M.Tech Project Dissertation

on

Developing a Software for Crew Scheduling

Master of Technology

Stage - I Project Report

by

Rupesh Anil Sonawane

(Roll No. 23m1533)

Supervisor:

Prof. Ashutosh Mahajan

Department of
Industrial Engineering and Operations Research (IEOR)

Indian Institute of Technology, Bombay
Mumbai 400076 India

Declaration

I affirm that this written submission is a representation of my own ideas, expressed in my own words. Wherever I have used the ideas or words of others, I have appropriately cited and referenced the original sources. I confirm that I have accurately acknowledged all sources used in the creation of this report. I also affirm that I have upheld all standards of academic integrity and honesty, ensuring that no idea, data, fact, or source has been misrepresented, fabricated, or falsified in this submission. I am aware that any breach of these principles may result in disciplinary action by the Institute and could also lead to legal consequences if sources are not properly cited or permissions were not obtained when required.

Name: Rupesh Anil Sonawane

Signature: 

Date: October 23, 2025

Acknowledgements

I would like to express my sincere gratitude to **Prof. Ashutosh Mahajan** from the Department of Industrial Engineering and Operations Research for his supervision and unwavering support throughout this project.
I am also thankful to **Prof. Madhu Belur** from IIT Bombay for his valuable suggestions and advice during the project. Additionally, I would like to acknowledge the collaboration of **Rishuv Gorka**, **Aadesh Jain** and **Sarvar E Abbas**

The works and contributions of others, mentioned in the reference section, have been instrumental in shaping this project. I am deeply indebted to all of them for their assistance and inspiration

Date: October 23, 2025

Abstract

This thesis work presents a comprehensive study on optimizing crew scheduling for mass transit metro systems using network based approach, with a specific focus on the Delhi Metro's Pink Line. Crew scheduling in large-scale metro systems, such as the Delhi Metro, presents a complex combinatorial optimization problem that involves generating feasible crew duties while adhering to multiple operational and regulatory constraints. Existing recursive and backtracking methods, though effective in small-scale setups, become computationally expensive and time-consuming when applied to dense timetables.

To address this challenge, this work introduces a network-based crew scheduling framework that models all train services as nodes within a directed acyclic network, with feasible service connections represented as edges. By constructing this service connectivity network, the scheduling problem transforms into an efficient path-tracing task, where valid crew duties correspond to constrained paths within the network. A stack-based Depth-First Search (DFS) algorithm is employed to explore these paths iteratively, enabling systematic traversal without the overhead of recursive backtracking.

This network-driven approach significantly enhances scalability and computational efficiency while maintaining strict compliance with operational constraints such as duty time limits, continuous driving rules, and required breaks. The proposed method demonstrates a substantial improvement in duty generation speed and solution robustness compared to previous recursive approaches, making it a practical and efficient solution for metro crew scheduling systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Efficient crew scheduling plays a vital role in the daily operation of any large-scale transportation system such as the Delhi Metro. Crew scheduling refers to the process of assigning locomotive pilots (train operators) to various train services throughout the day in such a way that operational requirements are met while adhering to labor regulations and minimizing overall costs. A well-designed crew schedule ensures that every scheduled train service is adequately staffed, while also maintaining fairness, safety, and compliance with organizational and statutory rules.

In the context of the Delhi Metro, crew scheduling is particularly challenging due to the dense network structure, high service frequency, and the need to respect multiple operational constraints. These include adherence to the Hours of Employment Regulations (HOER), which govern the maximum duty duration, driving time limits, required short and long breaks, and jurisdictional boundaries. Additional considerations such as rake change stations, break locations, and continuous driving limits must also be observed to ensure both safety and efficiency.

Traditional methods of manual or rule-based scheduling are not scalable to the complexity and size of the Delhi Metro network. Therefore, computational approaches are essential to automate and optimize the scheduling process. The crew scheduling problem is typically divided into two phases:

1. **Feasible Duty Generation** – The process of identifying and listing every potential and authorized duty (which are specific work sequences) that a crew member is able to execute within the specified restrictions.

2. **Duty Optimization** – Selecting the optimal set of duties that covers all required services while minimizing the total number of duties often formulated as a **Set Covering Problem (SCP)**.

In this project, the primary focus is on efficient generation of feasible duties. To achieve this, a network-based approach is adopted, where each train service is represented as a node in a graph and feasible connections between services form the edges. A Stack-based Depth First Search (DFS) technique is then employed to traverse this graph and trace all possible feasible duty paths while adhering to HOER and operational constraints.

Furthermore, various optimization and object-oriented programming (OOP) techniques are integrated to speed up the path generation process and ensure modular, maintainable code. Once the set of feasible duties is generated, optimization techniques such as Set Covering can be applied to select the optimal subset of duties.

Overall, the main goal of this work is to develop a fast and reliable network-based framework for feasible duty generation that can serve as a foundation for subsequent optimization and decision-making stages in Delhi Metro crew scheduling.

We analyze the train services on the Pink Line, which are managed by the Delhi Metro (as illustrated in Figure 1.1), and integrate the relevant crew allocation regulations and policies specific to this operational context.

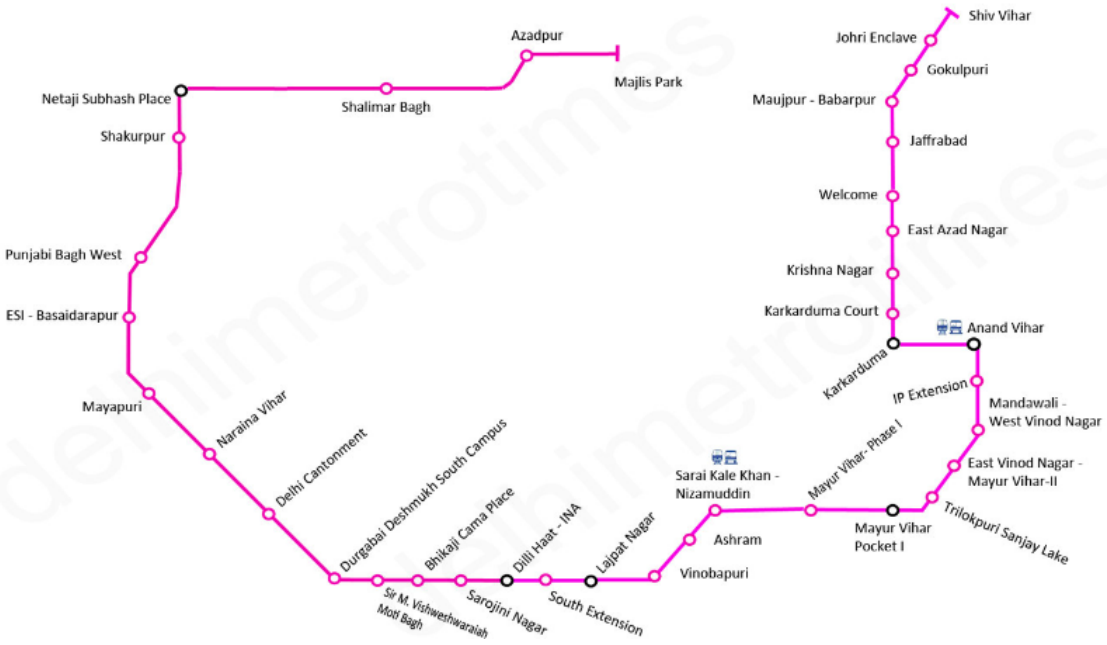

Figure 1.1: Delhi Metro Pink Line [13]

The complexities arise from the presence of dual crew control points, where crew members sign on and off within the same jurisdiction. This thesis introduces a heuristic-based optimization framework designed to efficiently generate feasible duty sets across all Delhi Metro lines while minimizing the total number of duties. The proposed

approach aims to support crew planners in preparing schedules more rapidly and conveniently. In addition, the method can substantially reduce crew-related costs and the time required to produce new schedules whenever modifications occur in the train timetable.

