Adding Ejection Chain to Nurse Rostering Simulated Annealing Solver

by

Roy Katz

Student Number: 5315492
Project Duration: November, 2024 - June, 2025
Thesis Committee: Dr. N. Yorke-Smith
Dr. A.L.D. Latour
Dr. ir. J.T. van Essen
Dr. L. Berghman

Acknowledgments

This thesis examines the impact of adding ejection chains to a commercial nurse rostering solver. It was completed as part of my Master's degree in Computer Science at TU Delft, within the Algorithmic research group, and the Artificial Intelligence Technology track.

I would like to extend my sincere gratitude to everyone who contributed to the completion of this thesis. First, I would like to thank my colleagues at ORTEC, especially my manager, Lotte Berghman, for her invaluable guidance and support whenever needed. I am also grateful to my mentor, Eva van Rooijen, for her dedication and effort throughout this process.

Additionally, I wish to thank my university supervisors: my thesis advisor, Neil Yorke-Smith, for his expertise and help in providing direction, and my daily supervisor, Anna Latour, for her insightful feedback on the writing, presentation and the overall project.

Roy Katz
Delft, June 2025

Summary

Effectively solving the Nurse Rostering Problem enhances nurse moral and leads to improved patient care [16]. While the use of ejection chains has shown promise in previous studies, studying their impact on real-life instances from two Dutch hospitals further deepens our understanding of their practical utility.

This thesis begins by discussing the Nurse Rostering Problem (NRP) as presented in literature, highlighting the lack of research conducted in real-life settings. We then describe the context of Dutch hospitals by defining the relevant hard and soft constraints. Afterwards, a review of previous NRP solutions is provided, covering various solvers and techniques, including ejection chains.

Ejection chains are an approach that combines multiple local search moves into a larger one, where each move forces (ejects) the next move. These moves are executed consecutively, forming a chain. Two recent ejection chain approaches were identified for further investigation, and a novel approach is also proposed.

Afterwards, the concept of the three ejection chains is explained. The first ejection chain, InfeasibleEC, is inspired by Kingston in [12], and explores regions of the search space that contain infeasible rosters. It starts with a swap that introduces a single hard constraint violation, but results in lower roster penalty. Then the ejection chain proceeds with a series of repairs, where each repair fixes the previous violation, but may introduce a single new violation. Four different approaches to explore the search space were implemented.

The second ejection chain described is EmulateEC by Curtois et al. [8]. This chain emulates human schedule planners by performing a series of vertical swaps. Each swap improves the penalty for one nurse but simultaneously increases the penalty for a different nurse. Thus, the next swap aims to improve the nurse whose penalty was worsened in the previous step. This series of swaps is repeated until a better roster is found.

The third ejection chain, RuinRecreateEC, repeatedly performs ruin and recreate operations in sequence. The idea is that each ruin and recreate operation makes a relatively small change to the roster, and by repeating the process multiple times, the final roster undergoes substantial diversification while still retaining much of the structure of the original roster.

All three ejection chains underwent individual parameter tuning using a sequential greedy tuning strategy to identify suitable parameter configurations based on two instances from the same hospital.

Once tuned, the chains were further tested by rerunning the chains on the initial instances as well as on additional instances from a second hospital to assess their robustness. We analysed the breakdown of the penalty based on the different soft constraints, and investigated the impact of the chains on finding better rosters throughout the running time. We could not find improvements on specific soft constraints penalties, nor on finding better rosters faster.

A significance test was conducted to determine whether the chains significantly reduce the penalty of the final rosters. There was insufficient evidence to conclude a statistically significant improvement.

Contents

List of Symbols

The table below is based on the table created by Faasse in [9]. Due to the same problem description and to ensure this thesis is complete it is added to this thesis as well.

Notation	
d	A day
e	An employee
s	A shift
Sets and Indices	
E	Set of employees ($e \in E$)
D	Set of days ($d \in D$)
S	Set of shift types ($s \in S$)
S^+	Set of shift types, with day off assignment o included ($S^+ = S \cup \{o\}$)
Parameters	
$a_{d,s}$	Coverage requirement on day d for shift type s in number of employees
$\varepsilon_{d,s}$	Preferred coverage on day d for shift type s in number of employees
b_s	Priority shift type indicator: 1 if shift type s is a priority shift type, 0 otherwise
c_e^{con}	Contractual workload for employee e in hours
$c_e^{\max}$	Maximum workload for employee e in hours: $c_e^{\max} = c_e^{\text{con}} + 10$
f_s	Duration of shift type s in hours
$g_{e,d}^{\text{on}}$	Day-on request indicator: 1 if employee e has requested to be assigned to a shift on day d, 0 otherwise
$g_{e,d}^{\text{off}}$	Day-off request indicator: 1 if employee e has requested a day off on day d, 0 otherwise
$h_{e,d,s}^{\text{on}}$	Shift-on request indicator: 1 if employee e has requested to be assigned to shift type s on day d, 0 otherwise
$h_{e,d,s}^{\text{off}}$	Shift-off request indicator: 1 if employee e has requested not to be assigned to shift type s on day d, 0 otherwise
j_e	Weight for the requests and preferred shift and day-off sequence lengths of employee e
$k_{e,s}$	Valid shift indicator: 1 if employee e has the required skill to work shift type s, 0 otherwise
$m_{e,d,s}$	Fixed assignment indicator: 1 if employee e has a fixed assignment of shift type s on day d, 0 otherwise
$n_{s,t}$	11 hrs rest indicator: 1 if there is at least 11 hrs rest between shift type s on day d, and shift type t on day $d+1$
$p_{s,t}$	8 hrs rest indicator: 1 if there is at least 8 hrs rest between shift type s on day d, and shift type t on day $d+1$
q_d	Sunday indicator: 1 if day d is a Sunday, 0 otherwise
r_e	Maximum number of Sundays that employee e can work in this period
β_s	Night shift indicator: 1 if shift type s is a night shift, 0 otherwise
γ_e	Maximum number of night shifts that employee e can work in this period
Decision Variables	
$x_{e,d,s}$	Working shift indicator: 1 if employee e works shift type s starting on day d, 0 otherwise
Auxiliary Variables	
$y_{d,s}$	Coverage shortage of shift type $s \in S$ on day $d \in D$: $y_{d,s} = \max\left\{a_{d,s} - \sum_{e \in E} x_{e,d,s}, 0\right\}$
$\varphi_{d,s}$	Preferred coverage shortage of shift type $s \in S$ on day $d \in D$: $\varphi_{d,s} = \max\left\{\varepsilon_{d,s} - \sum_{e \in E} x_{e,d,s}, 0\right\}$
y_d	Total coverage shortage on day $d \in D$: $y_d = \sum_{s \in S} y_{d,s}$
z_d	Coverage shortage on day d minus one: $z_d = \max\left\{\left(\sum_{s \in S} y_{d,s}\right) - 1, 0\right\}$
u_e	Worked overtime hours for employee e: $u_e = \max\left\{\left(\sum_{d \in D} \sum_{s \in S} x_{e,d,s} f_s\right) - c_e^{\text{con}}, 0\right\}$
$v_{e,d}$	Daily rest exception indicator: 1 if the daily rest exception is used on day d for employee e, 0 otherwise
$w_{e,d}$	Worked Sunday indicator: 1 if employee e works on day d which is a Sunday, 0 otherwise
ζ_e^{on}	Preferred length of consecutive shifts of employee e
ζ_e^{off}	Preferred length of consecutive days off of employee e
η_e^{on}	List of lengths of all consecutive shifts that employee e works
η_e^{off}	List of lengths of all consecutive days off that employee e has

1 Introduction

1.1. Why Nurse Scheduling

The nurse rostering problem (NRP) is a combinatorial optimization problem with noteworthy societal application, particularly within healthcare management. As the survey by Ngoo et al. [16] highlights, effectively solving this problem enhances hospital efficiency and boosts nurse morale, leading to improved patient care and overall experience. Although it is possible to create rosters manually, generating high quality rosters is both difficult and time consuming. Furthermore, Burke et al. [4] point out that this time-consuming act is usually performed by the chief nurse, whose main job is patient care, highlighting the need for automation. Thus, a lot of research has been conducted on leveraging computers to automate nurse rostering.

The importance of generating high-quality rosters becomes particularly apparent when considering the healthcare workers situation in the Netherlands. According to data collected by Central Bureau voor de Statistiek (CBS, Central Bureau for statistics) regarding healthcare workers in the Netherlands in 2021, healthcare workers report higher job satisfaction compared to the general workforce, but they are less satisfied with their working conditions [7]. Notably, satisfaction with possibility to determine your own working hours is 7.7% lower for healthcare workers, compared to employees in all sectors. Furthermore, 49.5% of care employees consider their workload as much too high, while only 57.6% believe they influence their schedule. Additionally, a projected shortage of 195 000 healthcare employees by 2033 [11] further emphasizes the urgent need for better rostering solutions that take nurse preferences into account.

1.2. Problem Statement

NRP is the problem of assigning shifts to nurses (or employees) over a scheduling period. Each *shift* is defined by a start and end time, specifying when the employee should work, and by the required *skills*, that determine which employees can be assigned to this specific shift. The resulting assignment is referred to as a *roster* or a schedule. To be *feasible*, the roster needs to follow a set of rules. Additionally, the objective is to find a roster that minimizes a penalty. The penalty is calculated based on having sufficient employees assigned for different shifts, the spread of overtime for different employees and the preference requests made by the employees. A feasible roster (i.e. one that satisfies all the rules) is always better than an infeasible roster. Among feasible rosters, the one with lower penalty is considered better. Due to the large search space (typically 50-150 employees, 3-10 shift types, 5-20 skill types and a scheduling period of a month) for this problem, ORTEC decided to look for a good roster, rather than guaranteeing an optimal roster. This is achieved by employing a heuristic solver and running it for a predetermined duration, striving to find as good roster as possible during that time. Chapter 2 defines the exact NRP researched in this thesis.

The current solver used to schedule nurses in Dutch hospitals is an evolutionary algorithm solver, but this solver can be improved. Recently, a hospital approached ORTEC with a roster produced by the solver, and showed that with 5 small changes, they found a roster that better follows the preferences of

the nurses. Demonstrating the opportunity to find better rosters by making multiple small changes consecutively with some intelligence behind it, as well as the need to explore the search space better, by escaping local minima.

In this thesis, solving this issue is explored by examining the impact of adding an ejection chain. An *ejection chain* is a series of local moves that are combined into a larger move. A chain starts with a move that forces (ejects) the algorithm into a series (a chain) of moves, each forcing the next move. The aim of the ejection chain is to both find a better roster than the one it started with, but also to escape from the local search space region to another one that can be explored further.

1.3. ORTEC

This thesis is written in collaboration and under the supervision of ORTEC B.V. The company has many different services, one of which is ORTEC Workforce Scheduler (OWS). This software is used to schedule nurses in the Netherlands, as well as schedule employees of many different companies around the world. ORTEC is working on creating a new solver for this service. This thesis will build on top of their new solver and check if the addition of an ejection chain can result in better schedules.

1.4. Research Questions

The research question that this thesis will answer is:

How do ejection chains impact the quality of rosters produced by a commercial nurse scheduling solver?

To address this question, multiple smaller research questions must be answered. First, it is essential to define how ejection chains can be applied to NRP. In literature, two relevant ejection chains have been identified. These two chains will be adapted to the NRP defined by ORTEC. The papers that describe them are discussed further in Section 3.3. Furthermore, a novel ejection chain was developed for this research. Detailed descriptions of the three ejection chains are provided in Chapter 4. This leads to the first sub question:

RQ1: How do different types of ejection chains differ in their impact on solution quality for nurse rostering?

To answer this question we analysed the effect of the ejection chains on the solver in terms of the penalty of the best roster found by the solver with and without ejection chain. Furthermore, we analysed the breakdown of the penalty, looking into the impact of the different chains on the different components of the penalty.

Although it is important to assess the impact of the ejection chains on the quality of the final roster, it is also essential to assess whether these effects are robust and generalize across different nurse rostering scenarios. This leads to the second research question:

RQ2: How robust is the impact of ejection chains across different nurse rostering instances?

To answer this question, the ejection chain was first developed and tuned using data from two departments within a single hospital. Afterwards, the ejection chain was evaluated on eight additional instances that originated from three different departments in a different hospital. By testing across multiple departments and hospitals, the study aims to determine if an improvement of the solver by an ejection chain is consistent across different real-life settings.

The primary metric used to evaluate NRP solvers is the quality of the final roster produced. On the other hand, it is also crucial to determine the speed in which rosters with lower penalties are found. Accordingly, this search will examine the relationship between the solver's running time and the penalty of the best roster identified during the search process. Tracking how solution quality evolves over time

provides insight into the efficiency and convergence properties of different ejection chains. This brings us to the third research question:

RQ3: How do ejection chains influence the progression of the solution quality as the solver runs?

By analysing the progression of the solution quality over time, this study aims to assess both the rate at which solutions of varying penalties are discovered and the solver's ability to escape from suboptimal regions of the search space.

1.5. Outline

This thesis proceeds with review of NRP found in literature as well as in ORTEC in Chapter 2. Followed by a description of methods used to solve NRP in Chapter 3. The implementation details of the three different ejection chains are in Chapter 4. A detailed explanation of the parameter tuning is in Chapter 5. Subsequently, the experiments and results will be presented in Chapter 6. The conclusion of the thesis is in Chapter 7.

2

Nurse Rostering Problem

This chapter explains the NRP presented in literature, as well as the version solved commercially by ORTEC. First, the definitions and notations used throughout the thesis are provided in Section 2.1. Subsequently, Section 2.2 describes the NRP commonly found in research. Afterwards, we focus on the NRP version employed by ORTEC. The criteria for feasible roster are detailed in Section 2.3. Followed by a description of the roster quality measures in Section 2.4.

2.1. Definitions and Notations

Before describing the problem, it is important to establish the definitions and notations used throughout this thesis. A key distinction is made between hard and soft constraints. *Hard constraints* are essential requirements that a solution (roster) must satisfy. A roster that violated one or more hard constraint is considered infeasible. In this study, the hard constraints include the maximum workload of each employee (defined by their respective contract), the skill requirement for each shift, the labour regulations from the Dutch Working Hours Act [1] and the collective labour agreement [14].

On the other hand, *soft constraints* determine the quality of feasible rosters. Violating a soft constraint does not render a roster infeasible but it does add a penalty that reduces the solution's quality. Therefore, among all feasible rosters, those with the lowest penalty are optimal solutions. The solver used in this research does not guarantee an optimal solution, but rather aims to find a roster of good quality, one that is feasible and has a low penalty.

Solutions take the form of a roster. A *roster* is created for a specific planning period, and specifies which shifts (including days off) are assigned to each nurse on ever day within that period. *Shifts* are defined by their start time, break time, end time and the skills required for assignment. Each nurse has a *contract* that specifies the minimum and maximum number of hours that they are expected to work during the scheduling period, as well as the skills they posses. Finally, for each shift type and each day within the scheduling period, there is a *coverage requirement*, which specifies the number of nurses that should be assigned to that shift on that day.

The following are additional characteristics of shifts:

Night shift - A shift with at least one hour of work assigned in between 00:00 - 06:00 AM.

Rest shift - These shifts are used to represent paid days off, such as sicknesses. They are used to count towards total hours assigned for the hard constraint: maximum workload, but not used for any other constraint.

Sunday shift - A shift that starts on a Sunday, or on a Saturday and ends on the Sunday.

Consecutive shifts - Shifts assigned to the same nurse that are consecutive in the schedule, and there is ≤ 32 hours between the end time of the first shift and the start time of the second shift.

NRP is defined formally in Theorem 1.

Theorem 1 (Nurse Rostering Problem Formulation)**.** *Let E be the set of employees, D the set of days, and S^+ the set of shifts. Define the assignment variables*

$$X = \{x_{e,d,s} \mid e \in E, d \in D, s \in S^+\},$$

where

$$x_{e,d,s} = \begin{cases} 1 & \text{if employee } e \text{ is assigned to shift } s \text{ on day } d, \\ 0 & \text{otherwise.} \end{cases}$$

Let C_h be the set of hard constraints *on assignments, and C_s be the set of* soft constraints*.*

The Nurse Rostering Problem is to find an assignment X such that:

1. *All hard constraints in C_h are satisfied,*
$$X \models C_h,$$
2. *The objective function*
$$f(X) = \sum_{c \in C_s} w_c \cdot \phi_c(X),$$
where each $\phi_c(X)$ measures the violation of soft constraint c, and $w_c = 1$ is its weight, is minimized.

Thus, the problem is:

$$\min_X f(X) \quad \text{subject to} \quad X \models C_h, \quad x_{e,d,s} \in \{0, 1\}.$$

2.2. Nurse Rostering Problem Variants

The nurse rostering problem was first presented by Warner [24] and Miller et al. [13], which inspired substantial research. These papers explored mathematical programming to create optimal rosters, while respecting different constraints, such as minimum number of nurses per shift.

The nurse rostering problem is a combinatorial optimization problem, where shifts are needed to be assigned to nurses over a scheduling period. The feasible solution space are all assignments that adhere to the hard constraints, while the optimal solutions are the ones within this space that minimizes the penalty. Burke [3] provided a clear description of the problem: given a set of nurses and shift, the shifts need to be assigned to the nurses in a way that adheres to some hard constraints (to ensure a feasible solution) and minimizes a penalty (the objective function). The objective function incorporates various soft constraints that are preferred not to be violated. Often there are different types of nurses and shifts. For nurses, these include different contracts (specifying different working hours) and skills that determine which shifts they can cover. Shifts differ in terms of different start time, end time and skills they require. The aim is to optimize many different objectives, such as adhering to legal requirements, accommodating nurse preferences, complying to hospital regulations, and more [17].

The survey by Ngoo et al. [16] notes that common hard constraints typically include *shift coverage* (ensuring that every shift is staffed by a nurse), and *skill requirements* (ensuring nurses posses the required skills for their assigned shifts). The study also categorizes common soft constraints into three types: series, successive series and counters. *Series constraints* restrict the number of consecutive occurrences, such as days off. *Successive series constraints* limit occurrences of series that immediately follow each other, for example, a series of days worked, that is immediately followed by a series of days off. Lastly, *counters constraints* limit cumulative quantities over specific time period, such as setting a maximum number of working hours a nurse can have within a month.

On the other hand, the NRP considered in this thesis follows the constraints used in the commercial solver from ORTEC B.V.. In this solver, coverage requirement is a soft constraint. The hard constraints mainly focus on legal rules for nurses in the Netherlands, ensuring the nurse has appropriate skills for their assigned shifts, and nurses do not exceed their maximum workload. The soft constraints address factors such as individual nurse preferences, distribution of unassigned shifts across days, spread of overtime hours between nurses and coverage requirements.

The differences between NRP formulation in this thesis and those found in literature become evident when inspecting the common datasets. Ngoo et al. [16] identified the three most common benchmark datasets as INRC-I [10], INRC-II [5] and Shift Scheduling [22]. The hard and soft constraints of these three datasets, along with those used in this thesis are summarized in Table 2.1. The main difference is that shift coverage is a hard constraint in most commonly used literature datasets, whereas it is a soft constraint in the ORTEC NRP. Additionally, in ORTEC NRP, focus on ensuring the roster adheres to the legal working requirements for nurses in the Netherlands.

