of different meetings, seniority spectrum of employees, etc. all of whom want to schedule meetings that are feasible and efficient.

At Ford, the information about frequently occurring meetings is currently being stored in MS Excel (Singh, 2023) with the following attributes: the meeting name, the duration of the meeting, its frequency, the owner of the meeting, the priority of the meeting (optional) and the required employees. The required employees are normally stored as names of the work teams which are later mapped to the actual employees. The calendars of all the workers are

securely stored and are available to look at through MS Outlook. The Ford Motor Company routinely schedules its recurrent meetings for employees four weeks ahead and these are updated at the end of every four weeks. Apart from recurrent meetings, they can also add meetings in the middle of the week if needed.

2.1.1 Current Method

Currently, meetings at The Ford Motor Company are scheduled manually via MS Excel. An employee, responsible for scheduling recurring meetings, goes through the meeting information and the schedules of each required employee and tries to efficiently place the meetings in the common free slots. The scheduler also needs to keep in mind other factors such as the frequency, the duration and the priority of the meetings. When the calendars become more and more busy, the scheduler has to decide between clashing meetings, while attempting to create a calendar with minimum clashes and maximum attendees, through human judgement and trial-and-error. Later, if there is a need for another meeting to be scheduled, the meeting owner would have to go through the busy calendars of employees to find preferred common slots.

The Drawbacks

The above mentioned method suffers from several major disadvantages, as listed below.

1. **Manual effort**: The most important drawback is the tedious manual work the person scheduling the meeting or the meeting owner has to do, in order to schedule a meeting. They mostly rely on trial-and-error and human judgement. This task becomes increasingly complicated with the increasing number of meetings, employees, and busy calendars, even without considering the attributes of the meetings. Even after a huge amount of manual effort, an optimal or near-optimal solution is not guaranteed.

2. **Lack of sufficient breaks**: While scheduling meetings, the goal is to always maximise the number of meetings scheduled with no or minimum clashes. This may lead to employees having little to no breaks, or more importantly, lack of ideal lunch breaks. In the long run, such a schedule can affect an employee's motivation in the work and the company.

3. **Imbalanced calendars**: While manually scheduling the meetings, it is practically very difficult to ensure that every employee has a well balanced calendar. As a result, usually such considerations are done only for a few selected employees due to their seniority or importance. This can lead to two major drawbacks:

   (a) Scattered meetings: For some employees, the meetings may not have been clustered efficiently, causing irregularly alternating meetings and breaks. This affects the focus time for work, possibly creating inefficiencies in the work environment.

   (b) Imbalanced weekdays: It is highly probable that a few employees may have extremely busy calendars on a few days of the week, while the other days of the week are completely vacant. While some employees may prefer this type of calendar, others may not. It comes down to taking into account the personal preferences of each employee.

4. **High costs for tweaking**: Reproducing the calendars with different or updated objectives will again require the same amount of work. It can become very cumbersome, restraining repeated changes in the objectives of a fast-paced work environment.

The Objectives

The main objective of this work is to create an easy-to-use algorithm that addresses the issues outlined above, and presents a more efficient and practical approach to calendar scheduling. The algorithm should fulfill the following objectives:

Primary Objectives:

- Maximise the number of scheduled meetings
- Minimise the number of meeting clashes

Secondary Objectives:

- Create a balanced calendar for each employee
- Satisfy various scheduling constraints
- Maximize the total priority score
- Accommodate different constraints for senior employees

2.1.2 Calendar Software Application

While there are a plethora of apps available online that ease the process of scheduling meetings, but we could not find any of them that actually solve the efficient calendar scheduling problem, let alone customising objectives and indicating preferences. We then look towards borrowing ideas from the history of scheduling problems to understand how they are tackled, and to find out if there are similar works already done that can be used and built upon. The following section goes deeper into the relevant literature review. There can be many ways for solving this problem and we will try to construct and present some of them.

2.2 Literature Review

The task of efficiently assigning work shifts to employees at different time slots has been around for years and it is a well researched area. The methods used to solve these problems also take into account, as much as possible, the constraints of each person and the ideal ways of completing the overall task. However, in many cases a fully optimal solution is not feasible, and the solution demands flexibility. Hence, these methods are often designed using a penalty based approach. Each new problem would pose new limitations and require new approaches to be designed. Some of the most famous problem categories are the Nurse Rostering Problems (Powell, 1974) and the Travelling Salesman problems (Dantzig et al., 1954).

One of the preferred choice of algorithms for practitioners is the heuristics based algorithms, which offer benefits in terms of resources required, ease of making changes, and providing near optimal solutions reasonably quickly (Vazirani, 2013). In recent years, the increase in the availability of resources and fast processing units has provided momentum to the discovery of many new algorithms, such as machine learning based approaches. However, training the machine learning models is still an expensive process, and doing it repeatedly, to make changes, takes time and uses significant computational power which hinders its practical implementations. Thus, heuristics based algorithms remain a popular choice in this problem space.

2.2.1 Shift Scheduling Problems

Burke et al. (1999) used a hybrid Tabu Search algorithm with heuristics for Nurse Rostering Problem in Belgian Hospitals. The problem consisted of various hard constraints, i.e., the mandatory requirements, and the soft constraints, i.e., preferences. The hard constraints would, for example, ensure that a nurse should not be scheduled for two shifts in a row or guarantee a minimum rest period. Soft constraints would take into account a nurse's preferred off-day or minimise the number of night shifts for a particular nurse. They showed that the quality of automatically produced schedules was much better than the manually produced schedules. The hybrid algorithm using heuristics was also demonstrated to outperform the normal Tabu Search and the Steepest Descent algorithm. This supports our premise that an automatic calendar scheduling algorithm performs better in quality and productivity than manually scheduled calendars. It further hints that encoding the hard and soft constraints may provide a good starting point for our problem.

Musliu et al. (2004) presented a local search algorithm with Tabu search for shift design problems. It was based on exploring local neighbourhood search space instead of the whole search space at each iteration/move of Tabu search. Additionally, the authors proposed an algorithm for generating a "good" initial solution based on knowledge about requirements and shift structure; and how the knowledge about the problem can significantly improve the solution when combined with classical approaches. After proving that the shift design problem is NP-hard and difficult to approximate, they proposed that local search techniques could be an effective approach to solving it. This was tested to be successful in practical cases. This suggests that it may help us improve our results if we follow this presumption of designing a similar hybrid model with good knowledge about the ideal meeting structure and schedule requirements. For the current study, we stick with exploring the entire search space at each iteration.

Van Den Bergh et al. (2013) did a comprehensive literature review on personnel scheduling problems. The authors classified the problems based on the labour contract (part-time or full-time), a variety of constraints (flexibility in shift length, start time, etc), and the methods used to solve these problems. The three major categories of techniques identified were Mathematical Programming Models, Heuristics and Metaheuristics approaches, and Decomposition methods, with the literature being heavily skewed towards the former methods (Van Den Bergh et al., 2013). The authors emphasised accounting for uncertainty in the algorithm, doing a thorough comparison between the methods, and implementing it on real world data, which has been lacking in various journals. Keeping this emphasis in mind, we aim to present a complete picture with a detailed comparison and robustness check of our new algorithm and do an implementation using real-world data for the novel approach.

Chu (2007) proposed a Goal Programming model with decomposition for crew duties assignment in the baggage services section of the Hong Kong International Airport. The models were divided into two phases: the duties generating phase, and the scheduling and rostering phase. The results of the models provided feasible crew schedules that minimised idle shifts. Randomly generated problem instances were used to test the models' performance, and the results showed that the GP models consistently provide efficient and feasible crew schedules. A minimax time-reversible heuristics with randomisation was shown to achieve optimal results in most cases and provides an effective solution to the crew assignment problem that can be easily applied to other workforce planning and scheduling problems. Following a similar pattern, we utilize the analogous randomisation and decomposition techniques for scheduling recurrent and ad-hoc meetings in our scheduling problem.

Özder et al. (2020) in their systematic review of personnel scheduling problems identify the main goals of personnel scheduling, which include minimising personnel cost, balanced workload distribution, and meeting individual needs. The authors explore various areas where scheduling problems were researched in recent years but we observe that none of them tackle meeting scheduling problems in the corporate setting. Creating efficient meeting schedules for various employees, to maximise scheduled meetings and minimise clashes, has a fundamentally different structure than shift designing, crew planning, or vehicle routing and thus demands that a new field be explored and researched upon. In our work, we learn from the already investigated works and design a novel algorithm to implement in an unexplored setting.

2.2.2 Heuristics Approaches

Heuristic methods are problem solving techniques that prioritise speed over accuracy. They are frequently used for problems which do not have deterministic polynomial-time solutions. Instead of finding the perfect solution, the heuristic algorithm aims at finding a satisfactory solution within a reasonable time frame. For this, they use various shortcuts, rule-of-thumbs, and trial-and-error methods to obtain good enough solutions for complex optimization problems. These algorithms also provide simplicity and interpretability in comparison to their wholly-deterministic counterparts, and have a greater flexibility to adapt to different types of problems. Heuristics algorithms are suitable for finding good initial solutions in resource deficit environments. Several heuristics algorithms have been developed and presented to serve specific needs. Within this research, we explore some of the standard heuristic approaches and construct a new greedy heuristic method.

CDS Heuristics

Clarke and Wright (1964) proposed a heuristic method to solve vehicle routing problems with an aim to find an efficient set of routes for a fleet of vehicles to deliver goods to a number of customers while minimising the total distance travelled. At its core, the CDS heuristic calculates "savings" associated with merging two routes into one, instead of having two separate routes. The "saving" is essentially the difference between the distance of two separate routes and the distance when they are merged. The routes with the highest savings are merged first, and this process continues iteratively until no more significant savings can be achieved or other constraints (like vehicle capacity) are met. It provides a good approximate solution in a short time. We draw inspiration from this approach and play around with the decision variables, for example, to decide whether it is more beneficial to schedule a meeting with full attendees or partial attendees.

NEH Heuristics

Nawaz et al. (1983) introduced a heuristics based algorithm that seeks to sequence jobs in a multi-machine environment where each job needs to be processed on each machine in the same order. It has since then been widely used in flow shop scheduling problems. NEH heuristics focus on balancing the solution quality and computational efficiency, generating near-optimal solutions with significantly less computational power as compared to exhaustive methods. It is based on the intuitive idea that jobs having greater total processing time on all the machines should be prioritised and placed as early in the sequence of jobs as possible. As more jobs are added, the heuristics evaluate various sequences and ensure the minimization of the makespan at every step. In this research, we also present

sorting the meetings by their duration and prioritise scheduling longer meetings. We compare the ordered meeting sequence with the randomised meeting-order results, to support the idea that instead of searching the whole search space, we can generate good enough solutions with heuristics.

Construction Heuristics

Construction Heuristics is a type of heuristic algorithm that computes the initial best solution for combinatorial optimization problems, which might not be feasible computationally, by iteratively adding new components to a partial solution until a complete solution is built. These methods tend to be fast and easy to implement. The various subcategories are discussed below.

- **Classical Construction Heuristics:** The three most commonly applied classical construction heuristics:

    1. **Greedy Algorithms**: It is based on the idea of adding the next component to the partial solution that yields the best immediate benefit or partial solution (Dijkstra, 1959). This approach will be exploited the most in this work.
    2. **Bin-packing problems**: The heuristics used to solve bin packing problem (Karmarkar & Karp, 1982) are based on ordering the items and placing them inside the bins according to certain criteria. For example, the First Fit places the item in the first bin that has enough space whereas the Best Fit tries to place the item in the bin which is most populated but has enough space. Both offer the benefit of minimising the number of used bins (Fischer, 1973). In this study, we also suggest another bin packing heuristics for a balanced item distribution.
    3. **Nearest Neighbour**: Commonly used in travelling salesman problems, it is a type of greedy algorithm that works by constructing a tour by always visiting the nearest unvisited slot, that is, the nearest unvisited neighbour next (Rosenkrantz et al., 1977). This gives an idea for designing a cluster enforcing methods for unused slots.

- **Advanced Construction Heuristics:** A few popular modern construction heuristics are:

    1. **Savings Heuristics**: The CDS heuristics (Clarke & Wright, 1964) formed the foundation of savings heuristics, based on the intuition of "saving" distance, cost or penalty score, widely used in graph problems (section 3.2.2).
    2. **Randomised Heuristics**: Randomisation is introduced in heuristic algorithms to avoid getting stuck in local optima (Feo & Resende, 1989). This may give rise to suboptimal solutions, but running it multiple times with varying entropy improves the results drastically. Randomisation is heavily utilised in our work to produce robust results.
    3. **Bio-inspired Heuristics**: Algorithms such as the Ant Colony Optimisation (Dorigo & Di, 2003) and the Genetic Algorithms (Goldberg, 2002) are approaches that are inspired by the natural phenomenon and propose a new era of advanced heuristic methods. Trying and testing such algorithms are left for future exploration.

2.2.3 Summary of key findings

Through the literature and background review, we observe that it is difficult to find a research study on scheduling methods that has been focused on and applied to solving calendar scheduling problems of meetings in a corporate setting. Most of the past research is build around the standard shift scheduling, graph traversal or machine management problems. The meeting scheduling problem is neither a subset nor similar to the nurse rostering problem, the travelling salesman problem or the flowshop problem. We note some major differences as follows:

- Difference from Nurse Rostering Problem (NRP): The basic structure of NRP involves assigning shifts to nurses while also considering their personal preferences. Only a fixed number of nurses are assigned to a shift and there is no hard constraint on which nurse should be assigned to which ward. In meeting scheduling, however, we have a variable number of meeting attendees that are required for a particular meeting to occur and there is a hard constraint between a meeting and the required attendees/employees.

- Difference from Travelling Salesman Problem (TSP): TSP and vehicle routing problems are majorly graph traversal problems that intend to minimise the total distance travelled or cost while also delivering the required products. Each node in the graph is free to be connected to any other node to create a specific route. While in our case, the meetings (nodes) cannot be connected freely to any other meeting since they are constrained to occur at the same time and day for all the attendees (route) and each attendee would have a different set of meetings to attend.

- Difference from Flow Shop Problem (FSP): If we consider meetings as jobs and attendees as machines of the flowshop problem, as for the case of scheduling meetings, each meeting (job) would require a certain number of attendees (machines) simultaneously. Thus making it fundamentally different from the flow shop problem where a job is completed by machines in an ordered manner one-by-one.

Since the meeting scheduling problems differ structurally from other traditional scheduling problems, it requires new approaches. We borrow knowledge from the previous techniques to design and implement new heuristics dedicated to solving calendar scheduling problems. We frequently follow the footsteps of various researchers to present a complete analysis of the problem and its solution. Lack of ample exploration on the current meeting scheduling problem makes this study novel in several aspects and we hope to expand it to various other sectors facing similar issues.

Chapter 3

Problem Description and Constraints Identification

Before we go on to try and test certain methods for our scheduling problem, it is important to classify the problem and understand the requirements in as much detail as possible. In this chapter, we discuss the details of the meetings and the restrictions while scheduling them. Every meeting has unique features in terms of its type, priority, and attendees. We present these features and how they influence the scheduling decisions. Along with the meeting attributes, the role of an employee and the various constraints also impact the assignment of a meeting in a calendar, as explained in this section.

3.1 Meeting Clusters

One of the major goals of this work is to do efficient clustering of meetings. A meeting cluster is defined as a group of consecutive meetings that occur one after another without breaks in between. A cluster cannot span for two or more days since the clusters break at the end of the day. The maximum length of the meeting cluster is, thus, equal to the maximum length of the office hours for a day (e.g. 9 hours), meaning the whole day would be packed with continuous meetings without any break.

3.2 Meeting Types

The meetings are divided into two major categories: Recurrent meetings and Ad-hoc meetings. Recurrent meetings are predefined and have a frequency of recurrence. This frequency is defined for four weeks.

$$\texttt{meeting frequency} = \frac{\texttt{number of times the meeting occurs in 4 weeks}}{4}$$

For example, a meeting with frequency 1 occurs every week for four weeks while a meeting with frequency 0.5 is a bi-weekly meeting. Recurrent meetings are generally scheduled on fairly vacant calendars of employees during the start of the month and the goal is to make a meeting calendar for each employee that is balanced and satisfies a number of objective constraints. On the other hand, ad-hoc meetings are set up as needed and are not pre-planned. These meetings need to be scheduled efficiently on generally busy calendars in the middle of the month. The

ad-hoc meeting owners are responsible for choosing among the free and feasible slots (found by the algorithm or manually) according to their preference.

3.3 Meeting Constraints

We identified a few major constraints to meet the needs of Ford Motor Company through periodic discussions with the employees. Some of these were critical and compulsory constraints (hard constraints that must be satisfied all the time) and some were recognised as preferable conditions (soft constraints that want to be satisfied if the costs are not too large).

3.3.1 Hard Constraints

A hard constraint is one that "must" be satisfied all the time for a particular meeting to be scheduled.

1. **Meeting Owner**: The owner of the meeting should always be present for the meeting to occur. Otherwise, we do not schedule that meeting.

2. **Threshold Attendees**: The percentage of attendees should be above a threshold percentage, for example, at least 80% of attendees should be present for the meeting to be scheduled.

3. **Consistent Day and Time**: The recurring meetings should be set up on the same day and time throughout the four weeks (or the month). For example, if a weekly meeting M1 is initially scheduled on Tuesday at 12 p.m. for week 1, it must occur on Tuesday at 12 p.m. for weeks 2, 3 and 4 as well. If we do not find such a common slot, the meeting is left unscheduled.

4. **Uniform Frequency Distribution**: The meetings should be uniformly distributed throughout four weeks according to its frequency. For instance, a meeting with a frequency of 0.5 is a biweekly meeting, so it should be scheduled in either week 1 and week 3 or week 2 and week 4. It can not be scheduled for two consecutive weeks.

5. **Partial Scheduling**: A meeting is scheduled only when its recurring frequency is fully satisfied and all its recurring meetings have found the same free time slots. Even if one of the meetings cannot be scheduled in any of the required weeks, the whole meeting type is left unscheduled.

3.3.2 Soft Constraints

A soft constraint is one that we want to be satisfied if the cost for having it is not too large.

1. **Start of the first Meeting Cluster**: We prefer the meetings to start as early as possible in the morning. The earliest is the start of office hours at Ford offices, which is 8 AM.

2. **Length of a Meeting Cluster**: We want to ideally maintain a meeting cluster length, that is, the duration of consecutive meetings to three hours to boost productivity and efficiency.

3. **Number of Meeting Clusters**: We want meetings to be more clustered than scattered to make space for more focused work times. Hence, we try to minimise the total number of meeting clusters.

4. **Presence of Lunch Break**: Each employee would be given at least one break in the afternoon for lunch, which should be preferably placed between 12 PM to 2 PM.

5. **Finishing Early of the last Meeting Cluster**: As we would want the meetings to start early, we also want them to finish early, practically by 3 PM, and not stretch till the end of the office hours.

3.3.3 Other Constraints

Apart from the hard and soft constraints of the algorithm, there are also other productivity constraints, which will be implemented in the algorithm based on requirement and feasibility. These constraints will be used to produce variation in the results.

1. **No-Meeting Friday**: Sometimes to enhance productivity, companies implement something called no meeting Friday. As the name suggests, all the meetings are organised in the first four days of the week and Friday is left for work only. This can be designed as a hard constraint where we can remove Friday from the vacant calendar and enforce no meeting on Friday. It can also be devised as a soft constraint, scheduling most of the meetings in the first four days but allowing only some of them to be set up on Friday. However, in this work, we will follow a strict no-meeting Friday rule.

2. **No Meeting Afternoon**: Similar to the objective of no meeting on Friday, some individuals may prefer distributed meetings scheduled every day before lunch and after lunch time becomes their focus time. While the algorithm will already impose the soft constraint of finishing early, no meetings in the afternoon is a hard constraint. This constraint is feasible only when the number of meetings is not too much or it may result in many unscheduled meetings.

3.4 Meeting Priority

An additional feature of meetings is their priority. A priority value is assigned to each meeting based on how preferred it is to be scheduled before other meetings. Factors affecting this value are the number of senior employees present and the objective of the meeting. In most cases, these values will be decided by the company. It is one of our secondary objectives to maximise the priority score. Priority value will also help in breaking ties between two meetings having exactly the same features. Meetings with higher priority values are preferred and scheduled first. To produce variations in the results, we will also change the maximising priority score into a primary objective and compare it with other results.

3.5 Senior Employees

Normally we would expect the calendars of senior employees to look significantly different from normal employees. In general, senior employees tend to organise and attend more meetings as compared to junior employees. It is again one of our secondary objectives to prioritise a better calendar organisation for senior employees. Therefore, we will tighten or relax the soft constraints for senior employees. For example, we will tighten the constraint for the presence of a lunch break and relax the constraint for the number of meeting clusters. Employees will be categorised into two classes: senior and not senior. While the seniority level is a wide spectrum, catering for each section of the spectrum poses new challenges and thus will be left for future work.

3.6 Balancing Calendars

Creating balanced calendars for each employee is one of our primary objectives, however, there is a trade off between allowing a no meeting day and having a balanced calendar throughout the week. This trade off will be built-in into the algorithm, which would prioritise keeping a no meeting day when the number of meetings is manageable otherwise it would prioritise keeping balanced week days when the number of meetings is large. More preference will be given to balancing the calendars of senior employees than normal employees.

3.7 Reproducibility

We have discussed a number of identified constraints, variations for the results and different primary and secondary objectives for the calendar scheduling problem at the Ford Motor Company in the United Kingdom. The sheer numerosity and the wide domain of the constraints hint at the subjectivity of the requirements which will quickly evolve with time. Therefore, it is important for the algorithm to be user friendly and easily reproducible, to readily account for any new constraint, variation, or objective. For that reason, we will stick with object oriented programming in Python wherever possible and read and store most of the data in MS Excel files that can be comfortably changed according to the needs.