Terminology used in the report

Trip Chart means creating a set of trips as per the given timetable by following concerned standard operating procedures and other requirements as per DMRC policies to optimize the manpower and other resources which shall be further allocated to the train operator as their rosters.

Roster means the tabular form of an assigned set of trips to the Train Operator's duty for a particular day that abided with the standard operating procedure and roster policy.

Crew Control is a designated location where crew members are managed, scheduled, assigned, and monitored as a part of an overall management system.

Crew Control Jurisdiction means the location of crew control, depot, outstation stabling, stepping back, and mainline governed by a particular crew control.

Stepping Back locations are the stations or terminals where the train terminates and starts again in reverse direction. It means a system to decrease turnaround time at terminal stations where a second TO enters the rear cab of an arriving train

Key Locations means the important locations in the context of train operations like crew control, depot, terminal stations and outstation stabling locations etc.

Sign On means reporting for duty at a depot or crew control office or any other designated place by train operator.

Sign Off means reporting by the train operator to depot or crew control at the end of their duties.

Service encompasses the movement of the train from the starting station to its destination station, making scheduled stops at various intermediate stations along the line.

Set means a sequence of services (or tasks) that are combined as a single duty generated by following the operational constraints and HOER guidelines.

1.2 Problem Description

Introduction

This chapter offers an introduction to the crew scheduling challenge, highlighting its crucial role in metro rail operations. We will examine the specific rules and regulations that govern this process within the Delhi Metro Rail Corporation (DMRC). Crew scheduling is essential for the effective deployment of human resources while simultaneously guaranteeing safety, uninterrupted service, and adherence to all regulatory standards.
The primary goal is to allocate crew members to a predefined set of train services. This allocation must satisfy all operational and legal restrictions while aiming to minimize the total required duties (thereby maximizing crew efficiency). The nature of this problem is combinatorial, and it has been the subject of extensive research in railway and metro systems globally.

1.3 Crew Allocation Rules

Delhi Metro crew allocation is governed by the "Hours of Work and Periods of Rest (HOER)" rules prescribed by the organization. These rules are subject to periodic updates. The key constraints considered in this study are as follows:

1. **Service Continuity:** The end station of a service must be the same as the start station of the following service in the duty.

$$\text{EndStation}(S_i) = \text{StartStation}(S_{i+1}), \quad \forall i \in \{1, 2, \ldots, k-1\}$$

2. **Time Feasibility:** The end time of a service must not exceed the start time of the subsequent service.

$$\text{EndTime}(S_i) \leq \text{StartTime}(S_{i+1}), \quad \forall i \in \{1, 2, \ldots, k-1\}$$

3. **Jurisdiction Constraint:** A duty must begin and end in the same jurisdiction, ensuring logistical feasibility.

4. **Rake Change Constraint:** Rake changes are allowed only at the stations **KKDA** and **PVGW**.

5. **Maximum Duty Hours:** The total time of a duty must not exceed 445 minutes. ***Special case:*** For morning duties (start before 6:00 AM) and evening duties (start after 23:30 PM), the maximum duty hours are restricted to 405 minutes.

$$\text{DutyDuration} \leq 445 \text{ minutes},$$
$$\text{DutyDuration} \leq 405 \text{ minutes for morning/evening shifts.}$$

6. **Maximum Driving Time:** The cumulative driving duration within a duty must not exceed 360 minutes. The driving duration is calculated as the sum of service times and any gap between consecutive services that is less than 30 minutes:

$$\text{DrivingDuration} = \sum_{i=1}^{k} \text{ServiceTime}(S_i) + \sum_{\substack{i=1 \\ \text{Gap}(S_i,S_{i+1})<30}}^{k-1} \text{Gap}(S_i, S_{i+1})$$

$$\text{DrivingDuration} \leq 360 \text{ minutes}$$

7. **Continuous Driving Limit:** No more than 180 minutes of continuous driving is allowed without a 30 min break.

$$\text{ContinuousDrivingTime} \leq 180 \text{ minutes}$$

8. **Break Conditions:** A valid path must have at least one break at a valid break station (KKDA or PVGW), and at least one break must occur within the jurisdiction of the first duty's start station.

   The total duration of all breaks must be less than 120 minutes.

   The breaks must satisfy one of the following cases:

   Case 1: There is exactly one break of at least 50 minutes.

   Case 2: There are multiple breaks. If there is any break of 30 minutes, there must also be at least one break of 50 minutes or more. Two breaks of 50 minutes or more are also valid. Three breaks of 30 minutes or more, 30 minutes or more and 50 minutes or more are also valid

1.4 Literature Review

Crew scheduling in the transportation sector has been an active area of research since the mid-20th century, initially motivated by challenges in airline and bus operations. The process of assigning crew members follows strict operational and regulatory frameworks that specify limits on duty durations and mandatory rest intervals. These regulations often differ across regions or countries and must be adapted to comply with local labor and transport policies.

Heil [9] offers a comprehensive review of advancements in railway crew scheduling. It traces the evolution of the field from its origins in airline and bus industries to its current state, highlighting the increasing complexity and scale of scheduling problems. The authors categorize over 130 studies published since 2000, focusing on three primary model formulations: set covering, set packing, and network flow models. These models aim to optimize various objectives, including schedule efficiency, robustness, and employee satisfaction. The paper also discusses the integration of operational constraints, such as legal regulations and practical operational rules, into scheduling models. Furthermore, it emphasizes the importance of developing decision support tools that allow railway operators to compare multiple feasible crew plans and select the most appropriate one for current situations.

Crew scheduling problems are typically formulated as mathematical models, often using either set covering or set partitioning, and are subsequently addressed using exact solution techniques or heuristics. For instance, Desrochers and Soumis [8] developed a column generation approach specifically for urban transit crew scheduling. Following this, Borndörfer et al. [6] applied analogous set partitioning and column generation methods for airline crew scheduling, while Lusby et al. [11] created a column generation-based heuristic tailored for ground crew rostering that incorporated work patterns. This body of research illustrates various methods for efficiently solving complex, large-scale crew scheduling problems.
In our specific context, involving 944 Delhi Metro services daily, we employ a network-based approach for feasible path generation that utilizes Depth-First Search (DFS) in conjunction with set covering to identify the optimal duties. The primary goal is to minimize the total quantity of duties, subject to operational constraints governing crew location, continuous driving time, mandated rest periods, and meal hours within a shift. A final schedule is defined as feasible if it satisfies two conditions: first, every task is assigned to precisely one duty, and second, each duty constitutes a valid sequence of tasks that can be performed by a single crew member.

Exploring Techniques for Duty Generation

Network Modeling for Crew Scheduling

Network models have been widely used to solve vehicle and crew scheduling problems, especially in mass transit systems. In these models, tasks such as shifts or duties are represented as nodes, and feasible transitions between tasks are represented as arcs, forming a network flow. This structure allows efficient representation of dependencies and precedence constraints, making it easier to enforce rules like connection gaps, breaks, and rake restrictions.

Directed Acyclic Graphs (DAGs) are particularly suitable for crew scheduling because they can model tasks and their dependencies while ensuring no cycles occur. In our approach, we represent the scheduling problem as a DAG with a start node, an end node, and service nodes in between. Each path from start to end represents a feasible duty sequence. We adopted this graph-based approach inspired by studies such as [10], which demonstrated the effectiveness of network representations in urban transit crew scheduling. Although we did not use column generation, the network structure provides a clear and intuitive way to model feasible sequences and enforce constraints dynamically. This approach simplifies path enumeration, allows efficient backtracking using a stack, and makes it easier to incorporate additional operational rules while traversing the network, making it both practical and easy to implement for large-scale scheduling scenarios.

Stack

A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle, where the last element added is the first to be removed [1]. The two primary operations are push, which adds an element to the top of the stack, and pop, which removes the top element. In the context of path enumeration, a stack is particularly useful because it naturally supports depth-first traversal, maintaining the current path and allowing efficient backtracking when a path reaches a dead-end or completes. We adopted the stack-based approach for path traversal in directed acyclic graphs (DAGs) based on methods described in [7], which showed that stacks can efficiently enumerate all paths while handling constraints effectively.