Table 2.1: Comparison of constraints across different NRP formulations. H means that the constraint is a hard constraint in this formulation, S means the constraint is a soft constraint in this formulation, N/A stands for not applicable and it means it is not a constraint in this formulation.

Constraint Description	**INRC-I**	**INRC-II**	**Shift Scheduling**	**ORTEC NRP**
Skill requirement	H	H	N/A	H
Shift coverage	H	H	H	S
One shift per day	H	H	H	H
Maximum number of assignments per nurse	S	S	N/A	N/A
Minimum number of assignments per nurse	S	S	N/A	N/A
Maximum number of consecutive working days	S	S	H	N/A
Minimum number of consecutive working days	S	S	H	N/A
Maximum number of consecutive days off	S	S	N/A	N/A
Minimum number of consecutive days off	S	S	H	N/A
Maximum number of consecutive working weekends	S	N/A	N/A	N/A
Required complete weekends	S	S	N/A	N/A
Identical shift types during the weekend	S	N/A	N/A	N/A
Minimum days off after night shift	S	N/A	N/A	N/A
Day on requests	S	N/A	S	S
Day off requests	S	S	S	S
Shift on request	S	N/A	N/A	S
Shift off request	S	S	N/A	S
Unwanted patterns	S	H	H	H
Coverage preference	N/A	S	S	S
Maximum number of weekends	N/A	S	H	H
Maximum number of specific shift per nurse	N/A	N/A	H	N/A
Minimum working hours	N/A	N/A	H	N/A
Maximum working hours	N/A	N/A	H	H
Required days off	N/A	N/A	H	H
Coverage surplus	N/A	N/A	S	H
Maximum number of Sundays	N/A	N/A	N/A	H
Maximum number of night shifts	N/A	N/A	N/A	H
Weekly rest	N/A	N/A	N/A	H
Rest after consecutive night shifts	N/A	N/A	N/A	H
Maximum consecutive assignment if includes one night shift	N/A	N/A	N/A	H
Fixed assignment	N/A	N/A	N/A	H
Consecutive days on	N/A	N/A	N/A	S
Consecutive days off	N/A	N/A	N/A	S
Uncovered shift spread	N/A	N/A	N/A	S
Overtime hours spread	N/A	N/A	N/A	S

2.3. Hard Constraints

The hard constraints described in this section, refer to C_h in Theorem 1. As mentioned previously, the hard constraints ensure that the rules set out in Dutch Working Hours Act, the regulations from the collective labour agreement, contractual working hours, and skill requirements for shift assignments are all satisfied. Although the Dutch Working Hours Act contains additional rules, ORTEC determined

that by enforcing the following hard constraints, and by defining breaks and shift lengths in accordance to legal standards, all the regulations are guaranteed to be followed.

The constraints relating to the Dutch Working Hours Act are the following:

Maximum number of Sundays per 52 weeks: Within any 52 weeks period, each employee must have at least 13 Sundays off. This is verified by checking how many Sunday shifts can be assigned for each employee by end of each week in the scheduling period, creating a personalized constraint for that week.

Maximum number of night shifts per 16 weeks: All employees can work at most 36 night shifts within any 16 week period. This is verified by checking how many night shifts each employee is allowed to do by the end of each week in the scheduling period, creating a personalized constraint for that week.

Daily rest: Between any two shifts, all employees need to have sufficient rest. This means that within 24 hours of starting a shift, an employee should receive at least 11 hours of consecutive rest. There is an exception, that every employee is allowed to have a rest of at least 8 hours, but only once every 7 days.

Weekly rest: This constraint ensures that each employee has a longer break on a (bi)weekly basis. Every employee needs to have one of the followings for each shift they are assigned:

- At least 36 hours of consecutive rest within 7×24 hours from the start time of this shift.
- At least 72 hours of consecutive rest within 14×24 hours from the start time of this shift.
- Two blocks of at least 32 hours of consecutive rest, summing up to at least 72 hours in total in these two blocks, within 14×24 hours from the start of any shift.

Rest after consecutive night shifts: After a series of at least 3 consecutive night shifts by an employee, the employee should have a rest of at least 46 hours after the end time of the last night shift.

Maximum series length if series includes at least one night shift: Employee can work at most a series of 7 consecutive shifts if at least one of the shifts is a night shift.

Except the hard constraints from labour rules, there are 5 additional hard constraints:

Min weekends off: Each employee needs to have a minimum of 22 weekends off per 52 weeks. A weekend off is translated to having at least 56 hours off in a row between Friday 4 PM and Monday 8 AM. This constraint is based on the collective labour agreement.

Required skills: Each employee posses a certain set of skills, and each shift requires a set of skills to be performed. An employee can only be assigned to a shift if they have all the skills required for the shift.

Maximum workload: Each employee has an individual contract specifying their contractual working hours per period, typically defined annually. For scheduling purposes, this is converted to a monthly working hours limit. To allow for flexibility, the monthly working hours limit is set to the calculated monthly workload + 10. As a result, it is possible for employees to work some over time. The hard constraint is that no employee may be assigned more than this buffered workload limit.

Fixed assignments: In each roster, there can be a fixed assigned shifts and days off. These cannot be altered. Furthermore, if an employee has a fixed day off on day d, they cannot be assigned a night shift on day $d - 1$.

Over assigning shifts: Each shift has a required and maximum number of employees that can be assigned per day in the scheduling period. It is a hard constraint that no more than the maximum number of employees are assigned to any shift.

2.4. Soft Constraints

The quality of a feasible roster is evaluated by the penalty, which is made from the soft constraints. They refer to C_s in Theorem 1. The penalty is calculated as the sum of all the soft constraints, with a weight of 1. This means w_c in Theorem 1 is 1 $\forall c \in C_s$. These penalties are summarized in Table 2.3.

The soft constraints in this NRP are:

Shift coverage: This constraint is essential to meet the work demand, as it penalized any required shift that is not sufficiently covered. Each shift type is classified as either a regular or priority shift,

altering the penalty weight for uncovered shift. A regular shift that is not covered has a penalty of 90, while a priority shift has a penalty of 490. For each shift type s and day d, there is a coverage requirement which is denoted as $a_{d,s}$. The coverage shortage ($y_{d,s}$) for each day and shift is calculated as: $y_{d,s} = max(0, a_{d,s} - \sum_{e \in E} x_{e,d,s})$ where $x_{e,d,s}$ indicated whether employee e is assigned to shift s on day d. This formulation ensures that there is no reward or penalty for scheduling extra employees. The binary input parameter β_s is used to denote if shift s is a priority shift. This means the overall coverage penalty is calculated as: $\sum_{d \in D} \sum_{s \in S} y_{e,d,s} \cdot (90 + 400 \cdot \beta_s)$

Coverage preference: This constraint provides schedule planners with the flexibility to differentiate between the minimum and preferred number of employees needed per shift. The previously described shift coverage constraint ensures that the minimum requirement is met, while the coverage preference addresses the optimal staffing level. For every preferred but uncovered shift, a penalty of 10 is applied. This preferred coverage for each day d and shift s is denoted as $\varepsilon_{d,s}$. The coverage preference shortage ($\varphi_{d,s}$) is calculated as $\varphi_{d,s} = max(0, \varepsilon_{d,s} - \sum_{e \in E} x_{e,d,s})$. The total coverage preference penalty is calculated as: $\sum_{d \in D} \sum_{s \in S} \varphi_{e,d,s} \cdot 10$. Therefore, when both the shift coverage and coverage preference are considered, the total penalty for a required shift that is not assigned is 100, and an unassigned priority shift is 500.

Shift coverage spread: This soft constraint aims to distribute the coverage shortages more evenly across the scheduling period, preventing substantial understaffing on any particular day. Let y_d denote the total coverage shortage on day d, i.e., the sum of coverage shortages $y_{d,s}$ of all shift types $s \in S$. Since a single shortage on a day ($y_d = 1$) cannot be spread, there is no penalty in this case. For additional shortages, the penalty is increased by an increase of a 100 for each additional shortage. This is made clearer with the following example. If $y_d = 1$, the penalty is 0. If $y_d = 2$, the penalty is 100. If $y_d = 3$, the penalty is 100 + 200 = 300. If $y_d = 4$ the penalty is 100 + 200 + 300 = 600, and so on. The formula for this penalty is: $\sum_{d \in D} \sum_{m=2}^{y_d} 100(m-1)$

Overtime hours spread: This soft constraint encourages the distribution of overtime hours among different employees, rather than concentrating all the hours on a few employees. The penalty is proportional to the quadratic of employee's overtime hours, so assigning x overtime hours to one employee will have a greater penalty than distributing those hours among multiple employees. Each shift $s \in S$ has an input parameter f_s denoting the duration of the shift. Furthermore, each employee e $\in$ E has a contractual working hours c_e^{con}. This means that the overtime of employee e $\in$ E is given by $u_e = max(0, \sum_{d \in D} \sum_{s \in S} (f_s \cdot x_{e,d,s}) - c_e^{con})$. We do not want to reward negative overtime, thus we ensure that u_e is at least a 0. The overtime spread penalty is: $\sum_{e \in E} u_e^2$.

Employee preferences: The remaining soft constraints relate to the individual preferences of the employee. These preferences can include requests for specific days or shifts on/off, preferred shift sequence lengths, and preferred day-off sequence lengths. The penalty weights of these preferences depend on both the employee's contractual hours and the total number of requests the employee has made. Employee with higher contractual hours and fewer requests will have higher weights for each request. Specifically, the weight of each request of employee $e \in E$ is denoted as j_e, and is calculated by dividing the total weight assigned to that employee by the number of requests they make. The total weight for employees with different contractual hours are summarized in Table 2.2.

Table 2.2: This shows the total weight of preference requests of an employee by the number of contractual working hours of the employee. This table was created by Faasse [9].

Contractual Working Hours per Week (λ)	Total Weight
$\lambda \geq 32$	100
$32 > \lambda \geq 24$	80
$24 > \lambda \geq 16$	60
$16 > \lambda$	40

The definitions of the different preference soft constraints are:

- **Day-on/off requests:** Employees have the option to request specific days to work, or specific days off. Each request has an associated weight of j_e, as previously discussed, resulting in a penalty of j_e if the request was not accommodated. Let the notations $g_{e,d}^{on}$ and $g_{e,d}^{off}$ represent the

respective day on and day off requests by employee e on day d. The penalty is then calculated as: $\sum_{e \in E} \sum_{d \in D} (j_e \cdot (x_{e,d,o} \cdot g^{on}_{e,d} + (1 - x_{e,d,o}) \cdot g^{off}_{e,d})$

- **Shift-on/off requests:** Similar to Day-on/off requests, employees can request to be assigned a specific shift on a specific day, or to not be assigned to a specific shift on a specific day. These requests are denoted as $h^{on}_{e,d,s}$ and $h^{off}_{e,d,s}$ respectively. The penalty for not fulfilling these requests is calculated as: $\sum_{e \in E} \sum_{d \in D} \sum_{s \in S} (j_e \cdot (x_{e,d,s} \cdot h^{off}_{e,d,s} + (1 - x_{e,d,s}) \cdot h^{on}_{e,d,s})$
- **Shift sequence length requests:** Employees may request a preferred shift sequence length, indicating how many consecutive shifts they wish to work before having a longer break. The precise definition of a shift sequence can be found in Section 2.1. Let η^{on}_e represent the list of lengths of consecutive shift sequences that employee e has, and ζ^{on}_e represent the requested consecutive sequence length. The penalty for not fulfilling shift sequence length requests is calculated as: $\sum_{e \in E} \sum_{\nu_s \in \eta^{\text{on}}_e} j_e \cdot (\nu_s - \zeta^{\text{on}}_e)^2$
- **Day-off sequence length requests:** Similar to shift sequence length requests, employees can also request a preferred length for consecutive days off. The length of a days off sequence is determined by the number of hours between two consecutive shifts, divided by 24, and rounded to the nearest integer. Sequences are only counted if the interval is at least 32 hours. Let η^{off}_e represent the list of lengths of consecutive days off sequences that employee e has, and ζ^{off}_e represent the requested length of consecutive days off. The penalty for not fulfilling day-off sequence length requests is calculated as: $\sum_{e \in E} \sum_{\nu_o \in \eta^{\text{off}}_e} j_e \cdot (\nu_o - \zeta^{\text{off}}_e)^2$
- **Joker Requests:** These are very similar to the fixed assignment hard constraint, but instead are modelled as a soft constraint, with a penalty of 10 000 000 000 if not satisfied.[1] Let the notations $G^{on}_{e,d}$ and $G^{off}_{e,d}$ represent the respective day on and day off joker requests by employee e on day d. Additionally, let $H^{on}_{e,d,s}$ and $H^{off}_{e,d,s}$ represent the shift on and shift off joker requests. The penalty is calculated as: $(10\,000\,000\,000 \cdot (\sum_{e \in E} \sum_{d \in D} \sum_{s \in S} (x_{e,d,s} \cdot H^{off}_{e,d,s} + (1 - x_{e,d,s})) \cdot H^{on}_{e,d,s} + \sum_{e \in E} \sum_{d \in D} (x_{e,d,o} \cdot G^{on}_{e,d} + (1 - x_{e,d,o}) \cdot G^{off}_{e,d}))$

The summary of the soft constraints is found in Table 2.3.

Table 2.3: Soft constraint penalty calculations, as described in Section 2.4. The used notation is clarified in Nomenclature section. This table is based on a table created by Faasse in [9].

Soft constraint	**Penalty**
Shift coverage	$\sum_{d \in D} \sum_{s \in S} y_{d,s} \, (90 + 400 b_s)$
Coverage preferrence	$\sum_{d \in D} \sum_{s \in S} \varphi_{e,d,s} \cdot 10$
Shift coverage spread	$\sum_{d \in D} \sum_{m=2}^{y_d} 100(m-1)$
Overtime hours spread	$\sum_{e \in E} u_e^2$
Day-on/off requests	$\sum_{e \in E} \sum_{d \in D} j_e \left(x_{e,d,o} g^{on}_{e,d} + (1 - x_{e,d,o}) g^{off}_{e,d} \right)$
Shift-on/off requests	$\sum_{e \in E} \sum_{d \in D} \sum_{s \in S} j_e \left((1 - x_{e,d,s}) h^{on}_{e,d,s} + x_{e,d,s} h^{off}_{e,d,s} \right)$
Preferred shift sequence length	$\sum_{e \in E} \sum_{\nu_s \in \eta^{\text{on}}_e} j_e \cdot (\nu_s - \zeta^{\text{on}}_e)^2$
Preferred day-off sequence length	$\sum_{e \in E} \sum_{\nu_o \in \eta^{\text{off}}_e} j_e \cdot (\nu_o - \zeta^{\text{off}}_e)^2$
Joker Requests	$(10\,000\,000\,000 \cdot (\sum_{e \in E} \sum_{d \in D} \sum_{s \in S} (x_{e,d,s} \cdot H^{off}_{e,d,s} + (1 - x_{e,d,s})) \cdot H^{on}_{e,d,s} + \sum_{e \in E} \sum_{d \in D} (x_{e,d,o} \cdot G^{on}_{e,d} + (1 - x_{e,d,o}) \cdot G^{off}_{e,d}))$

[1] The joker requests are fixed assignment hard constraints, modelled as a soft constraint by ORTEC for the sake of simplicity in instance generation.

3

Nurse Rostering Problem Solving

This chapter reviews previous research on solving the NRP, and introduces the simulated annealing solver used as the starting point for this thesis. Section 3.1 describes different algorithms applied to NRP in literature. Section 3.2 outlines other similar NRP solving techniques. Afterwards, Section 3.3 discusses the application of ejection chains for NRP. Lastly, Section 3.4 describes the current simulated annealing solver that serves as the basis for this research.

3.1. NRP Solvers

Many different approaches have been researched to solve the NRP. The survey by Ngoo et al. [16] provides a comprehensive summary of these methods. Since NRP is NP-Hard, some solvers opt to find a high-quality solution within a reasonable time frame, while others aim to find the optimal solutions but may not complete in practical running time for larger instances.

One solution methodology is Variable Neighbourhood Search (VNS), which aims to find a good solution within a limited time frame. The solver explores different local search moves and applies those that improve the roster. If no improvement is found after a certain number of iterations, the solver performs a perturbation move to diversify the solution. An example of such a solver is proposed by Zheng et al. [25]. The method begins with a random roster and iteratively improves it by applying two different local search moves. The first is a *vertical swap*, which exchanges shifts on selected days between two nurses. The second is a specialized vertical swap that only considers days where one of the nurses has a day off. If no improvement is made for some iterations, the solver applies a perturbation move, which selects a random day and a group of nurses, then performs a cyclic swap of shifts among those nurses on that day.

Another approach to solving the NRP, is through a mathematical optimisation that finds one of the optimal rosters. However, due to the large search space, this method is often not possible on larger instances. For example, Mischek et al. [15] proposed a solution using Integer Programming to find an optimal weekly roster on the INRC-II dataset, without considering the previous weeks. As the soft constraints in INRC-II are affected by previous weeks, the resulting rosters had a large penalty.

Substantial research has also focused on combining mathematical optimisation with metaheuristics. An example is the method proposed by Turhan et al. [23], which combined mixed integer programming (MIP) with simulated annealing. The process starts with a fix-and-relax algorithm, where some shifts are fixed and the rest are assigned using MIP. This solution is passed to the simulated annealing algorithm as the initial solution. The simulated annealing algorithm uses local search moves to improve the roster. When the algorithm fails to improve the roster for numerous iterations, it utilizes a fix-and-optimize algorithm to diversify the search, often also finding a better solution.

3.2. NRP Solver Techniques in Literature

As explained in [19], the main benefit of ejection chains is their ability to generate more complex neighbourhoods from simpler ones, which has shown promise in solving complicated combinatorial methods. This method both diversifies and intensifies the search process. This section will discuss ruin and recreate, and tabu lists, which are both methods that were explored in this thesis. Both methods are employed to improve the diversification in the search.

3.2.1. Ruin and Recreate

Ruin an recreate has been applied to solve NRP in various studies. The idea is to diversify the search by escaping local minima and changing the roster in ways that local search moves alone cannot (similar to ejection chains). The process starts with the ruin step, where different shifts are unassigned. This is followed by the recreate step, where shifts are being assigned [17].

The ruin recreate utilized by Rahimian et al. [17] is used to both intensify and to diversify the search. They introduced a cell penalty to identify parts of the roster that can be improved, it is calculated from determining the penalty contribution attributed to each shift assignment. During the ruin step, shifts with higher cell penalty weights are unassigned. This resulting roster is then passed as the initial roster to an integer programming solver (IP). Because the IP solver runs for a pre-determined time, it may not find the optimal assignments for the given roster. Thus, the ruin and recreate method does not guarantee intensification (it might not have an improved solution) and in those cases it only diversifies the search. An interesting aspect of their approach, is the use of hard constraint relaxation, which allows the ruin and recreate to violate some hard constraints, but these violations are given very high penalties to direct the solver towards feasible solutions.

The paper by Stølevik et al. [21] employs ruin and recreate to escape local minima by diversifying the solution. Their solver begins with a constraint programming approach to find an initial feasible roster. Afterwards, it applied VNS to improve the solution, where the ruin and recreate method is used to escape local minima. During the ruin step, all shifts assigned to a selected set of nurses are unassigned. The recreate step reruns the CP solver, but it is restricted to assign shifts only to nurses unassigned in the ruin phase.