Chapter 4

Existing Methods

After we have classified and understood the specific scheduling problem in detail, we test some of the standard approaches used to solve such problem. The literature review suggests that a direct research reference to the corporate meeting scheduling problem may be difficult to find, however, the CSP can be related to other variants of scheduling problem to find similarities. These similarities shape the initial direction of design and implementation of a new algorithm. Within the scope of this research, it is not feasible to test a wide range of applicable models. Therefore, based on the findings from the literature review, our approach is to implement various heuristic methods, used in bin packing problem style, to achieve our current objective. Later, we create different variant of the heuristic method and expand the basic algorithms, based on an example case.

Ethical approval

Ethics approval for this research was granted, with ERGO approval number 83165.

4.1 Slot Assignment

For ease of design and algorithm simplicity, we convert the meeting duration into slots. Initially, each 30 minutes of a working day is considered as one single slot. Even if a slot is partially occupied, the whole slot is regarded as busy. A meeting lasting between 1 minute to 30 minutes will occupy a single slot, and if a meeting is 50 minutes long, we assign two slots to the meeting. It is assumed that either a slot is occupied or it is unoccupied but it cannot be partially occupied. We identify that this may not be the optimal way to divide the day into slots, and hence, more granular slot assignments will be done later, where every 5 minutes will be considered as a slot. This is just an example of how our algorithm will be designed to be easily customisable as per the needs of the organisation.

4.2 Bin Packing Problem

The bin packing problem is a type of optimization problem, where the goal is to fit items of different sizes into a minimum number of bins of fixed capacity. Analogously for calendar scheduling, meetings of different lengths are taken as items and the days are considered as the bins of fixed length. There are several heuristic methods designed to solve the bin packing problem. We test two of the standard heuristics approaches on the corporate meeting scheduling problem and suggest some variations to satisfy slightly different objectives. We probe the first fit and the best fit heuristic approaches with the goal to utilise the minimum number of days (bins) of a calendar for

meetings, ideal for enforcing constraints such as the no-meeting Friday. Furthermore, we construct a new balanced bin heuristic method that is well suited for packing the items uniformly across bins, producing balanced calendars through distributed meeting scheduling.

4.2.1 First Fit, Best Fit and Balanced Heuristics

Items are sorted in descending order of meeting length or duration measured in minutes. The First Fit Decreasing algorithm works by placing the next item in the first bin that has enough space to accommodate it. While it may give non optimal solutions for the naive descending order sorting, it is known that there exists at least one ordering of items when the first fit generates the ideal solution. On the other hand, the Best Fit Decreasing algorithm works by placing the new item into the bin that is most occupied or populated and is simultaneously free enough for the new item. Both these standard heuristics minimise the number of bins used. We learn from the best fit heuristics and create our balanced bin packing algorithm that uses a fixed number of bins and works by placing the next meeting (item) into the day (bin) that is least occupied. Since the items are already sorted, this ensures a balanced distribution of items across the bins.

Implementation and Results: Analogous to the bin packing problem, we use heuristic algorithms to place meeting items into the calendar bins. These are implemented for an example consisting of 10 meetings of different lengths and priority scores (table 4.1) that need to be scheduled over a week of five working days. For better visualisation, the bin size is trimmed down to 180 minutes (figure 4.1) and meeting IDs uniquely identifying each meeting is mentioned on the respective bars. The meeting data is read in Python from an Excel sheet. For the initial testing, we do not apply any constraints and identify if the bin packing algorithm is worth exploring for the current problem.

Let *'I'* be the set of n meetings that need to be scheduled. Each item I_i of set *'I'* has three attributes: a unique identifier Meeting ID, the duration of the meeting in minutes and a priority value of the meeting. Let *'B'* be the set of days (bins) available for the scheduling, which is five in our case. The algorithms are defined as follows:

Steps for First Fit Decreasing Algorithm:

- **Step 1** Sort the meetings in **non increasing** order based on the meeting length. After the ordering, we say, without loss of generality, that $I_1 \geq I_2 \geq I_3 \geq \cdots \geq I_n$ duration wise.
- **Step 2** Let $i = 1$ and start an incremental loop, running until $i = n$.
- **Step 3** Select an item I_i from the sorted list of meetings.
- **Step 4** Loop over the days and schedule the meeting I_i in the first bin that has enough space to accommodate it. If enough free space is not found, leave the meeting I_i unscheduled.
- **Step 5** Do $i = i + 1$. Exit loop when $i > n$.

Steps for Best Fit Decreasing Algorithm:

- **Step 1** Sort the meetings in **non increasing** order based on the meeting length. After the ordering, we say, without loss of generality, that $I_1 \geq I_2 \geq I_3 \geq \cdots \geq I_n$ duration wise.
- **Step 2** Let $i = 1$ and start an incremental loop, running until $i = n$.
- **Step 3** Select an item I_i from the sorted list of meetings.

Step 4 Sort the days or bins in **non increasing** order according to the space already occupied in the day. So the busiest day would be the first bin and the least busy day would be the last bin.

Step 5 Loop over the sorted days and schedule the meeting I_i in the first bin that has enough space to accommodate it. If enough free space is not found, leave the meeting I_i unscheduled.

Step 6 Do $i = i+1$. Exit loop when $i > n$.

Steps for Balanced Heuristic Algorithm:

Step 1 Sort the meetings in **non increasing** order based on the meeting length. After the ordering, we say, without loss of generality, that $I_1 \geq I_2 \geq I_3 \geq \cdots \geq I_n$ duration wise.

Step 2 Let $i = 1$ and start an incremental loop, running until $i = n$.

Step 3 Select an item I_i from the sorted list of meetings.

Step 4 Sort the days in **non decreasing** order according to the space already occupied in the day. So the least busy day would be the first bin, and the busiest day would be the last bin.

Step 5 Loop over the sorted bins and schedule the meeting I_i in the first bin that has enough space to accommodate it. If enough free space is not found, leave the meeting I_i unscheduled.

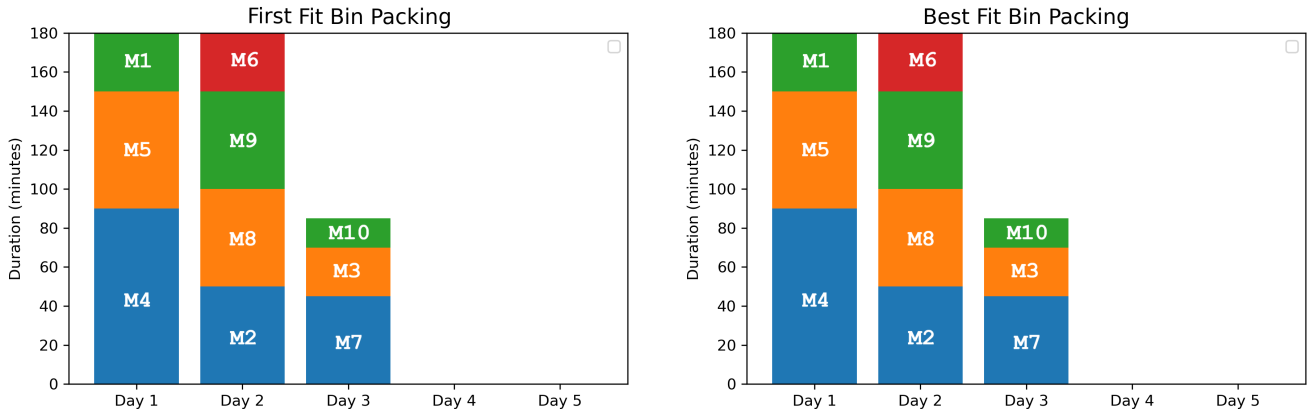
Step 6 Do $i = i+1$. Exit loop when $i > n$.

In Figure 4.1, we see the results of the three heuristic algorithms used for placing meetings into days of fixed length similar to the bin packing problem. As expected, the first fit and the best fit algorithm minimise the number of bins, that is, the number of days used and yield the same packing sequence solution for the given toy example. The first two results show that two days at the end of the week are free from any meetings while the other two days at the start are fully packed and one day in the middle is partially busy. On the other hand, balanced fit heuristics focuses on making a balanced schedule for each day of the week as depicted in figure 4.1 (c). We observe that all the five days of the week have almost the same load of scheduled meetings.

Table 4.1: Example meetings for Bin Packing Problem

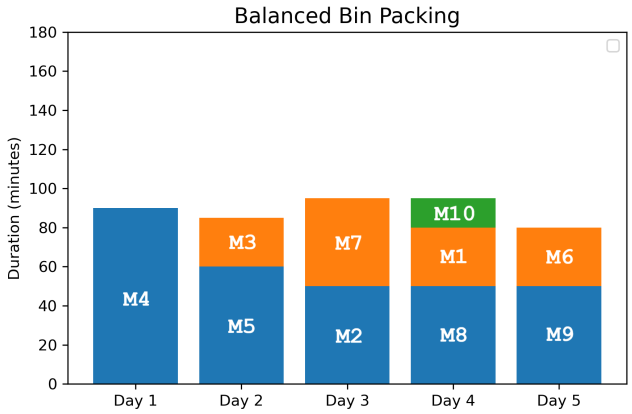
Meeting ID	Duration (min)	Priority value
M1	30	10
M2	50	5
M3	25	7
M4	90	4
M5	60	3
M6	30	2
M7	45	6
M8	50	8
M9	50	9
M10	15	1

Drawbacks: Some major drawbacks were identified in the initial algorithm designing phase. In the bin packing way, the meetings are packed one after another without any gaps, however, this is not feasible in the real work scenario. We want to have breaks between ideal meeting clusters and more importantly presence of a lunch break in the middle of the day. Constraints such as the required employees were not implemented because certain apparent issues were seen during the construction of the algorithm. Consider the following scenario as an example.



(a)

(b)



(c)

Figure 4.1: Result visualisation for Bin Packing Problem

Employee A has his first meeting M1 at the start of the day, and another meeting M2 after meeting M1. Meeting M2 also requires Employee B to attend. Employee B does not have any meetings before M2. In this case, it is not possible to control the tight packing heuristics to leave the initial space in the bin empty to ensure meeting M2 occurs at the same time for both employees. Thus, in the standard bin packing design we do not have command over the bins unless we create dummy items to form gaps or breaks.

These problems can be solved by dividing the days into slots to gain more control through slot indexing, but it changes the fundamental structure of the bin packing problem and the heuristics around it. So, we do not call the indexed days as bins, or consider our problem as a bin packing problem as we make further changes. Instead of imagining the days as vertical bins where items are stuffed tightly, we will alternatively consider them as horizontal containers where we have the option to place items anywhere in the middle as long as there is enough space. This will be named as the **horizontal packing** method. Hence, we divide a day into slots of 30 minutes and index them. Any new model will be built upon these indexed time slots.

4.3 Horizontal Packing Problem

Learning from the bin packing problem, we create a more flexible packing heuristics and call it the horizontal packing heuristics. If we remove the flexibility and constraints, this problem will become exactly the same as the original bin packing problem. As a starting constraint, we introduce already busy slots in the calendar. These already busy slots are input by the employees indicating their prior commitments. With this constraint, the primary objective is to efficiently add the maximum number of meetings to an already somewhat busy calendar. We also introduce the link between the meetings and the employees, that is, the meetings will have a list of attendees that must be present for the meeting to occur. If even one of the attendees is absent, the meeting will not be scheduled (coded as a hard constraint). We will later change this to a soft constraint to allow up to a certain threshold of attendees to be absent for the meeting to still be set up.

As seen in the analogous bin packing problem results (figure 4.1), the first fit and best fit methods are expected to produce close results for items sorted in descending order. So, we implement the first fit heuristics method for its ease of application. We again have a number of predefined meetings that need to be scheduled in a specific week as listed in table 4.2. The duration of the same meetings will now be converted, presented and used in terms of the number of slots required by the meeting. As mentioned earlier, there is no partially occupied slot. Either a slot is busy or it is completely free. Figure 4.2 (a) shows the slot wise already busy calendar of the four employees for a particular day in a week (the red slots indicate busy slots).

Table 4.2: Example meetings for Horizontal Packing Problem

Meeting ID	**Duration (in slots of 30 mins)**	**Priority value**	**Required Employees**
M1	1	10	E1 E2 E3 E4
M2	2	5	E2 E3
M3	1	7	E1 E3 E4
M4	3	4	E1 E4
M5	2	3	E1 E2
M6	1	2	E1 E3
M7	2	5	E1 E2

The approach is a first fit heuristic which tries to maximise an objective function. This algorithm will be named the **First Fit Horizontal Packing of Ad-hoc meetings**, since we are trying to schedule a small number of meetings to an already busy calendar of employees. The algorithm searches for a common slot among the required employees and schedules the meeting at the first free slot if found otherwise the meeting is left unscheduled. The two main objective functions used for this first fit packing approach are:

1. **Maximising the number of meetings scheduled**: The most intuitive way to achieve this is by sorting the meetings in ascending order based on their duration or lengths, thus scheduling more number of shorter meetings before the longer meetings.

2. **Maximising the total priority score**: This objective is met by sorting the meetings based on their priority values. Higher priority meetings are picked up first and are scheduled before the lower priority meetings.

While sorting the meetings based on their length or priority, whenever two values are the same, the second feature is used to break the tie. That is, the priority value is used to break ties in the case of maximising the number of meetings, and the meeting length is used in the case of maximising the priority score. Again, let *'I'* be the set of n meetings that need to be scheduled. Each item I_i of set *'I'* has four attributes: a unique identifier Meeting ID, the duration of the meeting specified in the number of slots, a priority value of the meeting and the required employees. We have four employees each with 16 time slots (including busy slots) in our example case.

Steps for Horizontal First Fit Packing Algorithm:

Step 1 Sort the meetings in non decreasing order based on the meeting length or slots occupied. After the ordering, we say, without loss of generality, that $I_1 \leq I_2 \leq I_3 \leq \cdots \leq I_n$ duration wise.

Step 2 Let $i = 1$ and start an incremental loop, running until $i = n$.

Step 3 Select an item I_i from the sorted list of meetings.

Step 4 Loop over the free time slots of each of the required employees for the meeting I_i and schedule the meeting in the first common unoccupied slot(s) that is found among the required employees.

Step 5 If a free common slot(s) is not found for all the required employees, leave the meeting I_i unscheduled.

Step 6 Update employees' free and busy time slots.

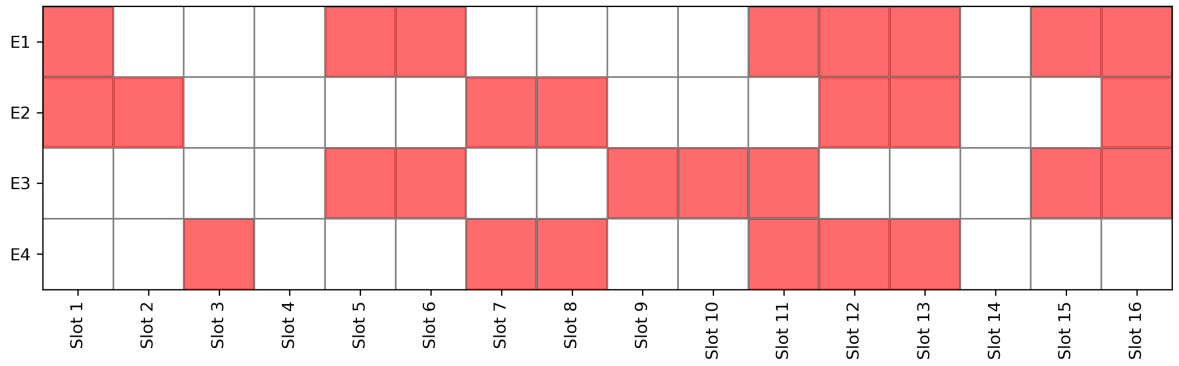
Step 7 Do $i = i + 1$. Exit loop when $i > n$.

Variation to the above heuristics: The above algorithm is built with the objective of maximising the number of scheduled meetings. For the objective of maximising the priority score of the meetings, we change the first step of the heuristics to sort the meetings according to their priority value in ascending order. This would change the objective of the algorithm.

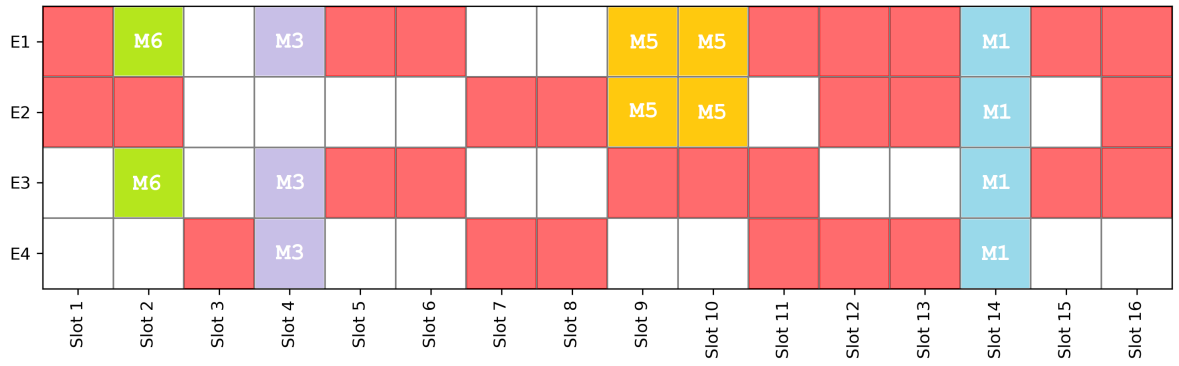
Results and Discussion: Figure 4.2 (b) and (c) respectively show the results of first fit heuristics for the two different objectives. To maximise the number of scheduled meetings (objective 1), the meetings were sorted in ascending order of their durations. The formed meeting sequence gave rise to the result where four meetings are scheduled out of seven, as shown in figure 4.2 (b). While maximising the priority (objective 2), the meetings were sorted based on their priority value. The resulting sequence gave rise to a slightly different set of four meetings being scheduled at different slots (figure 4.2 (c)) than the previous sequence. It is important to note that in this example, the number of meetings scheduled by both the algorithms is the same, however, the specific meetings that are scheduled are different. For clarity in the figures, each meeting is colour coded and mentioned at their respective slots. In general, we would expect the sequence generated by priority (objective 2) to schedule less than or equal to the number of meetings scheduled by the sequence generated by duration (objective 1), because the latter tries explicitly to schedule as many meetings as possible.

The ideas of slot assignment, indexing and horizontal placement of meetings establish a promising structure to build upon. We see that the first fit heuristics schedules the same number of meetings for two different meeting sequences where one is preferred due to its higher total priority score. This suggests that we need to try different random sequences and score the generated results on certain criteria to decide the best calendar. Since only two hard constraints were applied for this case, we will next design a penalty based scoring method which would serve two purposes. Firstly, it would help to quantify and compare different calendars. Secondly, it would also help to implement soft constraints through greedy heuristics built upon the penalty scoring. Though the First Fit Heuristics

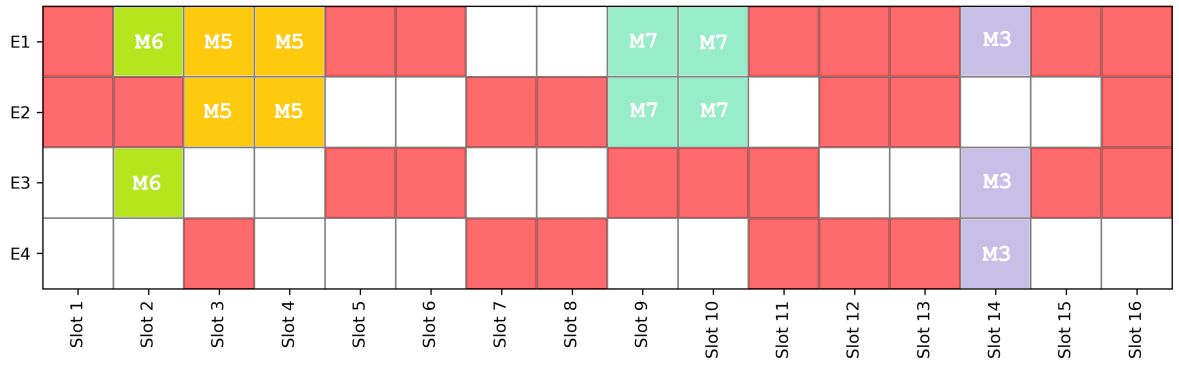
may work well in the case where there are only hard constraints, it can not enforce soft constraints. A penalty based greedy approach can ensure minimum deviations from the soft constraints while also following the hard constraints and doing efficient scheduling.



(a)



(b)



(c)

Figure 4.2: Results of First Fit Heuristics for Horizontal Packing Problem

4.4 First Fit Heuristics with Hard Constraints

Although the first fit heuristics does not have enough flexibility to incorporate soft constraints, it is capable of performing fast meeting schedules that satisfies all the hard constraints. This model serves as the most rudimentary structure for calendar scheduling. The improved greedy methods will be expanded on this structure. We created a

complete First Fit Heuristics obeying all the hard constraints as mentioned below:

1. A meeting does not occur without the meeting owner.
2. 100% of attendees are required for a meeting to set up.
3. Recurrent meetings are scheduled on the same day and time whenever it reoccurs.
4. Frequency of meetings is uniformly distributed across the four weeks.