Using a stack provides several advantages for path enumeration. It is memory efficient since only the current path needs to be stored, avoiding the need to keep all possible paths in memory simultaneously. It supports backtracking easily, as nodes can be pushed when moving forward and popped when backtracking. The implementation is straightforward and simple to debug. It enables deterministic traversal order, allowing predictable depth-first exploration of paths, and it is flexible, as it can be combined with pruning or memoization to handle additional constraints efficiently. For service

or duty graphs, the stack can be used by pushing a service node onto the stack when exploring a valid connection, continuing to push nodes as long as subsequent services satisfy all constraints such as connection gap, rake, or breaks, and popping nodes to backtrack if a dead-end is reached to explore alternative paths. Complete feasible paths can be recorded when the end node is reached, with the stack containing the sequence. This approach allows enumeration of all feasible duty sequences without using recursion, making it easier to dynamically enforce constraints at each step while traversing the graph.

In this study, all crew assignment policies and operational guidelines relevant to the Delhi Metro are taken into account to create conflict-free duty schedules that remain valid across multiple operating days. Conventional formulations for metro crew scheduling often overlook such system-specific requirements and thus cannot be directly implemented in the Delhi Metro environment. Considering that DMRC manages 944 services each day, a network-based strategy has been developed where feasible duties are identified using a stack-driven depth-first search (DFS) procedure. After generating the complete duty set, a set covering model is employed to compute the minimum number of duties required, while satisfying all operational constraints such as rest intervals, continuous driving limits, and maximum allowable duty duration per day or week. This framework consolidates insights from prior studies and provides a practical and efficient scheduling methodology.

Chapter 2

Solution Methodology

2.1 Introduction

This chapter describes the overall methodology adopted to solve the crew scheduling problem for the Delhi Metro using a network-based approach. The primary objective is to generate feasible duties efficiently while satisfying all operational and regulatory constraints defined under the Hours of Employment Regulations (HOER).

The proposed framework consists of two major stages. First, the services operated by the Delhi Metro throughout the day are represented in the form of a directed graph, where each service acts as a node and feasible service connections form the edges. Second, a Stack-based Depth First Search (DFS) technique is applied to this graph to trace all possible duty paths that satisfy the defined constraints such as duty duration, continuous driving limits, and break requirements.

This chapter explains the design choices, data representation, and algorithmic flow used in the development of the solution. The approach not only ensures the feasibility of generated duties but also focuses on computational efficiency to handle the large volume of services present in the Delhi Metro network.

2.2 Graph Construction

To model feasible connections between services, we construct a directed graph where each node represents a service, including dummy start and end nodes. The graph captures all possible feasible transitions between services according to the operational constraints.

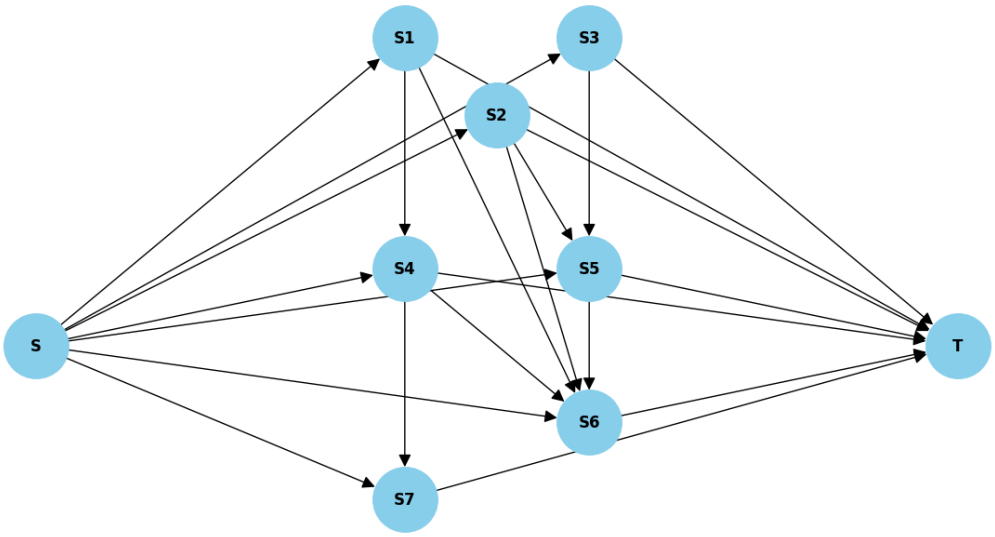

Figure 2.1: Service Network with Dummy Nodes

Nodes

- Each service in the dataset is represented as a node in the graph.
- Two additional dummy nodes, start and end, are added to represent the beginning and end of a duty sequence.

Start and End Connections

- The dummy start node is connected to all valid services that can begin a duty sequence.
- All services are connected to the dummy end node to allow paths to terminate properly.

Edges and Feasible Connections

Edges represent feasible transitions between services. An edge from service s_1 to service s_2 is added if the following conditions are satisfied:

1. The end station of s_1 matches the start station of s_2.
2. The connection gap between services satisfies $0 \leq \text{start}_2 - \text{end}_1 \leq 120$ minutes. (The maximum and cumulative break duration is 120 minutes, which is why we set the upper limit to 120 minutes.)
3. Rake constraints:

- If s_1 and s_2 share the same rake, the connection is allowed directly.
- If the rakes are different, the connection is allowed only at KKDA or PVGW stations, and the gap must be at least 30 minutes.

Modeled Constraints in the Graph

By constructing the graph this way, the following constraints are implicitly enforced in all feasible paths:

- **Station continuity:** Services must connect at the same station.
- **Maximum connection gap:** Gaps between services cannot exceed 120 minutes.
- **Rake consistency:** Connections respect the same rake or allowed rake-change rules.
- **Minimum rake-change gap:** When changing rakes, a minimum gap of 30 minutes is enforced at allowed stations.
- **Start/End handling:** All paths originate from the dummy start node and terminate at the dummy end node.

This graph structure allows efficient enumeration of all feasible duty sequences while automatically respecting the operational and rake-change constraints. It ensures that only valid service connections are considered, eliminates infeasible paths early, and provides a flexible framework to incorporate additional constraints, such as break durations, duty limits, and jurisdiction requirements, without modifying the underlying graph structure.

2.3 Path Tracing in a Graph

2.3.1 Path Feasibility Check on Partial Paths

The function `is_path_acceptable()` plays a crucial role in validating whether a currently building (partial) path in the duty enumeration process remains feasible under the operational constraints. Since the path is being dynamically constructed during graph traversal (using a stack-based approach), not all constraints can be checked at every intermediate step. Therefore, this function focuses only on those constraints that can be evaluated incrementally — i.e., constraints that depend on the **current subset of services** rather than the entire completed path.

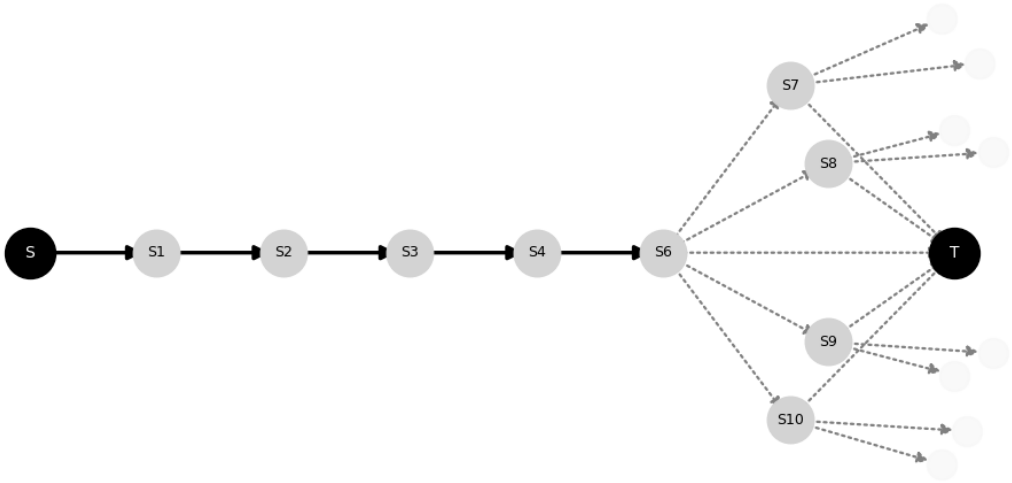

Figure 2.2: Tracing Path

The main checks implemented in this function are as follows:

1. Driving Time Limit Check

The first condition ensures that the total accumulated driving time in the current path does not exceed the global limit of 360 minutes. This limit represents the maximum allowable driving time for a single crew within a duty.

$$\text{TotalDrivingTime} \leq 360$$

If this limit is exceeded at any stage, the path is immediately rejected to avoid unnecessary exploration of infeasible extensions.