The research by Reid et al. [20] uses ruin and recreate on an NRP that is similar to the one solved in this thesis, with the hard constraints mainly focusing on labour laws (in their case, the British labour laws). Furthermore the coverage requirement in their formulation is also a soft constraint. The ruin step of their solver, considers all shift assignments, and unassign them with some probability. This is followed with the recreate step, where all employees that can work more shifts are being assigned a random shift on a random day, if such as assignment has a fitness above some threshold. In both the ruin and the recreate steps, the threshold to unassign and reassign shifts decrease throughout the solver runtime.

3.2.2. Tabu Lists

Another common technique to diversify the search in NRP solvers is the use of tabu lists. Tabu lists have been used in multiple studies, such as [6, 2, 18]. Bester et al. [2] implemented tabu search by maintaining a list of their recent moves, to prevent reversing recent moves. An exception was made for moves is in the tabu list that result in a new best found roster. In their work, the tabu list was used in combination with ejection chains, demonstrating that these approaches can be integrated. Ramli et al. [18] used a different tabu list strategy, as they opted to maintain a tabu list of entire rosters, preventing the search from revisiting to the same roster too frequently.

3.3. NRP Ejection Chains

The technique we will research is ejection chain. This technique combines multiple local search moves to create a larger move. This combination is achieved by applying one move that changes the solution. This change forces (ejects) another move to be applied, repeating in a sequence (chain). We are focusing on this technique, because one of the solutions generated by the ORTEC solver was improved by a human schedule planner in a Dutch hospital by manually applying an ejection chain. Our goal is to automate this process.

From the literature, four distinct ejection chain approaches for the NRP were identified. The first

approach was introduced by Dowsland [6]. However, the NRP addressed by Downsland is substantially different from the problem tackled in this thesis. As Downsland's NRP focuses on creating *cyclic* schedule, which are schedules that repeat on a weekly basis. The advantages of such schedules include having a balanced coverage, nurses having predictable schedule and new rosters do not need to be generated as often, as noted by Bester et al. [2]. On the other hand, cycles schedules lack flexibility, which are required in situations such as a nurse going on holiday [2].

The ejection chain proposed in the paper by Dowsland [6], begins by assigning an under-covered shift to a nurse. Afterwards, a different nurse receives the entire shift plan of the first nurse, this repeats in a chain. For example, consider nurses A, B and C, the chain starts by assigning a new shift to Nurse A, then nurse B takes over the entire-previous shift plan of nurse A. Afterwards, nurse C takes the full shift plan of nurse B, and so forth. The chain ends successfully if the penalty after these swaps is lower than before the chain started.

The next ejection chain applied to the NRP was proposed by Bester et al. [2], and is called the stepping stone ejection chain. The chain is a closed sequence of alternating horizontal and vertical swaps. A *horizontal swap* exchanges shift assignments between two different days for the same nurse, while a *vertical swap* switches the shift assignments between two nurses on the same day. The chain begins with a horizontal swap between two shifts (A and B), where shift A is a day-off shift, while shift B is not. This is followed by a vertical swap between shift B and a third shift, shift C - a shift on the same day as shift B but is assigned to a different nurse. Next, a horizontal shift involving shift C is performed. This process repeats until the original column is involved in a horizontal swap, closing the sequence. In other words, all shifts involved in the chain undergo an anti clockwise rotation. They demonstrated that their chain outperformed horizontal and vertical swaps.

Burke et al. introduced a novel ejection chain in their paper [4], designed to mimic how human schedule planners create rosters. The chain starts by improving some Nurse A's penalty at the expense of some Nurse B, though a vertical swap between nurses A and B. The next step is to improve nurse B's penalty, since it was just worsened. This is done by doing a vertical swap between nurse B and some other nurse, C, such that nurse B's penalty improves at the expense of nurse C. This repeats until the roster's penalty is lower than the best roster found so far. If the maximum chain depth or time limit is reached without improvement, all swaps are reversed. At each step of the chain, all possible vertical swaps are evaluated, and the swap that produces the best roster is applied. Additionally, each swap in the chain must satisfy the condition that the penalty after the swap, excluding the penalty increase from nurse C (the nurse who's penalty worsened) is lower than the best roster penalty found so far.

The most recent ejection chain approach is introduced by Kingston in [12]. This method explores the region of the search space containing a single defect, which is a single hard constraint violation. The chain begins with a local search move that reduces the penalty but the associated roster has one defect. The chain then proceeds as a sequence of repair moves, where each repair removes the current defect (fixes the violation), but may introduce a new one. The chain terminates successfully when a repair eliminates the defect without introducing a new one, and resulting in a roster that has a lower penalty than the best penalty found so far. Importantly, any repair that introduces multiple defects is disregarded. In effect, the chain is a path through a graph where nodes represent rosters and edges correspond to repair moves. Since each node can have multiple outgoing edges, if one repair does not lead to a successful chain, the next repair is tried. To prevent exhaustive search, each chain has a maximum allocated time. Furthermore, a tabu list of rosters is maintained to avoid cycles within the chain.

Although substantial research has been conducted on ejection chains in the NRP, several gaps and limitations remain:

- One limitation relates to the datasets used for evaluation. Both recent ejection chain studies [12, 4] were tested on the Shift Scheduling dataset. As shown in Table 2.1, this instance differs from the real-life instances explored in this thesis.
- The two most recent ejection chain papers [12, 4] also lack an in-depth analysis beyond the final penalty. Since the ejection chain made up the core of their solvers, these studies do not investigate its role as an additional move within a broader solver framework. Furthermore, there is a need to

investigate the effect of the chains on the penalty breakdown of the final roster, to determine if they are particularly effective at improving specific soft constraints.

- Lastly, limited research has been conducted on how different parameters impact the effectiveness of the chains. For example, there is a lack of research about the impact of maximum chain depth on performance. Although some prior work, such as [4], discusses the effects of certain parameters, such as the maximum time allocated to each chain, the overall understanding is inadequate.

It is worth noting that while Bester et al. [2] investigated the impact of ejection chain compared to other local moves and examined different parameter configurations, this was done using an earlier ejection chain variant, which will not be considered in this research.

3.4. ORTEC Solver

ORTEC has developed multiple nurse scheduling solvers. Recently, the simulated annealing (SA) solver created by Faasse [9] has demonstrated a lot of promise in his thesis. As a result, in this thesis we implemented ejection chains that extend that solver. The descriptions of the chains are in Chapter 4. This section will describe the SA solver in detail. The solver begins with the construction of an initial roster, which is described in Section 3.4.1. Afterwards, the roster is improved by applying various local search moves, described in Section 3.4.2. These moves are used within the broader simulated annealing algorithm, which will be explained in Section 3.4.3.

3.4.1. Construction Method

Before running the simulated annealing algorithm, a construction method is used to greedily assign shifts to employees. It begins by sorting all shifts required to meet the coverage requirement. The exact sorting criteria, determined by extensive testing by Faasse, can be found in [9]. Afterwards, for each shift, all employees with the required skills are sorted (according to a criteria also described in [9]). The algorithm then iterates through the list of eligible employees for each shift, greedily assigning the shift, as long as the employee is available (they do not have a shift on that day), and assigning the shift does not violate any hard constraint. Otherwise, the algorithm considers the next employee in the list. This process is repeated for all shifts.

3.4.2. Neighbourhood Structures

Neighbourhood structures are the set of local search operations that the solver uses to move from one roster to another. The neighbourhood operations are the followings:

- **Vertical Swap (V_x):** The vertical swap is performed on two different nurses, e.g. A and B, by exchanging their shifts within a consecutive block of x days. After the swap, nurse A is assigned all the shifts that nurse B originally had during these x days, and vice versa.
- **Horizontal Swap (H_x):** The horizontal swap involves exchanging two blocks of x consecutive shifts within the schedule of the same nurse. Specifically, two non-overlapping blocks of equal length but occurring in different periods are selected and their shifts are switched.
- **Change (C_x):** These operations assign a specific shift to a nurse for x consecutive days. If the nurse already has a shift assigned on one of those days, that previous shift is unassigned.
- **Coverage-focused (UC):** There are three different coverage focused operations. These operations aim to improve the coverage, which is the most important soft constraint. The operations are the following:
  - **UC1:** Assigning an under-covered shift to a nurse, who currently has a day off on the specific day.
  - **UC2:** Assigning an under-covered shift to a nurse who currently has a day off on the specific day, while also unassigning a shift from a different day, to help avoid the maximum workload hard constraint.
  - **UC3:** Assigning an under-covered shift to a nurse, who already has a different shift currently assigned on that day.

To reduce the search space, there are multiple cases where a move is not considered. These include:

- **No change:** A move is not considered if it does not change the roster.
- **Violation:** A move is not attempted if applying it will cause a hard constraint violation in the roster.
- **Non-required shift:** A change or swap is not considered if it involves assigning a shift on a day when that shift is not required.
- **Equivalent operation:** A move is not considered if it is equivalent to another move of a smaller block size. For swaps, this means the first or last shift in the block are the same in both blocks. For a change, this means that the first or last shift in the block is the same as the shift being assigned.

3.4.3. Simulated Annealing

The solver employs a simulated annealing algorithm. At each iteration, a random neighbour of the current roster is generated by applying a randomly selected move. If the resulting roster is feasible with a lower penalty, it is accepted and the solver continues from this roster. Otherwise, if the new roster is feasible but does not improve the penalty, it may still be accepted with a certain probability. This probability decreases as the penalty increase grows. Furthermore, in the beginning of the search operations that increase the penalty are more likely to be accepted, compared to the end of the search. Specifically, the probability of accepting a move is calculated as: $P(\Delta) = e^{\frac{-\Delta}{T}}$ Such that Δ is the increase in penalty and T is the temperature. The temperature T decreases over time, making it more likely to accept worse solutions at the start of the search and less likely as the search continues.

4

Approach

This section describes the three different ejection chains researched. This first ejection chain mainly focuses on exploring the infeasible search space by trying to find a better roster, through having infeasible rosters as intermediate steps. It was mainly inspired by the paper by Kingston [12]. The implementation details describing the chain are in Section 4.1. Afterwards, the second ejection chain implementation is described, which is inspired by the chain by Curtois et al. [4]. It focuses on emulating the way schedule planners operate. The implementation of it is explained in Section 4.2. The third ejection chain is a chain of ruin and recreate, exploring the combination of the concepts - ruin and recreate and ejection chains. It is discussed in Section 4.3. Henceforth, the first ejection chain discussed is called InfeasibleEC , the second chain is referred to as EmulateEC , while the third ejection chain is named RuinRecreateEC .

4.1. InfeasibleEC

The aim of InfeasibleEC is to leave the current local minima, while also finding better rosters. This is done by allowing infeasible rosters as intermediate steps. The chain starts after applying a swap that improves the penalty of the roster, but also causes a single *defect* (a single hard constraint violation). This new roster is considered interesting, due to its lower penalty, and the potential to fix the defect while maintaining the lower penalty. A sequence of repairs is used to fix the roster, while maintaining a lower penalty. A repair is defined as a swap which removes (or helps remove) a defect, but potentially introduces a new defect. This means that when applying a repair, the resulting roster could have a new defect. In that case, another repair needs to be applied to fix the new defect. More information about these repairs can be found in Section 4.1.1. In order to start an ejection chain, or for a repair to be accepted, the resulting roster needs to follow the below criteria:

$$\begin{gathered} p_{\text{resulting\_roster}} < p_{\text{pre-ejection\_roster}} \\ f(\text{resulting\_roster}) \leq 1 \end{gathered} \tag{4.1}$$

where the *pre-ejection roster* is the roster before the start of the ejection chain and $f(r)$ is the number of defects that are present in roster r. If $f(current\_roster) == 0$, then the ejection chain is finished and results in a successful ejection chain as the ejection chain found a better roster than the original roster (lower penalty, while maintaining feasibility). On the other hand, if the number of defects equals 1, then the ejection chain will continue by trying to repair its infeasibility. A useful way to visualize these ejection chains, is to consider them as a path in a graph, where each roster is a node, and an edge between nodes I and II signifies the existence of a repair that turns roster I into roster II. The ejection chain starts from a node that is a feasible roster, and finds a path through nodes (that represent infeasible rosters) until it finds a roster that is feasible and with a better penalty than the original roster. Each of these intermediate nodes must have better penalty than the pre-ejection roster.

4.1.1. Repair Generators

Repair generators take as input the defect in the roster and return different swaps that might solve it. Each hard constraint needs to have its own generator. The repair can be part of different neighbourhoods, can have different block lengths and different methodologies. Different methodologies means fixing the violation in a different way. To make the idea of repair methodology more clear, consider an example for the hard constraint 'rest after consecutive night shifts'. It is possible to fix such a violation by using the following vertical swap repairs:

- Having a vertical swap that reduces the number of consecutive night shifts to a value lower than 3.
- Having a vertical swap that combines the current block of night shifts with the next block of night shifts, resulting in one large block of night shifts that has sufficient rest afterward.
- Having a vertical swap that increases the rest after the consecutive night shifts.

These methodologies in the respective order can be found in Figure 4.1. The repairs shown are all vertical swaps of block length 1. It is important to note that these examples are valid repairs, although they can cause a new defect. E.g. the third example results in a daily rest violation for employee 4.

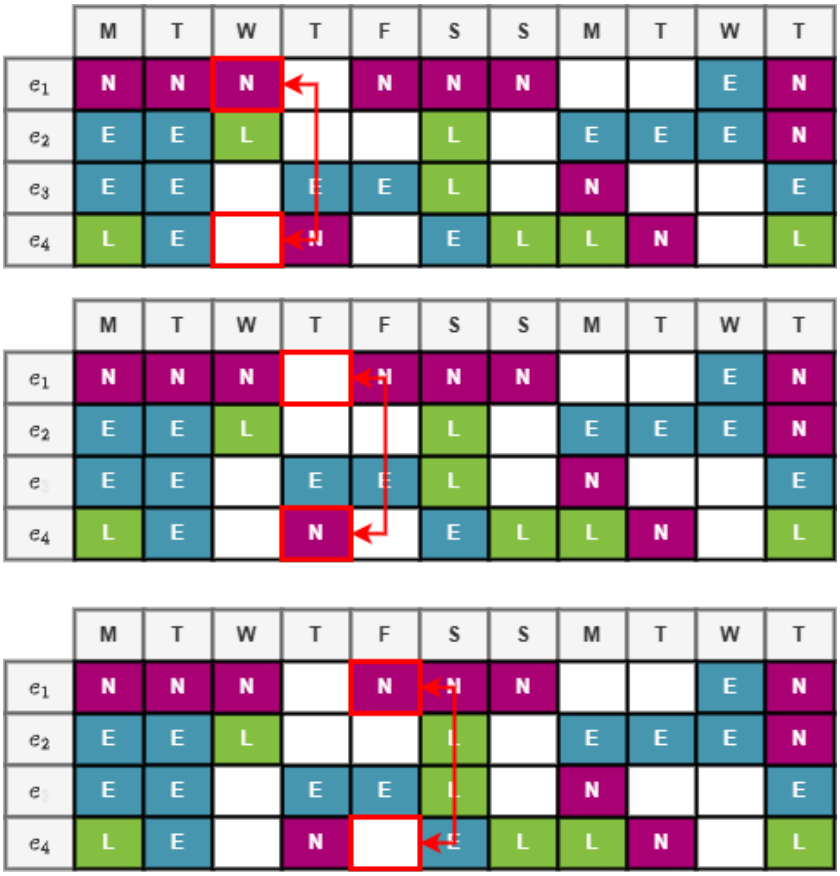

Figure 4.1: An example of three different repair methodologies for a violation of the hard constraint, rest after consecutive night shifts. First repair is an example of decreasing the number of night shifts to be lower than 3. The second repair is an example of combining two different night shift blocks. Lastly, the third repair example shows increasing the rest time. N stands for night shift, E for early shift, and L for late shift.

Due to the large amount of possibilities in designing repairs for each hard constraint, we opted to use data insights for deciding which repairs to check. We used information from previous research and preliminary experimentations to make these decisions.

The main repair consist of a vertical swap, because it is the most common swap in literature [4] and the one that shift planners opt to use to find better rosters manually [4]. Moreover, these swaps are intuitively better than the other swap types. With respect to horizontal swaps, they keep the shifts that

are harder to assign (e.g. Sunday shifts), rather than move them to a different day. Secondly, in the case where all shifts are assigned, a horizontal swap will never be successful, as the solver does not allow overstaffing. For the same reason, vertical swaps are also better than change operators that assign a new shift. On the other hand, change operators that unassign shifts, make the roster worse by reducing shift coverage, so it is better to avoid these swpas as well. Lastly, the coverage focused swaps, which are mainly aimed to make a feasible roster better, will only repair an infeasible roster on very rare occasions. Due to these reasons, we decided to solely use vertical swaps as repairs.

During preliminary experiments, we attempted to use vertical swaps of greater length, as well as horizontal swaps. Both were not effective in fixing the defect, without making the roster have multiple defects. Due to the above reasons, it was decided to only focus on vertical swaps of length one for the repair generator.

We also decided not to attempt repairing the hard constraints: fixed assignment constraint, the required skill constraint, and over assigned shift constraint. These were excluded as they do not help as intermediary step. In the case of a violation of the required skills, the shift that was recently assigned (which the employee is not qualified to do), will need to be unassigned, and thus having it as an intermediately step is not beneficial. The same is true in the case where a fixed shift is being unassigned. As this shift will need to be reassigned to the same person in later steps.

After gaining insights into the problem, analysing initial results of the preliminary experiments and looking into previous research, we came up with 8 different repair generators for 8 different hard constraints. These generators test multiple vertical swaps of block-length 1. The swaps that cannot repair the defect are deleted from the list. The remaining items are repairs: they might fix the defect or result in a roster that is *closer* to being feasible. For example, in the case of a hard constraint violation concerning max workload, a swap that reduces the employee's workload, though still exceeding the limit, would be considered as it would result in a roster that is closer to feasibility. In terms of implementation, the repairs themselves are constructed lazily (a new repair is only made when needed, instead of all the repairs made at once). The pseudo code for the generators for each hard constraint can be found in Appendix B.

4.1.2. Cycles

One major pitfall that needs to be considered when implementing an ejection chain is the possibility of cycles. If the search has cycles, it will repeat, and rather than exploring new search space regions, it will get stuck and reconsider rosters that were already visited before. Figure 4.2 shows a possible example of a cycle. In this example, the cycle happens in the repair operation when solving the hard constraint max night shifts. In the example, the possible repairs are ordered first by employee id, and than chronologically. As a result, when fixing the max night shift defect on employee 1, the chain first attempts a vertical swap with employee 2. The generator realizes that on Tuesday, employee 2 has an off day, while employee 1 has a night shift. Thus, the ejection chain attempts the repair of a vertical swap between employees 1 and 2 on Tuesday. This new roster achieves the criteria (it has one defect and better penalty than the pre-ejection roster). In the next iteration, the ejection chain tries to solve the infeasibility of this new roster. The new defect is a max night shift violation of employee 2. Due to the sorting criteria, it first looks at the first employee, which is employee 1. It finds the first day where employee 2 has a night shift and employee 1 does not have a night shift, which is Tuesday. After this swap, the result is the original roster we tried to fix. This means that if we try to solve this roster, it will find repairs that will cause the two rosters to continuously alternate. This is very harmful, as it wastes a lot of time trying the same rosters over and over, without finding any new solutions. This highlights the importance of avoiding cycles.