Since the threshold for attendees is 100%, the meeting owner is safely included in the attendees list and the first constraint becomes a subset of the second constraint. Out of these hard constraints, the percentage of attendees is relaxed later in the greedy penalty based method. This complete first fit algorithm works on any type of calendar, busy or vacant, as input by the user.

The results produced are similar to figure 4.2, with the variation that the meetings are now scheduled across four weeks and have a predefined frequency of occurrence. The previous first fit algorithm for a week was a subset of the complete algorithm for four weeks, lacking the frequency parameter and accordingly scheduled meetings. Hence, the visualisation remains similar for a week. As before, the results change with different sorting. The formal definition of the algorithm to maximise the number of meetings scheduled is given below.

Let *'I'* be the set of n meetings that need to be scheduled. Each item I_i of set *'I'* has five attributes: a unique identifier Meeting ID, the duration of the meeting specified in the number of slots, a priority value of the meeting, the required attendees and its frequency. We have four employees each with 16 time slots (including busy slots) a day for four weeks. Each week has five working days. Thus, weeks are indexed from 1 to 4, each week has five days indexed from 1 to 5 and each day has sixteen time slots indexed from 1 to 16.

Steps for Complete First Fit Method

Step 1 Sort the meetings in non decreasing order based on the meeting length or number of slots required. After the ordering, we say, without loss of generality, that $I_1 \leq I_2 \leq I_3 \leq \cdots \leq I_n$ duration wise.

Step 2 Let $i = 1$ and start an incremental loop, running until $i = n$.

Step 3 Select the item I_i from the sorted list of meetings.

Step 4 Loop over the all time slots. Choose the next first slot(s) that is free for all the required attendees of meeting I_i, for the first occurrence of the meeting. The chosen slot is uniquely identified by the tuple (W_x, D_x, T_x), with the values being the week (W_x), the day (D_x) and the time (T_x).

Step 5 From the frequency, calculate the required number of occurrences of the meeting and an *increment value* for even distribution across the weeks. This *increment value* provides all the week numbers ($w_1, \cdots, w_n$) for which the meeting needs to be scheduled if the chosen slot acts as the first occurrence of the meeting I_i.

Step 6 For the chosen slot (D_x, T_x), check if the same slot is free for all the required weeks $w_1, \cdots, w_n$ calculated from the frequency. This is checked for all the required employees. (W_x will be a subset of $w_1, \cdots, w_n$)

Step 7 If the chosen slot passes all the checks, the meeting is scheduled at this slot for all the required employees and the weeks $w_1, \cdots, w_n$ calculated in step 5. Otherwise, go back to Step 4 and continue the loop from the next slot (W_x, D_x, T_x+1).

Step 8 If none of the chosen slots passes the requirements, the meeting I_i is left unscheduled.

Step 9 Update the calendar and do $i = i + 1$. Exit loop when $i > n$.

The above heuristics algorithm applies all the hard constraints while scheduling the meetings. It first goes through all the time slots and checks if the required employees are free at the given time slot for the required length. While spreading the meetings across the weeks according to the frequency, the same time slot is again verified if it is unoccupied for the required weeks. A meeting is scheduled only after a fit time slot satisfying all the constraints is found. The same structure will be used for the greedy penalty based algorithm to enforce hard constraints, however, the meeting will be scheduled in the first fit time slot that is found but in the time slot that gives the minimum penalty score.

Chapter 5

New Greedy Approach

In this chapter, we introduce a novel approach to solving the corporate meeting scheduling problem. We enhance the basic structure of the first fit algorithm to introduce a penalty based greedy approach to choose the best time slot for setting up a meeting instead of using the first fit time slot. The final algorithm outperforms all the previous methods and is robust and reproducible. A comparative study of the final algorithm with the real world data is presented in chapter 7 to support its usefulness. The algorithm can be readily implemented and expanded to different sectors and requirements. The data is available on GitHub to support further research efforts (Singh, 2023).

5.1 Greedy Heuristics

Greedy approaches are based on deciding the next step which will yield the largest immediate benefit. In our problem, this benefit is measured in terms of decrement in penalty score and a greedy approach is used in choosing the best time slot for the current meeting. Although this approach is not far sighted, it offers good enough initial solutions. The presence of a penalty will help in incorporating various constraints in the algorithm. We will apply all the identified hard and soft constraints through different penalty values and the algorithm design.
A brief summary of the soft and hard constraints that are enforced is mentioned as follows:

1. Presence of Meeting owner for the meeting (Hard Constraint)
2. Presence of up to a threshold percentage of attendees (Hard Constraint)
3. Same day and time for recurrent meetings (Hard Constraint)
4. Even distribution of frequent meetings (Hard Constraint)
5. Percentage of meeting attendees (Soft Constraint)
6. Start of the first meeting cluster (if any) each day (Soft Constraint)
7. Length of each meeting cluster (Soft Constraint)
8. Total number of meeting clusters (Soft Constraint)
9. Presence of Lunch Break (Soft Constraint)
10. End of the last meeting cluster each day (Soft Constraint)

Note:

(a) The percentage of attendees is bifurcated into two constraints (constraint number 2 and 5), one hard condition and the other as a soft constraint. For example, if the decided threshold percent of attendees is 80%, the

meeting is left unscheduled if the percentage of available attendees falls below the threshold of 80% (hard constraint). However, the algorithm will allow invitees to be absent if the attendance, without them, is still greater than or equal to 80% (soft constraint) with the ideal attendance being 100%. The meeting owner is not counted among the required attendees while imposing the threshold.

(b) The total number of meeting clusters does not have an ideal value but we would want to minimise it as much as possible. Given that there is an ideal cap on the length of the meeting clusters, there will be a trade-off between the two constraints relying on the penalty values.

Whenever a hard constraint is violated, the particular meeting is not scheduled. It is assumed that the office hour starts at 9 AM in the morning and ends at 5 PM in the evening. Table 5.1 summarises the ideal value for the soft constraints as per the deduced requirements of the Ford Motor Company. Our flexible algorithm allows these values to be easily updated whenever needed. Table 5.2 summarises the penalty value for deviation from the ideal condition for each of the constraints. The total penalty score is calculated by summing up the penalties of calendars for each employee. Considering senior employees with greater importance, the penalty factor for them is, in general, higher than the normal employees and is mentioned separately in table 5.2. The list below presents the quantifying way to capture the deviation from ideal values.

1. **Start of the first meeting cluster each day**: If the meeting cluster does not start at 9:00 AM, the deviation is counted as the number of slot differences between the slot at 9:00 AM and the meeting cluster start slot.

2. **Length of each meeting cluster**: If the length of any meeting cluster is greater than six time slots, the deviation is measured as the number of slots exceeding 6 slots (3 hours).

3. **Floating Lunch Break**: If a lunch break is not present between 12:00 PM to 2:00 PM, the deviation is calculated as the number of slots after 2:00 PM when an hour long break is found. If a break is not found, the slots are counted till the end of the office hour.

4. **Percentage of attendees**: If the number of attendees is less than the total expected attendees the deviation is quantified as the number of missing attendees for each meeting.

5. **End of the meeting cluster each day**: If any meeting cluster ends after 3PM, the deviation is counted as the number of slots the cluster extends after the slot ending at 3:00 PM.

Table 5.1: Summary of the ideal values for each soft constraint

Constraint	**Assumed Ideal Value**
Start of the first meeting cluster each day	9:00 AM (start of office hour)
Length of each meeting cluster	3 hours (6 time slots)
Total number of meeting clusters	As small as possible
Floating Lunch Break	1 hour (2 slots) break between 12pm-2pm
Percentage of attendees	100% (full attendance)
End of meeting cluster each day	3:00 PM (finishing early)

Table 5.2: Penalty factors for different constraints as a function of the deviation value (d)

Constraint	**Penalty for Normal Employees**	**Penalty for Senior Employees**
Start of the first meeting cluster each day	$15d$	$30d$
Length of each meeting cluster	$100d$	$200d$
Total number of meeting clusters	$30d$	$60d$
Floating Lunch Break	10^d	10^{2d}
Percentage of missing attendees	$500d$	$500d$
End of meeting cluster each day	$60d$	$120d$

We can change the penalty value according to the changing importance of soft constraints. Imposing a very heavy penalty is equivalent to applying a hard constraint, as the algorithm wants to minimise the total penalty and hence, it will avoid very heavy penalties. With the defined penalties, the primary framework of the greedy algorithm works by selecting one meeting at a time and scheduling it at the time slot chosen greedily to minimise total penalty score. After every meeting that has been scheduled, an updated penalty score is calculated. If the meeting is not scheduled a heavy penalty is added to the score. The penalty score for different constraints are also reported separately for result comparison.

5.1.1 Algorithm Structure

The inputs are given in CSV file formats which are read in Python. There are three CSV data files: meetings_data, employee_data and employee_schedule. The 'meetings_data' file contains the information of all the meetings that the company wants to schedule and the information is divided into the six predefined features of a meeting. The 'employee_data' file provides the employee details. For the simple case, it currently has the employee ID and a field indicating if the employee is senior. More characteristics about an employee can simply be added to this file whenever necessary. The 'employee_schedule' file consists of the calendars of the employees for four weeks. The calendar of each employee is divided into time slots and is uniquely indexed by its slot number, day number and week number together. Employees can mark a slot as busy if they do not want any meeting to be scheduled in that specific slot.

Let *'I'* be the set of n meetings that need to be scheduled. Each item I_i of set *'I'* has six attributes: a unique identifier Meeting ID, the duration of the meeting in number of slots, a priority value of the meeting, the required attendees, its frequency and the meeting owner. We have assumed to have four employees each with 16 time slots (including busy slots) a day for four weeks. Each week has five working days. The weeks (W) are indexed from 1 to 4, each week has five working days (D) indexed from 1 to 5 and each day has sixteen 30 minute time slots (T) indexed from 1 to 16 uniformly covering hours from 9:00 AM to 5:00 PM.

Steps for Greedy Penalty based Algorithm:

Step 1 Sort the meetings in non decreasing order based on the meeting duration or the number of slots re-

quired. After the ordering, we say, without loss of generality, that $I_1 \leq I_2 \leq I_3 \leq \cdots \leq I_n$ duration wise. Initialise the `Total Penalty` $= 0$.

Step 2 Let $i = 1$. Start a loop and run until $i \leq n$.

Step 3 Select the item I_i from the sorted list of meetings. For the meeting I_i, extract the information about the meeting duration, required employees, meeting owner and frequency.

Step 4 Initialise `Meeting Penalty` $= 10000$. This is the cost for not scheduling the meeting I_i. Set `Default Penalty` $=$ `Total Penalty` $+$ `Meeting Penalty`.

Step 5 Let $w = 1$ and start another loop for weeks until $w \leq 4$

Step 6 Create a nested loop, inside the week loop, for the day and the time slot. Let $d = 1$ and run this loop until $d \leq 5$. Inside the day loop, let $t = 1$ and exit the loop when $t > 16$.

Step 7 For the particular time slot uniquely identified by (w,d,t), check if the meeting owner is free for the required duration.

Step 8 If the meeting owner is not available, continue the loop over the time slots. Increment t or d or w according to the nested loop conditions.

Step 9 If the meeting owner is available, loop over the required employees and access their availability.

Step 10 If all the required attendees are unoccupied, temporarily schedule the meeting at the time slot (w,d,t) and calculate the penalty of this temporary calendar as `Temp Penalty`.

Step 11 If all the required attendees are not available, calculate the percentage of employees who are free and check if it is above the predefined attendance threshold.

Step 12 If the attendance percentage is below the threshold, continue the loop over the time periods.

Step 13 If the free attendees are above the threshold, schedule the meeting temporarily i at the time slot (w,d,t). Calculate the penalty for missing attendees and the penalty of the temporary calendar. Add them as the `Temp Penalty`.

Step 14 If `Temp Penalty` $<$ `Default Penalty`, set `Default Penalty` $=$ `Temp Penalty` and store the time slot index (w,d,t) as the `Best Slot` and the attendees as `Corresponding Attendees`. Revert the temporary schedule.

Step 15 Increment t or d or w according to the nested loop current position and go back to Step 7 until all the values of t,d,w are tested.

Step 16 Finish the nested loop at Step 5 and Step 6. Then finally schedule the meeting I_i as per the value of the `Best Slot` and `Corresponding Attendees`. If none of the slots is feasible, leave the meeting unscheduled.

Step 17 Update `Total Penalty` = `Default Penalty`.

Step 18 Do $i = i + 1$. When $i > n$, finish the loop at Step 2.

5.1.2 Penalty Calculation

We calculate the penalty of calendars for choosing time slots and comparing them. A penalty is always calculated and summed for all the schedules of each of the employees. Thus, every time we calculate the new penalty, the function hovers over all the employees and their schedules. For each employee, it checks if the employee is senior, according to the data read from the 'employee_data' file, to decide which set of penalty factors will be used. After the penalty factors have been decided, a nested loop is created to loop over all the four weeks and their five working days. Finally, all the deviations from the ideal condition are measured and multiplied by the penalty factors and

added to the global penalty variable. All the deviations and penalties are computed day wise.

The penalty function gives us the total penalty value where deviations are multiplied by penalty factors and summed together. However, we will play around with different values of the multiplicative factors to find good enough near-optimal penalty weights. The company can also change these factors corresponding to the changing importance of soft constraints. Therefore, changing the penalty function will change the overall penalty score for the same calendar. This makes it unreliable for comparison between the different sets of penalty factors. Hence, the total deviation value for each soft constraint is reported separately for such comparison.

Exponential Penalty: Another interesting feature of the penalty calculation is based on the gravity of damage a large deviation can do in regards to a small deviation. This particularly applies to the condition for the presence of a lunch break. We ideally want a lunch break between 12:00 PM and 2:00 PM. So, when the deviation is 30 minutes or 1 hour (in factors of slot length), the lunch break is still within 3:00 PM which can be acceptable. However, when the deviation goes beyond 1 hour, the lunch break gradually moves out of the acceptable range. For instance, a lunch break at 4:00 p.m. requires a much more severe penalty than a lunch break at 3:00 p.m. Thus, this penalty should not be computed linearly and we use an exponential function instead. In the exponentially determined penalty, the base is set as 10 and power is the deviation value, imposing heavier penalties for large deviations.

Hard Constraint Penalty: One of the mandatory conditions for the meeting to occur is the presence of the meeting owner. If the meeting owner is unavailable, the algorithm does not schedule that specific meeting but we want the algorithm to primarily maximise the number of meetings scheduled. In other terms, the algorithm should schedule a meeting even if the penalties of the soft constraints are extremely high. Although the hard constraints, integrated into the algorithm design, are deciding whether a meeting is feasible, we also apply a large penalty for every unscheduled meeting. This forces the algorithm to prioritise scheduling meetings above any soft constraint. The penalty score not only directs the algorithm to follow soft constraints as much as possible but also conditions the primary objective in the greedy structure.

Threshold Relaxation: We have set the minimum percentage of attendees as 80% for a meeting to be feasible. We notice that for the cases where the number of meeting attendees is less than five, in order to fulfil the attendee threshold condition, we actually require all the attendees. This is because even one missing attendee would make the percentage fall below 80%. To introduce more flexibility, for such cases, we allow at maximum one missing attendee even though the value drops down the threshold value. This threshold relaxation is an additional feature, utilised when a large number of meetings are clashing, and can be easily removed.

5.1.3 Mathematical Formulation

Variables

Let the variables be defined as follows:

1. d_1 for deviation in the start of the first meeting cluster
2. d_2 for deviation in the length of each meeting cluster
3. d_3 for the total count of meeting clusters

4. d_4 for deviation in the lunch break from the designated slots
5. d_5 for percentage of missing attendees
6. d_6 for deviation in the end of the last meeting cluster
7. `Unsch` for penalty of an unscheduled meeting
8. m for the number of meetings to be scheduled

Objective

Penalty for an unscheduled meeting `Unsch` = `Large Finite Positive Integer`
The objective is to maximise the number of scheduled meetings, that is, to minimise the number of unscheduled meetings. Let Z be the total penalty for all the meetings left unscheduled, which is also the objective function.

$$\texttt{min } Z = \sum_{i=1}^{m} \texttt{Unsch}$$

Constraints

The constraints are designed to apply a penalty whenever a soft constraint is violated and the secondary goal is to minimise this penalty. The combined penalty for deviation from various soft constraints in a day is:

$$\texttt{Penalty} = 15 \times d_1 + 100 \times d_2 + 30 \times d_3 + 10^{d_4} + 500 \times d_5 + 60 \times d_6$$

We have 4 weeks and 5 working days, hence we add the deviations $d^{[d,w]}$ for all the working days d in all four weeks w and the secondary objective to satisfy calendar preferences becomes:

$$\texttt{min Penalty} = \sum_{w=1}^{4} \sum_{d=1}^{5} 15 \times d_1^{[d,w]} + 100 \times d_2^{[d,w]} + 30 \times d_3^{[d,w]} + 10^{d_4^{[d,w]}} + 500 \times d_5^{[d,w]} + 60 \times d_6^{[d,w]}$$

Non-negativity Constraint: $d_1 \geq 0, d_2 \geq 0, d_3 \geq 0, d_4 \geq 0, d_5 \geq 0, d_6 \geq 0$

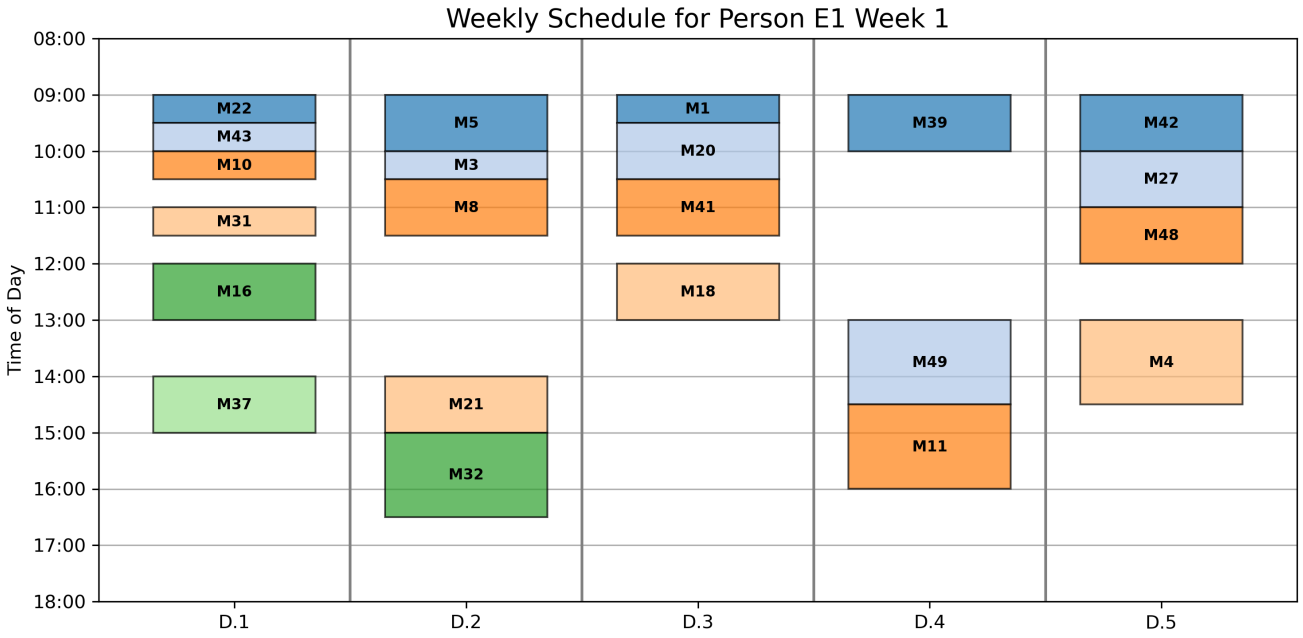
5.1.4 Results and Visualisation

For simulating the algorithm, we take the fully vacant calendars of four employees and the penalty factors from table 5.2 along with threshold relaxation. Since the vacant calendars offer a lot of flexibility and space, we created an example list of 51 meetings to test the algorithm in restraining conditions. These meetings are listed in table 5.3. When the meetings are sorted based on their duration in non decreasing order (with the priority value used for breaking ties), the algorithm produces the following result.