2. Duty Duration Limit Check

The second required check involves calculating the total duty duration. This duration is determined by measuring the time elapsed between the start time of the initial service (excluding the dummy `SOURCE` node) and the end time of the final service in the sequence. The resulting effective duty duration is subsequently evaluated against a permissible time limit, which is variable depending on whether the assigned duty is classified as a morning/evening shift or a normal shift.

$$\text{DutyDuration} = \text{EndTime}_{\text{last}} - \text{StartTime}_{\text{first}}$$

$$\text{DutyDuration} \leq \begin{cases} 405, & \text{if start time is before 6:00 AM or after 11:30 PM} \\ 445, & \text{otherwise (normal duty)} \end{cases}$$

If the duty duration exceeds the corresponding threshold, the path is pruned immediately.

3. Continuous Driving Rule

A key operational rule limits the crew from driving continuously for more than 180 minutes without a break of at least 30 minutes. The function iterates over consecutive services in the path to compute a running total of driving and gap times.

- If the gap between two services is shorter than the SHORT_BREAK duration (30 minutes), it is added to the continuous driving counter.
- If the gap exceeds or equals 30 minutes, the counter resets, indicating a valid rest break.
- If the accumulated continuous driving time ever exceeds 180 minutes, the path is deemed infeasible and is immediately discarded.

$$\text{ContinuousDrivingTime} \leq 180 \quad \text{before any valid break}$$

4. Early Pruning of Infeasible Paths

All the above checks enable **early pruning** — meaning that paths violating any of these rules are rejected before reaching the sink node (`T`). This greatly improves computational efficiency by preventing unnecessary exploration of infeasible extensions.

In summary, the `is_path_acceptable()` function acts as a real-time filter that enforces time-based operational constraints while the path is still being constructed. It ensures that only feasible partial paths are extended further in the graph traversal, significantly reducing computational overhead and ensuring operational compliance throughout the enumeration process.

Algorithm 1 Check if a Partial Path is Acceptable:

`is_path_acceptable()`

Require: *path*, *end_service*, *total_driving_time*, $DRIVING\_TIME\_LIMIT$, $DUTY\_TIME\_LIMIT$

Ensure: Returns TRUE if path satisfies all operational constraints

Condition 1: Driving Time Limit
if $total\_driving\_time > DRIVING\_TIME\_LIMIT$ **then**
    return FALSE
end if
Condition 2: Duty Time Limit
$first\_start \leftarrow path[1].start\_time$
if $path[-1] == end\_service$ **then**
    $last \leftarrow path[-2]$
else
    $last \leftarrow path[-1]$
end if
$total\_duty \leftarrow last.end\_time - first\_start$
if $first\_start < MORNING\_SHIFT\_CUTOFF$ **or** $first\_start > EVENING\_SHIFT\_CUTOFF$ **then**
    $effective\_duty\_limit \leftarrow MORN\_EVEN\_DUTY\_TIME\_LIMIT$
else
    $effective\_duty\_limit \leftarrow DUTY\_TIME\_LIMIT$
end if
if $total\_duty > effective\_duty\_limit$ **then**
    return FALSE
end if
Condition 3: Continuous Driving Limit
$continuous\_drive \leftarrow 0$
$end\_index \leftarrow$ length(path) - 1 if $path[-1] == end\_service$ else length(path)
for $i = 1$ to $end\_index - 1$ **do**
    $current \leftarrow path[i]$
    $continuous\_drive \leftarrow continuous\_drive + current.service\_time$
    if $i < end\_index - 1$ **then**
        $next \leftarrow path[i + 1]$
        $gap \leftarrow next.start\_time - current.end\_time$
        if $gap < SHORT\_BREAK$ **then**
            $continuous\_drive \leftarrow continuous\_drive + gap$
        else
            $continuous\_drive \leftarrow 0$
        end if
    end if
    if $continuous\_drive > CONTINUOUS\_DRIVE\_LIMIT$ **then**
        return FALSE
    end if
end for
return TRUE

2.3.2 Path Feasibility Check on Fully Constructed Paths

The final validation stage is performed by a small set of tightly-focused functions: `get_jurisdiction_groups()`, `is_final_path_valid()` and `has_required_breaks()`. Together these functions apply the constraints that can only be verified once a path is fully constructed (i.e., when it terminates at the sink node T). The checks are arranged to maximize early rejection of infeasible paths while keeping expensive work to a minimum. Figure 2.3 shows the fully constructed duties in which service_ids are shown. Each row represents a unique duty.

```
573,275,855,862,314,353,372,564
825,832,861,868,320,542,552,368,939,569,943
270,518,537,344,920,565,926,937,940,376
819,826,278,523,538,347,922,566,929,941,942,576
260,513,298,888,894,345,371
502,514,299,879,886,903,909,357,562
501,511,293,873,880,907,913,360,559
```

Figure 2.3: Fully constructed duties

Overview.

- **get_jurisdiction_groups(station)** returns the set of jurisdiction groups a station belongs to. It is cached (via `lru_cache`) and returns a `frozenset` for stability in cached results and fast set operations.
- **is_final_path_valid(path)** is the entrypoint for fully-constructed paths. It first checks jurisdiction overlap between the first duty's start station and the last duty's end station. If that passes, it delegates to `has_required_breaks()`, passing the precomputed start-jurisdictions to avoid recomputation.
- **has_required_breaks(path, start_jurisdictions)** evaluates the break-specific rules (minimum/maximum/case rules, jurisdiction requirement and cumulative duration) and returns whether the path satisfies them.

Detailed behaviour and elimination logic

get_jurisdiction_groups(station)

- Purpose: compute the set of jurisdiction group IDs that include the given station (derived from the global `jurisdiction_dict`).
- Implementation notes: uses `@lru_cache(maxsize=None)` to store previously computed results, eliminating redundant recomputation for frequently queried stations.

- **Jurisdiction match:** the function ensures that the first station of a duty and the last station of the previous duty belong to at least one common jurisdiction group by checking that their jurisdiction sets are **not disjoint** (i.e., `not set1.isdisjoint(set2)`).

is_final_path_valid(path)

1. **Compute start/end stations:** extract the start station of the first duty (path[1]) and the end station of the last real service (path[-2] if sink is present).
2. **Jurisdiction overlap (cheap, early check):** call `get_jurisdiction_groups` for both stations and test if the two group sets overlap using `isdisjoint`. If they are disjoint, the path is immediately rejected (`return False`). This early rejection avoids running the more expensive break checks for paths that already fail a necessary condition.
3. **Break validation:** after jurisdictions check the function call `has_required_breaks()`. It passes the already-computed start-jurisdiction set to avoid recomputing it inside the break-checker.
4. **Return:** the path is valid only when both jurisdiction overlap and the break rules pass.

Why the order matters The jurisdiction overlap check is intentionally performed first because it is:

- **Cheap** (cached lookups + single set disjoint test).
- **Highly discriminative** — many infeasible paths fail jurisdiction rules, so failing them early avoids running full break-analysis.

has_required_breaks(path) This function enforces the break rules that are definable only on a complete path:

1. **Minimum path length**: if the path is shorter than a threshold (`len(path) < 6`) it is considered invalid. This quickly rejects obviously too-short sequences.
2. **Use cached start_jurisdictions if provided:** if `start_jurisdictions` is passed in (from `is_final_path_valid`), the function reuses it — avoiding recomputation of the same jurisdiction groups.
3. **Extract times and compute gaps:**
   - Build arrays of service start and end times for path[1:-1] (excluding dummies).