Three different methods were implemented to avoid these cycles. The methods used were identified during the parameter tuning, which can is in Chapter 5. The three methods are:

Tabu List of Swaps

One possible solution to reduce the number of cycles, is to use a tabu list of swaps for each ejection chain. This ensures that any swap that was applied will not repeat in the same chain (which would undo the original swap). This reduces the number of cycles, including the cycle shown in Figure 4.2. This is because the original swap between employees 1 and 2 will be added to the tabu list, not allowing

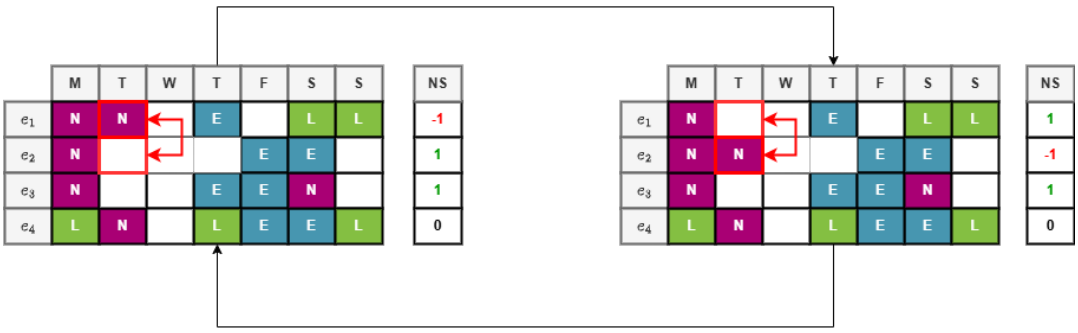

Figure 4.2: An example showing a possible cycle. It is a schedule for 4 employees throughout the week. The NS column shows the remaining allowed night shifts per employee for the scheduling period (originally employee 1 has one too many night shifts; afterwards this holds for employee 2). N are ight shifts, E are early shifts and L are late shifts.

to undo that swap by swapping the shifts on Tuesday between employees 2 and 1. The disadvantages of this tabu list are that it does not remove every cycle, and that it removes possible repairs that do not result in a cycle. On the other hand, its advantage is that checking such a list is very fast and does not require a lot of memory.

Tabu List of Rosters

Another potential solution is the use of a tabu list of rosters. This is very similar to the previous method, but instead of maintaining a list of previous swaps, it keeps track of all the rosters that were explored during an ejection chain. The advantage of this method is that it removes all cycles during an ejection chain. The disadvantage is that a roster is larger than a swap, and thus it requires more memory and takes longer to check if a roster has been visited before.

Randomization

The third method that will be explored is to randomize the order of employees and days when generating repairs. On its own, it will make it less likely to have cycles, as if a swap occurs, it is less likely that the next swap will undo it. The advantage of this method is that it is very fast, and does not require additional memory. The disadvantage is that it does not remove the possibility of cycles, but rather makes them less frequent.

4.1.3. Synthetic Constraints

Preliminary experiments showed that the min weekends off hard constraint was violated the most frequently in the VPA instance, but those chains were successful only on rare occasions. The min weekends off constraint being calculated on a rolling horizon basis. This means that for every week, there is a maximum number of weekends the employee can work during the scheduling period until the end of that week. When checking the occurrences of these violations, it was apparent that most occurrences of the defect were relating to a single employee, during a week where the employee was not allowed to work the weekend. In other words, the employee must be assigned a days off during that weekend, referred to as *synthetic days off* - a day off constraint that is made from other constraints. Because if the employee would work on those days, it would result in a violation. [1]

The synthetic days off are a result of the history (the period before the current scheduling period) and hard constraints. Thus, we fixed the issue of having high penalty due to the chain exploring the synthetic days off, by checking if a violation is a synthetic day off given the history. If that is the case, the chain does not consider that violation. Consider the example of maximum number of Sundays hard constraint, when an employee has 0 allowed Sundays during a specific period, solely based on the history. A roster that has a Sunday shift assigned to the employee during that period is not considered as a valid roster to be an intermediate step in InfeasibleEC.

[1] Previously, in 4.1.1, we discussed that we would not consider fixed day off violations as valid defect for the chain to fix. The chain performing badly due to this synthetic day-off further supports this decision.

4.1.4. Search Methods

As mentioned previously, a good way to visualize InfeasibleEC is by considering the chain as a path in a graph, where each node is a roster and an edge is a repair. Naturally, there are multiple ways to explore a graph looking for a successful path. Different search methods explore the search space in a different way and order. In this chapter, we explain the four different methods that were implemented.

Depth First Search Until Feasibility (DFSF)

The first method follows a depth first search of the different nodes, until a roster that is feasible and has better penalty than the original roster is found. An example of such a search can be found in Figure 4.3. The search starts from the N0, the initial infeasible roster with lower penalty, and it explores the nodes in a depth first manner. This means exploring all the nodes within the depth limit of the first neighbour explored, so after N1, it explore N2 and N3. As they do not have any neighbours, the search backtracks to N1 and then to N0. Moving to N0's next neighbour, which is N4, and then to N5. As N5 is a feasible roster, the chain ends successfully and the solver continues its search from N5.

It is important to note that the search is limited by both a maximum depth and a time restriction. This means that as soon as a node of maximum depth or with no possible repair is reached, we back track to a previous node, continuing the search. The ejection chain can end in one of three scenarios: it explored all nodes within the depth limit, it reached the time limit or it found a feasible roster.

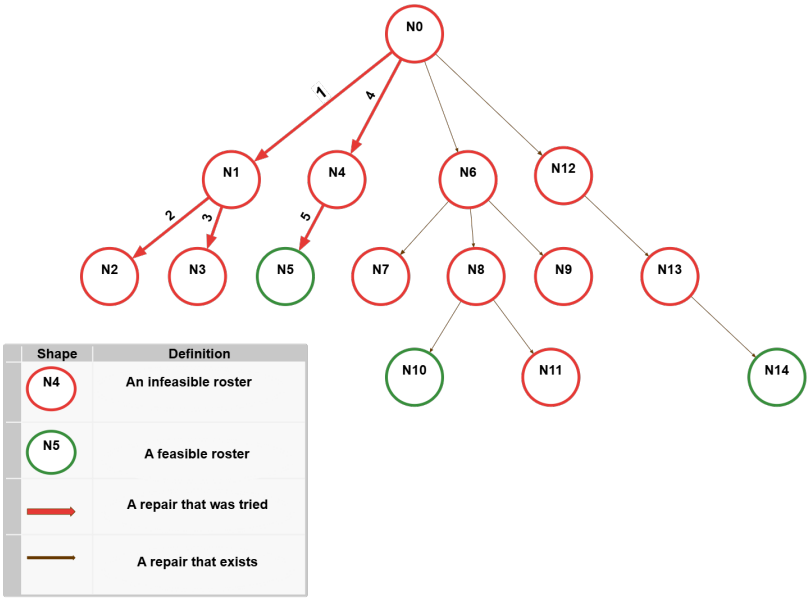

Figure 4.3: Order of nodes and edges explored during an InfeasibleEC chain that uses depth first search until feasibility.

Depth First Search (DFS)

This search method is very similar to the previously described method. The difference is that in this variant, feasible rosters are stored and instead of ending the ejection chain, the search continues by backtracking to the previous node. This approach explores the full search space of the graph (within the depth and time limits). When the search finishes, the feasible roster with the lowest penalty that was found is used.

An example of such a search methodology can be found in Figure 4.4. The chain starts the same way as DFSF, but after node 5, it backtracks back to N4 then to N0, and continues the search. By the end of the search, three feasible rosters are found: N5, N10 and N14. The solver continues from the roster of the one that has the lowest penalty.

Naturally, this kind of search mainly explores rosters that are very different from the original roster, as it spends the majority of its search time exploring nodes close to the maximum depth. This is the first advantages of depth search with respect to breadth search. The other advantage is related to the

implementation: for breadth first search, there is a need to either store the different rosters, which is not ideal in terms of memory, or follow edges more often which takes significant time.

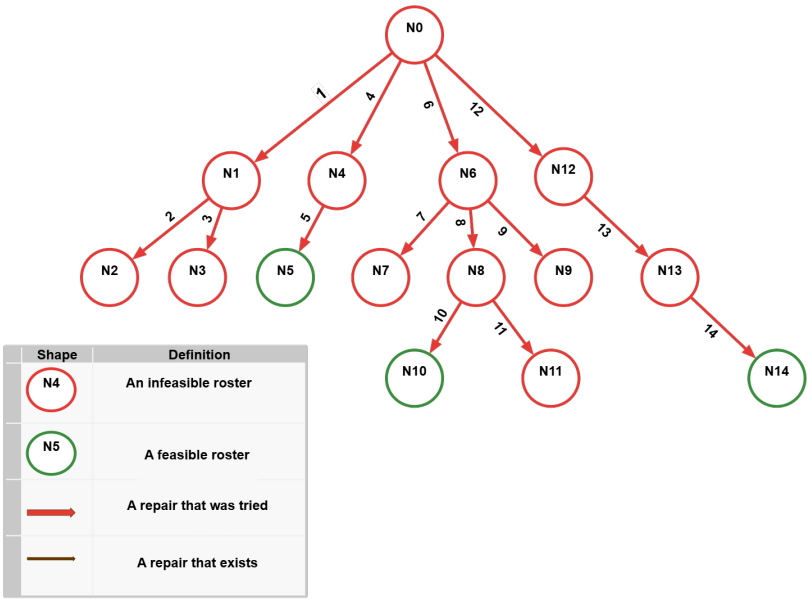

Figure 4.4: Order of nodes and edges explored during an InfeasibleEC chain that uses depth first search

Depth First Search Without Backtracking (DFSN)

The aim of this method is to quickly explore the search space, spending minimum time per ejection chain, regardless of the shape of the graph. This is done by choosing a single path to explore. In other words, at each node, you choose the first edge you find that results in a node that meets the criteria, and continue the ejection chain from that node. The ejection chain ends if a feasible roster is found, the maximum depth is reached or the time limit is reached.

An example of such ejection chain can be found in Figure 4.5. Once again, the chain starts from N0, moves to N1, and lastly N2. As N2 does not have any neighbours, the chain ends, this time unsuccessfully. This means the solver will continue from the pre-ejection chain roster.

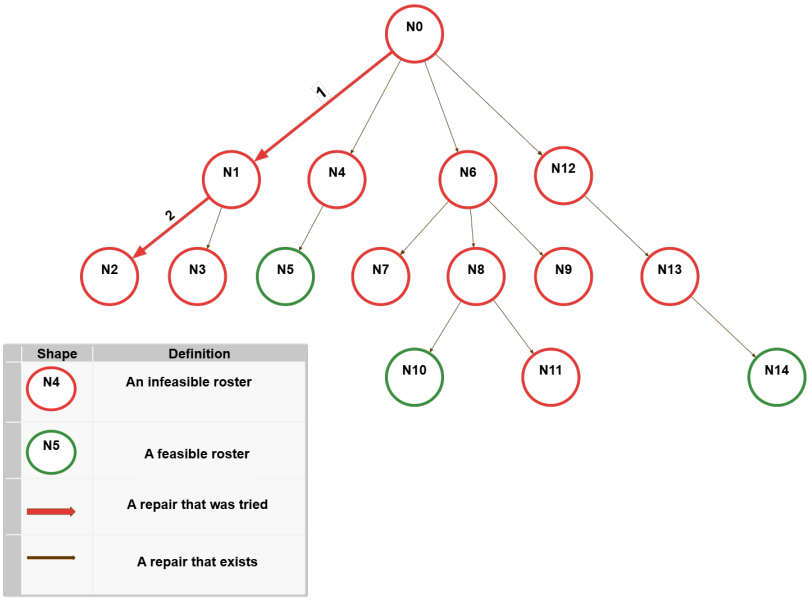

Figure 4.5: Order of nodes and edges explored during an InfeasibleEC chain that uses depth first search without backtracking

Breadth First Search Without Backtracking (BFSN)

The final search method that we implemented is a breadth first search without backtracking. In this method, each node explores all its neighbours, and the most promising neighbour will be chosen to continue the ejection chain. To choose the most promising neighbour, the specific instance is studied during the run. This is done by keeping track of statistics regarding the ejection chain nodes, and when they result in a successful chain. These statistics are used to estimate which neighbour is easiest to repair, continuing the search from that neighbour.

An example of this type of search can be found in Figure 4.6. It starts by exploring all the neighbours of N0: N1, N4, N6 and N12. By comparing all the chains, it considers N6 easiest to fix, thus it continues from N6. This time it finds N7, N8 and N9, finding N8 easiest to fix. Lastly, it explores the neighbours of N8, which are N10 and N11. As N10 is feasible, the chains ends successfully and the solver continues the search from N10.

We use a comparator method that compares all neighbouring rosters, and identifies the neighbour that is estimated to be the easiest to fix. To compare two rosters, we first check if both rosters are feasible. If so, we choose the roster with the lower penalty. If only one roster is feasible, we choose the feasible roster. If both rosters are not feasible, we choose the roster that seems more likely to be repaired based on history. First, we check the type of hard constraint violation and if those differ, we choose the one that during previous ejection chains had a better ratio of being repaired. If the violated hard constraints are the same, we choose the roster where the employee that has the violation had a better ratio of being repaired in previous ejection chains. This comparator is shown mathematically below:

Let S_1 and S_2 be two nurse schedules. Define:

- $P(S)$: penalty of schedule S.
- $F(S)$: feasibility indicator, where $F(S) = 1$ if S is feasible, otherwise $F(S) = 0$.
- $h(S)$: hard constraint violation in schedule S.
- $R(h)$: resolution ratio for hard constraint violation h, defined as the number of times a hard constraint of same type as h is resolved divided by its occurrence.
- $R_e(h)$: employee-specific resolution ratio for hard constraint violation h.
- $E(h)$: indicator function where $E(h) = 1$ if the hard constraint type of h or the affected employee was encountered before, otherwise $E(h) = 0$.

The preference relation $S_1 \succ S_2$ (i.e., S_1 is better than S_2) is determined by the following criteria:

1. **Feasibility Check:**

$$\text{If } F(S_1) > F(S_2), \text{ then } S_1 \succ S_2. \tag{4.2}$$

$$\text{If } F(S_1) = F(S_2) = 1, \text{ then select the schedule with the lower penalty:}$$

$$\text{If } F(S_1) = F(S_2) = 1 \land P(S_1) < P(S_2) \Rightarrow S_1 \succ S_2. \tag{4.3}$$

2. **Encountered Defect or Employee Check:**

$$\text{If } F(S_1) = F(S_2) = 0 \land E(h(S_1)) < E(h(S_2)), \Rightarrow S_1 \succ S_2. \tag{4.4}$$

3. **Hard Constraint Violation Check (if infeasible):**

$$\text{If } h(S_1) \neq h(S_2), \text{ then select the schedule with the higher resolution ratio:}$$

$$\text{If } F(S_1) = F(S_2) = 0 \land E(h(S_1)) = E(h(S_2)) \land h(S_1) \neq h(S_2) \land R(h(S_1)) > R(h(S_2)) \Rightarrow S_1 \succ S_2. \tag{4.5}$$

4. **Same Hard Constraint Violation Type:**

$$\text{If } h(S_1) = h(S_2), \text{ then compare employee-specific resolution ratios:}$$

$$\text{If } F(S_1) = F(S_2) = 0 \land E(h(S_1)) = E(h(S_2)) \land h(S_1) = h(S_2) \land R_e(h(S_1)) > R_e(h(S_2)) \Rightarrow S_1 \succ S_2. \tag{4.6}$$

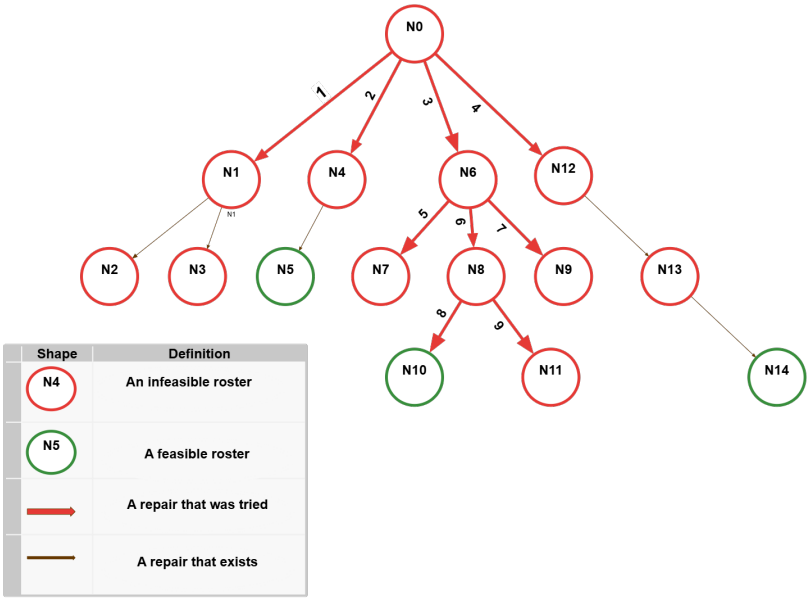

Figure 4.6: Order of nodes and edges explored during an InfeasibleEC chain that uses breadth first search without backtracking.

4.1.5. Example

An example of InfeasibleEC is shown in Figure 4.7. The example contains an initial swap followed by two repairs. The initial swap is a vertical swap that moves a night shift from employee 2 to employee 1 on Wednesday. This results in an infeasible roster, as employee 1 now has an excessive number of night shifts. But the resulting roster has potential, due to its lower penalty (both employee 1 and 2 have a schedule that fits their consecutive shift preference better), and the roster has a single defect. The first repair swaps the shifts on Monday between employee 1 and 3. This fixes the maximum night shift defect, but it introduces a new defect. The new defect is a daily rest violation for employee 3 as this employee starts a night shift on Monday followed by a morning shift on Tuesday which implies insufficient rest between these two shifts. Once more, the roster needs to be fixed with a repair. The second repair is a horizontal swap moving the night shift of employee 3 from Monday to Sunday. This results in a

successful ejection chain. As the new roster is feasible, and the penalty has decreased with respect to the pre-ejection roster. This decrease is due to the fact that the soft constraint sequence preference has a lower penalty in the roster after the chain compared to before the chain.

This example shows the two different advantages of ejection chains. First, we improved the roster by finding a new roster with lower penalty, by passing through intermediate infeasible rosters. The second advantage is that we found a roster that is part of a different search space region. It is important to note that the new roster can be improved further: a morning shift that still needs to be scheduled on Sunday can now be assigned to employee 2 without making the schedule infeasible. This morning shift could not be assigned in the initial roster without a violation.