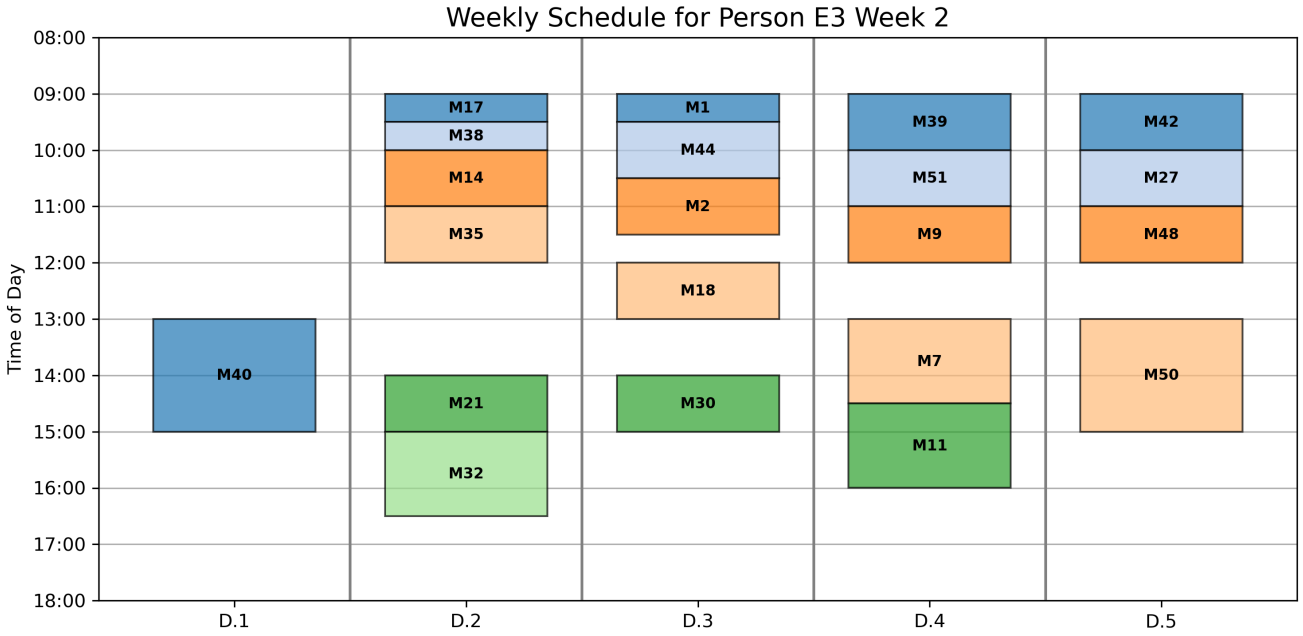
Total Penalty: 53195	Meeting Clusters Count: 160	Finish Early Deviation: 48
Priority Score: 249	Meeting Start Deviation: 41	Floating Lunch Deviation: 0
Unscheduled meetings: 2	Cluster Length Deviation: 2	Partial Attendees Meetings: 4

We visualise a few weekly schedules for two random employees and weeks. Figure 5.1a presents the first week schedule of Employee E1 and figure 5.1b shows the second week calendar of Employee E2. As the meetings were sorted according to their lengths in non decreasing order, we see a trend of smaller meetings being clustered together and scheduled frequently in the first half of the day and larger meetings being scheduled later. As per the

results, there is no deviation from the preference of having a lunch break between 12:00 PM and 2:00 PM. This can also be seen in the plotted two example calendars. Almost all the meeting clusters have the ideal or less than 3 hours duration. The algorithm generates promising results, as all expect 2 meetings are scheduled, and only 4 meetings are set up with partial attendees. The fulfilment of soft constraints are also easily visible in the figures 5.1a and 5.1b.



(a)



(b)

Figure 5.1: Visualisation for (a) First Week Calendar of the employee E1 and (b) Second Week Calendar of the employee E3

Table 5.3: List of meetings used for Greedy penalty-based Method

Meeting ID	Duration	Priority	Attendees	Frequency	Owner
M1	1	10	E1 E2 E3 E4	1	E1
M2	2	5	E2 E3	0.25	E1
M3	1	7	E1 E3 E4	0.5	E2
M4	3	4	E1 E4	0.75	E3
M5	2	3	E1 E2	1	E4
M6	2	6	E2 E4	0.5	E1
M7	3	2	E3 E4	0.75	E2
M8	2	3	E3 E1	0.5	E4
M9	2	5	E1 E2 E3	0.25	E4
M10	1	9	E1 E2 E4	1	E3
M11	3	5	E1 E2 E3 E4	1	E3
M12	2	5	E2 E3	0.25	E1
M13	6	6	E1 E3 E4	0.5	E2
M14	2	3	E1 E3	0.5	E2
M15	4	4	E2 E4	0.5	E3
M16	2	5	E1 E2	0.25	E4
M17	1	6	E3 E4	1	E1
M18	2	2	E1 E2 E3	0.75	E4
M19	4	3	E2 E3 E4	0.25	E1
M20	2	1	E1 E2 E3 E4	0.5	E3
M21	2	6	E1 E3 E4	1	E2
M22	1	3	E1 E2 E4	1	E3
M23	2	4	E2 E3 E4	0.5	E1
M24	3	5	E2 E3	0.25	E4
M25	4	6	E1 E4	0.75	E2
M26	1	9	E2 E4	0.5	E3
M27	2	7	E1 E3	1	E2
M28	3	11	E1 E2	0.25	E4
M29	4	6	E3 E4	0.5	E1
M30	2	2	E2 E3 E4	0.75	E1
M31	1	9	E1 E2 E4	1	E3
M32	3	5	E1 E2 E3 E4	1	E3
M33	2	5	E2 E3	0.25	E1
M34	6	6	E1 E3 E4	0.5	E2
M35	2	3	E1 E3	0.5	E2
M36	4	4	E2 E4	0.5	E3
M37	2	5	E1 E2	0.25	E4
M38	1	6	E3 E4	1	E1
M39	2	2	E1 E2 E3	0.75	E4
M40	4	3	E2 E3 E4	0.25	E1
M41	2	1	E1 E2 E3 E4	0.5	E3
M42	2	6	E1 E3 E4	1	E2
M43	1	3	E1 E2 E4	1	E3
M44	2	4	E2 E3 E4	0.5	E1
M45	3	5	E2 E3	0.25	E4
M46	4	6	E1 E4	0.75	E2
M47	1	9	E2 E4	0.5	E3
M48	2	7	E1 E3	1	E2
M49	3	11	E1 E2	0.25	E4
M50	4	6	E3 E4	0.5	E1
M51	2	2	E2 E3 E4	0.75	E1

Sequence Variation:

The plotted results are for the meeting sequence where the meetings are sorted in non decreasing order, however, it does not guarantee the optimal solution. In methods like the first fit algorithm, it is known that there exists a sequence for which the heuristics will generate the optimal solution. Similarly, we can assume that there exist other orderings of the meetings that can produce better results. Therefore we test a few random orderings of meetings and compare their results with the sorted sequence. If there are significant improvements in the results, our assumption will lead the algorithm in the correct direction.

Random Meeting Sequence R1: ['M35', 'M46', 'M33', 'M11', 'M29', 'M50', 'M51', 'M26', 'M15', 'M34', 'M45', 'M1', 'M37', 'M10', 'M13', 'M8', 'M30', 'M27', 'M48', 'M38', 'M24', 'M16', 'M40', 'M9', 'M3', 'M6', 'M4', 'M19', 'M5', 'M44', 'M12', 'M2', 'M42', 'M47', 'M32', 'M36', 'M14', 'M39', 'M22', 'M41', 'M20', 'M17', 'M21', 'M23', 'M28', 'M31', 'M49', 'M25', 'M43', 'M18', 'M7']

Total Penalty: 15505	Meeting Clusters Count: 176	Finish Early Deviation: 56
Priority Score: 261	Meeting Start Deviation: 70	Floating Lunch Deviation: 0
Unscheduled meetings: 0	Cluster Length Deviation: 4	Partial Attendees Meetings: 9

Random Meeting Sequence R2: ['M43', 'M2', 'M37', 'M21', 'M42', 'M17', 'M35', 'M18', 'M13', 'M38', 'M45', 'M24', 'M30', 'M15', 'M48', 'M36', 'M8', 'M20', 'M19', 'M27', 'M40', 'M9', 'M10', 'M23', 'M5', 'M29', 'M25', 'M44', 'M6', 'M14', 'M46', 'M22', 'M47', 'M11', 'M39', 'M41', 'M7', 'M34', 'M49', 'M28', 'M3', 'M33', 'M12', 'M31', 'M16', 'M1', 'M4', 'M51', 'M32', 'M26', 'M50']

Total Penalty: 21555	Meeting Clusters Count: 179	Finish Early Deviation: 108
Priority Score: 261	Meeting Start Deviation: 86	Floating Lunch Deviation: 9
Unscheduled meetings: 0	Cluster Length Deviation: 6	Partial Attendees Meetings: 6

Random Meeting Sequence R3: ['M1', 'M5', 'M11', 'M42', 'M41', 'M37', 'M20', 'M34', 'M3', 'M16', 'M22', 'M15', 'M40', 'M48', 'M50', 'M46', 'M18', 'M8', 'M27', 'M25', 'M39', 'M7', 'M38', 'M47', 'M28', 'M51', 'M49', 'M21', 'M23', 'M31', 'M2', 'M12', 'M24', 'M33', 'M43', 'M14', 'M17', 'M9', 'M19', 'M30', 'M4', 'M45', 'M35', 'M29', 'M26', 'M6', 'M36', 'M32', 'M13', 'M44', 'M10']

Total Penalty: 14370	Meeting Clusters Count: 167	Finish Early Deviation: 56
Priority Score: 255	Meeting Start Deviation: 64	Floating Lunch Deviation: 0
Unscheduled meetings: 1	Cluster Length Deviation: 4	Partial Attendees Meetings: 6

From the results of the random sequence of meetings, we can observe drastic improvements in the total penalty score in comparison to the sorted meeting sequence. All three of the random sequences outperform the sorted sequence in the total penalty, the priority score and the number of scheduled meetings, three of our primary objectives. Among the three random sequences, the sequence R3 provides the lowest total penalty value with one unscheduled meeting, while the sequence R1 provides the best total penalty value with all meetings being scheduled. We can also compare other soft constraints of the problem and notice that sequence R3 has a higher number of meetings scheduled with partial attendance than sequence R1.

The above results confirm our hypothesis that there exists sequences on which the method provides better results than the sorted sequence. Unfortunately, if we need to schedule n meetings, there are $n!$ possible meeting orderings that the algorithm can run on. This is a very large number even for a small number of meetings, and it is absolutely infeasible to run the algorithm on all the possible meeting ordering sequences. Since we do not have a defined way of generating different sequences of meeting that would be better for our algorithm, we will rely on random sequences as before. To find one of the best sequences that provides a near optimal solution, it is necessary to test a sufficiently large number of random sequences. According to different requirements, a sequence can be chosen that improves certain metrics better without changing the penalty factors. Based on this idea, we create a competitive greedy heuristic where different ordering sequences will be competing against each other to provide better scheduling outcomes.

5.2 Competitive Heuristics

The competitive heuristics is built upon competing meeting sequences that can minimise the total penalty score. Five hundred random meeting sets are generated and tested for the lowest penalty score. The shuffling of the meetings are controlled through seed, to produce the same random sequences for later comparison. For a given penalty factor value, the best sequence is chosen. The following steps formalise the heuristics.

Steps of the Competitive Heuristics:

Step 1 Set `min_penalty` = `Large Finite Positive Integer`.

Step 2 Set $t = 1$ and `seed` $= t + 1$.

Step 3 Read the meeting data and shuffle it with the set `seed` as M_{seed}.

Step 4 Run the **Greedy penalty based algorithm** on the meeting sequence M_{seed} and store its total penalty as `current_penalty`.

Step 5 If `current_penalty` $<$ `min_penalty`, set `min_penalty` = `current_penalty` and set store M_{seed}.

Step 6 Do $t = t + 1$. Exit when $t > 500$.

The above algorithm gives the best sequence and the lowest penalty score out of the five hundred randomly generated meeting sets. The answer meeting sequence is as follows along with the result produced by the greedy algorithm.

Best Meeting Sequence: ['M6', 'M36', 'M14', 'M16', 'M31', 'M12', 'M9', 'M5', 'M48', 'M30', 'M45', 'M28', 'M2', 'M35', 'M22', 'M43', 'M51', 'M46', 'M15', 'M47', 'M4', 'M24', 'M50', 'M32', 'M20', 'M42', 'M1', 'M40', 'M29', 'M11', 'M25', 'M37', 'M18', 'M21', 'M27', 'M41', 'M7', 'M17', 'M39', 'M10', 'M38', 'M19', 'M49', 'M23', 'M26', 'M13', 'M33', 'M8', 'M34', 'M3', 'M44']

Total Penalty: 8544	Meeting Clusters Count: 158	Finish Early Deviation: 0
Priority Score: 249	Meeting Start Deviation: 38	Floating Lunch Deviation: 4
Unscheduled meetings: 2	Cluster Length Deviation: 6	Partial Attendees Meetings: 5

The five hundred meeting sequences generate their corresponding penalty values. Each sequence is taken up in an

iteration and fed to the greedy algorithm. We sort these five hundred penalty scores in decreasing order and present the iteration convergence curve of the competitive heuristics algorithm in figure 5.2.

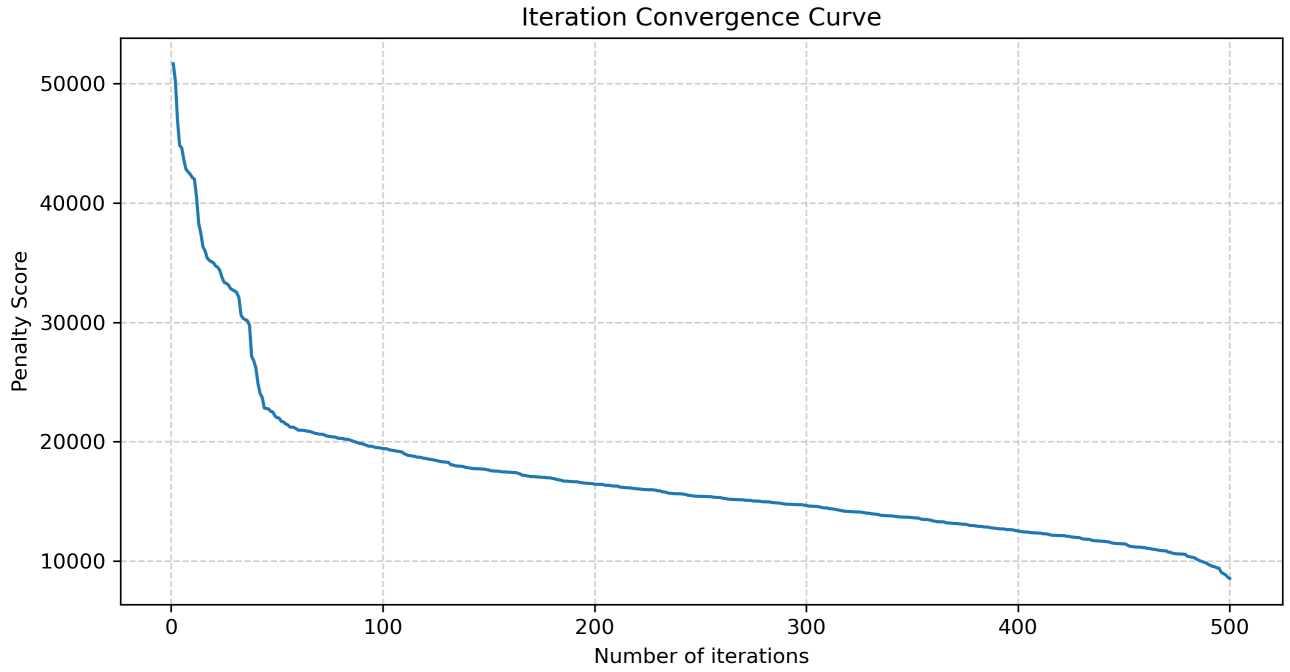


Figure 5.2: Iteration Convergence Curve

5.3 Random Sequence - A probabilistic method

We have used five hundred randomly produced meeting sequences, and the algorithm picks up the best out of them as the solution for our scheduling problem. In this section, we explain why these random sequences work and offer a solution close to the optimal sequence. Let us consider a simple example where, in a hypothetical situation, we have all possible meeting combinations and their corresponding penalty score. The penalty values, linked with their respective meeting sequence, are stored in an array and sorted in ascending order. Then the first value of the penalty array is the optimal solution with the lowest penalty score. We want to find this optimal solution or get as close to it as possible using limited resources, computational power and time.

To begin with, let us set the target to reach a solution that is in the top 10% of all the solutions. The top 10% solutions are assumed to be good enough solutions. Then, the probability that a solution, produced by an arbitrarily shuffled meeting sequence in the greedy penalty based approach, lies in the top 10% is 0.1.

$$P(\textit{a randomly chosen meeting sequence is in top } 10\%) = 0.1$$

So the probability is very low that a random set gives a good enough answer. However, let us modify our method to produce two candidate solutions, from two different random meeting sets, and select the solution with the lower penalty score. It gives rise to three cases: (a) both the solutions are within the top 10% (b) Only one of them is in the top 10% and (c) none of the two solutions are inside the top 10% range. While the first two cases are favourable

for us, only the last case is unfavourable. Hence, the new probability that the solution lies in the top 10% is:

$$P(\textit{at least one of two randomly chosen meeting sequences is in top } 10\%) = 1 - 0.9 \times 0.9 = 0.19$$

The chances of getting a good enough solution, therefore, increased from 0.1 to 0.19. Similarly, if we choose three arbitrary candidate meeting sequences, the probability increases to 0.271 and so on. In general,

$$P(\textit{at least one of n randomly chosen meeting sequences is in top } 10\%) = 1 - 0.9^n$$

As we can see, the probability increases as an exponential function of n. For the n = 100, we already attain a probability of 99.9973% of finding at least one of the top 10% solutions. Finally, when the number of meetings is large, the solution space expands drastically and the top 10% of the solutions may not represent a good enough solution. In such cases, the target percentage (t) is set as top 1% or smaller. So, in the generalized case,

$$P(\textit{at least one of n randomly chosen meeting sequences is in top } t\%) = 1 - (1 - \frac{t}{100})^n$$

Table 5.4: Probability of attaining near optimal solutions with 500 randomly chosen meeting ordering sequences

Percentage	**Probability**
Top 0.1%	0.39362
Top 0.5%	0.91843
Top 1%	0.99343
Top 5%	0.99999...
Top 10%	0.99999...

For the competitive heuristic algorithm, we used five hundred sequences sampled randomly. Table 5.4 depicts the probability of how close the found solution is from the optimal solution when we use 500 candidate meeting sets. We see that attaining a solution up to the top 0.5% has a high probability (greater than 0.9), affirming the method of random sequence heuristics.

We define a term called 'Acceptable Percentage' to denote the solution schedules that are considered to lie within an acceptable proximity of the optimal solution. As before, this proximity is measured in percentage bound near the best answer. Figure 5.3 represents the plot of probability against the acceptable percentages for three varying sets of 100, 200 and 500 random meeting sequences. The figure illustrates that as the number of samples increases, there is a stronger chance of converging towards the optimal solution. Although the probability of finding the most optimum solution is low, the chances of finding a good enough solution is very high.

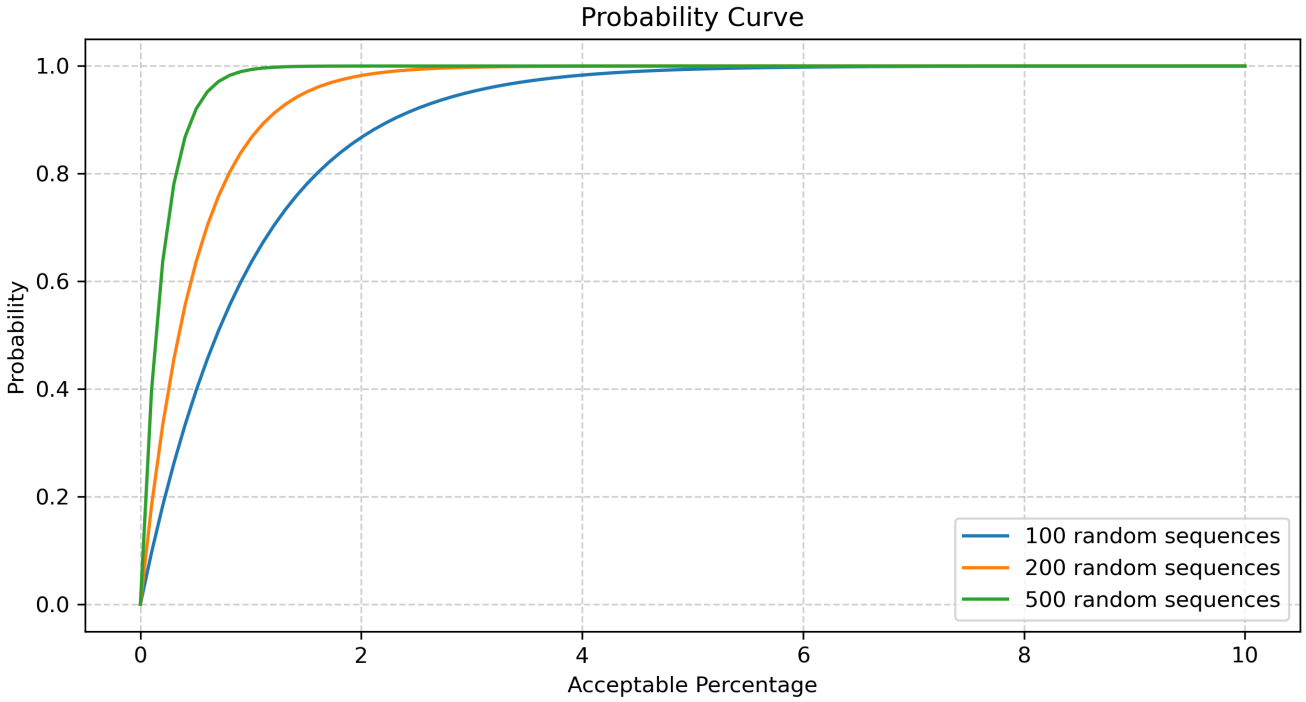


Figure 5.3: Probability Curve for Acceptable Percentage

Choice of Penalty Factors: The values of penalty factors are estimated and guessed according to the requirements of the Ford Motor Company. We need to tweak it as the preferences change. We use a grid based trial and error method to make a good choice.

Chapter 6

Data Analysis

In this chapter, we analyse the real world scheduling data provided by the Ford Motor Company. The data was completely anonymised to protect individual identity and only contains relevant information. Thus, we will often use terms such as Person 1, Person 2, etc and Meeting 1a, Meeting 1b, etc even for the real world data. The provided data contains the meetings as well as the calendar information of all the involved employees. The calendars are not directly presented in the data file, however, it can be readily deduced using scheduled meetings' mapping.