- Compute the list of gaps between consecutive services:

$$\text{gaps}[i] = \text{start}_{i+1} - \text{end}_i$$

4. **Select candidate breaks:**
   - Iterate over gaps and consider a gap a candidate break only if the station where the gap occurs is one of the allowed break stations (`BREAK_STATIONS` (KKDA, PVGW) and the gap is at least SHORT_BREAK.
   - For each such candidate break, add the gap duration to the `breaks` list and check whether that break station shares any jurisdiction with the first duty (by calling `get_jurisdiction_groups(station)` and testing `isdisjoint` against `start_jurisdictions`). If at least one candidate break lies in the same jurisdiction, flag `has_break_in_same_jurisdiction`.
5. **Basic existence/jurisdiction checks:**
   - If no candidate breaks were found, reject the path.
   - If no candidate break lies in the same jurisdiction as the first duty, reject the path.
6. **Cumulative break duration:** sum all candidate break durations and reject the path if total_break > CUMULATIVE_BREAKS_DURATION. This enforces the global cap on rest time.
7. **Short / Long break logic (case rules):**
   - Classify candidate breaks into `SHORT` and `LONG` using boolean tests.
   - If there is exactly one break, it must be a LONG break (single long-break rule).
   - If there are multiple breaks and any `SHORT` break exists, there must also be at least one `LONG` break (shorts cannot occur alone).
   - Multiple `LONG` breaks are allowed.
8. If all checks pass, the function returns `True`, otherwise `False`.

Algorithm 2 Final Path Validation Including Break Rules:

`is_final_path_valid()`

Require: *path*, $BREAK\_STATIONS$, $SHORT\_BREAK$, $LONG\_BREAK$, $CUMULATIVE\_BREAKS\_DURATION$

Ensure: TRUE if path satisfies jurisdiction and break constraints

Step 1: Jurisdiction Check
start_station ← *path*[1].*start_station*
end_station ← *path*[−2].*end_station*
start_groups ← get_jurisdiction_groups(*start_station*)
end_groups ← get_jurisdiction_groups(*end_station*)
if *start_groups* is disjoint with *end_groups* **then**
    return FALSE
end if
Step 2: Required Breaks Check
if length(*path*) less than 6 **then**
    return FALSE
end if
Extract *start_times* and *end_times* from path[1:-1]
gaps ← consecutive differences between end and start times
breaks ← [ ]
$has\_break\_in\_jurisdiction \leftarrow FALSE$
for $i = 0$ to length(*gaps*)-1 **do**
    station ← path[i+1].end_station
    gap ← *gaps*[*i*]
    if $station \in BREAK\_STATIONS$ **and** $gap \geq SHORT\_BREAK$ **then**
        Append *gap* to *breaks*
        break_groups ← get_jurisdiction_groups(*station*)
        if *break_groups* intersects *start_groups* **then**
            $has\_break\_in\_jurisdiction \leftarrow TRUE$
        end if
    end if
end for
if *breaks* is empty OR not *has_break_in_jurisdiction* **then**
    return FALSE
end if
total_break ← sum of *breaks*
if $total\_break > CUMULATIVE\_BREAKS\_DURATION$ **then**
    return FALSE
end if
has_short ← any break $\geq SHORT\_BREAK$ and < LONG_BREAK
has_long ← any break $\geq LONG\_BREAK$
if length(*breaks*) == 1 **then**
    return *has_long*
else if *has_short* and not *has_long* **then**
    return FALSE
end if
return TRUE

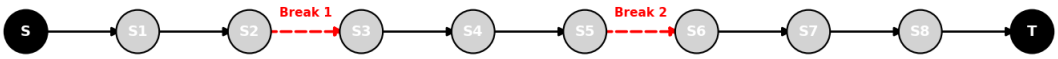

Figure 2.4: Fully constructed duty with two breaks

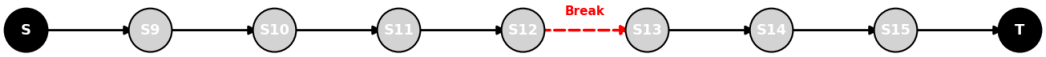

Figure 2.5: Fully constructed duty with only one breaks

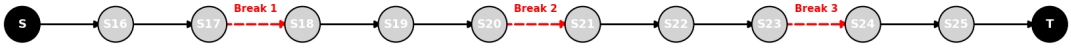

Figure 2.6: Fully constructed duty with three breaks

```
---  Duty Timeline ---
+------------+-------+-------+------+------+--------------+------------+------------+-------------------------------+
| Service_ID | Start |  End  | From |  To  | Service Time | Gap (mins) |   Break    |          Rake Change          |
+------------+-------+-------+------+------+--------------+------------+------------+-------------------------------+
|     18     | 06:15 | 06:21 | VND  | KKDA |      6       |     0      |     -      |               -               |
|    583     | 06:21 | 06:36 | KKDA | MUPR |      15      |     4      |     -      |               -               |
|    586     | 06:40 | 06:55 | MUPR | KKDA |      15      |     0      |     -      |               -               |
|     28     | 06:55 | 08:01 | KKDA | PVGW |      66      |     45     | 45 mins OK | Rake changed at PVGW after 28 |
|     70     | 08:46 | 09:52 | PVGW | KKDA |      66      |     51     | 51 mins OK | Rake changed at KKDA after 70 |
|    119     | 10:43 | 11:50 | KKDA | PVGW |      67      |     0      |     -      |               -               |
|    442     | 11:50 | 12:25 | PVGW | PVGW |      35      |     0      |     -      |               -               |
|    159     | 12:25 | 13:31 | PVGW | KKDA |      66      |     -      |     -      |               -               |
+------------+-------+-------+------+------+--------------+------------+------------+-------------------------------+

---  Constraint Summary ---
+----------------------+------------+-------------------------------+
|      Constraint      | Condition  |            Status             |
+----------------------+------------+-------------------------------+
|  Total Driving Time  | 340 <= 360 |               OK              |
|  Cumulative Breaks   | 96 <= 120  |               OK              |
|    Duty Duration     | 436 <= 445 |               OK              |
| Max Continuous Drive | 168 <= 180 |               OK              |
|  Jurisdiction Match  |   2 == 2   |               OK              |
|     Rake Changes     |     2      | PVGW after 28, KKDA after 70 |
+----------------------+------------+-------------------------------+
```

Figure 2.7: Valid duty with two breaks

Figure 2.7 and 2.8 show the valid duties which are accepted by our approach because they satisfy all the mentioned constraints of DMRC. Below are some invalid duty examples which are correctly rejected by our approach while tracing the path.

--- Duty Timeline ---

Service_ID	Start	End	From	To	Service Time	Gap (mins)	Break	Rake Change
57	08:17	08:23	VND	KKDA	6	0	-	-
620	08:23	08:38	KKDA	MUPR	15	6	-	-
628	08:44	08:59	MUPR	KKDA	15	31	31 mins OK	Rake changed at KKDA after 628
90	09:30	10:37	KKDA	PVGW	67	0	-	-
428	10:37	11:10	PVGW	PVGW	33	52	52 mins OK	Rake changed at PVGW after 428
445	12:02	12:37	PVGW	PVGW	35	33	33 mins OK	Rake changed at PVGW after 445
458	13:10	13:47	PVGW	PVGW	37	0	-	-
186	13:47	14:53	PVGW	KKDA	66	0	-	-
766	14:53	15:08	KKDA	MUPR	15	3	-	-
771	15:11	15:26	MUPR	KKDA	15	-	-	-

--- Constraint Summary ---

Constraint	Condition	Status
Total Driving Time	313 <= 360	OK
Cumulative Breaks	116 <= 120	OK
Duty Duration	429 <= 445	OK
Max Continuous Drive	136 <= 180	OK
Jurisdiction Match	2 == 2	OK
Rake Changes	3	KKDA after 628, PVGW after 428, PVGW after 445