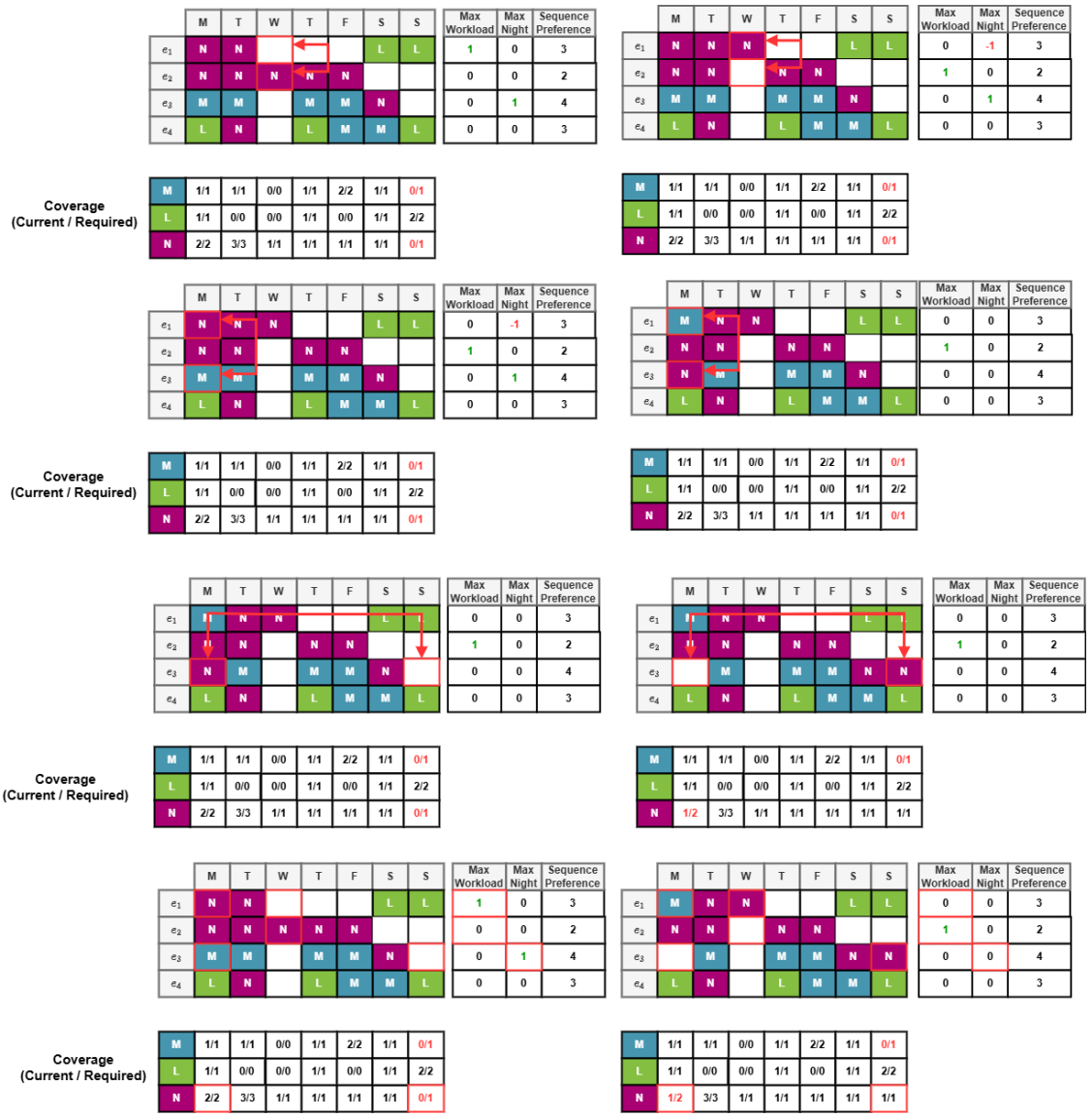

Figure 4.7: A potential ejection chain for a roster with 4 employees throughout a week. Each schedule has a table showing the coverage required per shift per day. As well as a table to help keep track of hard constraints and soft constraints. The column 'Max Workload' shows the number of work shifts an employee can still do during the scheduling period. Max Night shows the number of night shifts an employee can still do during this scheduling period. Lastly, 'Sequence Preference' is the number of consecutive shifts that an employee prefers before having time off. Each row shows on the left the roster before the swap and on the right the resulting roster. M are morning shifts, L are late shifts and N are night shifts.

4.2. EmulateEC

In this section, we discuss a second ejection chain that we call "EmulateEC". This ejection chain emulates how human nurse schedule planners operate: by ejecting a chain of vertical swaps, where each swap improves the penalty of some nurse A, while lowering the penalty of some nurse B. This forces the next swap, which should improve the penalty of nurse B. But this new swap might lower the penalty of some nurse C. In that case, we try to find a swap that improves the penalty of nurse C. This process continues until we find a roster that has a better penalty than the original roster. Unlike InfeasibleEC , this ejection chain does not allow for infeasibility in the intermediate rosters. Instead, it aims to search through the search space of lower penalty rosters. This is an implementation of the ejection chain that was discussed in the paper by Burke et al. [4].

This ejection chain can be visualized as a graph, where each node is a roster. In this graph, an edge between nodes I and II, signifies a vertical swap that turns node I into node II. It is important to understand that both node I and node II represent feasible rosters, so they do not have defects. The ejection chain explores all neighbours of the current node and chooses the node that has the best penalty to continue the search. In order to consider a node, it must satisfy one of the following two criteria:

$$P(R_{\text{resulting\_roster}}) < P(R_{\text{best}}) \quad \text{or} \quad P(R_{\text{resulting\_roster}}) - P(N^{C}_{\text{resulting\_roster}}) + P(N^{C}_{\text{current}}) < P(R_{\text{best}}) \tag{4.7}$$

where

$P(R_{\text{resulting\_roster}})$ is the penalty of the new roster,
$P(R_{\text{best}})$ is the penalty of the best roster found so far,
$P(N^{C}_{\text{resulting\_roster}})$ is the penalty contribution of nurse C in the resulting roster,
$P(N^{C}_{\text{current}})$ is the penalty contribution of nurse C in the current roster.

The resulting roster roster is the roster after the swap that is currently being attempted, and current roster, is the roster that this swap is being attempted on.

The criteria states that for a node to be considered, it needs to satisfy at least one of two criteria. The first option is that the resulting roster has a lower penalty than the best roster found so far. If this is the case, then the ejection chain ends successfully. The second option is that the roster's penalty, without considering the increase of employee C (whose penalty got worse), has a lower penalty than the best roster. If this is the case, the ejection chain continues with the next move.

To make it more concrete, an example of the penalties is given in Table 4.1. You can see that initially there is a swap between nurses A and B which improves nurse B's roster, but which at the same time makes nurse A's roster and the total penalty worse. This swap satisfies the second criteria, as when not considering nurse B it has a lower penalty. This is because: $84 - 19 + 15 = 80 < 82$. Afterwards, the chain attempts to improve nurse B's penalty by doing a swap between nurse B and C. Once again the resulting roster adheres to the criteria as: $83 - 14 + 12 = 81 < 82$. This is followed by a swap between nurses C and E. This swap results in a roster with lower penalty, and thus the ejection chain ends successfully.

Table 4.1: A table showing the penalties of 5 different nurses during a run of EmulateEC.

Nurse	Best Roster	Current Roster	A-B swap	B-C swap	C-E swap
A	15	21	18	18	18
B	16	15	19	16	16
C	12	12	12	14	11
D	20	20	20	20	20
E	19	15	15	15	16
Total penalty	82	83	84	83	81

The search method in this ejection chain explores all the neighbours of each node, before continuing with the best node. Unlike InfeasibleEC, this method does not allow for infeasibility, thus deciding on the best roster to in each step is done by choosing the roster that has the lowest penalty out of all

neighbours satisfying (Equation (4.7)). After moving to a new neighbour, the ejection chain continues by exploring all the neighbours of the new roster. This process continues until either the time limit is reached, or the maximum ejection chain length is reached, or a roster with better penalty than the best roster is found. Pseudo code for the ejection chain can be found in Algorithm 1.

Algorithm 1 EmulateEC

```
1: function EmulateEC(endTime, remainingDepth)
2:     if depth ≤ 0 ∨ now > endTime then return false
3:     end if
4:     bestSwap ← null
5:     minPenalty ← ∞
6:     for blockLength = 1 to 4 do
7:         for all Vertical swaps s of block length blockLength not in tabu list that satisfy Equation (4.7)
   do
8:             p ← PenaltyOf(s)
9:             if p < minPenalty then
10:                minPenalty ← p
11:                bestSwap ← s
12:            end if
13:        end for
14:    end for
15:    if bestSwap == null then
16:        return false
17:    end if
18:    tabuList.add(bestSwap)
19:    apply(bestSwap)
20:    if newPenalty < bestPenalty then return true
21:    end if
22:    return EmulateEC(endTime, remainingDepth − 1)
23: end function
```

Example:
An example of an EmulateEC chain can be found in Figure 4.8. The chain starts with a vertical swap between employee 3 and employee 1 on Friday. This swap makes the penalty of the roster worse, but when you do not consider the impact on employee 1 it is better, as the preferences of employee 3 are better followed. Next, we execute a vertical swap between employee 1 and 4 on Wednesday and Thursday. Once again, the new roster is worse than the pre-ejection chain roster, but if we do not consider nurse 4, the roster has improved. The chain finishes succesfully with a swap between employee 4 and employee 1, as the new roster better follows the preferences of employees 1, 3 and 4.

4.3. RuinRecreateEC

The third ejection chain implemented is a chain of ruin and recreates. The idea is to combine several smaller ruin and recreate moves into a single larger compound operation.

Preliminary experiments on InfeasibleEC showed that substantial modifications to the roster are helpful for searching different regions of the search space. Ruin and recreate operations, as shown in previous studies [17, 21, 20], are another effective technique for this purpose. However, Faasse [9] found that there was minimal impact on the quality of the final roster, whether the solver started from an empty roster, or from a roster constructed by greedy assignments. In our ruin and recreate, the "recreate" phase involves greedily assigning shifts. This means that if too many shifts are removed (during the ruin phase) the search will essentially restart from scratch, with a roster similar to one found by greedy assignments.

To address this issue, the chain of ruin and recreates applies several smaller ruin and recreate operations consecutively. This approach ensures that the chain makes substantial changes, diversifying the search, yet not so drastic that the solver effectively resets to an empty or ineffective starting point. By combining multiple smaller ruin and recreates in a sequence, the solution cannot change the roster excessively,

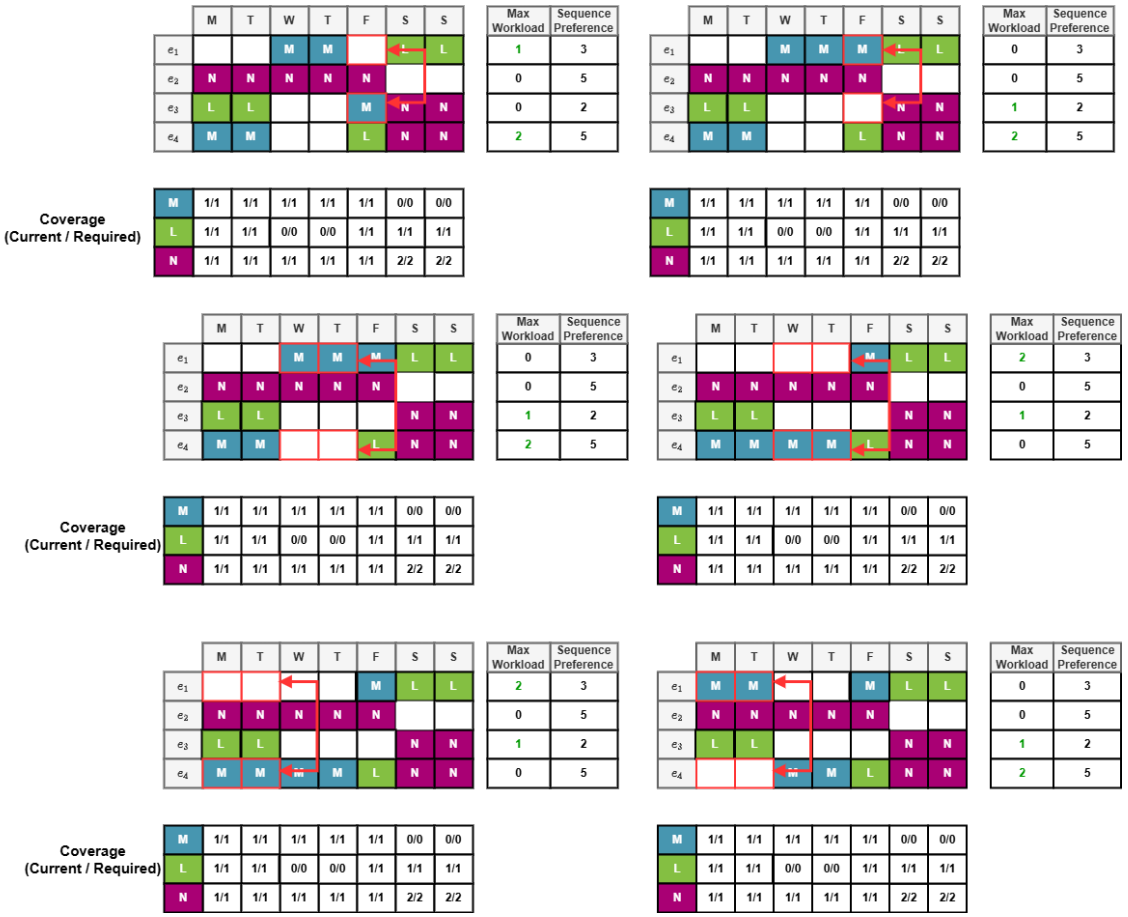

Figure 4.8: An example of an EmulateEC chain for a roster with 4 employees throughout a week. Each schedule has a table showing the coverage required per shift per day. As well as a table to help keep track of hard constraints and soft constraints. The column 'Max Workload' shows how many shifts an employee can still do during the scheduling period. Sequence preference shows the number of consecutive shifts an employee prefers to do before having a longer break. Each row shows on the left the roster before the swap and on the right the resulting roster. M are morning shifts, L are late shifts and N are night shifts.

resulting in a recreated roster that is more similar to the initial roster.

The psudeo code for this algorithm is presented in Algorithm 2. The idea of the algorithm is to perform a chain of n ruin and recreate operations. In each step, the algorithm selects k shifts to unassign (out of all shifts that have not yet been unassigned during the current chain) by evaluating the change in penalty of unassigning each shift. Shifts whose unassignment results in a penalty increase less than a threshold p are added to a candidate list. To bias the selection towards better candidates, the penalty differences in the candidate list are offset such that lower penalty increases correspond to higher selection probabilities. This is done by computing for each penalty difference p_i: $p_{i_{offset}} = max(p) - p_i$, where $max(p)$ is the highest penalty difference in the list. Afterwards, using a roulette wheel selection, we choose a shift to unassign. This involves generating a random number between 0 and the sum of all offset penalties. The shift corresponding to the point at which the cumulative sum exceeds the random number is selected. This process introduces a probabilistic aspect into the shift selection, while maintaining that shifts resulting in a lower penalty increase are more likely to be chosen.

Algorithm 2 RuinRecreateEC(n, p, m, k)

```
1:  function RuinRecreateEC(n - ejection chain length, k - maximum number of unassignments in
    each step, p - maximum allowed increase of penalty from an unassignment, m - maximum allowed
    increase of penalty of ejection chain)
2:      tabu_recently_unassigned ← new List()
3:      tabu_recently_assigned ← new List()
4:      for j = 0 to n − 1 do
5:          for i = 0 to k − 1 do
6:              shift ← get_shift_to_unassign_not_in_tabu(tabu_recently_assigned)
7:              if penalty_increase(unassigning(shift)) > p then
8:                  continue
9:              end if
10:             unassign(shift)
11:             tabu_recently_unassigned.add(shift)
12:             swaps_done.add(shift, unassign)
13:         end for
14:         while can_assign_new_shift() do
15:             shift ← get_shift_to_assign_not_in_tabu(tabu_recently_unassigned)
16:             assign(shift)
17:             tabu_recently_assigned.add(shift)
18:             swaps_done.add(shift, assign)
19:         end while
20:         if penaltyDifference > m then
21:             reverseEverything(swaps_done)
22:             return
23:         end if
24:     end for
25: end function
```

After the ruin operation is completed, the algorithm proceeds with the recreate operation. In this phase, shifts are greedily assigned. The method that is used to choose which shifts to assign, mirrors the method used for unassignment. The assignment decision is made using the same offset calculation and roulette wheel selection approach previously described, allowing for randomization in shift selection, while preferring shifts that lower the penalty more. This process is repeated until no more shifts can be assigned.

Similarly to the other two ejection chains, RuinRecreateEC is also susceptible to cycles. Avoiding cycles is particularly important here as this method requires more computation time than the other two ejection chains. To prevent cycles, a shift that is assigned during the recreate phase cannot be unassigned during any of the ruin operations of the same ejection chain. The same is true vice versa, a shift that is being unassigned during the ruin phase, cannot be reassigned during any of the recreate phases of the same ejection chain. This is enforced by the use of tabu lists, that keep track and restrict the assignment and

unassignments.

Like the previous ejection chains, the purpose of this ejection chain is to move the solution to a different region of the search space. Unlike the previous ejection chains, this chain does not aim to find an improved roster. This chain will be invoked less frequently as the overall strategy is to use the solver to thoroughly search the current local search space, and then invoke RuinRecreateEC to move to another region of the search space, which the solver can subsequently explore. The frequency of applying the ejection chain is a parameter and the final value will be determined experimentally.

5

Parameter Tuning

This section describes the parameter tuning conducted to find decent parameter configurations for the ejection chains. The chapter includes the parameter tuning set up with reasoning, as well as the results. First, we outline the experimental settings, including information about the processor used and the instances used in Section 5.1. Afterwards, Section 5.2 discusses the parameter tuning performed on the different ejection chains.

5.1. Experimental Settings

The ejection chains were tuned on instances from real nurse rostering problems at one Dutch hospital. We use instances from June 2024 from two different departments. The first is from the Gastroenterology department, while the second is from the Coronary Care Unit department. The characteristics and abbreviations of the instances are shown in Table 5.1, which also include additional instances from a second hospital used in Chapter 6. All tuning were run on an Intel(R) Xeon(R) Gold 6146 CPU @ 3.20 GHz processor, with 8 GB of RAM. The only exceptions are the parameter tuning of EmulateEC and RuinRecreateEC, which were conducted on a processor with similar specifications but 6 GB of RAM. The project was developed using .NET Framework 4.8. The tuning process involved comparing penalties after 10-minute runs.

5.2. Parameter Tuning

This section starts by describing the parameter tuning methodology in 5.2.1. The parameters tuning of InfeasibleEC and EmulateEC are described in 5.2.2. The tuning of parameters in RuinRecreateEC are discussed in 5.2.3.

5.2.1. Methodology

Finding a suitable value for a parameter involved running the solver for 10 runs of 10 minutes each for each parameter configurations, then selecting the parameter value that resulted in the lowest average penalty. Due to the large differences in average penalties between the two instances (VPA and CCU), it was essential to first normalize the penalties. This was done by computing the average penalty for each parameter value per instance, then dividing each average by the lowest average penalty found on that instance. Lastly, the normalized averages were averaged across both instances, and the parameter value resulting in the lowest average was selected for the next steps. This procedure is outlined in Algorithm 3.

All three ejection chains have multiple parameters, which require experimentations to determine well performing values. To ensure a fair evaluation of the ejection chains, parameter tuning was conducted on the instances from the first hospital (VPA and CCU), while the experiments in Chapter 6, also contain instances from the second hospital. This separation allows for a more robust assessment of the algorithms' performance.