6.1 Descriptive Statistics

We perform descriptive statistics to explain the data and its variation. There are 27 recurring meetings scheduled over 4 weeks for a total of 37 employees. For the manually scheduled calendar, we first calculate the top ten busiest employees. Table 6.1 and 6.2 list the top most occupied persons based on number of meetings and total meeting duration (in minutes), respectively. We see that person p1, person p4 and person p2 are the top three busiest employees in both aspects. Later the overlap of busy employees between the two tables decrease, however some similarities remain as the total duration is expected to increase when the number of meetings increases.

Table 6.1: Busiest Employees based on #meetings

Employee ID	Number of meetings
p1	33 meetings
p4	29 meetings
p2	27 meetings
p7	27 meetings
p8	27 meetings
p3	26 meetings
p12	25 meetings
p20	25 meetings
p5	24 meetings
p14	23 meetings

Table 6.2: Busiest Employees based on duration

Employee ID	Total meeting duration
p1	1375 minutes
p4	1160 minutes
p2	1100 minutes
p3	1065 minutes
p7	1060 minutes
p8	1060 minutes
p20	1030 minutes
p12	980 minutes
p5	955 minutes
p34	950 minutes

We further explore the calendars of the most busy employee p1 by quantifying the weekly and daily distribution of meetings and duration as summarised in table 6.3 and 6.4. We observe that the meetings are neither distributed evenly across the weeks nor the days. Mondays tend to be the busiest day of the week while Friday is the least occupied day. It may indicate the attempt to keep Friday as free as possible. Among the weeks, week 1 and week 3 are the most loaded. As these meetings are manually scheduled, dividing the meetings equally across the two dimensions is a heavy task and is also reflected through the statistics.

Table 6.3: Meeting Distribution of Person 'p1'

Person 'p1'						
	Monday	Tuesday	Wednesday	Thursday	Friday	Weekly Total
Week 1	3	2	2	1	1	9
Week 2	3	0	3	1	1	8
Week 3	3	2	2	1	2	10
Week 4	3	0	2	1	0	6
Daily Total	12	4	9	4	4	-

Table 6.4: Duration Distribution of Person 'p1'

Person 'p1'						
	Monday	Tuesday	Wednesday	Thursday	Friday	Weekly Total
Week 1	120	135	75	45	50	425
Week 2	120	0	100	45	50	315
Week 3	120	105	75	45	50	395
Week 4	120	0	75	45	0	240
Daily Total	480	240	325	180	150	-

We count the total number of meetings that have been scheduled each day over all four weeks to see if the observation for person p1 matches the overall statistics. A meeting is counted once for the day it was scheduled. We perform it once with the recurring frequencies and once without including it. These counts, however, are not repeated for individual attendees, to see if the meetings were disturbed without involving attendees' calendars. We observe that without the recurrent frequency, Monday has 3 different repeating meetings, Tuesday has 5 meetings, Wednesday and Thursday both have 6 meetings and Friday has 7 meetings assigned to it. Here the results do not show the same trend as the individual person p1 but have an opposite distribution. More meetings are reserved for Friday than Monday. If we include the frequencies, we notice a similar grouping of meetings with Monday, Tuesday, Wednesday, Thursday and Friday having 12, 9, 16, 17 and 16 meetings respectively.

To further investigate why the trend of person p1 did not match the meeting distribution skeleton, we present collective statistics which include all the employees. If we sum the number of meetings for all employees over four weeks, we get the following results: Monday - 412, Tuesday - 122, Wednesday - 142, Thursday - 56, Friday - 40. These results correspond well with the statistics of employee p1. A possible explanation can be that the meetings

scheduled on Monday, in general, have a higher frequency of recurrence and a large number of attendees. This increases the load on Monday on an individual level. The presented contrasting statistics also depict the weakness in the manual method of scheduling meetings, in that it often produces inconsistent trends.

Duration Statistics: We plot the duration range box plot for three types of employees, on the basis of how busy they are, as high, medium and low. We take one example from each type of employee and plot a comparison graph shown in figure 6.1. It is interesting to note that the busier personnel tend to have longer meetings than less occupied persons. There is a consistent decrease in the median of the meeting duration as we move from highly busy to less busy individuals.

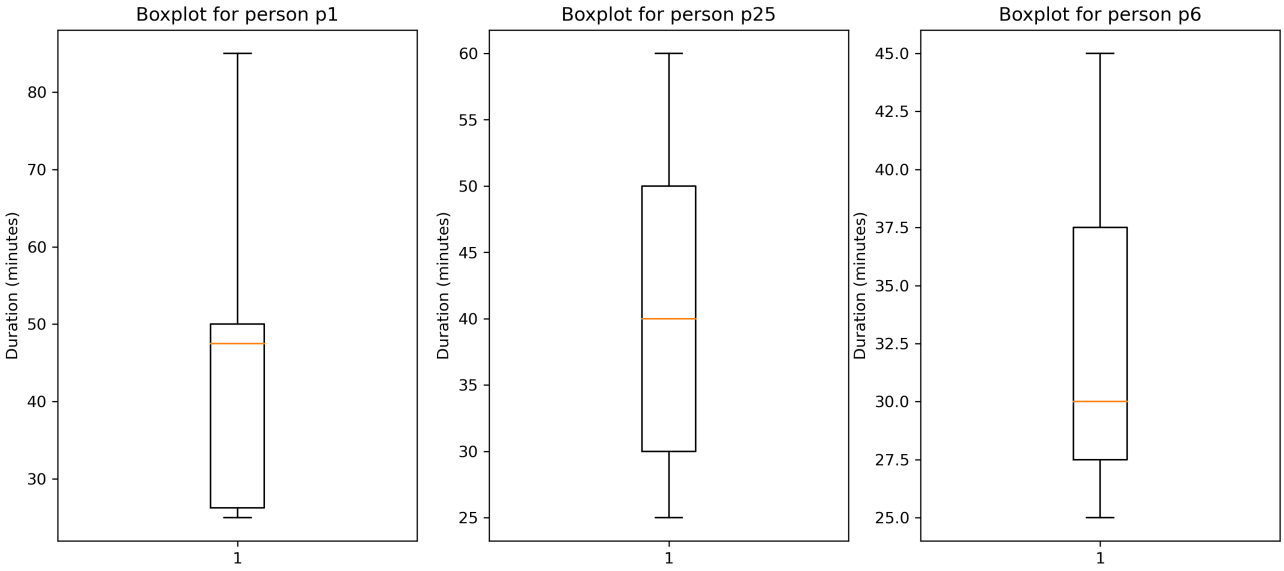


Figure 6.1: Comparison Boxplot

6.2 Data Cleaning

There are in total 27 unique meetings mentioned in the data, however, some meetings have missing information. The most common type of missing data is a lack of information on required attendees. In some cases, the meeting is scheduled for only one attendee, who is also the meeting owner, which again is the same type of missing data lacking attendee details. There exist 13 meetings with incomplete data, which is removed for unbiased comparison between manual scheduling and automated scheduling. Out of the remaining 14 meetings, 6 of them do not have a meeting owner. For such cases, one of the senior attendees is chosen as the meeting owner for completeness.

We also need information on the seniority of the employees, which is not explicitly provided in the data. It is mentioned that the members of the group *LL5+* are considered most important. The team includes nine personnel with the following unique IDs: p1, p4, p8, p14, p7, p2, p3, p12 and p6. We consider these members as the senior employees for our algorithm. This data is also consistent with the top 10 most busy employees, where 8 of these senior employees were found to be the most busy ones. Lastly, we do not have the priority values for the meetings, which is coded as an additional feature in the algorithm, that can be omitted. Nevertheless, we include this feature and assign random values to the meeting as priority value. The final list of cleaned and corrected meetings is shown in table 6.5.

Table 6.5: List of cleaned and corrected meetings

Meeting ID	Duration (minutes)	Priority Value	Recurrent Frequency	Meeting Owner	Required Attendees
1b	85	2	0.25	p1	p1, p2, p3
1c	55	3	0.25	p1	p1, p2
2a	50	5	1	p4	p4, p7, p8, p24, p25, p26, p27, p28, p29, p30, p31, p32, p33, p34, p35, p36
3a	25	6	0.25	p1	p1, p2, p3, p4, p7, p8, p12, p14, p17
3b	25	1	1	p2	p1, p2, p3, p4, p5, p7, p8, p9, p11, p12, p14, p15, p16, p17, p18, p19, p20, p21, p22, p23, p24, p25, p26, p27, p28, p29, p30, p31
3f	50	7	0.5	p4 / p2	p1, p2, p4, p18, p20, p24, p34
3g	25	8	0.25	p14 / p13	p1, p2, p3, p4, p5, p6, p7, p8, p12, p14, p15, p16
2b	60	3	0.5	p3 / p12	p3, p5, p12, p20, p21, p22, p23
2d	30	11	1	p6	p32, p33, p34, p35, p36
2e	45	9	1	p4	p1, p2, p3, p4, p5, p6, p7, p8, p12, p14, p20, p21, p22, p23
4a	60	10	1	p4	p1, p2, p3, p4, p5, p7, p8, p9, p11, p12, p14, p15, p16, p17, p18, p19, p20, p21, p22, p23, p24, p25, p26, p27, p28, p29, p30, p31, p32, p33, p34, p35, p36
4b	30	6	1	p1	p1, p2, p3, p4, p5, p6, p7, p8, p9, p11, p12, p14, p15, p16, p17, p18, p19, p20, p21, p22, p23, p24, p25, p26, p27, p28, p29, p30, p31, p32, p33, p34, p35, p36
4e	30	7	1	p1 / p2	p1, p2, p3, p4, p5, p7, p8, p9, p11, p12, p14, p15, p16, p17, p18, p19, p20, p21, p22, p23, p24, p25, p26, p27, p28, p29, p30, p31, p32, p33, p34, p35, p36
4f	50	12	0.25	p3	p2, p3, p4, p5, p7, p8, p9, p11, p12, p14, p15, p16, p17, p18, p19, p20, p21, p22, p23, p24, p25, p26, p27, p28, p29, p30, p31, p32, p33, p34, p35, p36

Chapter 7

Performance Analysis

We carry out a performance analysis on the cleaned data of 14 meetings as given in table 6.5. First, we calculate the penalty score and other metrics for the manually scheduled calendars of 37 employees. Then we execute the algorithm to do an automated meeting scheduling. Finally, we expand on what the different scores mean, and do a comparison between the manual and the automated process. We also visualise the calendars built by the algorithm and by the manual scheduling process. This section shows the real world implementation of the heuristic method and its improvement over the old approach.

7.1 Score of Manual Method

We first prepare the data and extract the manually scheduled calendars of the employees. After the data cleaning process, we convert the meeting durations in terms of the number of slots of 30 minutes, for ease of application. The number of qualifying meetings, after data cleaning and having all the information, is in the smaller range for four weeks and hence we expect the calendars to be sparse and less deviating from the ideal conditions. We pass this processed data to the penalty score function and get the following results:

Total Penalty: 34985	Meeting Clusters Count: 608	Finish Early Deviation: 0
Priority Score: 91	Meeting Start Deviation: 508	Floating Lunch Deviation: 0
Unscheduled meetings: 0	Cluster Length Deviation: 0	Partial Attendees Meetings: 0

For the small number of real meetings that we tested here, most of the metrics have the ideal values and all the meetings are scheduled. Although we notice a large number of meeting clusters indicating that the meetings may not be clubbed efficiently. Furthermore, for the manual calendars, the meetings do not tend to start at the beginning of the day (preferred condition), as depicted by the high value of start deviation. The calendar, however, performs well on other soft constraints.

7.2 Score of Automated Method

We again take the extracted and cleaned meetings from the data provided and use it in the greedy based algorithm. Ideally, we would run the complete competitive heuristics method to find the near optimal meeting sequence and the result. However, even if one unsorted meeting sequence (as provided by the data) produces a better schedule in

an automated approach than the manual method, it will prove that the heuristics will find at least a solution better than the old solution. Hence, we leave the meeting set unsorted ('1b', '1c', '2a', '3a', '3b', '3f', '3g', '2b', '2d', '2e', '4a', '4b', '4e', '4f') and just implement the greedy based scheduling to obtain the following:

Total Penalty: 17285	Meeting Clusters Count: 385	Finish Early Deviation: 0
Priority Score: 91	Meeting Start Deviation: 32	Floating Lunch Deviation: 0
Unscheduled meetings: 0	Cluster Length Deviation: 0	Partial Attendees Meetings: 0

Owing to the small number of meetings, we again see minimum deviation from the ideal state. All the meetings are scheduled with full attendees and satisfy all the soft constraints except one. To quantify how tolerable the deviation score for the constraint 'start of the first meeting cluster' is, we compute the total number of days in the calendar for 37 employees, which is equal to 740 days. The value of 32 is small in comparison, hence showing that the above result also performs for the 'start of the first meeting cluster' constraint.

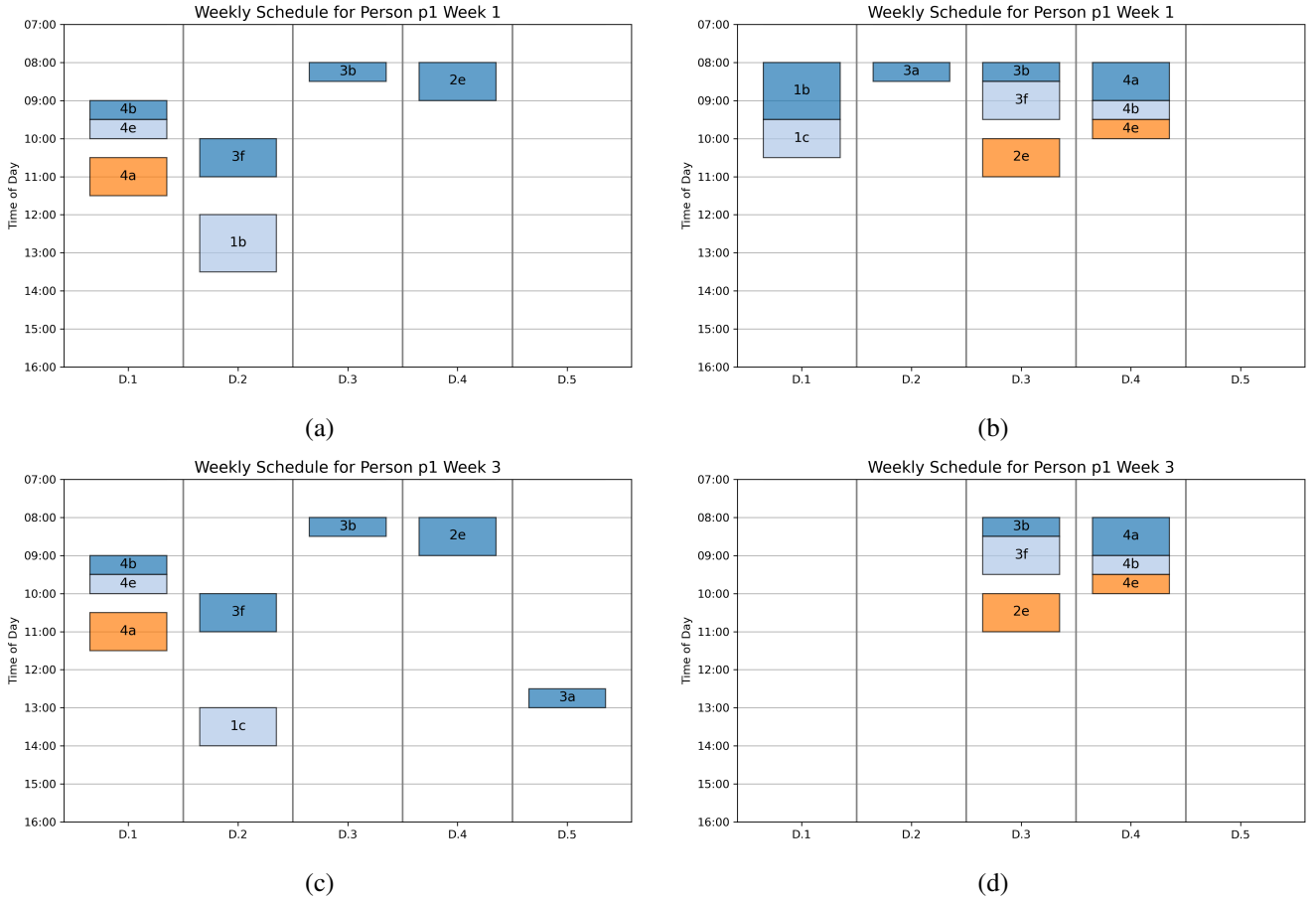


Figure 7.1: Weekly visualisation for person p1 for weeks 1 and 3. The left hand side figures (a) and (c) represent the original calendars while the right hand side images (b) and (d) represent the calendars generated by the algorithm

7.3 Performance Comparison

We observe that the total penalty score for the automated process is less than half the penalty score of the manual calendars. This demonstrates significant improvement in the calendar when using the automated algorithm. Apart from the penalty score, there is also a huge advantage in the soft constraint score. The number of meeting clusters is considerably reduced and there is a drastic drop in the deviation from the constraint of starting a meeting cluster as early as possible. It is important to note that the meeting sequence in the automated method was not optimised or worked upon and yet the basic algorithm still produced remarkably better results.

7.3.1 Visualisation of meeting clusters

To study features of the schedules, we visualise example weekly calendars as shown in figure 7.1 and 7.2. Through the figures, we notice that the meetings are more scattered in the original schedules while they are effectively clustered in the algorithmic calendars. The clustering of meetings also allows for more focus time in the schedules produced by the greedy method.

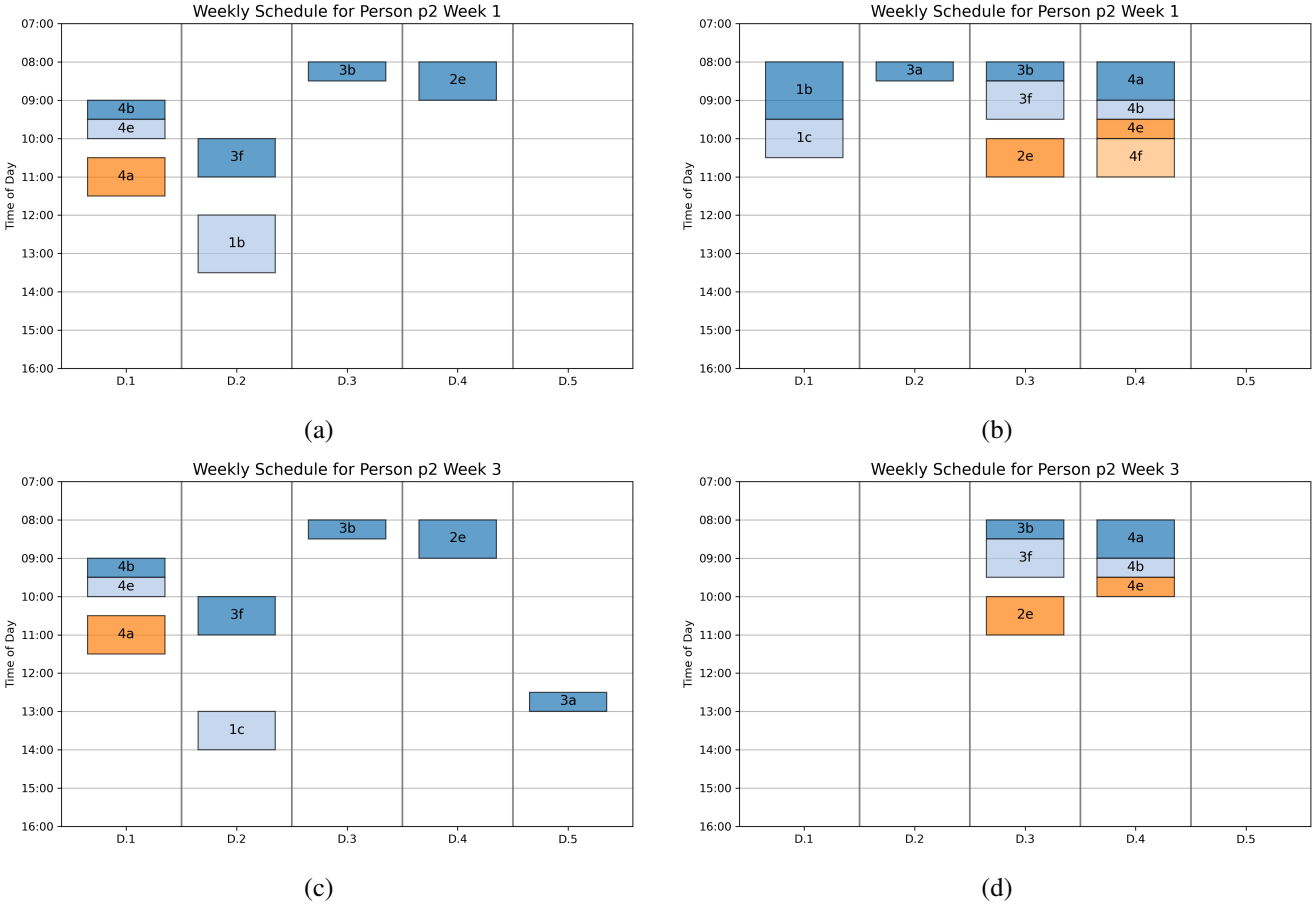
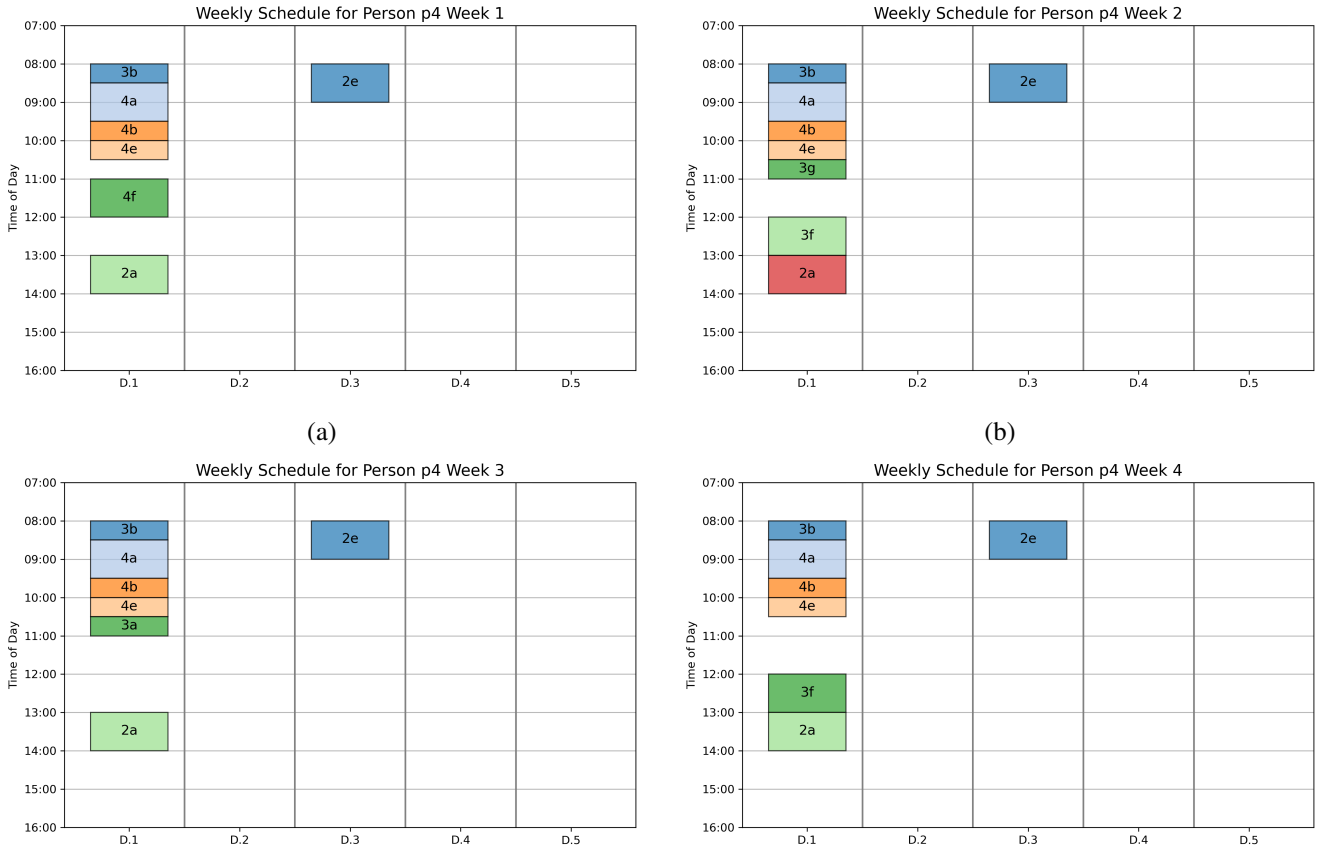