Figure 2.8: Valid duty with three breaks

--- Duty Timeline ---

Service_ID	Start	End	From	To	Service Time	Gap (mins)	Break	Rake Change
261	17:32	18:37	PVGW	KKDA	65	0	-	-
841	18:37	18:53	KKDA	MUPR	16	2	-	-
848	18:55	19:11	MUPR	KKDA	16	0	-	-
300	19:11	20:17	KKDA	PVGW	66	57	57 mins OK	Rake changed at PVGW after 300
344	21:14	22:20	PVGW	KKDA	66	0	-	-
920	22:20	22:36	KKDA	MUPR	16	0	-	-
565	22:36	22:53	MUPR	MUPR	17	0	-	-
926	22:53	23:08	MUPR	KKDA	15	0	-	-
370	23:08	24:14	KKDA	PVGW	66	0	-	-
563	24:14	24:29	PVGW	MKPD	15	-	-	-

--- Constraint Summary ---

Constraint	Condition	Status
Total Driving Time	360 <= 360	OK
Cumulative Breaks	57 <= 120	OK
Duty Duration	417 <= 445	OK
Max Continuous Drive	195 <= 180	Error
Jurisdiction Match	1 == 1	OK
Rake Changes	1	PVGW after 300

Figure 2.9: Invalid duty- violating maximum continuous drive constraint

--- Duty Timeline ---

Service_ID	Start	End	From	To	Service Time	Gap (mins)	Break	Rake Change
798	16:34	16:49	KKDA	MUPR	15	3	-	-
803	16:52	17:07	MUPR	KKDA	15	75	75 mins OK	Rake changed at KKDA after 803
835	18:22	18:37	KKDA	MUPR	15	3	-	-
842	18:40	18:55	MUPR	KKDA	15	0	-	-
294	18:55	20:01	KKDA	PVGW	66	0	-	-
531	20:01	20:37	PVGW	PVGW	36	42	42 mins OK	Rake changed at PVGW after 531
543	21:19	21:54	PVGW	PVGW	35	0	-	-
356	21:54	23:00	PVGW	KKDA	66	0	-	-
928	23:00	23:16	KKDA	MUPR	16	0	-	-
931	23:16	23:22	MUPR	SVVR	6	1	-	Invalid rake change at SVVR
372	23:23	24:29	KKDA	PVGW	66	-	-	-

--- Constraint Summary ---

Constraint	Condition	Status
Total Driving Time	358 <= 360	OK
Cumulative Breaks	117 <= 120	OK
Duty Duration	475 <= 445	Error
Max Continuous Drive	190 <= 180	Error
Jurisdiction Match	2 == 1	Error
Rake Changes	2	KKDA after 803, PVGW after 531

Figure 2.10: Invalid duty- violating multiple constraints

2.4 Performance and pruning tricks used

The implementation contains several deliberate choices that improve runtime and pruning effectiveness:

- **Two-stage validation strategy:** cheap incremental pruning during path construction (`is_path_acceptable`) followed by final checks (`is_final_path_valid` and `has_required_breaks`) avoids expensive computations on obviously-feasible partial paths that later fail more global rules.
- **Cached jurisdiction lookup:** `get_jurisdiction_groups` is memoized with `lru_cache`. For stations that appear frequently, this reduces repeated work from O(jurisdiction_dict) to an O(1) cache hit.
- **Immutable cached results:** returning a `frozenset` ensures the cached object is immutable and safe for reuse in set operations and dictionary keys.
- **Early short-circuits:**
  1. `is_final_path_valid` first runs the cheap jurisdiction overlap test and returns immediately on failure — avoiding the more expensive break analysis.
  2. `has_required_breaks` returns early for short paths (length check), and rejects as soon as a required condition fails (e.g., no breaks or no jurisdictional match), which reduces wasted computation.

- **Work limited to path length:** all computations (gaps, break selection, classification) are linear in the number of services in the path (O(k)). There is no nested heavy computation per gap.
- **Passing precomputed data:** `is_final_path_valid` passes the start-jurisdiction set to `has_required_breaks` to avoid recomputing it.

How invalid paths are eliminated (end-to-end)

1. Many infeasible paths are already pruned earlier by `is_path_acceptable` during graph traversal (driving time, duty time, continuous-driving). Only those that pass the incremental tests reach the final validation stage.
2. `is_final_path_valid` first eliminates paths with no jurisdiction overlap — a cheap, high-value test.
3. Only if that passes, `has_required_breaks` performs the break-specific checks and rejects paths that do not meet the allowed break cases, cumulative duration or jurisdictional break requirement.
4. Because of early-exit logic and caching, most invalid paths are detected and discarded with minimal extra work.

In summary the implemented functions form an efficient final-path validator that uses cached jurisdiction overlap checks and linear-time break validation to quickly eliminate infeasible paths, maintaining computation cost proportional to the path length.

2.5 Stack-Based Path Exploration Using DFS

The function `stack_based_all_paths()` enumerates all feasible paths through the service network graph G using a stack-based depth-first search (DFS) approach. This method systematically explores all possible service sequences while applying operational constraints at each step to prune infeasible paths early.

2.5.1 Stack Structure and Purpose

The `stack` is the core data structure guiding the DFS exploration. Each element is a tuple:

$$(\text{current\_service}, \text{path\_so\_far}, \text{total\_driving\_time})$$

where **current_service** is the node being visited, **path_so_far** records the path from the start node S, and **total_driving_time** accumulates driving time (including short allowable gaps).

Initially, the stack holds only the start node S with zero driving time. At each step, the top element is popped and expanded; for every neighbor, a new candidate path is pushed onto the stack if it satisfies `is_path_acceptable()`.

2.5.2 Path Exploration Process:

The exploration follows these steps:

1. **Pop the current node:** The top element of the stack provides the current service node, the path constructed so far, and the total accumulated driving time.
2. **Visit neighbors:** For each successor node in the directed graph G, the function considers moving to that service. The corresponding service object is retrieved from the graph.
3. **Update driving time:**
   - Compute the gap between the end time of the last service in the path and the start time of the neighbor.
   - Add the neighbor's service duration to the total driving time.
   - If the gap is shorter than the minimum `SHORT_BREAK`, it is included in the driving time counter.
4. **Form new path:** The neighbor is appended to the current path, forming a candidate extension.
5. **Constraint check:** The candidate path is sent to `is_path_acceptable()`:
   - Only constraints that can be evaluated incrementally on a *partial path* are checked.
   - These include driving time limits, duty duration limits, and continuous driving rules.
   - If the partial path fails any check, it is immediately discarded (early pruning).
6. **Terminal node check:**
   - If the neighbor is the sink node T, the path has reached completion.
   - The completed path is then sent to `is_final_path_valid()` to verify constraints that can only be checked on fully constructed paths, such as break compliance and jurisdiction rules.

- Valid paths are then recorded in the CSV file, excluding the dummy start and end nodes.

7. **Push for further exploration:**

   - If the neighbor is not the sink T and the path is acceptable so far, a new tuple containing the neighbor, the updated path, and the updated driving time is pushed onto the stack. It allows DFS to continue exploration.