Table 5.1: This table shows the characteristics of the different instances. It is based on a table made by Faasse in his thesis [9]. The notations: numbers of required shifts (S), priority shifts (PS), preferred number of shifts (PrS), night shifts (NS), shift types (ST), employees (E), average maximum working hours per employee (MH), average required working hours per employee (RH), requests (R, sum of the following four request types), day-on requests (DOnR), day-off requests (DOffR), shift-on requests (SOnR) and shift-off requests (SOffR). Note that the required shifts include the priority and night shifts, and that the maximum and required working hours are counted for the whole month.

Dept.	Month	Abbr.	S	PS	PrS	NS	ST	E	MH	RH	R	DOnR	DOffR	SOnR	SOffR
First hospital															
Gastroenterology	June	VPA	330	0	390	60	3	24	128.5	115.6	105	0	96	9	0
Coronary Care Unit	June	CCU	624	0	624	166	8	50	140.9	104.4	383	340	0	43	0
Second hospital															
Obstetrics	April	O4	951	80	951	242	15	85	88.0	86.7	186	76	88	22	0
	May	O5	992	86	992	252	15	85	92.1	93.7	104	86	18	0	0
	June	O6	956	92	956	240	15	84	90.4	91.4	98	85	13	0	0
Trauma	April	T4	429	36	429	60	10	38	114.0	90.8	108	27	62	6	13
	May	T5	447	40	447	62	10	37	120.7	97.2	154	41	90	10	13
	June	T6	438	44	438	60	10	36	116.2	97.7	65	11	40	0	14
Vascular Surgery	April	V4	441	33	441	60	10	46	121.6	77.5	56	0	51	1	4
	June	V6	563	47	563	60	12	45	124.1	100.9	24	0	24	0	0

Evaluating all the possible combinations of parameter values requires too much computation time, making exhaustive grid search infeasible. Instead we opted for a sequential greedy tuning strategy, which substantially reduces the number of configurations by optimizing one parameter at a time while keeping the others fixed. Although this approach may fail to capture complex interactions between parameters, the long runtime needed to evaluate each configuration, made it the only practical method to identify high-performing configurations within a reasonable time frame. Furthermore, the primary aim of this thesis is not to find the optimal parameters, but to identify reasonable configurations that can be used to evaluate the impact of the ejection chains on the solver performance.

Tuning was performed independently for all four InfeasibleEC search methods. This approach was chosen because the four search methods operate in fundamentally different ways, and as a result, each algorithm is likely to perform best with its own specific parameter configurations. This results in tuning six different algorithms (three ejection chain types with four search methods for one of them), each having a large number of potential parameter values.

This would require an infeasible amount of time to properly assess. This was solved by selecting the values tested during the tuning process. For example, when tuning parameters other than maximum ejection chain length, a baseline value of 10 was used for the maximum chain length. To tune this parameter, one value above and one below the baseline were tested. Based on the best performing value, additional values were subsequently selected for further evaluation. Moreover, each chosen parameter value can substantially influence the tuning of subsequent parameters, so it is important to maintain reasonable baseline values throughout the process. Further details on the parameters, their baseline values, and the rationale behind the tuning order are provided in Sections 5.2.2 and 5.2.3.

Algorithm 3 Parameter Evaluation

Input: Set of parameter values $P = \{p_1, p_2, \ldots, p_n\}$, dataset instances $D = \{d_1, d_2\}$, number of runs $r = 10$
for each parameter value $p_i \in P$ **do**
  for each dataset instance $d_j \in D$ **do**
    Run model with p_i on d_j for r runs
    Compute average score $\text{avg}_{i,j}$
  end for
end for
for each dataset instance $d_j \in D$ **do**
  Find $\min_j = \min_i(\text{avg}_{i,j})$
  for each parameter value $p_i \in P$ **do**
    Compute normalized score: $n_{i,j} = \text{avg}_{i,j}/\min_j$
  end for
end for
for each parameter value $p_i \in P$ **do**
  Compute final score: $s_i = \frac{1}{|D|} \sum_{j=1}^{|D|} n_{i,j}$
end for
Output: Best parameter $p^* = p_{\arg_i \min s_i}$

5.2.2. Parameters for InfeasibleEC and EmulateEC

This section discusses the various parameters for both InfeasibleEC and EmulateEC. Due to the large number of parameters, and the presence of six chains, only a subset of the tuning results will be shown here. Specifically, this section displays the penalty results for one search methods for InfeasibleEC (DFSN). We chose to display DFSN, as its final parameter configuration caused the chain to execute more often, making the relationship between parameter values and average penalty clearer. The graphs corresponding to the other search methods of InfeasibleEC and EmulateEC are in Appendix A.

Roster penalty to compare with:
The first parameter tuned is which roster penalty is used for comparison with the current roster penalty throughout the ejection chain. As described in Section 4.1, the criteria for executing InfeasibleEC is that the current roster must have a lower penalty than the pre-ejection chain roster. Another viable option is to compare against the penalty of the best roster found so far (instead of the pre-ejection chain roster). This parameter is tuned first, as it has the highest impact on the ejection chain behaviour. There is a substantial difference in the number of ejection chains attempted depending on the chosen reference. For example, in the VPA instance, on average $> 57\,000$ InfeasibleEC with DFSN chains were attempted if the comparison is made to the pre-ejection chain roster, whereas on average < 60 were attempted if compared to the best roster found so far. Furthermore, there is support for both options in literature, as the ejection chain by Kingston in [12] compares to the pre-ejection chain roster, while the ejection chain by Curtois in [8], compared the best roster found so far.

Both options offer distinct advantages. Comparing to the best roster results in less time spent executing ejection chains, and when a successful chain is found, it guarantees the discovery of a new best solution. On the other hand, comparing to the pre-ejection chain roster leads to more frequent ejection chains, and potentially moving the current roster to a new unexplored search space regions.

The penalties achieved across the 10 runs for InfeasibleEC DFSN are presented in Figure 5.1. The figure indicates that the choice of comparison baseline does not have a substantial impact on the final roster penalty. However, the value of this parameter does have a large impact on the solver behaviour. This is evident in Tables 5.2 and 5.3, which show the average time spent on ejection chains over ten-10 minute runs, as well as the number of ejection chains attempted and their success rate. It is evident that a lot less time is spent on ejection chains when comparing to the best roster, instead of the pre-ejection chain roster. Additionally, in the CCU instance, the success rate of ejection chains increases when comparing with the best roster, while for VPA instance it decreases. Figure 5.1 reveals that using the pre-ejection chain penalty as the baseline resulted in a slightly lower average final penalty during the experiments.

Thus, this setting will be used for subsequent experiments.

Assessing the precise impact of parameter configurations on the different InfeasibleEC search methods proves challenging. This difficulty rises as configuration that perform best on one search methods, often yield worse results on other search methods. Similarly, these variations exist between different instances. This variability is also due to the inherent randomness in the solver, which leads to different penalty outcomes with the same parameter settings. For the choice between comparing the current roster to either the best found roster or to the pre-ejection chain roster, the effect on performance is variable across different search methods and instances. Notably, InfeasibleEC with BFSN is the only configuration that had the same parameter value achieve a lower average penalty across both VPA and CCU (it performed better on both when comparing to the best roster). This suggests the solver's performance is relatively insensitive to this parameter choice, moreover we hypothesise that this is because there is no large difference between the performance of the solver with and without the use of InfeasibleEC. This is especially evident in the CCU instance using DFS, when comparing to pre-ejection chain roster, the solver spends majority of its 600 seconds (> 540 seconds) on the ejection chain, whereas it only spends < 4 seconds when comparing to the best roster. Despite the large difference in computation time distribution, the final average penalties remain very close (215 712 and 216 148, respectively).

For EmulateEC, the solver achieves a lower average penalty when comparing to the best roster, rather than to the pre-ejection chain roster. This is particularly noticeable in the VPA instance, where substantially less time is allocated to ejection chains when comparing with the best roster. One possible explanation is that allocating excessive time on EmulateEC is worse than spending it on the other parts of the solver.

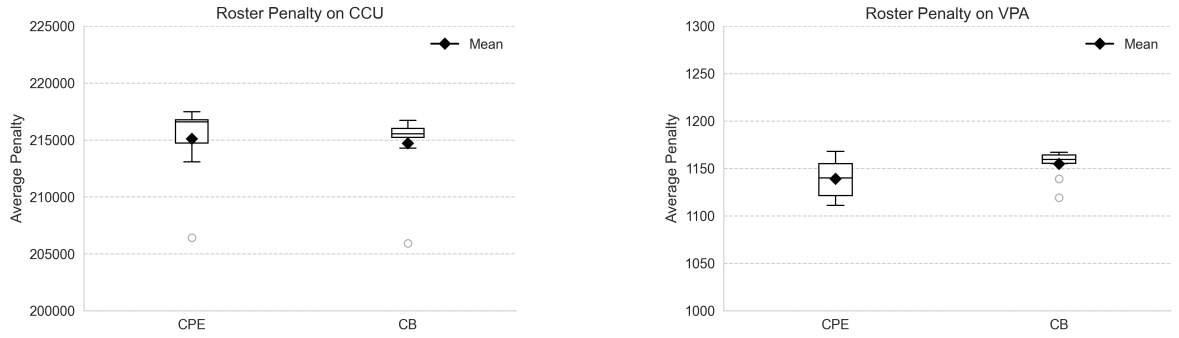

Figure 5.1: Parameter tuning InfeasibleEC with DFSN, on which roster to compare with. CPE indicates the runs where the current roster was compared to pre-ejection chain roster. CB indicates the runs where the current roster was compared to best roster found so far.

Table 5.2: Ejection chain performance metrics for the CCU instance for different type of Rosters being compared to averages of ten runs. CPE indicates the runs where the current roster was compared to pre-ejection chain roster. CB indicates the runs where the current roster was compared to best roster found so far.

Ejection Chain Type	Roster Compared	Number of ECs	Number of Successful ECs	Success Rate (%)	Time Spent on Successful Chains (s)	Time Spent on Unsuccessful Chains (s)
InfeasibleEC - DFS No Backtrack	CPE	2375.4	701.3	29.5	11.4	31.0
	CB	1070.8	448.1	41.8	5.6	7.6
InfeasibleEC - BFS No Backtrack	CPE	2178.2	782.7	35.9	39.0	37.9
	CB	941.9	501.6	53.3	25.5	7.1
InfeasibleEC - DFS	CPE	1841.8	644.7	35.0	64.9	33.6
	CB	719	370.6	51.5	38.6	6.3
InfeasibleEC - DFS Until Feasibility	CPE	2156.8	814.0	37.7	18.4	39.6
	CB	940.0	506.3	53.9	8.8	7.8
EmulateEC	CPE	3559.7	390.5	11.0	38.0	261.3
	CB	1738.5	256.3	14.7	24.7	121.2

Table 5.3: Ejection chain performance metrics for the VPA instance for different type of Rosters being compared with. CPE indicates the runs where the current roster was compared to pre-ejection chain roster. CB indicates the runs where the current roster was compared to best roster found so far.

Ejection Chain Type	Roster Compared	Number of ECs	Number of Successful ECs	Success Rate (%)	Time Spent on Successful Chains (s)	Time Spent on Unsuccessful Chains (s)
InfeasibleEC - DFS No Backtrack	CPE	57869.3	43963.7	76.0	111.3	47.3
	CB	69.3	49.8	71.9	0.25	0.1
InfeasibleEC - BFS No Backtrack	CPE	37705.0	32425.9	86.0	321.4	17.9
	CB	64.3	49.5	77.0	1.0	0.1
InfeasibleEC - DFS	CPE	6607.6	5701.9	86.3	536.6	4.3
	CB	42.4	32.3	76.2	3.2	0.1
InfeasibleEC - DFS Until Feasibility	CPE	54312.4	47261.5	87.0	170.7	24.9
	CB	69.4	54.1	78.0	0.3	0.2
EmulateEC	CPE	9129.9	3592.1	39.3	218.2	184.0
	CB	19.9	7.7	38.7	0.5	0.5

Type of tabu list:

The next parameter tuned is the type of tabu list used. The first option is a tabu list of rosters, while the second is a tabu list of swaps. Both types were described in Section 4.1.2. As explained previously, improper handling of cycles can cause the ejection chain to have a negative impact on the performance of the solver. Thus, the method used to avoid cycles is an important parameter to tune, and therefore it is the next parameter tuned. The baseline value is a tabu list of rosters, as this approach completely eliminates cycles within an ejection chain. On the other hand, the tabu list of swaps may offer potential improvements by being faster and using less memory, while permitting some cycles.

The results for tuning the type of tabu list on InfeasibleEC DFSN are shown in Figure 5.2. There is a slight advantage to using a tabu list of rosters rather than a tabu list of swaps. Although maintaining a roster-based tabu list requires more computation time, it rejects swaps more accurately. This difference is illustrated in Table 5.4, which compares the number of swaps rejected when using a tabu list of swaps compared to a tabu list of rosters. Tabu list of swaps rejects more swaps, compared to a tabu list of rosters. This indicates that many of the swaps rejected by a tabu list of swaps are rejected incorrectly. This happens more frequently than the tabu list of swaps not rejecting a swap that should be.

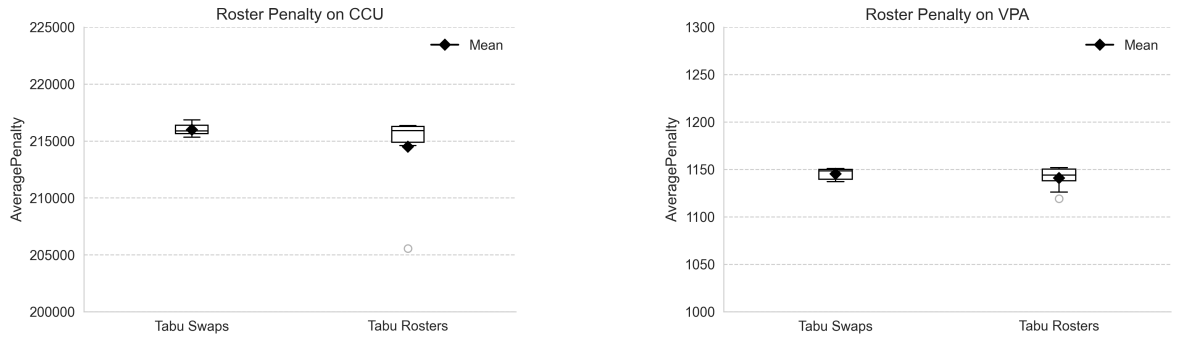

Figure 5.2: Parameter tuning InfeasibleEC with DFSN on which tabu list to use.

One type of tabu list consistently outperforms the other on both instances, across all four search methods. This effect appears connected to the choice of roster being compared to. InfeasibleEC with DFSN accomplished a lower average penalty when using a roster based tabu list in combination with comparison to the pre-ejection chain roster. On the other hand, the other three search methods performed better when comparing to the best roster and using a swap based tabu list. A plausible explanation for this pattern, is that comparing to pre-ejection chain roster typically involved running more ejection chains and using a tabu list of swaps restrict potentially beneficial moves, and thus when

Table 5.4: The average number of swaps rejected when tuning the type of tabu list parameter on both the VPA and CCU instances.

Instance	Type of Tabu List	InfeasibileEC - DFS No Backtrack	InfeasibileEC - BFS No Backtrack	InfeasibileEC - DFS	InfeasibileEC - DFS Until Feasibility	EmulateEC
VPA	Swaps	12270	4650	370.3	21193	0
	Rosters	9207	3989	96.8	12353	0
CCU	Swaps	590	425	1104	1094	0
	Rosters	469	380	683	705	0

comparing to pre-ejection chain roster, this negative impact occurs more frequently. For EmulateEC, no meaningful difference is observed between swap based and roster based tabu lists, as neither tabu list type filers any roster.

Order of employees and days:

The logical next parameter to tune is the order in which other employees and days are considered when generating a repair, either in a random sequence or in a predetermined manner. This parameter is relevant because it is also related to cycle reduction within chains. The baseline value is to randomize the order, as this can help with decreasing the frequency of cycles. The other option for this parameter is to sort the days and employees according to an estimate of which employees and days are more likely to be part of a repair that yields a feasible roster. For these estimates, we compute the ratio of successful to unsuccessful ejection chains that include a repair involving the specific day or employee throughout the run. The more frequently an employee or day participates in a successful chain, the earlier they are considered in subsequent chains. It is important to note that this parameter will not be tuned for EmulateEC and InfeasibleEC BFSN, as both systematically visit all possible neighbours of the current roster, making the visiting order irrelevant.

The results for the different type of order of employees on InfeasibleEC DFSN is shown in Figure 5.3. Using a random order provides a slight advantage over a dynamic order. Moreover, Table 5.5 shows that using a random order has a higher success rate than dynamic order. As the idea behind dynamic ordering is to increase the success rate, together with the fact that the penalty has decreased, it was decided to use random order instead. As a result, we decided to skip the experiments with dynamic order of days.

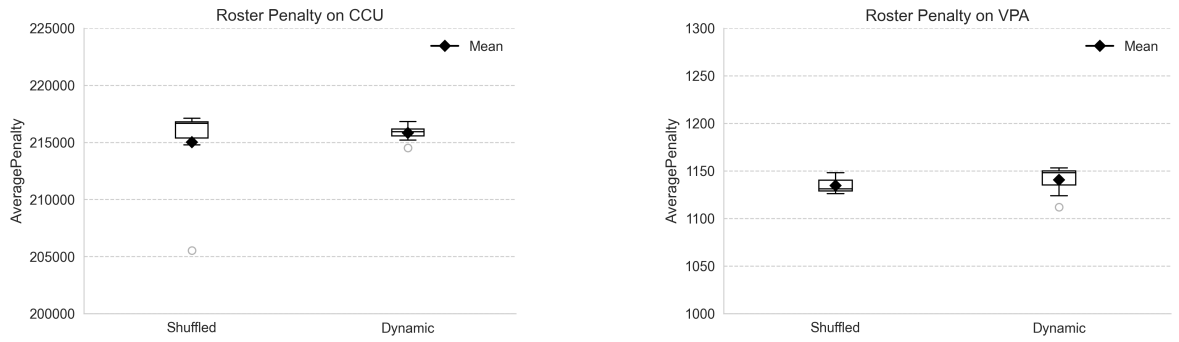

Figure 5.3: Parameter tuning InfeasibleEC with DFSN on how to order employees.

Table 5.5: The average success rate of ten 10-minute runs of ejection chains when using shuffled or dynamic ordering.

Instance	Order Type Employees	InfeasibileEC - DFS No Backtrack	InfeasibileEC - DFS	InfeasibileEC - DFS Until Feasibility
VPA	Shuffled	76.1	70.7	86.9
	Dynamic	72.2	69.5	87.0
CCU	Shuffled	29.4	49.1	37.3
	Dynamic	27.5	48.6	35.5

Maximum depth and maximum time:

The next two parameters are related and have a large impact on the overall performance of the ejection chains. The first is the maximum depth of the chain, and the second is the maximum time allowed before terminating a chain. These parameters are tuned last, as I gained some intuition for reasonable values during the construction of the ejection chains by experimenting with different settings. The chosen baseline values are a maximum chain depth of 10, and a maximum time limit per chain of 100 milliseconds.

The results for different maximum chain depths on InfeasibleEC DFSN are shown in Figure 5.4. The results indicate a relationship between the maximum chain depth and solution quality: as the depth increases, the average penalty decreases for both VPA and CCU instances. The final value selected for this parameter is 30, although the figure suggests that even greater depths might yield even further improvements. However, in practice, a chain of depth 30 is never reached, due to the nature of the problem and the maximum time allotted per chain. Thus, there is no benefit to increasing the maximum depth beyond 30.