Figure 7.2: Weekly visualisation for person p2 for weeks 1 and 3. The left hand side figures (a) and (c) represent the original calendars while the right hand side images (b) and (d) represent the calendars generated by the algorithm

We also observe that the meetings are wrapped before lunch time (no meeting after lunch) in the automated calendars. Upon further investigation, we found that indeed there are no meetings after lunch in any of the schedules

generated by the algorithm. This depicts that the number of meetings are small enough to allow to be set up always before lunch. We note that the automated calendars are constructed using an unsorted meeting sequence, which is not yet optimised.

Note: While visualising the weekly schedules of employees p1 and p2, we observe that the same meetings are not present in the same week. This is because the automated algorithm works for the combined four weeks and hence different meetings are scheduled in different weeks.



(a) (b)

(c) (d)

Figure 7.3: The four weeks calendar of employee p4 generated by the final improved sequence

7.4 Result and Discussion

The final result of the heuristic method on the real data is refined by improving the unsorted meeting sequence as used before. A total of 200 iterations are run to find the best sequence out of the two hundred randomly shuffled meeting sets. Given a larger number of employees, the total time taken for the algorithm to complete increases to two hours. However, it is only required to run the best sequence finding heuristics once. Based on the lowest penalty score, we get the following sequence and metric score results:

Best Meeting Sequence: [’3b’, ’4a’, ’4b’, ’4e’, ’4f’, ’1c’, ’1b’, ’3g’, ’2e’, ’2a’, ’2b’, ’3f’, ’3a’, ’2d’]

Total Penalty: 13355	Meeting Clusters Count: 307	Finish Early Deviation: 0
Priority Score: 91	Meeting Start Deviation: 5	Floating Lunch Deviation: 0
Unscheduled meetings: 0	Cluster Length Deviation: 0	Partial Attendees Meetings: 0

The metric score shows a drastic improvement over the score of the original manually scheduled calendar. All the soft constraints are closely satisfied, with deviation from four of them being exactly zero. The total number of days in the overall calendar is $4\,weeks \times 5\,days \times 37\,employees = 740\,days$. In comparison to the total number of days, only 5 days have a deviation for the start of the first meeting cluster, which is negligible. There are in total 307 meeting clusters which allows us to safely infer that more than half of the days in the overall calendar are free from any meeting. This can significantly improve focus time and productivity.

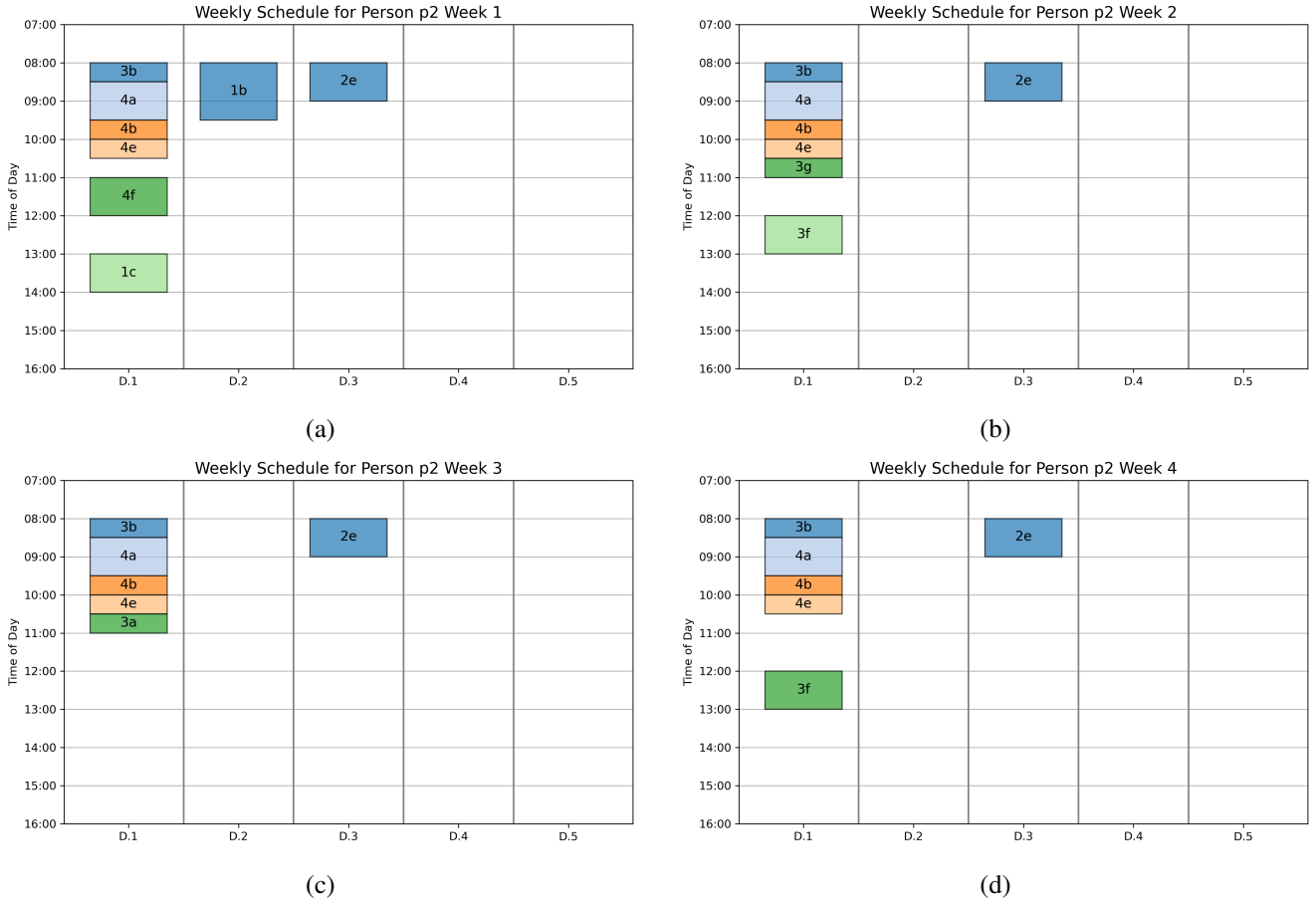


Figure 7.4: The four weeks calendar of employee p2 generated by the final improved sequence

We visualise the weekly calendars of the most busy employees based on the cleaned data. The new busiest personnel is person p4 and the employees p2, p7 and p8 are the next most busy persons with the same number of meetings. Figure 7.3 and 7.4 present the final four weeks calendar schedules of person p4 and p2 respectively. The enhanced meeting sequence lets the algorithm implement a better clustering with the ideal cluster length. The smaller meetings are grouped together to form ideal meeting clusters and the longer meetings form independent clusters. We also see that the calendars generally have up to three no-meeting days in a week.

Although the number of recurring meetings scheduled from the real world case is small and the calendars are largely vacant, the results showed the effective clustering done by the automated scheduling algorithm. The penalty score decreases by more than half of the original score. The features can be seen through the calendar visualisation. The new approach gives better results for most of the randomly shuffled meeting sequences, as tested for unsorted sequences in sections 5.1.3 and 7.2. The superiority of the designed method is consistently proved throughout the real case calendar analysis (chapter 7).

7.5 Cost Analysis

An internal survey was conducted by the Ford Motor Company, in their PowerTrain Manufacturing team, to collect data about the relevant problems in the current meeting scheduling process. Based on the collected data, we present a cost estimation and savings analysis that an automated scheduling method may bring in the future. The questionnaire was designed to target various categories of employees. The Ford Motor Company broadly classifies the employees into five major group according to their level of seniority and salary range. We pick relevant data from the survey for the cost analysis and present a summary of weighted average values in table 7.1. The salary of each category is estimated using the salary data present on Glassdoor ("How much does Ford Motor Company (UK) pay in 2023? (175 salaries)", n.d.).

Table 7.1: Summary of questionnaire data

Salary Group	**Average time spend in scheduling meeting per week (minutes)**	**Average time spend in rescheduling meeting per week (minutes)**	**Estimated Salary per annum (GBP)**
Category 1	36.667	7.333	100,000
Category 2	101.3	6.05	80,000
Category 3	39.6	7.808	60,000
Category 4	26.964	8.536	40,000
Category 5	39.143	20	30,000

There are approximately 200 employees in the PowerTrain Manufacturing sector. We divide the employees in the same ratio as in the responses of the questionnaire. The percentage response and employee division is given in table 7.2.

Table 7.2: Division of Employees in different categories

Salary Group	**Percentage Response**	**Equivalent number of employees**
Category 1	6%	12
Category 2	21%	42
Category 3	28%	56
Category 4	30%	60
Category 5	15%	30

For the cost estimation, we calculate the total time spent in scheduling and rescheduling meetings per week, multiplied by the number of employees and salary. The cost estimated for each category is given in following sections. The total cost is the sum of cost of each group. We use the following structure for estimating cost and we use a shrinkage factor to account for annual leave, holidays, sick leave, training, etc.

$$\texttt{Weekly Impact} = \texttt{Number of Employees} \times \texttt{Time spent per week}$$

$$\texttt{Weekly Percentage Impact (Unadjusted)} = \frac{\texttt{Weekly Impact}}{\texttt{Working hours per week}}$$

$$\texttt{Weekly Percentage Impact (Adjusted)} = \texttt{Weekly Percentage Impact (Unadjusted)} \times \texttt{Shrinkage Factor}$$

$$\texttt{Estimated Cost Savings} = \texttt{Weekly Percentage Impact (Adjusted)} \times \texttt{Salary}$$

Category 1

$\texttt{Weekly Impact} = 12 \times \frac{44}{60}\ \texttt{hours} = 8.8\ \texttt{hours}$
$\texttt{Weekly Percentage Impact (Unadjusted)} = \frac{8.8}{37.5} = 0.235$
$\texttt{Weekly Percentage Impact (Adjusted)} = 0.235 \times 1.28 = 0.300$
$\texttt{Estimated Cost Savings} = 0.300 \times £100,000 = \boldsymbol{£30,000}$

Category 2

$\texttt{Weekly Impact} = 42 \times \frac{107.35}{60}\ \texttt{hours} = 75.145\ \texttt{hours}$
$\texttt{Weekly Percentage Impact (Unadjusted)} = \frac{75.145}{37.5} = 2.004$
$\texttt{Weekly Percentage Impact (Adjusted)} = 2.004 \times 1.28 = 2.565$
$\texttt{Estimated Cost Savings} = 2.565 \times £80,000 = \boldsymbol{£205,200}$

Category 3

$\texttt{Weekly Impact} = 56 \times \frac{47.408}{60}\ \texttt{hours} = 44.247\ \texttt{hours}$
$\texttt{Weekly Percentage Impact (Unadjusted)} = \frac{44.247}{37.5} = 1.180$
$\texttt{Weekly Percentage Impact (Adjusted)} = 1.180 \times 1.28 = 1.510$
$\texttt{Estimated Cost Savings} = 1.510 \times £60,000 = \boldsymbol{£90,600}$

Category 4

$\texttt{Weekly Impact} = 60 \times \frac{35.5}{60}\ \texttt{hours} = 35.5\ \texttt{hours}$
$\texttt{Weekly Percentage Impact (Unadjusted)} = \frac{35.5}{37.5} = 0.95$
$\texttt{Weekly Percentage Impact (Adjusted)} = 0.95 \times 1.28 = 1.216$
$\texttt{Estimated Cost Savings} = 1.216 \times £40,000 = \boldsymbol{£48,640}$

Category 5

$\texttt{Weekly Impact} = 30 \times \frac{59.143}{60}\ \texttt{hours} = 29.572\ \texttt{hours}$
$\texttt{Weekly Percentage Impact (Unadjusted)} = \frac{29.572}{37.5} = 0.790$

$$\texttt{Weekly Percentage Impact (Adjusted)} = 0.790 \times 1.28 = 1.011$$
$$\texttt{Estimated Cost Savings} = 1.011 \times £30,000 = £\mathbf{30,330}$$

Final Cost Savings

The total cost savings is $£\mathbf{404,770}$ per year for the PowerTrain Manufacturing team. Dividing by the number of employees, the cost savings is approximately $£\mathbf{2,000}$ per year per employee. Bias in the calculation may come due to the salary estimation and employee division in various categories. Nevertheless, the total cost is still representative of high savings that can be achieved using an automated scheduling process.

7.6 Time Complexity Analysis

To estimate the worst case time complexity of the algorithm we define the following variables:

1. M - Set of all the meetings to be scheduled
2. E - List of all the employees
3. w - number of weeks to be planned ahead
4. d - number of working days in a week
5. t - number of time slots in a day

Although the algorithm can schedule up to any number of weeks in the future, in this particular case the Ford Motor Company schedules meetings up to 4 weeks. Hence we assume that the variable w can have a maximum value of 4, which is a fixed constant. Similarly, in the worst case, the maximum number of working days in week can be 7 (a constant) and the maximum number of time slots in a day will again have a constant upper limit.

Algorithm 1 Simplified Pseudocode for Penalty Calculation

```
1:  function PENALTY(E)                                   ▷ E ← Set of all Employees
2:      total ← 0
3:      for every e in E do
4:          for w in range(1,4) do
5:              for d in range(1,5) do
6:                  d1 ← Calculate deviation in the start of first meeting cluster for (e,w,d)
7:                  d2 ← Calculate deviation in length of each meeting cluster for (e,w,d)
8:                  d3 ← Calculate total count of meeting clusters for (e,w,d)
9:                  d4 ← Calculate deviation in the lunch break slot for (e,w,d)
10:                 d5 ← Calculate deviation in the end of last meeting cluster for (e,w,d)
11:                 total ← total + 15 × d1 + 100 × d2 + 30 × d3 + 10^d4 + 60 × d5
12:             end for
13:         end for
14:     end for
15:     return total
16: end function
```

Time complexity for calculating various deviations:

1. $\boldsymbol{d1}$: to calculate the deviation in the start of the first meeting cluster, the program iterates over all the time slots of a day. So the time complexity of this step is O(t).

2. ***d*2**: to calculate the deviation in the length of the each meeting cluster, the program iterates over all the time slots of a day. So the time complexity of this step is O(t).
3. ***d*3**: to calculate the total count of meeting clusters, the program iterates over all the time slots of a day. So the time complexity of this step is O(t).
4. ***d*4**: to calculate the deviation in the lunch break slot, the program iterates over all the time slots of a day. So the time complexity of this step is O(t).
5. ***d*5**: to calculate the deviation in the end of the last meeting cluster, the program iterates over all the time slots of a day. So the time complexity of this step is O(t).

To calculate the total penalty, the function iterates through all the w weeks, d days and t time slots of the calendar. However, since all these variables have a constant upper bound, the final time complexity of calculating the total penalty of any calendar for one employee in Big O notation will be O(1). However, we calculate the penalty for all the employees, therefore, the time complexity of total penalty calculation is O($|E|$), where $|E|$ is the size of the set of all employees.

Algorithm 2 Simplified Pseudocode for Meeting Scheduling

```
1: function SCHEDULE(M,A,E)                                   ▷ M ← Set of all Meetings,
                                                                A ← Set of list of Attendees,
                                                                E ← Set of all Employees
2:     for every m in M do
3:         numMissingAttendees ← 0
4:         default penalty ← INF
5:         threshold ← 0.2 × |A_m|
6:         for w in range(1,4) do
7:             for d in range(1,5) do
8:                 for t in range(1,16) do
9:                     for every a in A_m do
10:                        if a is not free at (t,d,w) then
11:                            numMissingAttendees ← numMissingAttendees + 1
12:                        end if
13:                    end for
14:                    if numMissingAttendees ≤ threshold then
15:                        current penalty ← PENALTY(E)        ▷ Temporarily schedule m at (t,d,w)
16:                        current penalty ← current penalty + 500 × numMissingAttendees
17:                        if current penalty < default penalty then
18:                            default penalty ← current penalty
19:                            (T,D,W) ← (t,d,w)
20:                        end if
21:                    end if
22:                end for
23:            end for
24:        end for
25:        if default penalty < INF then
26:            Schedule meeting m at (T,D,W)
27:        end if
28:    end for
29: end function
```

Final Time Complexity Calculation:

- Loop at line 2 has a time complexity of $\mathbf{O}(|M|)$, where $|M|$ is the size of the set of all meetings.
- Loop at line 6 has a time complexity of O(w). Since w has a constant upper bound, the time complexity becomes **O(1)**.
- Loop at line 7 has a time complexity of O(d). Since d has a constant upper bound, the time complexity becomes **O(1)**.
- Loop at line 8 has a time complexity of O(t). Since t has a constant upper bound, the time complexity becomes **O(1)**.
- Loop at line 9 has a time complexity of $\mathbf{O}(|A_m|)$, where $|A_m|$ is the size of set of attendees.
- At line 10, checking if an employee a is free in the slot (t,d,w) is an **O(1)** time complexity operation.
- Calculating current penalty from the `PENALTY` function at line 15 is an $\mathbf{O}(|E|)$ operation, where $|E|$ is the size of the set of all employees.
- Scheduling a meeting at `(t,d,w)` at lines 15 and 16 is an **O(1)** time complexity operation.

Hence, the total time complexity of the meeting scheduling algorithm is $O(|M|) \times (O(|A_m|) + O(|E|))$. Since the maximum number of attendees can never be greater the total number of employees, the final time complexity becomes $\boldsymbol{O(|M| \times |E|)}$.

Chapter 8

Conclusion

The overall design, implementation and analysis of this work suggest the effectiveness of an automated calendar scheduling algorithm. The algorithm in the study was built from scratch and presents a start point for solving meeting scheduling problems. The new approach provides the flexibility to include individual preferences although it can be enhanced further. There are a number of improvements that can be made to the method and we discuss a few of them that were identified but not implemented due to time constraints.

8.1 Conclusion

In conclusion, this study on the corporate meeting scheduling problem has yielded promising results, confirming the immense potential of the automated scheduling process compared to the traditional manual scheduling methods. The innovative automated approach, carefully designed to operate within resource constraints, demonstrated its effectiveness through a pragmatic application of greedy penalty-based heuristics. This study presents a pioneering effort to solve calendar scheduling problems in corporate settings by comprehensively addressing individual preferences while satisfying organisational constraints - a feat that was previously less explored.

The solution calendars produced by the new greedy approach depicted desired results, accomplished in half the time of what would have originally required extensive manual effort. Moreover, the cost analysis illustrates the substantial savings this method promises to deliver in the future. Beyond the financial aspect, the introduction of this method contributes to enhancing collaboration and organisational productivity. The availability of efficient breaks, focused work time and optimally clustered meetings may lead to greater employee satisfaction and ultimately the overall success of the company.