2.5.3 Stack-based DFS Algorithm for Path Enumeration

Algorithm 3 Crew Path Enumeration using Stack-based DFS:

`stack_based_all_paths()`

Require: Directed graph G, start node S, end node T
Require: Driving time limit $D_{\max}$, duty time limit $L_{\max}$
Require: Short break B_{short}, Long break B_{long}
Ensure: All valid crew paths are saved to a CSV file

Initialize stack $S_t \leftarrow [(S, [S], 0)]$
while $S_t \neq \emptyset$ **do**
    $(current, path, driveTime) \leftarrow \text{Pop}(S_t)$
    for each *neighbor* in successors($current$) **do**
        $serviceTime \leftarrow$ service time of *neighbor*
        $last \leftarrow$ last service in *path*
        $gap \leftarrow \max(0, neighbor.start\_time - last.end\_time)$
        $newDriveTime \leftarrow driveTime + serviceTime$
        if $gap < B_{\text{short}}$ **then**
            $newDriveTime \leftarrow newDriveTime + gap$
        end if
        $newPath \leftarrow path + [neighbor]$
        if is_path_acceptable($newPath, T, newDriveTime, D_{\max}, L_{\max}$) **then**
            if $neighbor = T$ **and** is_final_path_valid($newPath$) **then**
                Save $newPath$ to CSV
            else
                Push $(neighbor, newPath, newDriveTime)$ onto S_t
            end if
        end if
    end for
end while

2.5.4 Key Features of the Stack-Based DFS Approach:

- **Incremental path construction:** The path is built step-by-step, and constraints are checked along the way, ensuring early pruning of infeasible sequences.
- **Exploring different neighbors:** At each service node, all successors are considered, enabling the algorithm to enumerate all possible paths through the network.
- **Separation of partial vs. complete path validation:** `is_path_acceptable()` handles constraints that can be checked incrementally, while `is_final_path_valid()` handles constraints that require a fully formed path.
- **Efficient path storage:** The stack keeps only the necessary information to continue traversal; paths are extended only if feasible.
- **Output:** All valid paths are eventually written to the CSV file in a human-readable format, excluding the dummy source and sink nodes.

In summary the stack-based DFS effectively explores the service network graph by maintaining a stack of candidate paths. It incrementally builds paths from the start node S to the end node T, pruning invalid partial paths early using `is_path_acceptable()`. Completed paths reaching T are validated using `is_final_path_valid()`, ensuring all operational constraints are met. The final valid paths are written to an output CSV file for further analysis or downstream processing.

2.6 Mathematical Formulation

In this section, we present the mathematical formulation for the duty selection from the generated feasible duties. Set of services $S = \{1, 2, \ldots, n\}$ and a set of paths (duties) $P = \{1, 2, \ldots, m\}$. Each duty $p \in P$ covers a subset of services $A_p \subseteq S$. The objective is to select the minimum number of duties such that every service is covered exactly once.

Decision Variables

$$x_p = \begin{cases} 1, & \text{if duty (path) } p \text{ is selected,} \\ 0, & \text{otherwise;} \end{cases} \quad \forall p \in P$$

$$y_s = \begin{cases} 1, & \text{if service } s \text{ is covered,} \\ 0, & \text{otherwise;} \end{cases} \quad \forall s \in S$$

Objective Function

The objective is to minimize the total number of selected duties:

$$\min Z = \sum_{p \in P} x_p \tag{2.1}$$

Constraints

(1) Service Coverage Constraint Each service must be covered by exactly one selected duty:

$$\sum_{p \in P : s \in A_p} x_p = y_s, \quad \forall s \in S \tag{2.2}$$

(2) Service Assignment Constraint Each service must be covered:

$$y_s = 1, \quad \forall s \in S \tag{2.3}$$

2.7 Code Customizability and Readability

The crew scheduling solution has been implemented using an Object-Oriented Programming (OOP) approach, ensuring modularity, reusability, and maintainability. All operational rules and configuration parameters are defined at the start of the code, avoiding hard-coded numbers and making it easy to modify rules without changing the core logic.

Key Features

- **Service Representation:** Each service is represented as a `Service` object, encapsulating all attributes such as `service_id`, `rake_num`, `start_station`, `end_station`, `start_time`, `end_time`, and additional operational details.

  Example: To fetch the start station of the second service in a path, we can directly access:

  ```
  path[1].start_station
  ```

  This demonstrates how OOP makes it easy to query any attribute of a service in a readable way.

- **Readability and Maintainability:** Functions like `load_services()` separate data loading from processing logic. Dummy start (`S`) and end (`T`) nodes standardize pathfinding.

- **Customizable Rules:** All business rules (break durations, duty time limits, allowed stations, rake gaps, etc.) can be easily updated at the top of the code, minimizing errors and improving maintainability.

Path Tracing Using the Graph

The scheduling paths are explored using a graph-based representation of services. Each node represents a service, and edges represent feasible transitions between services. Fetching neighboring services is simple:

```
for neighbor_id in G.successors(current_service.service_id):
    neighbor = G.nodes[neighbor_id]["data"]
```

Here, `G.successors(current_service.service_id)` fetches all valid next services from the current service, and `G.nodes[neighbor_id]["data"]` gives access to the full `Service` object for each neighbor. This approach allows easy traversal, path building, and application of all operational constraints during scheduling.
The complete implementation, including the service dataset and graph-based crew path generation code, is available in the GitHub repository [12].

Chapter 3

Results and Conclusion

3.1 Characteristics of the Crew Scheduling Graph

Before performing crew path enumeration and duty generation, a directed graph is constructed to represent all feasible connections between train trips. Each node in the graph corresponds to an individual trip, while each edge indicates a valid connection that satisfies operational and temporal feasibility constraints, such as minimum rake gap, maximum connection gap, and rake continuity.

Parameter	**Value**
Number of nodes	944
Number of edges	23555
Number of rakes (Mainloop)	51
Number of stations (Mainloop)	11
Time required to build the graph	0.5 sec

Table 3.1: Characteristics of the Crew Scheduling Graph

3.2 Time Requirement to Trace Paths

This comparison helps in evaluating the efficiency of the network based scheduling algorithm and estimating the time required for exploring large datasets. As the number of paths grows, the time to trace all possible paths increases, which is critical for planning computations on larger networks. For instance, tracing roughly 50,000 paths may take around 15 seconds, whereas tracing 12 lakh paths may take close to 50 minutes. These rough numbers help set expectations and guide optimization or parallelization efforts.

No. of Paths	Time (mins)
50,000	0.25
100,000	1
200,000	2.5
400,000	10
600,000	22
1,048,342	50

Table 3.2: Time required to trace numbers of paths for 944 services.

3.3 Comparison with Existing Approach

In the previous approach, path tracing was implemented using recursion and backtracking without any graph-based representation. Each service was explored sequentially, and feasible paths were built recursively. While functional, this method was computationally expensive and difficult to optimize for large datasets.
In the latest approach, a graph-based representation is used where each node corresponds to a service, and edges represent feasible transitions between services. Path tracing is implemented using a stack-based Depth-First Search (DFS), which efficiently explores all feasible paths while adhering to operational constraints.

For a dataset of 944 services:

- The old approach required approximately 5 hours to trace all feasible paths.
- The latest graph-based approach traces 13 lakh paths in only 50 minutes.

This corresponds to a time reduction of:

$$\frac{50}{300} = 0.1667 \approx 16.67\%$$

of the time required by the old approach. Moreover, the set of paths traced by the new approach is a superset of those traced by the existing method, ensuring completeness while significantly improving performance. Further optimizations are planned to reduce computation time even more.

3.4 Optimal Solution

After generating all feasible crew duties using the stack-based Depth-First Search (DFS) approach on the constructed graph, an optimization model based on the *Set Covering Method* was applied to select the minimal set of duties that covers all train trips. The objective of this model is to minimize the total number of duties while ensuring that every trip is covered exactly once by a qualified crew duty.

Parameter	Value
Total feasible duties	1,048,342
Optimal number of duties	110
Time required to solve (mins)	155

Table 3.3: Summary of Optimal Solution for Crew Scheduling

3.5 Computing Resources

All experiments related to duty generation and optimal solution finding for the Delhi Metro crew scheduling problem were executed on the **Passpoli Server**. The specifications of the computing environment are summarized in Table 3.4.

Server Name	Passpoli Server
Processor	AMD Ryzen Threadripper PRO 5975WX
Number of Cores	32
RAM	128 GB

Table 3.4: Specifications of the computing environment used for experiments

The computation times and results reported in the previous sections were generated using this server configuration. All algorithms and optimization models were executed under identical hardware and software conditions to ensure consistency of performance evaluation.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

In this work, we successfully modeled all 944 services of the Delhi Metro's Pink Line's mainloop using a network-based Directed Acyclic Graph (DAG), enabling efficient representation of service connectivity and operational constraints. By leveraging a stack-based DFS algorithm, we were able to enumerate all feasible crew duties from the graph effectively. Compared to the existing approach using recursion and backtracking—which takes up to 5 hours to generate feasible duties—our network and stack DFS-based method reduces the computation time dramatically to approximately 50 minutes, while ensuring that all DMRC operational rules and constraints are satisfied.