A pattern is less evident for the other search methods. As hypothesised earlier, this difference may arise due to the difference in computation time dedicated to executing the ejection chains. Specifically, InfeasibleEC with DFSN executes substantially more ejection chains, potentially making the beneficial effect of increased depth clearer. On the other hand, the other search methods spend minimal time on ejection chains relative to the total running time, reducing the impact of ejection chains, and thus their parameters. For EmulateEC, there is no clear pattern between maximum chain depth and the final penalty, which aligns with the fact that EmulateEC consistently operates with a maximum depth of 3 in all iterations.

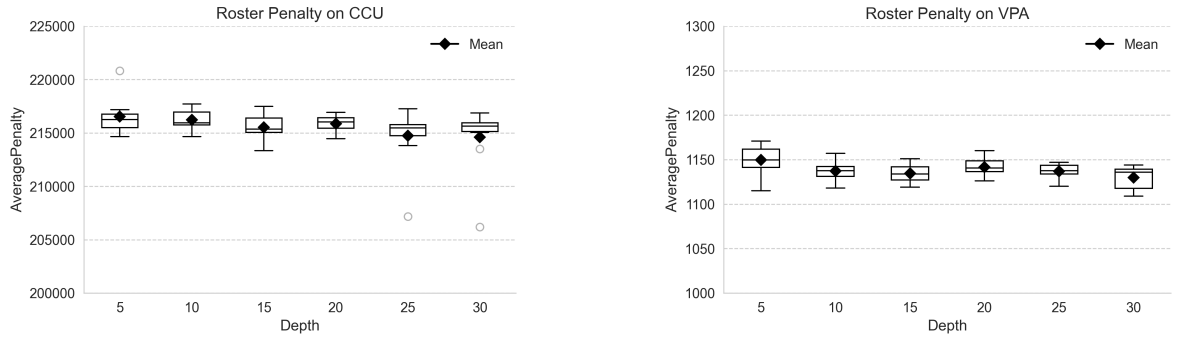

Figure 5.4: Parameter tuning InfeasibleEC with DFSN on the maximum allowed depth.

The penalties for different maximum allocated time for chains on InfeasibleEC DFSN are shown in Figure 5.5. Allowing longer run times per ejection chain increases their success rate, but also results in more time spent on the ejection chains rather than on the other parts of the solver. The effect on the penalties of InfeasibleEC are variable across different maximum times, instances and search methods.

Both maximum depth and maximum time influence when a chain is terminated, thus it is important to consider the impact they have on each other when tuning. When the optimal value for the maximum time exceeded the baseline value of 100 ms, the maximum depth parameter was re-tuned. This is

important because longer chains often terminated due to time constraints, therefore with a longer time limit, it was necessary to reevaluate whether allowing for even deeper chains improve performance further.

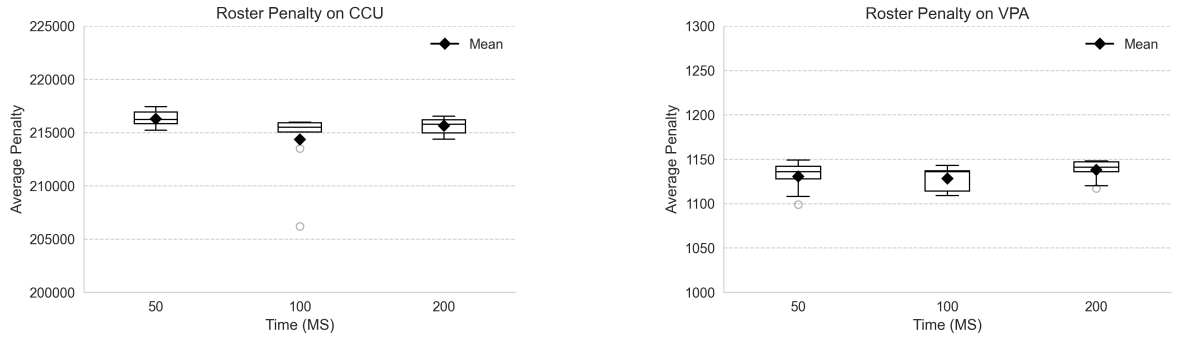

Figure 5.5: Parameter tuning InfeasibleEC with DFSN, on the maximum allowed time per ejection chain.

Filtering required days:

The parameters of EmulateEC are the same as InfeasibleEC, except EmulateEC has an additional parameter designed to reduce the number of swaps attempted in each step. The rationale behind this parameter is that the employee being targeted for an improvement in the current step, had their penalty worsened in the previous step. Meaning that the the new changes may be suboptimal. To leverage this information, a new parameter was introduced, which states that any swap must have at least one day in common with the previous swap. The effect of this filtering is shown in Figure 5.6. As illustrated, applying this filter negatively affects the final penalty, and thus it will not be used.

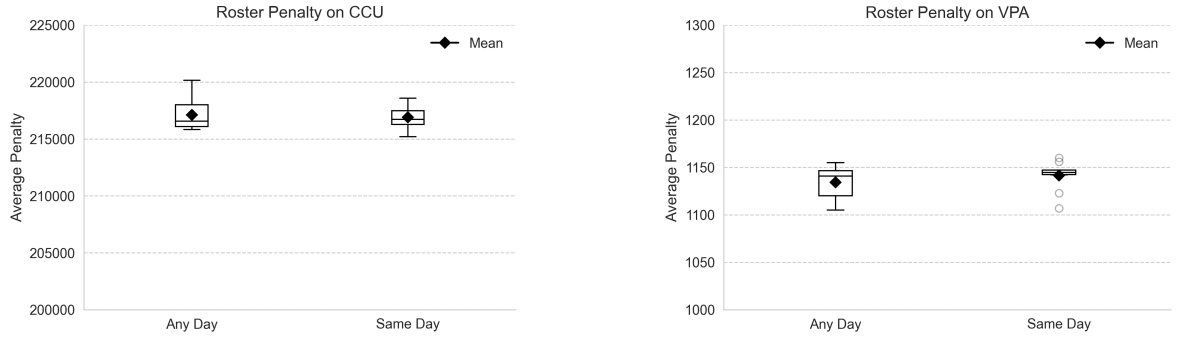

Figure 5.6: Parameter tuning EmulateEC on whether a swap in the ejection chain needs to use at least one day from the previous swap in the ejection chain. Same day indicates the runs where the swap needs at least one day in common with previous swap. Any day indicates the runs where the swaps are not filtered.

Table 5.6 summarises the final parameter values used for experiments for InfeasibleEC and EmulateEC , as well as the baseline values used for the tuning.

Table 5.6: The final parameter configuration for the InfeasibleEC chains and EmulateEC based on the parameter tuning experiments. The first row, the baseline, shows the values used as baseline for tuning the other parameters. CPE refers to comparing to pre-ejection chain roster. CB refers to comparing to best roster. TR refers to tabu lists of rosters. TS refers to tbu list of swaps. N/A refers to parameter that is not applicable to the specific ejection chain.

Ejection Chain Type	Roster Compared	Type of Tabu List	Employee Ordering	Day Ordering	Maximum Depth	Maximum Time Per Ejection Chain (ms)	Day Filtering
Baseline	N/A	TR	Shuffled	Shuffled	10	100	Not Used
InfeasibileEC - DFS No Backtrack	CPE	TR	Shuffled	Shuffled	30	100	N/A
InfeasibileEC - BFS No Backtrack	CB	TS	N/A	N/A	15	50	N/A
InfeasibileEC - DFS	CB	TS	Shuffled	Shuffled	10	200	N/A
InfeasibileEC - DFS Until Feasibility	CB	TS	Shuffled	Shuffled	10	75	N/A
EmulateEC	CB	TR	N/A	N/A	10	50	Not Used

5.2.3. Parameters for RuinRecreateEC

RuinRecreateEC has a different set of parameters compared to InfeasibleEC and EmulateEC. This section discusses the tuning of these parameters.

Maximum penalty increase allowed from an unassignment and from the full chain:

The first parameter that was tuned is the maximum penalty increase allowed from a single unassignment during the ruin operations. Setting this value low restricts the available moves, while setting it high risks substantially worsening the roster during the search. The second parameter tuned is the maximum overall increase in penalty permitted from a single run of the chain. While the maximum increase per shift parameter ensures that no single unassignment drastically worsens the penalty, a long chain of moves can still cumulatively lead to a much higher overall penalty. Therefore, it is also important to set a limit on how much the penalty can worsen during a single chain. These parameters correspond to p and m in Algorithm 2. They were tuned first, as they control the crucial decision of the penalty increase allowed from a single chain, and only after these values are set, tuning the next parameters become meaningful.

The impact of the maximum penalty increase from a single unassignment parameter can be seen in Figure 5.7, with a value of 100 yielding the best performance. The results suggest a slight parabolic impact, with the solver performing better on both VPA and CCU instances with this parameter being set to 100 and 200, compared to being set to 0 or 300.

The penalties for different maximum allowable increase in penalty per ejection chain are presented in Figure 5.8. The figure shows a clear pattern, where the lower values resulted in lower penalties. However, this is because the solver tends to perform better when not using RuinRecreateEC, and the lower the value, the more chains are being rejected. This can be seen when using the value of 0, which yielded the lowest average penalty, but none of the ejection chains were successful under this strict constraint. The problem is that with the current parameter values, the ejection chain has a worsening effect on the solver. To allow the ejection chain process to succeed in some runs, a more permissive value of 50 was selected. This compromise enables the chain to finish successfully while still controlling the potential increase in penalty.

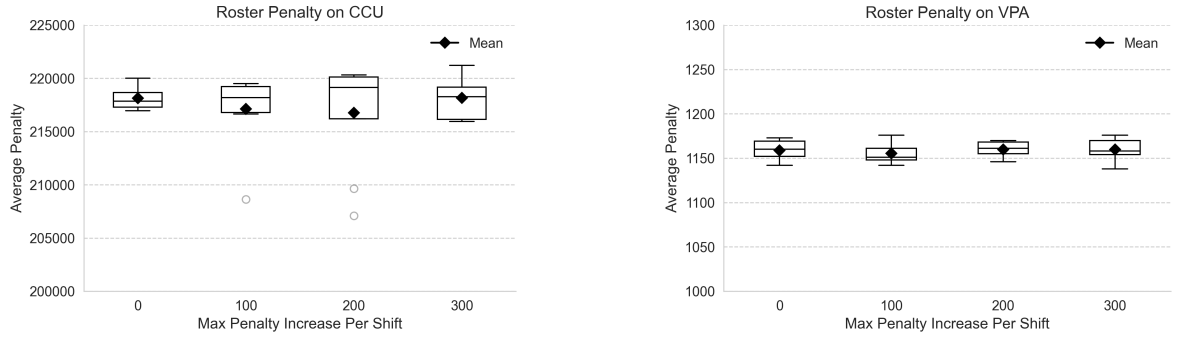

Figure 5.7: Parameter tuning RuinRecreateEC on the maximum penalty increase per shift unassignment during the chain.

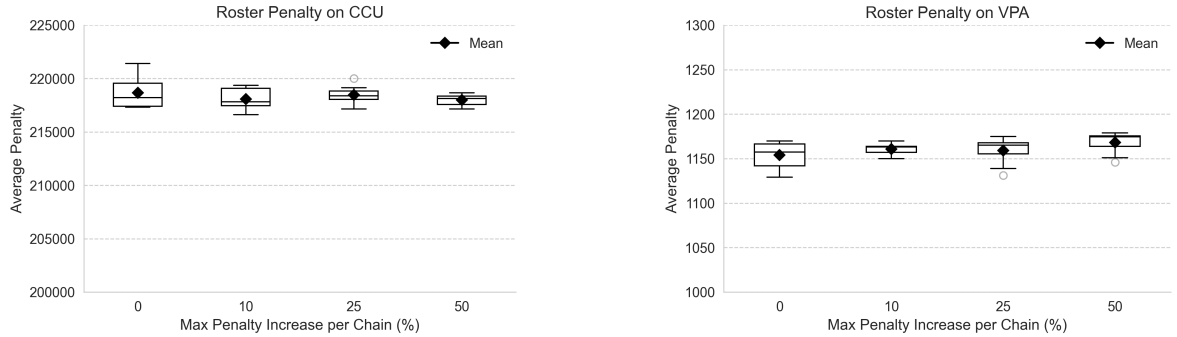

Figure 5.8: Parameter tuning RuinRecreateEC on the maximum penalty percentage increase of the roster compared to the pre-ejection chain roster.

Number of shifts unassigned and maximum depth:

The next parameter tuned is the number of shifts to unassign during the ruin operations, denoted as k in Algorithm 2. This is a key parameter, as the previous parameters ensured that the chain does not substantially worsen the roster, while k controls the extent of change introduced to the roster. A baseline value of 10 was chosen, as unassigning 10 shifts produces a large change without completely altering the roster. The following parameter tuned was the maximum depth of the ejection chain, denoted as n in Algorithm 2. Since both parameters are closely related they were tuned in sequence. As the goal is to sufficiently explore different regions of the search space without fully altering the current roster, a baseline depth value of 5 was selected.

While both parameters influence the scale of modifications, their effects are distinct, as reducing the number of unassigned shifts per step limits the possible reassignments available, which is not equivalent to simply increasing the chain depth. The impact of these parameters on the penalty is shown in Figures 5.9 and 5.10. The values that yielded the best results are the maximum chain length of 5, and a maximum of 5 shift unassignments per ruin step. Unsurprisingly, the maximum number of unassignments per move shows lower penalties when set lower. Once again, this is due to the solver performing better without the intervention of the chain. Surprisingly, this trend did not extend to the maximum chain depth, as the depth of 5 outperformed both higher and lower depths. This suggests that when using the parameter configuration found so far, the chain may have positive impact.

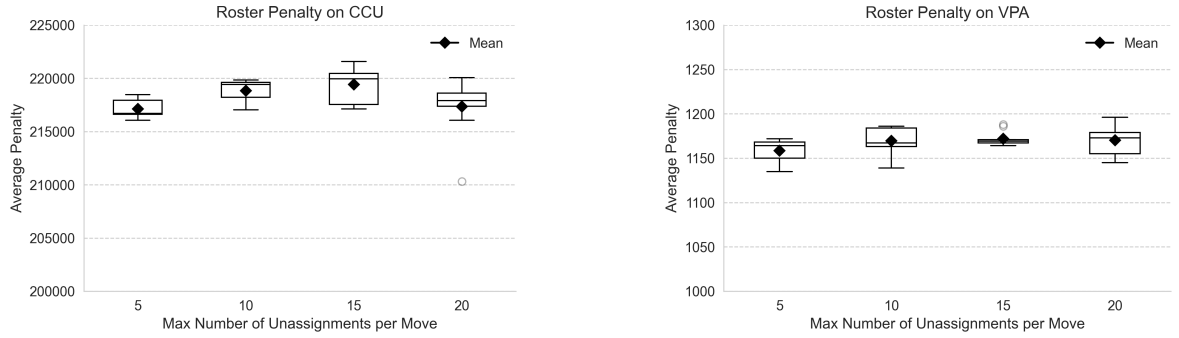

Figure 5.9: Parameter tuning RuinRecreateEC on the maximum number of shifts to unassign in each ruin step of the chain.

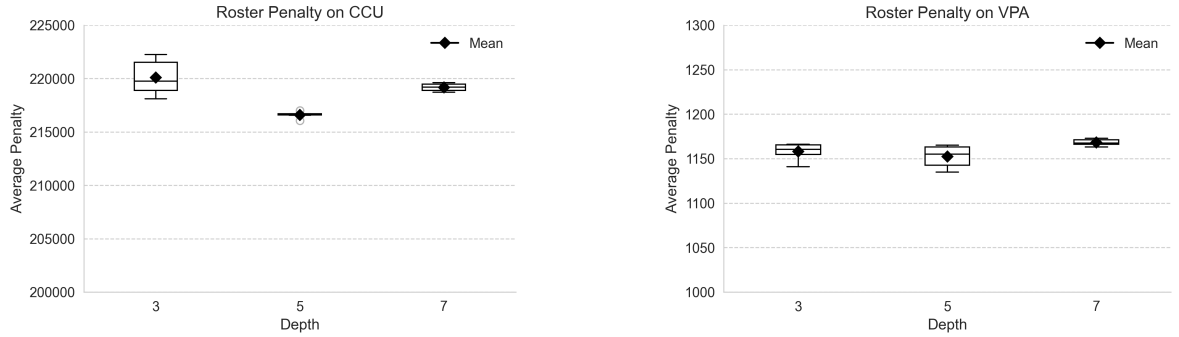

Figure 5.10: Parameter tuning RuinRecreateEC on the maximum allowed depth.

Frequency of executing chain:

The final parameter tuned is the number of local neighbourhood moves attempted between each iteration of the ejection chain. Unlike the other ejection chains, which can only be executed when the roster meets a specific criteria, RuinRecreateEC can be invoked at any point. This parameter decides when to execute the chain, greatly influencing the solver's performance. The baseline value chosen is to execute the chain every 500 000 steps, allowing the chain to be executed approximately 3-4 times during the 10 minute run time. This frequency should provide sufficient opportunities to search the current search space region before moving to a next region using the chain. The effect of varying this parameter on the penalty is illustrated in Figure 5.11. The best results were achieved by executing the ejection chain every 250 000 local search moves.

Tuning results for the frequency of running RuinRecreateEC further supports that the chain has a positive impact on average penalties with the current parameter configuration. As executing the chain every 250 000 steps results in a lower average penalty than every 500 000 steps.

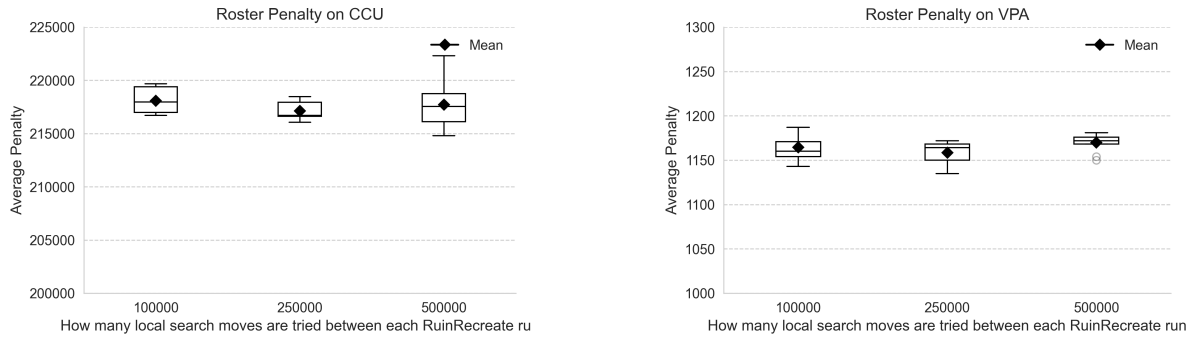

Figure 5.11: Parameter tuning RuinRecreateEC on the number of moves in between each chain execution.

Table 5.7 shows the baseline and final parameter values for RuinRecreateEC.

Table 5.7: The final parameter configuration for the RuinRecreateEC based on the parameter tuning experiments. The first row, the baseline, shows the values used as baseline for tuning the other parameters.