In this study, we introduced a novel approach to corporate meeting scheduling, employing an efficient greedy heuristic method that balances speed, ease of use, and interpretability. Notably, our work stands at the forefront of academic innovation in this domain. It addresses a critical gap in the existing literature, offering a comprehensive solution to the calendar meeting scheduling problem - a gap we identified in prior research. A greedy penalty based heuristic method was not earlier found in the literature review. The new algorithm harnesses the power of three different ideas, that is, it greedily chooses time slots based on minimum penalty value in a heuristic iteration approach. This type of hybrid models were less explored and offers a starting point for future research.

The significance of this research extends beyond academia. Our method's cost-effectiveness, intuitive visualization, and reasonable computational complexity, akin to a manageable 2nd order polynomial time complexity, present promising prospects for real-world application. It offers a valuable tool for businesses and organizations seeking to optimize their meeting schedules while considering diverse preferences and operational constraints. This development is poised to streamline scheduling processes, bolstering productivity, employee satisfaction, and ultimately contributing to the success of various ventures.

Although the study represents a remarkable achievement, it also opens doors for future improvements. There is scope for future development and refinement in the design of the algorithm, potentially uncovering more effective scheduling strategies. Future enhancements can also focus on handling dynamic constraints, optimising for faster scheduling processes, improving user interfaces and integrating AI techniques to predict customer satisfaction. Embracing real-time data and exploring machine learning integration mark future trajectories for optimizing this algorithm to cater to evolving organisational needs.

8.2 Future Work

The future scope for the study is to expand it in various other settings while also incorporating more flexibility and different situation-specific preferences by using the penalty method. For the current calendar scheduling problem, new adjustments can be made in the algorithm structure for making it more robust. Enhancing the ease of using the designed approach through user-friendly interfaces is another area for future exploration.

8.2.1 Algorithm Improvement

The five major aspects recognised for improving the algorithm in the future are as follows:

1. **Decreasing Slot Size**: Currently, for making the slot assignment easier, the slot length was taken to be 30 minutes. As per the algorithm's design, a slot is either regarded as fully occupied or unoccupied but it cannot be considered as partially occupied. This makes the slot length of 30 minutes inefficient for meetings of size less than 30 minutes. This can be remarkably refined by introducing slots of 15 minutes, or even 5 minutes.

2. **Optimising Penalty Factors**: In this study, the values of penalty factors were estimated and guessed through trial-and-error and grid based approach tailored around the requirements. The method used is not very reliable and it can be improved by designing another heuristic or machine learning approach to optimise and estimate factor values. However, more research is needed to construct one of the optimisation techniques.

3. **Increasing Senior Employee Spectrum**: In the algorithm, an employee is either considered either a senior or not a senior personnel, but in reality, the seniority level of the employees is a spectrum. This work is left for the future, to infuse granular seniority level for better calculation of the penalties for different types of employees.

4. **Flexible day and time for recurrent meetings**: One of the hard constraints was defined on the assumption that the recurring meetings need to be scheduled on the same day and time whenever they recur. This

assumption can be removed to provide more flexibility, where the recurring meetings can be allowed to defer from the original time up to a threshold. It may reduce the number of clashing meetings.

5. **Partially scheduling recurring meetings**: The constructed heuristics does not schedule a meeting if it can not be fully scheduled according to its frequency across four weeks. This constraint can be relaxed to allow partial scheduling of a meeting even if its repeating frequency can not be fully satisfied. This may increase the number of meetings scheduled.

There can be more such structural improvements to enhance flexibility and increase details in the algorithm. The present algorithm, while highly effective, offers scope for evolution. Future enhancements can also focus on handling dynamic constraints, optimising for faster scheduling processes and integrating AI techniques to predict customer satisfaction. Continuous innovation remains vital to ensure that the scheduling algorithm evolves in correspondence with the ever-changing needs of modern organisations.

8.2.2 User Experience

Another major aspect of the study that requires further work is enhancing the user experience while using the method. The method is built to be used and easily edited by non-technical personnel. Designing intuitive scheduling interfaces that allow employees to easily input their availability and preferences, makes the scheduling process more user-centric. In the current method, the data is input in the form of MS Excel or CSV files, which is easy to build but does not offer a great user experience. An interactive software will act as a bridge between the algorithm and the user. The presence of such a platform makes it more likely for the user to embrace the new method and continue using it. Designing a user-friendly interface, to minimise the time and effort required for users to become proficient, is left for future exploration.

8.3 Next Steps

The industrial sponsor recognises the value of this project outcomes and its potential to reduce non-value added administration time across its global operations. From the findings of this research, the Ford Motor Company has already begun exploring opportunities to fund the continuation of this research project in order to address the opportunities for future work and incorporate this methodology into its current scheduling practices.

Appendix A

User Guide

A.1 Requirements

The algorithm is designed in python which has the following technical requirements.
Software requirements:

- Anaconda
- Spyder
- Any CSV editor
- Any text editor

Python library requirements:

- csv
- random
- copy
- math
- mathplotlib
- datetime

A.2 Input Files

All the input files are loaded in CSV format. The program can read the following three inputs:

- **Meeting_Data.csv**: Containing the information about the meetings to be scheduled (Singh, 2023).
- (Optional) **Employee_Calendar.csv**: Containing the details of employee calendars for a warm start, giving information about pre-scheduled meetings (Singh, 2023).
- (Optional) **Senior_Employees.csv**: Containing data about which employees are senior (Singh, 2023).

The respective example files are present in the GitHub repository, with same names, which provides a guide about the column order and how the data should be arranged. In the Employee_Calendar.csv, zero represent free slot and any other entry represent occupied slot. In the Senior_Employees.csv file, 1 denotes senior employee and zero is used for normal employees.

Note: The files Employee_Calendar.csv and Senior_Employees.csv are marked as optional for the algorithm. If an employee calendar is not explicitly input, the whole calendar is assumed to be vacant. Similarly, if information about senior employees is not provided, all the employees are considered as normal employees.

A.3 Steps

The whole program is broadly divided into three parts:

1. **Competitive Heuristics:** For finding the best meeting sequence heuristically.
2. **Greedy penalty-based Scheduling:** For calculating detailed penalty metric score given a meeting sequence.
3. **Weekly Calendar Visualisation:** For visualising weekly schedule given an Employee ID and a week number.

We first find the best meeting sequence using the Competitive Heuristics and then calculate the detailed penalty score and save the calendar output for the found sequence using Greedy Scheduling. Finally, we visualise the output calendar using the Calendar Visualiser.

A.3.1 Steps for Competitive Heuristics

In the competitive heuristics code, follow these steps:

Step 1 If a warm start is needed where pre-occupied slots are present in the employee calendar, input/change the file name with its path at line **212**. Otherwise comment out the line **212**.

Step 2 [Optional] If you want to change the penalty multiplicative factors, change the respective values at lines **232, 233, 234, 235, 236** and **304**. Normal penalty scores are present at line **223** for reference.

Step 3 [Optional] Change the threshold percentage of attendees at line **324**.

Step 4 Input the meetings data file name along with the path at line **380**.

Step 5 Modify the number of employees at line **386** inside the class *EmployeeSchedule*.

Step 6 [Optional] The details of senior employees is input at line **389**. Otherwise **comment out** this line.

Step 7 Increase or decrease the number of iterations at the line **378**.

Step 8 Run the Python Script.

Print Output: The total penalty score at each iteration is printed along with the final answer penalty score and best meeting sequence.

Saved Files: Penalty score at each iteration is saved in `Result_File.txt` file and the best penalty score and meeting sequence is stored in `Results.txt` file.

A.3.2 Steps for Greedy penalty-based Scheduling

From the `Results.txt` file note the lowest penalty score and map it to its iteration number in the `Results_File.txt`. In the meeting scheduling code, follow these steps:

Step 1 Modify the number of employees at line **170** inside the class *EmployeeSchedule*, if needed.

Step 2 Input the meetings data filename along with the path at line **214**.

Step 3 [Optional] If a warm start is needed where pre-occupied slots are present in the employee calendar, uncomment the line **217** and input the file name with its path. Otherwise leave the line **217** commented.

Step 4 [Optional] The details of senior employees is input at line **220**. Otherwise **comment out** this line.

Step 5 [Optional] If you want to change the penalty multiplicative factors, change the respective values at lines **240, 241, 242, 243, 244** and **311**. Normal penalty scores are present at line **231** for reference.

Step 6 [Optional] Change the threshold percentage of attendees at line **331**.

Step 7 Put the iteration number of best sequence + 87876 at line **398**.

Step 8 Run the Python Script.

Print Output: The total penalty score along with various other soft constraints penalty score is print as output. The program also prints the meetings that are scheduled and what is the percentage of attendees in each meeting.

Saved Files: The updated employee calendar is stored as `New_Employee_Calendar1.csv` in the same folder as the code.

A.3.3 Steps for Visualisation

Using the `New_Employee_Calendar1.csv` or any other calendar file in the specific format (Singh, 2023), we perform the following visualisation steps:

Step 1 Identify the employee ID and the week number that you want to visualise.

Step 2 Change the week number at line **12** and employee ID at line **14**.

Step 3 Input/change the employee calendar file name along with its path at the line **19**.

Step 4 Run the Python Script.

Saved Files: The visualised image is stored in the same folder as the code.

Appendix B

Python Codes

The Competitive Heuristics Code:

```
# -*- coding: utf-8 -*-
"""
@author: ss7u22
"""

import csv
import random
import copy
import math

class DailySchedule:
    def __init__(self):
        self.slots: dict[int, str] = {}
        for l in range(16):
            self.slots[l] = '0'

    def build(slots):
        obj = DailySchedule()
        obj.slots = slots
        return obj

    def __deepcopy__(self, memo):
        id_self = id(self)
        _copy = memo.get(id_self)
        if _copy is None:
            _copy = DailySchedule.build(copy.deepcopy(self.slots, memo))
            memo[id_self] = _copy
        return _copy

    # Number of meeting cluster in a day
    def count_clusters(self):
        is_slot_started = False
        num_clusters = 0
        for i in range(16):
            if self.slots[i] != '0':
```

```
                is_slot_started = True
            else:
                if is_slot_started:
                    num_clusters += 1
                is_slot_started = False
        if is_slot_started:
            num_clusters += 1
        return num_clusters

    # Check if meeting starts at 8AM everyday
    def meeting_start(self):
        is_meeting_started = False
        deviation = 0
        for i in range(16):
            if self.slots[i] != '0':
                is_meeting_started = True
                break
        if is_meeting_started:
            deviation = i
        return deviation

    # Check the length of meeting clusters each day
    # Return deviation from the ideal length
    def cluster_length(self):
        is_slot_started = False
        exceeding_slot = 0
        length = 0
        for i in range(16):
            if self.slots[i] != '0':
                is_slot_started = True
                length += 1
            else:
                if is_slot_started:
                    if length > 6:
                        exceeding_slot += length - 6;
                is_slot_started = False
                length = 0
        if is_slot_started:
            if length > 6:
                exceeding_slot += length - 6;
        return exceeding_slot

    # Check if the meeting finish by 3PM each day
    def finish_early(self):
        last_busy_slot = 0
        exceeding_slot = 0
        for i in range(16):
            if self.slots[i] != '0':
                last_busy_slot = i
        if last_busy_slot > 12:
            exceeding_slot = last_busy_slot - 12
```

```
        return exceeding_slot

    # Deviation from the floating lunch
    def floating_lunch(self):
        consecutive_slots = False
        start = 6
        deviation = 0
        while consecutive_slots == False and start+1 < 16:
            if self.slots[start] == '0' and self.slots[start+1] == '0':
                consecutive_slots = True
                if start > 8:
                    deviation = start - 8
            start += 1
        if consecutive_slots == False:
            deviation = 8
        return deviation

class WeeklySchedule:
    def __init__(self):
        self.days: dict[int, DailySchedule] = {}
        for k in range(5):
            self.days[k] = DailySchedule()

    def build(days):
        obj = WeeklySchedule()
        obj.days = days
        return obj

    def __deepcopy__(self, memo):
        id_self = id(self)
        _copy = memo.get(id_self)
        if _copy is None:
            _copy = WeeklySchedule.build(copy.deepcopy(self.days, memo))
            memo[id_self] = _copy
        return _copy

class Employee:
    def __init__(self):
        self.weeks: dict[int, WeeklySchedule] = {}
        self.is_senior = True
        for j in range(4):
            self.weeks[j] = WeeklySchedule()

    def build(is_senior, weeks):
        obj = Employee()
        obj.is_senior = is_senior
        obj.weeks = weeks
        return obj

    def __deepcopy__(self, memo):
        id_self = id(self)
```

```
        _copy = memo.get(id_self)
        if _copy is None:
            _copy = Employee.build(copy.deepcopy(self.is_senior, memo), copy.deepcopy(
    self.weeks, memo))
            memo[id_self] = _copy
        return _copy

class EmployeeSchedule:
    def __init__(self, num_employees: int):
        self.employees: dict[str, Employee] = {}
        for i in range(num_employees):
            employee = Employee()
            self.employees[f'p{i+1}'] = employee

    def build(employees):
        obj = EmployeeSchedule(0)
        obj.employees = employees
        return obj

    def __deepcopy__(self, memo):
        id_self = id(self)
        _copy = memo.get(id_self)
        if _copy is None:
            _copy = EmployeeSchedule.build(copy.deepcopy(self.employees, memo))
            memo[id_self] = _copy
        return _copy

    def print(self):
        for i, emp in self.employees.items():
            print(f'{i}:')
            emp.print()


def read_meetings(filename):
    meetings = []
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            meeting = {
                'Meeting ID': (row[0]),
                'Duration': math.ceil(int(row[1])/30),
                'Priority': int(row[2]),
                'Required Employees': [(e) for e in row[5].split()],
                'Frequency': float(row[3]),
                'Meeting Owners': [(e) for e in row[4].split()]
            }
            meetings.append(meeting)
    return meetings

def read_employee_schedule(filename, employee_schedulex: EmployeeSchedule):
```

```
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            employee_id = (row[0])
            week_id = int(row[1])
            day = int(row[2])
            F_slots = [(slot) for slot in row[3:]]
            for i in range(16):
                employee_schedulex.employees[employee_id].weeks[week_id-1].days[day-1].
    slots[i] = F_slots[i];

def senior_employee(filename, employee_schedulex: EmployeeSchedule):
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            employee_id = (row[0])
            is_senior = int(row[1])
            if is_senior == 0:
                employee_schedulex.employees[employee_id].is_senior = False
            else:
                employee_schedulex.employees[employee_id].is_senior = True

# For a warm start read a employee calendars
# read_employee_schedule('Employee_Calendar.csv',employee_schedulex)
total_penalty = 0
penalty_type_value = {
    "Cluster Count":0,
    "Meeting Start Deviation":0,
    "Cluster Length Deviation":0,
    "Finish Early Deviation":0,
    "Floating Lunch Deviation":0,
    "Incomplete meetings":0
    }

normal_penalty = [30,15,100,60,10]

def calculate_penalty_for_employee(employee_n: Employee):
    factor = 1
    if employee_n.is_senior :
        factor = 2
    penalty1 = 0
    for i in range(4):
        for j in range(5):
            penalty1 += (employee_n.weeks[i].days[j].count_clusters())*30*factor
            penalty1 += (employee_n.weeks[i].days[j].meeting_start())*15*factor
            penalty1 += (employee_n.weeks[i].days[j].cluster_length())*100*factor
            penalty1 += (employee_n.weeks[i].days[j].finish_early())*60*factor
            penalty1 += 10**(employee_n.weeks[i].days[j].floating_lunch())*factor

```

```
    return penalty1


def calculate_penalty(employee_schedulex1: EmployeeSchedule):
    total = 0
    for empId, employee in employee_schedulex1.employees.items():
        total += calculate_penalty_for_employee(employee)
    return total

def Employee_free_slot(required_employees,i,j,k,meeting_length,temp_schedule):
    free_slot = True
    for empl in required_employees:
        for l in range(meeting_length):
            if (k+l) < 16:
                if temp_schedule.employees[empl].weeks[i].days[j].slots[k+l] != '0':
                    free_slot = False
            else:
                free_slot = False
    return free_slot

def Free_attendees(required_employees,i,j,k,meeting_length,temp_schedule):
    free_slot = True
    employees_free = []
    for empl in required_employees:
        free_slot = True
        for l in range(meeting_length):
            if (k+l) < 16:
                if temp_schedule.employees[empl].weeks[i].days[j].slots[k+l] != '0':
                    free_slot = False
            else:
                free_slot = False
        if free_slot:
            employees_free.append(empl)

    return employees_free

def schedule_weekly_meeting(i,no_of_meetings,increment,new_combined,meeting_length,
    temp_schedule,j,k,meeting_id,required_employees,can_attendee,default_penalty,
    partial_attendees,pos):
    possible = True
    scheduled = 1
    weeks = [i]
    week = i

    while scheduled < no_of_meetings and possible == True:
        if week + increment < 4:
            weekly_slots_free = True
            for empl in new_combined:
                for l in range(meeting_length):
                    if temp_schedule.employees[empl].weeks[week+increment].days[j].slots
    [k+l] != '0':
```

```
                        weekly_slots_free = False
            if weekly_slots_free:
                weeks.append(week+increment)
                scheduled += 1

            week += increment
        else:
            possible = False

    if possible == True and scheduled == no_of_meetings:
        for week_c in weeks:
            for empl in new_combined:
                for l in range(meeting_length):
                    temp_schedule.employees[empl].weeks[week_c].days[j].slots[k+l] = (
    meeting_id)

        current_penalty = calculate_penalty(temp_schedule)

        # Penalty for less than 100% attendees
        current_penalty += (len(required_employees)-len(can_attendee))*500
        if current_penalty < default_penalty:
            default_penalty = current_penalty
            partial_attendees = new_combined
            pos = [weeks,j,k]
    return default_penalty,partial_attendees,pos

def schedule_meeting(meeting):
    global total_penalty
    global employee_schedulex
    meeting_id = meeting['Meeting ID']
    meeting_length = meeting['Duration']
    required_employees = meeting['Required Employees']
    frequency = meeting['Frequency']
    meeting_owners = meeting['Meeting Owners']

    required_employees = [i for i in required_employees if i not in meeting_owners]
    combined_employees = required_employees + meeting_owners

    total_attendees = len(required_employees)
    threshold_attendees = min(math.ceil(0.8*total_attendees),total_attendees-1)

    no_of_meetings = int(frequency * 4)
    increment = int(4//no_of_meetings)

    default_penalty = total_penalty + 10000 * no_of_meetings
    pos = []
    partial_attendees = []
    for i in range(4):
        for j in range(5):
            for k in range(16):
                temp_schedule = copy.deepcopy(employee_schedulex)
```

```
                slots_free = True
                owners_slots_free = True

                slots_free = Employee_free_slot(required_employees, i, j, k, meeting_length, temp_schedule)
                owners_slots_free = Employee_free_slot(meeting_owners, i, j, k, meeting_length, temp_schedule)

                if slots_free and owners_slots_free:
                    default_penalty,partial_attendees,pos = schedule_weekly_meeting(i, no_of_meetings,increment,combined_employees,meeting_length,temp_schedule,j,k, meeting_id,required_employees,required_employees,default_penalty,partial_attendees, pos)


                elif owners_slots_free and slots_free == False:
                    can_attendee = Free_attendees(required_employees, i, j, k, meeting_length, temp_schedule)
                    if len(can_attendee) >= threshold_attendees:
                        new_combined = can_attendee + meeting_owners

                        default_penalty,partial_attendees,pos = schedule_weekly_meeting(i, no_of_meetings,increment,new_combined,meeting_length,temp_schedule,j,k,meeting_id,required_employees,can_attendee,default_penalty,partial_attendees,pos)

    total_penalty = default_penalty

    if pos == []:
        return False, "None"

    for week in pos[0]:
        for empl in partial_attendees:
            for l in range(meeting_length):
                employee_schedulex.employees[empl].weeks[week].days[pos[1]].slots[pos[2]+l] = (meeting_id)

    penalty_type_value["Incomplete meetings"] += (len(combined_employees)-len(partial_attendees))


    if len(partial_attendees)!=len(combined_employees):
        percentage = len(partial_attendees)*100/len(combined_employees)
        sr = f"Partial {percentage}%"
        return True, sr
    return True, "Full"

ans_penalty = 9999999
meeting_sequence = []
# To find the best solution iterate over random sequences

with open('Result_File.txt', 'a') as the_file:
```

```
377
378     for till in range(200):
379
380         meeting_data = read_meetings('Meeting_Data.csv')
381         # Set the seed to produce same meeting sequences
382         random.seed(till)
383
384         random.shuffle(meeting_data)
385         current_sequence = []
386         employee_schedulex = EmployeeSchedule(37)
387
388         # Read the seniority information
389         senior_employee("Senior_Employees.csv", employee_schedulex)
390
391         # Iterate through each meeting in the meeting data
392         for meeting in meeting_data:
393             scheduled, printout = schedule_meeting(meeting)
394             if scheduled:
395                 current_sequence.append(meeting['Meeting ID'])
396
397         print("Iteration",till,"value",total_penalty)
398         addd = "Iteration "+str(till)+" value "+str(total_penalty)+"\n"
399         the_file.write(addd)
400
401         if total_penalty < ans_penalty:
402             ans_penalty = total_penalty
403             meeting_sequence = current_sequence
404
405 print("---------------------------------")
406 print("Total Penalty:",ans_penalty)
407 print("---------------------------------")
408 print(meeting_sequence)
409
410 # Save the result in a text file
411 with open('Results.txt','w') as file:
412     file.write(str(ans_penalty))
413     for x in meeting_sequence:
414         add = "'"+x+"', "
415         file.write(add)
```