Furthermore, the project implements Object-Oriented Programming (OOP) principles, making the code highly readable, modular, and customizable for future modifications or scaling. Overall, the approach demonstrates a significant improvement in computational efficiency, maintainability, and adaptability for practical crew scheduling applications on the Delhi Metro's Pink Line.

4.2 Future Work

There are several directions in which this work can be extended to improve efficiency, scalability, and usability:

- **Further Optimization and Speedup:** The current stack-based DFS approach can be further accelerated by exploring parallelization techniques to leverage multi-core processing [5]. Graph preprocessing or simplification techniques can also help in minimizing the number of edges without compromising the feasibility of duties.
- **Scalability to Other Lines:** The approach can be extended to include all lines of the Delhi Metro, taking into account the unique operational constraints

of each line. Developing a centralized scheduler capable of handling multi-line duty allocation simultaneously can improve efficiency and coordination, similar to approaches used in multi-line metro crew planning and replanning [4].

- **Integration with Real-Time Operations:** Incorporating real-time operational data, such as train delays and maintenance schedules, can enable dynamic updates to feasible duties. This allows rescheduling in response to unexpected events, such as crew unavailability or service disruptions, enhancing operational robustness, as explored in real-time public transit rescheduling studies [3].

- User-Friendly and Maintainable Code: Future work can focus on developing a graphical user interface (GUI) or dashboard for visualising schedules and allowing manual adjustments. Enhancing the modularity of the OOP structure can make the code more maintainable and easier to adapt for other metro networks or future extensions, as highlighted in studies on constraint programming-based metro crew scheduling [2].

Appendix

Meetings and Consultations

To deeply understand the existing crew scheduling method, specifically the **Recursion Backtracking** approach developed by IIT Bombay students under the guidance of **Prof. Narayan Rangaraj** and **Prof. Madhu Belur**, I have conducted multiple meetings with them and their students. These consultations were crucial for grasping the intricacies of their existing code and the operational constraints of the Delhi Metro Rail Corporation (DMRC).

- **June 24, 2025** : Meeting with Deepankar and Rishav to understand their approach of **column generation** and their code.
- **June 26, 2025** : Meeting with Rishuv Gorka to understand the existing approach and code they used (**Recursion and Backtracking + Set Covering**) currently utilized by the Delhi Metro.
- **July 10, 2025** : Meeting with Aadesh Jain to understand the different **constraints** of Delhi Metro crew scheduling.
- **August 10, 2025** : Meeting with Sarvar E Abbas to get some clarity on the operational **constraints** of the Delhi Metro.
- **August 27, 2025** : Meeting with Rishuv to understand the old coded constraints and to document all constraints of Delhi Metro in the report.
- **September 18, 2025** : Meeting with **Prof. Madhu Belur** in the EE department of IIT Bombay for explaining the new network based approach and guidance.
- **September 23, 2025** : Given presentation of the **network-based approach** developed in this project to **Prof. Narayan Rangaraj**, **Prof. Madhu Belur**, **Sarvar E Abbas**, and **Aadesh Jain** in the IEOR department (IE: 105) from 3:30 PM to 4:30 PM.

Nomenclature

For easy understanding, some of the abbreviations are mentioned here which are going to be used in the subsequent sections of the report.

- **HOER**: Hours of Employment and Period of Rest
- **DMRC**: Delhi Metro Rail Corporation
- **CC**: Crew Control

- **MKPR**: Majlis Park
- **MUPR**: Maujpur-Babarpur
- **SVVR**: Shiv Vihar
- **SAKP**: Shakurpur
- **PVGW**: Punjabi Bagh West
- **VND**: East Vinod Nagar
- **IPE**: I.P. Extension
- **KKDA**: Karkarduma
- **MVPO**: Mayur Vihar Pocket-1
- **NZM**: Nizamuddin

Configuration Input Variables

- **MIN_RAKE_GAP_MINUTES:** Minimum required gap between different rakes.
- **ALLOWED_RAKE_CHANGE_STATIONS:** Stations where rake changes are allowed.
- **MAX_CONNECTION_GAP_MINUTES:** Maximum allowed gap between consecutive services.
- **BREAK_STATIONS:** Stations where breaks are permitted.
- **CUMULATIVE_BREAKS_DURATION:** Maximum cumulative duration of breaks in a duty.
- **SHORT_BREAK:** Duration of a short break in minutes.
- **LONG_BREAK:** Duration of a long break in minutes.
- **DUTY_TIME_LIMIT:** Maximum total duty time for a standard shift.
- **MORN_EVEN_DUTY_TIME_LIMIT:** Maximum duty time for morning and evening shifts.
- **MORNING_SHIFT_CUTOFF:** Cutoff time (in minutes) for morning shift duties.
- **EVENING_SHIFT_CUTOFF:** Cutoff time (in minutes) for evening shift duties.

- **CONTINUOUS_DRIVE_LIMIT:** Maximum continuous driving time without a break longer than 30 minutes.
- **DRIVING_TIME_LIMIT:** Maximum driving time including short breaks.
- **Jurisdiction Buckets:** DMRC Pink Line operates with two jurisdiction groups. The model ensures that the start station of the first service and the end station of the last service in a duty belong to the same jurisdiction for logistical convenience.

Bibliography

[1] Introduction to stack data structure, operations, and applications in algorithms. https://www.geeksforgeeks.org/dsa/introduction-to-stack-data-structure-and-algorithm-tutorials/.

[2] Anonymous. A constraint programming-based approach to the crew scheduling problem of the taipei mass rapid transit system, 2014. https://ideas.repec.org/a/spr/annopr/v223y2014i1p173-19310.1007-s10479-014-1619-1.html.

[3] Anonymous. Real-time integrated re-scheduling for public transit, 2020. https://www.researchgate.net/publication/333844771_Real-Time_Integrated_Re-scheduling_for_Public_Transit.

[4] Anonymous. Unified crew planning and replanning optimization in multi-line metro systems, 2023. https://arxiv.org/abs/2509.14251.

[5] Z. A. Aziz and A. Abdulqader. Python parallel processing and multiprocessing: A review, 2021. https://www.researchgate.net/publication/354221786_Python_Parallel_Processing_and_Multiprocessing.

[6] R. Borndörfer, M. Grötschel, and A. Martin. A column generation approach to airline crew scheduling, 2007. https://link.springer.com/chapter/10.1007/3-540-32539-5_54.

[7] L. Chen, S. Gupta, and H. V. Jagadish. Stack-based algorithms for pattern matching on dags, 2005. https://www.vldb.org/archives/website/2005/program/paper/wed/p493-chen.pdf.

[8] M. Desrochers and F. Soumis. A column generation approach to the urban transit crew scheduling problem, 1989. https://pubsonline.informs.org/doi/10.1287/trsc.23.1.1.

[9] Jens Heil, Kai Hoffmann, and Uwe Büscher. Railway crew scheduling: Models, methods and applications. *European Journal of Operational Research*, 283(2):405–425, 2020. https://doi.org/10.1016/j.ejor.2019.06.016.

[10] H. Jin, Y. Wang, and Y. Liu. Column generation-based optimum crew scheduling incorporating network representation for urban rail transit systems, 2022. https://www.sciencedirect.com/science/article/pii/S036083522200225X.

[11] Richard M. Lusby, Anders Dohn, Troels Martin Range, and Jesper Larsen. A column generation-based heuristic for rostering with work patterns, 2012. Journal of the Operational Research Society, 63(2): 261–277, https://doi.org/10.1057/jors.2011.18.

[12] GitHub Repository. Crew scheduling using network approach: Stack-based dfs approach for feasible duty generation in delhi metro, 2025. https://github.com/rupesh-source/Crew_Scheduling_Network.

[13] Delhi Metro Times. Delhi metro pink line, 2025. https://www.delhimetrotimes.in/delhi/maps/delhi-metro-pink-line.html.