The Greedy penalty-based scheduling for a given meeting sequence:

```
# -*- coding: utf-8 -*-
"""
@author: ss7u22
"""

import csv
import random
import copy
import math

class DailySchedule:
    def __init__(self):
        self.slots: dict[int, str] = {}
        for l in range(16):
            self.slots[l] = '0'

    def build(slots):
        obj = DailySchedule()
        obj.slots = slots
        return obj

    def __deepcopy__(self, memo):
        id_self = id(self)
        _copy = memo.get(id_self)
        if _copy is None:
            _copy = DailySchedule.build(copy.deepcopy(self.slots, memo))
            memo[id_self] = _copy
        return _copy

    # Number of meeting cluster in a day
    def count_clusters(self):
        is_slot_started = False
        num_clusters = 0
        for i in range(16):
            if self.slots[i] != '0':
                is_slot_started = True
            else:
                if is_slot_started:
                    num_clusters += 1
                is_slot_started = False
        if is_slot_started:
            num_clusters += 1
        return num_clusters

    # Check if meeting starts at 8AM everyday
    def meeting_start(self):
        is_meeting_started = False
        deviation = 0
        for i in range(16):
```

```
            if self.slots[i] != '0':
                is_meeting_started = True
                break
        if is_meeting_started:
            deviation = i
        return deviation

    # Check the length of meeting clusters each day
    # Return deviation from the ideal length
    def cluster_length(self):
        is_slot_started = False
        exceeding_slot = 0
        length = 0
        for i in range(16):
            if self.slots[i] != '0':
                is_slot_started = True
                length += 1
            else:
                if is_slot_started:
                    if length > 6:
                        exceeding_slot += length - 6;
                is_slot_started = False
                length = 0
        if is_slot_started:
            if length > 6:
                exceeding_slot += length - 6;
        return exceeding_slot

    # Check if the meeting finish by 3PM each day
    def finish_early(self):
        last_busy_slot = 0
        exceeding_slot = 0
        for i in range(16):
            if self.slots[i] != '0':
                last_busy_slot = i
        if last_busy_slot > 12:
            exceeding_slot = last_busy_slot - 12
        return exceeding_slot

    # Deviation from the floating lunch
    def floating_lunch(self):
        consecutive_slots = False
        start = 6
        deviation = 0
        while consecutive_slots == False and start+1 < 16:
            if self.slots[start] == '0' and self.slots[start+1] == '0':
                consecutive_slots = True
                if start > 8:
                    deviation = start - 8
            start += 1
        if consecutive_slots == False:
```

```
            deviation = 8
        return deviation

class WeeklySchedule:
    def __init__(self):
        self.days: dict[int, DailySchedule] = {}
        for k in range(5):
            self.days[k] = DailySchedule()

    def build(days):
        obj = WeeklySchedule()
        obj.days = days
        return obj

    def __deepcopy__(self, memo):
        id_self = id(self)
        _copy = memo.get(id_self)
        if _copy is None:
            _copy = WeeklySchedule.build(copy.deepcopy(self.days, memo))
            memo[id_self] = _copy
        return _copy

class Employee:
    def __init__(self):
        self.weeks: dict[int, WeeklySchedule] = {}
        self.is_senior = True
        for j in range(4):
            self.weeks[j] = WeeklySchedule()

    def build(is_senior, weeks):
        obj = Employee()
        obj.is_senior = is_senior
        obj.weeks = weeks
        return obj

    def __deepcopy__(self, memo):
        id_self = id(self)
        _copy = memo.get(id_self)
        if _copy is None:
            _copy = Employee.build(copy.deepcopy(self.is_senior, memo), copy.deepcopy(
    self.weeks, memo))
            memo[id_self] = _copy
        return _copy

class EmployeeSchedule:
    def __init__(self, num_employees: int):
        self.employees: dict[str, Employee] = {}
        for i in range(num_employees):
            employee = Employee()
            self.employees[f'p{i+1}'] = employee

```

```
    def build(employees):
        obj = EmployeeSchedule(0)
        obj.employees = employees
        return obj

    def __deepcopy__(self, memo):
        id_self = id(self)
        _copy = memo.get(id_self)
        if _copy is None:
            _copy = EmployeeSchedule.build(copy.deepcopy(self.employees, memo))
            memo[id_self] = _copy
        return _copy

    def print(self):
        for i, emp in self.employees.items():
            print(f'{i}:')
            emp.print()

# Define the number of employees
employee_schedulex = EmployeeSchedule(37)

def read_meetings(filename):
    meetings = []
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            meeting = {
                'Meeting ID': (row[0]),
                'Duration': math.ceil(int(row[1])/30),
                'Priority': int(row[2]),
                'Required Employees': [(e) for e in row[5].split()],
                'Frequency': float(row[3]),
                'Meeting Owners': [(e) for e in row[4].split()]
            }
            meetings.append(meeting)
    return meetings

def read_employee_schedule(filename, employee_schedulex: EmployeeSchedule):
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            employee_id = (row[0])
            week_id = int(row[1])
            day = int(row[2])
            F_slots = [(slot) for slot in row[3:]]
            for i in range(16):
                employee_schedulex.employees[employee_id].weeks[week_id-1].days[day-1].
    slots[i] = F_slots[i];

```

```
def senior_employee(filename, employee_schedulex: EmployeeSchedule):
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            employee_id = (row[0])
            is_senior = int(row[1])
            if is_senior == 0:
                employee_schedulex.employees[employee_id].is_senior = False
            else:
                employee_schedulex.employees[employee_id].is_senior = True


meeting_data = read_meetings('Meeting_Data.csv')

# Read the employee calendar for a warm start
# read_employee_schedule('Employee_Calendar.csv',employee_schedulex)

# Read the details about employee seniority
senior_employee("Senior_Employees.csv", employee_schedulex)
total_penalty = 0
penalty_type_value = {
    "Cluster Count":0,
    "Meeting Start Deviation":0,
    "Cluster Length Deviation":0,
    "Finish Early Deviation":0,
    "Floating Lunch Deviation":0,
    "Incomplete meetings":0
    }

normal_penalty = [30,15,100,60,10]

def calculate_penalty_for_employee(employee_n: Employee):
    factor = 1
    if employee_n.is_senior :
        factor = 2
    penalty1 = 0
    for i in range(4):
        for j in range(5):
            penalty1 += (employee_n.weeks[i].days[j].count_clusters())*30*factor
            penalty1 += (employee_n.weeks[i].days[j].meeting_start())*15*factor
            penalty1 += (employee_n.weeks[i].days[j].cluster_length())*100*factor
            penalty1 += (employee_n.weeks[i].days[j].finish_early())*60*factor
            penalty1 += 10**(employee_n.weeks[i].days[j].floating_lunch())*factor

    return penalty1

def calculate_penalty(employee_schedulex1: EmployeeSchedule):
    total = 0
    for empId, employee in employee_schedulex1.employees.items():
        total += calculate_penalty_for_employee(employee)
```

```
252     return total
253 
254 def Employee_free_slot(required_employees,i,j,k,meeting_length,temp_schedule):
255     free_slot = True
256     for empl in required_employees:
257         for l in range(meeting_length):
258             if (k+l) < 16:
259                 if temp_schedule.employees[empl].weeks[i].days[j].slots[k+l] != '0':
260                     free_slot = False
261             else:
262                 free_slot = False
263     return free_slot
264 
265 def Free_attendees(required_employees,i,j,k,meeting_length,temp_schedule):
266     free_slot = True
267     employees_free = []
268     for empl in required_employees:
269         free_slot = True
270         for l in range(meeting_length):
271             if (k+l) < 16:
272                 if temp_schedule.employees[empl].weeks[i].days[j].slots[k+l] != '0':
273                     free_slot = False
274             else:
275                 free_slot = False
276         if free_slot:
277             employees_free.append(empl)
278 
279     return employees_free
280 
281 def schedule_weekly_meeting(i,no_of_meetings,increment,new_combined,meeting_length,
    temp_schedule,j,k,meeting_id,required_employees,can_attendee,default_penalty,
    partial_attendees,pos):
282     possible = True
283     scheduled = 1
284     weeks = [i]
285     week = i
286 
287     while scheduled < no_of_meetings and possible == True:
288         if week + increment < 4:
289             weekly_slots_free = True
290             for empl in new_combined:
291                 for l in range(meeting_length):
292                     if temp_schedule.employees[empl].weeks[week+increment].days[j].slots
    [k+l] != '0':
293                         weekly_slots_free = False
294             if weekly_slots_free:
295                 weeks.append(week+increment)
296                 scheduled += 1
297 
298             week += increment
299         else:
```

```
            possible = False

    if possible == True and scheduled == no_of_meetings:
        for week_c in weeks:
            for empl in new_combined:
                for l in range(meeting_length):
                    temp_schedule.employees[empl].weeks[week_c].days[j].slots[k+l] = (
    meeting_id)
        #print(weeks,j,k)
        current_penalty = calculate_penalty(temp_schedule)

        # Penalty for less than 100% attendees
        current_penalty += (len(required_employees)-len(can_attendee))*500
        if current_penalty < default_penalty:
            default_penalty = current_penalty
            partial_attendees = new_combined
            pos = [weeks,j,k]
    return default_penalty,partial_attendees,pos

def schedule_meeting(meeting):
    global total_penalty
    global employee_schedulex
    meeting_id = meeting['Meeting ID']
    meeting_length = meeting['Duration']
    required_employees = meeting['Required Employees']
    frequency = meeting['Frequency']
    meeting_owners = meeting['Meeting Owners']

    required_employees = [i for i in required_employees if i not in meeting_owners]
    combined_employees = required_employees + meeting_owners

    total_attendees = len(required_employees)
    threshold_attendees = min(math.ceil(0.8*total_attendees),total_attendees-1)

    no_of_meetings = int(frequency * 4)
    increment = int(4//no_of_meetings)

    default_penalty = total_penalty + 10000 * no_of_meetings
    pos = []
    partial_attendees = []
    for i in range(4):
        for j in range(5):
            for k in range(16):
                temp_schedule = copy.deepcopy(employee_schedulex)
                slots_free = True
                owners_slots_free = True

                slots_free = Employee_free_slot(required_employees, i, j, k,
    meeting_length, temp_schedule)
                owners_slots_free = Employee_free_slot(meeting_owners, i, j, k,
    meeting_length, temp_schedule)
```

```
348
349                 if slots_free and owners_slots_free:
350                     default_penalty,partial_attendees,pos = schedule_weekly_meeting(i,
    no_of_meetings,increment,combined_employees,meeting_length,temp_schedule,j,k,
    meeting_id,required_employees,required_employees,default_penalty,partial_attendees,
    pos)
351
352
353                 elif owners_slots_free and slots_free == False:
354                     can_attendee = Free_attendees(required_employees, i, j, k,
    meeting_length, temp_schedule)
355                     if len(can_attendee) >= threshold_attendees:
356                         new_combined = can_attendee + meeting_owners
357
358                         default_penalty,partial_attendees,pos = schedule_weekly_meeting(
    i, no_of_meetings,increment,new_combined,meeting_length,temp_schedule,j,k,meeting_id
    ,required_employees,can_attendee,default_penalty,partial_attendees,pos)
359
360     total_penalty = default_penalty
361
362     if pos == []:
363         return False, "None"
364
365     for week in pos[0]:
366         for empl in partial_attendees:
367             for l in range(meeting_length):
368                 employee_schedulex.employees[empl].weeks[week].days[pos[1]].slots[pos[2
    ]+l] = (meeting_id)
369
370     penalty_type_value["Incomplete meetings"] += (len(combined_employees)-len(
    partial_attendees))
371
372
373     if len(partial_attendees)!=len(combined_employees):
374         percentage = len(partial_attendees)*100/len(combined_employees)
375         sr = f"Partial {percentage}%"
376         return True, sr
377     return True, "Full"
378
379 # counter for storing different result files
380 ok = 1
381 # Function to write the updated employee schedules to a CSV file
382 def write_updated_schedules_to_csv():
383     global ok
384     global employee_schedulex
385     with open(f'New_Employee_Calendar{ok}.csv', 'w', newline='') as file:
386         writer = csv.writer(file)
387         writer.writerow(['Employee ID', 'Week', 'Day', 'Slot 1', 'Slot 2', 'Slot 3', '
    Slot 4', 'Slot 5', 'Slot 6', 'Slot 7', 'Slot 8', 'Slot 9', 'Slot 10', 'Slot 11', '
    Slot 12', 'Slot 13', 'Slot 14', 'Slot 15', 'Slot 16'])
388         for employee_id, schedule in employee_schedulex.employees.items():
```

```
            for week, schedule_week in schedule.weeks.items():
                for day, schedule_day in schedule_week.days.items():
                    sch = [employee_id,week+1,day+1]
                    for slot, meeting_id in schedule_day.slots.items():
                        sch.append(meeting_id)
                    writer.writerow(sch)
    ok+=1

# The best solution seed from the competitive heuristics
random.seed(139+87876)
random.shuffle(meeting_data)

unscheduled = 0
unsch_id = []
priority_score = 0

# Sort the meeting for ordered sequence
# meeting_data.sort(key=lambda x: (x['Duration'], x['Priority']))

sequence = []
for meeting in meeting_data:
    sequence.append(meeting['Meeting ID'])

# Print the meeting sequence
for i in sequence:
    print("'",i,"', ",end="",sep="")

# Iterate through each meeting in the meeting data
for meeting in meeting_data:
    scheduled, printout = schedule_meeting(meeting)
    if scheduled:
        print(meeting['Meeting ID'],' scheduled ',printout)
        priority_score += meeting['Priority']
    else:
        unscheduled += 1
        unsch_id.append(meeting['Meeting ID'])

# Write the updated employee schedules to a CSV file
write_updated_schedules_to_csv()

# Update Penalty Dictionary
for empId, employee_n in employee_schedulex.employees.items():
    for i in range(4):
        for j in range(5):
            penalty_type_value["Cluster Count"] = penalty_type_value["Cluster Count"] +
    employee_n.weeks[i].days[j].count_clusters()
            penalty_type_value["Meeting Start Deviation"] = penalty_type_value["Meeting
    Start Deviation"] + employee_n.weeks[i].days[j].meeting_start()
            penalty_type_value["Cluster Length Deviation"] = penalty_type_value["Cluster
     Length Deviation"] + employee_n.weeks[i].days[j].cluster_length()
            penalty_type_value["Finish Early Deviation"] = penalty_type_value["Finish
```

```
    Early Deviation"] + employee_n.weeks[i].days[j].finish_early()
437             penalty_type_value["Floating Lunch Deviation"] = penalty_type_value["
    Floating Lunch Deviation"] + employee_n.weeks[i].days[j].floating_lunch()
438
439 print("---------------------------------")
440 print("Total Penalty:",total_penalty)
441 print("---------------------------------")
442 print("Unscheduled meetings:",unscheduled)
443 print("Priority Score:",priority_score)
444 print("Meeting Clusters Count:",penalty_type_value["Cluster Count"])
445 print("Meeting Start Deviation:",penalty_type_value["Meeting Start Deviation"])
446 print("Cluster Length Deviation:",penalty_type_value["Cluster Length Deviation"])
447 print("Finish Early Deviation:",penalty_type_value["Finish Early Deviation"])
448 print("Floating Lunch Deviation:",penalty_type_value["Floating Lunch Deviation"])
449 print("Partial Attendees Meetings:",penalty_type_value["Incomplete meetings"])
```

Weekly calendar visualisation code:

```
# -*- coding: utf-8 -*-
"""
@author: ss7u22
"""

import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from datetime import datetime, timedelta
import csv

# Select Week
week = '4'
# Select Employee
person = 'p4'

schedule = []

# Read the weekly data
with open('Employee_Calendar.csv', newline='') as csvfile:
    csvreader = csv.DictReader(csvfile)
    for row in csvreader:
        if row['Employee ID'] == person and row['Week'] == week:
            day = int(row['Day'])
            slot = 1
            while slot < 17:
                if row[f"Slot {slot}"] != '0':
                    Meeting_code = row[f"Slot {slot}"]
                    if (int((slot-1)//2)+8) < 10:
                        minute = 3
                        if (slot)%2 == 1:
                            minute = 0
                        start_time_str = f"0{(int((slot-1)//2)+8)}:{minute}0"
                    else:
                        minute = 3
                        if (slot)%2 == 1:
                            minute = 0
                        start_time_str = f"{(int((slot-1)//2)+8)}:{minute}0"

                    duration = 0
                    while slot < 17 and row[f"Slot {slot}"] == Meeting_code:
                        duration += 1
                        slot += 1

                    slot += (-1)
                    start_time = datetime.strptime(start_time_str, '%H:%M')
                    end_time = start_time + timedelta(minutes=duration*30)
                    schedule.append([(start_time, end_time), Meeting_code, day])

                slot += 1
```

```
schedule.append([(datetime(1900, 1, 1, 7, 0), datetime(1900, 1, 1, 7, 0)), '', 1])
schedule.append([(datetime(1900, 1, 1, 7, 0), datetime(1900, 1, 1, 7, 0)), '', 2])
schedule.append([(datetime(1900, 1, 1, 7, 0), datetime(1900, 1, 1, 7, 0)), '', 3])
schedule.append([(datetime(1900, 1, 1, 7, 0), datetime(1900, 1, 1, 7, 0)), '', 4])
schedule.append([(datetime(1900, 1, 1, 7, 0), datetime(1900, 1, 1, 7, 0)), '', 5])


# Create figure and axes
fig, ax = plt.subplots(figsize=(8, 5))

# Define a custom color map for events
colors = plt.cm.tab20.colors


for day in (range(1, 6)):
    day_schedule = [[event, code, event_day] for event, code, event_day in schedule if event_day == day]
    for i, [(start_time, end_time), event, _] in enumerate(day_schedule):
        ax.bar(day, height=(end_time - start_time),
                bottom=(start_time), width=0.7, color=colors[i % len(colors)],
                alpha=0.7, label=event, edgecolor = "black",zorder=7)
        ax.text(day,(end_time - start_time)/2 + start_time + timedelta(seconds = 300), event,ha="center",
                color = "black", size = 12,zorder=9)

    if day < 5:
        ax.axvline(x=day + 0.5, linestyle='-', color='gray')

# Set plot aesthetics
ax.set_xticks((range(1, 6)))
ax.set_xticklabels(['D.{}'.format(day) for day in (range(1, 6))])  # Display day labels
ax.yaxis_date()
ax.yaxis.set_major_locator(mdates.HourLocator(interval=1))
ax.yaxis.set_major_formatter(mdates.DateFormatter('%H:%M'))

# # Customize x-axis
ax.set_ylim(datetime(1900, 1, 1, 16, 0), datetime(1900, 1, 1, 7, 0))
ax.set_ylabel('Time of Day')

# Add title
title = "Weekly Schedule for Person " + person + " Week " + week
ax.set_title(title, fontsize=14)
ax.yaxis.grid(True,zorder=0)

plt.tight_layout()
# Save the plot
plt.savefig(f"Final_Algo_Real_Person{person}-Week{week}_Sorted.png",dpi=300)
# Show the plot
plt.show()
```

References

Burke, E. K., De Causmaecker, P., & Vanden Berghe, G. (1999, January). *A hybrid Tabu search algorithm for the nurse rostering problem*.

Chu, S. C. K. (2007). Generating, scheduling and rostering of shift crew-duties: Applications at the Hong Kong International Airport. *European Journal of Operational Research*, *177*(3), 1764–1778.

Clarke, G., & Wright, J. W. (1964). Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. *Operations Research*, *12*(4), 568–581.

Dantzig, G. B., Fulkerson, R., & Johnson, S. M. (1954). Solution of a Large-Scale Traveling-Salesman problem. *Journal of the Operations Research Society of America*, *2*(4), 393–410.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271.

Dorigo, M., & Di, G. A. (2003). Ant colony optimization: a new meta-heuristic. *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*.

Feo, T. A., & Resende, M. G. C. (1989). A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, *8*(2), 67–71.

Fischer, M. J. (1973). Near-optimal bin packing algorithms.

Goldberg, D. E. (2002, January). *Genetic Algorithms in Search, Optimization, and Machine Learning*.

How much does ford motor company (uk) pay in 2023? (175 salaries). (n.d.). https://www.glassdoor.co.uk/Salary/Ford-Motor-Company-UK-Salaries-E152869_P4.htm

Karmarkar, N., & Karp, R. M. (1982). An efficient approximation scheme for the one-dimensional bin-packing problem. *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, 312–320. https://doi.org/10.1109/SFCS.1982.61

Musliu, N., Schaerf, A., & Slany, W. (2004). Local search for shift design. *European Journal of Operational Research*, *153*(1), 51–64.

Nawaz, M., Enscore, E. E., & Ham, I. (1983). A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, *11*(1), 91–95.

Özder, E. H., Özcan, E., & Eren, T. (2020). A Systematic Literature Review for personnel scheduling Problems. *International Journal of Information Technology and Decision Making*, *19*(06), 1695–1735.

Powell, M. J. D. (1974). A Nurse-Scheduling Model. *Management Science*, *20*(5), 822–829.

Rosenkrantz, D. J., Stearns, R. E., & Lewis, P. (1977). An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, *6*(3), 563–581.

Singh, S. (2023, September). *Calendar Scheduling Algorithm*. https://github.com/saumyas16/Calendar-Scheduling-Problem

Van Den Bergh, J., Belien, J., De Bruecker, P., Demeulemeester, E., & De Boeck, L. (2013). Personnel scheduling: A literature review. *European Journal of Operational Research*, *226*(3), 367–385.

Vazirani, V. V. (2013, March). *Approximation algorithms*. Springer Science & Business Media.