Ejection Chain Type	Max Penalty Increase per Unassignment	Max Penalty Increase per Chain (%)	Max Number of Unassignments per Ruin Step	Max Depth	Number of Steps Between Two Chain Executions
Baseline	N/A	20	10	5	250000
RuinRecreateEC	100	50	5	5	250000

6

Experiments and Results

This chapter describes the experiments conducted to answer the research questions. We start by explaining the experiment settings in Section 6.1. Section 6.2 discusses the final results of the ejection chains on the instances from the first hospital. Afterwards, Section 6.3 discusses the breakdown of the penalty structure from the various soft constraints. Section 6.4 summarises the impact on the final penalty for the instances from the second hospital. Section 6.5 shows how the penalty changes over time for the different chains. Afterwards, we show the finding of a significance test in Section 6.6. We address the research questions in Section 6.7. Lastly, the research limitations are described in Section 6.8.

6.1. Experimental Settings

The ejection chains are evaluated on instances from real nurse rostering problems at two different Dutch hospitals. For the first hospital, we have instances from June 2024 from two different departments. The first is from the Gastroenterology department, while the second is from the Coronary Care Unit department. For the second hospital, we used instances from April, May and June 2024 from three different departments. The characteristics and abbreviations of these instances are shown in Table 5.1. All experiments were conducted on an Intel(R) Xeon(R) Gold 6146 CPU @ 3.20 GHz processor with 8 GB of RAM. The project was developed using .NET Framework 4.8. Some experiments were performed on instances from the first hospital, while others were conducted on instances from the second hospital. Moreover, certain experiments ran for 10 minutes, while others were run for 30 minutes. Each of the following experiments specifies the settings it used.

6.2. Comparison

Settings: This comparison was conducted on instances from the first hospital using ten 10-minute runs and ten 30-minute runs.

The experiments evaluated the solver without the ejection chain, and with each of the 3 ejection chains (using four search methods for InfeasibleEC). The final penalties after the ten 10-minute runs for VPA and CCU are presented in Figures 6.1 and 6.2 respectively, while the corresponding results after 30 minutes are shown in Figures 6.3 and 6.4.

After 10 minutes, the average penalty on the VPA instance is lowest when using the InfeasibleEC with DFSN. Furthermore, InfeasibleEC that utilizes either DFS or DFSF also yields lower average penalties compared to not using any ejection chain. On the other hand, EmulateEC, RuinRecreateEC and InfeasibleEC with BFSN all result in higher average penalties than the configuration without an ejection chain.

The results for the CCU instance are comparable to those observed for the VPA instance, except for InfeasibleEC BFSN that performed much better on the CCU instance. As shown in Figure 6.2, the best performing algorithm is InfeasibleEC DFSF. For CCU, all four search methods for InfeasibleEC result in lower average penalties than the solver without an ejection chain, while both EmulateEC and

RuinRecreateEC have higher average penalties.

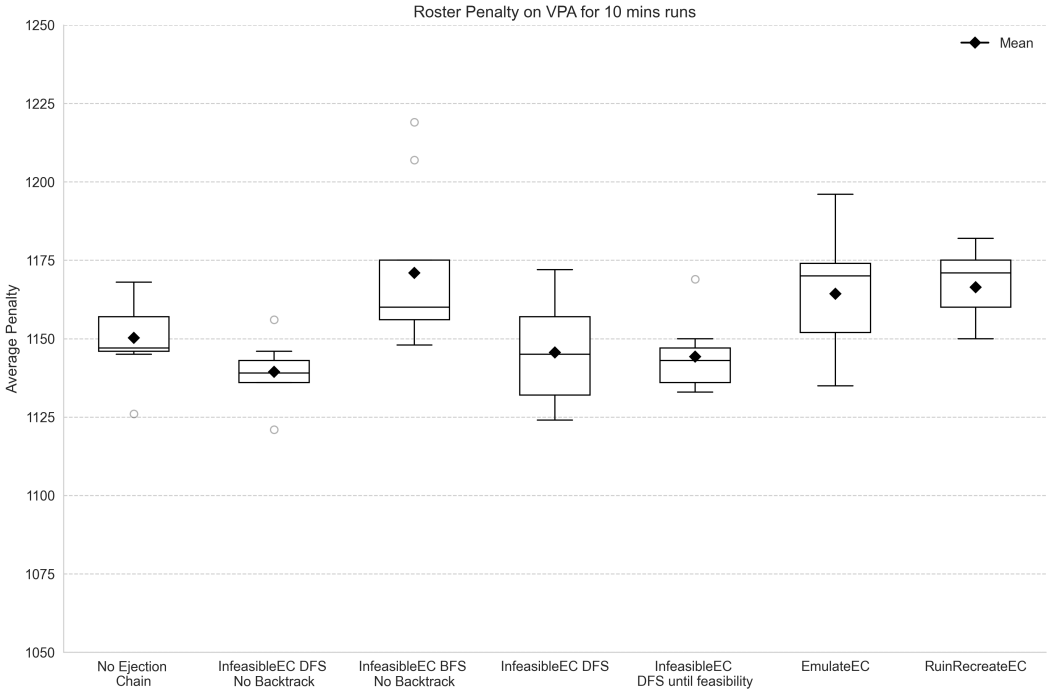

Figure 6.1: Running all ejection chains for 10 mins on the VPA instance.

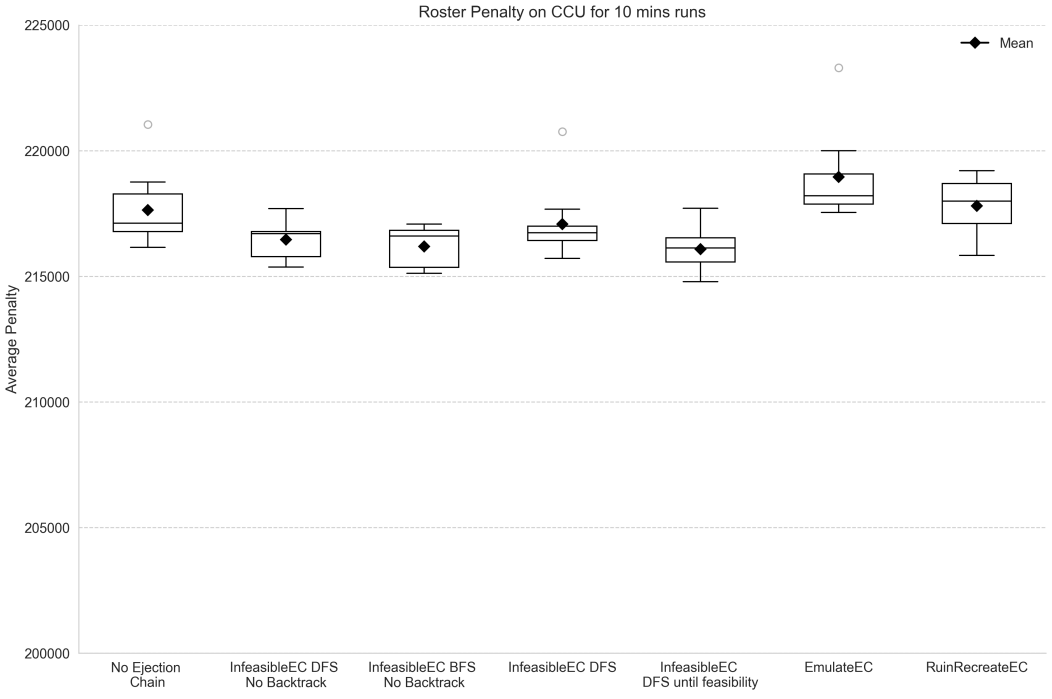

Figure 6.2: Running all ejection chains for 10 mins on the CCU instance.

In contrast, the results of the ten 30-minute runs for both VPA and CCU instances differ substantially compared to the 10-minute runs. InfeasibleEC with DFSN, which performed best after 10 minutes, produced the worst average penalty after 30 minutes. Its average penalty is > 850, while all other methods have achieved average penalties, between 805 and 815.

For the CCU instance, InfeasibleEC with DFSN once again has achieved the best performance, yielding an average penalty of 213 333. The other ejection chains produced similar average penalties, with the exception of RuinRecreateEC, which exhibited a substantially higher mean penalty. Notably, RuinRecreateEC was unable to generate rosters that satisfy all joker requests within the 30-minute time limit on two occasions. Consequently, its average is not shown in Figure 6.4, as it is a lot higher than 225 000. This outcome was unexpected, given that RuinRecreateEC consistently found rosters that adhered to the joker requests within the shorter, 10-minute runs.

A possible explanation is that RuinRecreateEC changes the roster more extensively compared to the other ejection chains. It is primarily used to diversify the search, but this comes at the expense of intensifying exploration within the current region of the search space. As a result, the algorithm may not sufficiently explore the current area to find a roster that satisfies all joker requests, since the search is frequently redirected to new regions.

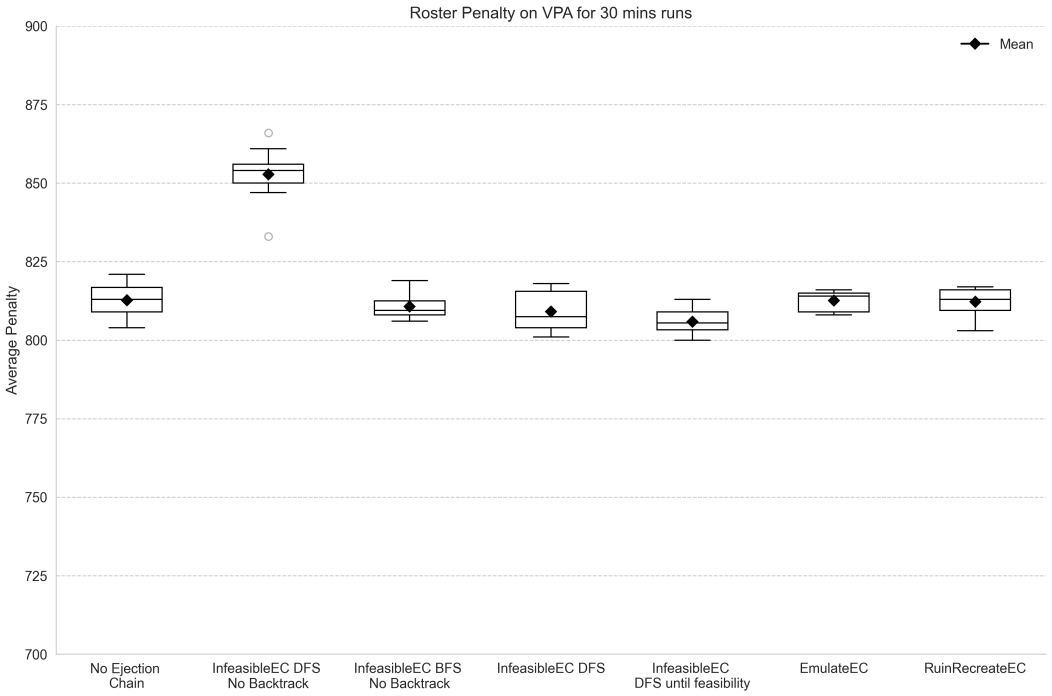

Figure 6.3: Running all ejection chains for 30 mins on the VPA instance.

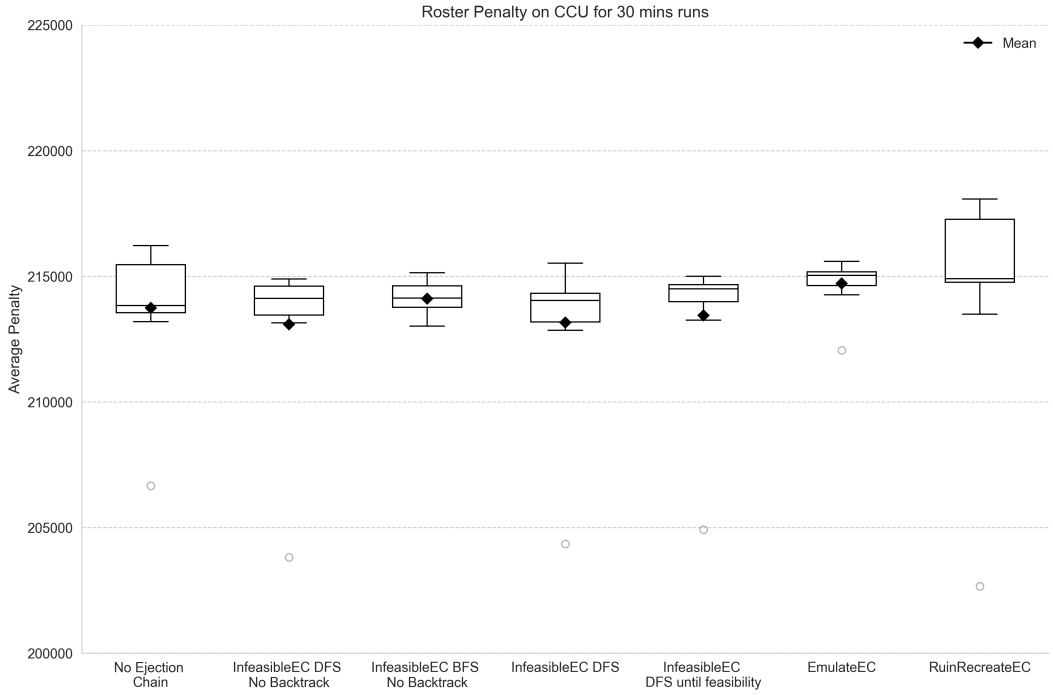

Figure 6.4: Running all ejection chains for 30 mins on the CCU instance.

6.3. Soft Constraints Analysis

Settings: This experiment was conducted on instances from the first hospital using ten 10-minute runs.

To answer RQ1, we gained deeper insights into the sources of the penalties, by analysing the impact of the different chains on the soft constraint violations. Tables 6.1 and 6.2 present the average breakdown of penalties for the various soft constraints after ten 10-minute runs on both the VPA and CCU instances, respectively.

For the VPA instance, nearly half of the total penalty arises from violations to the coverage requirements, with coverage penalties consistently ranging between 480 and 500 across all methods. Notably, the InfeasibleEC using DFSN achieved the lowest total penalty (1139.4) and produced the smallest overtime hours spread penalty (76.8). Although overtime hours spread penalties contributed less to the total penalty, it remained steady at around $75 - 100$ in all configurations. Importantly, penalties associated with coverage spread and employee requests are negligible (all zero) for all methods, indicating effective satisfaction of these soft constraints within the given computation time.

In contrast, the penalty breakdown for the CCU instance reveals a very different structure, with a much larger average penalty. Here, the penalty is dominated by violations of the preference of consecutive days off (with a minimum around 200 000 for all configurations) as well as by preference of consecutive shifts on (with a minimum around 12 000).

These penalties mainly arise from the same group of employees who were on extended leave. For example, one employee prefers 2 consecutive days off, but has also requested consecutive 23 days off (possibly for holiday) during the scheduling period. This results in a penalty of $(23 - 2)^2 \cdot 100 = 44\,100$. Although this penalty increase is not ideal, it is constant across all solvers and therefore does not affect their comparative evaluation.

The coverage penalty for the CCU instance is substantially lower, and consistently close to 0, indicating that nearly all required shifts are being covered. Similar to the VPA instance, the coverage spread and requests penalty are always negligible. However, the overtime hours spread is higher, generally ranging between 500 and 600 for most methods, except for RuinRecreateEC, which stands out with a penalty of 740.

Main Takeaways: No clear pattern has emerged linking the use of ejection chains to improved satisfaction of specific soft constraints.

Table 6.1: The division of the penalty for different soft constraints. Shows the average of ten 10-minute runs on the VPA instance.

Ejection Chain	Total Penalty	Coverage Penalty	Coverage Spread Penalty	Overtime Hours Spread Penalty	Requests Penalty	Consecutive Shifts on Preference Penalty	Consecutive Days off Preference Penalty
No Ejection Chain	1150.2	495.6	0	77.2	0	232.3	345.2
InfeasibileEC - DFS No Backtrack	**1139.4**	495.6	0	**76.8**	0	225.3	341.7
InfeasibileEC - BFS No Backtrack	1171.0	492.2	0	101.4	0	233	344.4
InfeasibileEC - DFS	1145.6	**483.3**	0	97.8	0	**224.7**	**339.8**
InfeasibileEC - DFS Until Feasibility	1144.2	494.4	0	83.6	0	225.6	340.7
EmulateEC	1164.3	496.7	0	81.3	0	238.1	348.2
RuinRecreateEC	1166.4	488.9	0	93.3	0	233.6	340.7

Table 6.2: The division of the penalty for different soft constraints. Shows the average of ten 10-minute runs on the CCU instance.

Ejection Chain	Total Penalty	Coverage Penalty	Coverage Spread Penalty	Overtime Hours Spread Penalty	Requests Penalty	Consecutive Shifts on Preference Penalty	Consecutive Days off Preference Penalty
No Ejection Chain	217212.0	50	0	**512.0**	0	14137.7	202512.0
InfeasibileEC - DFS No Backtrack	216301.7	12.5	0	526.8	0	13650.0	**202112.0**
InfeasibileEC - BFS No Backtrack	216073.3	**0**	0	535.8	0	**12662.5**	202875.0
InfeasibileEC - DFS	217252.8	**0**	0	602.8	0	13975.0	202675.0
InfeasibileEC - DFS Until Feasibility	**216026.5**	50	0	537	0	13150.0	202287.5
EmulateEC	219136.3	42.9	0	536.3	0	15585.7	202971.4
RuinRecreateEC	217682.0	14.3	0	740.0	0	14171.0	202757.0

6.4. Instances from the Second Hospital

Settings: This experiment was conducted on instances from the second hospital using ten 10-minute runs and ten 30-minute runs.

To address RQ2, we need to analyse the impact of the chains on additional instances, and examine its robustness. We evaluated the roster penalties generated by different ejection chains on eight additional instances from a second hospital. Tables 6.3 and 6.4 report the average final penalties after 10 and 30 minutes of solver runtime, respectively.

After 10 minutes, the solver without an ejection chain achieved the lowest penalty on 3 out of 8 instances, specifically O5, T4 and T5. No other configuration outperformed the others on more than 2 instances. The InfeasibleEC variant utilizing BFSN achieved the lowest penalty on 2 instances (V4 and V5). Similarly, RuinRecreateEC produced the lowest average penalties on 2 instances (O4 and O6). Lastly, InfeasibleEC using DFSN yielded the best result on instance T6.

The results after 30 minutes differ notably. Two solver configurations each achieved the lowest average penalty on 3 instances. InfeasibleEC with DFSN, obtained the lowest average penalties for T4, V4 and V6. Meanwhile, RuinRecreateEC produced the lowest average penalties for O5, O6 and T6. Additionally, EmulateEC had the lowest average penalty on O4, and InfeasibleEC with BFSN produced the lowest average penalty on T5. It is important to highlight that after 30 minutes, the solver without any ejection chain did not produce the best penalty on any of the eight instances.

Main Takeaways: No ejection chain consistently outperformed the solver without ejection chains across all instances, but the solver without ejection chains did not achieve the lowest average penalty in the 30-minute runs on any instance.

Table 6.3: Average penalty of ten 10-minute runs for the solver with different ejection chains on the instances from the second hospital

Ejection Chain	O4	O5	O6	T4	T5	T6	V4	V6
No Ejection Chain	143401.6	**122770.7**	108874.6	**158456.0**	**17489.9**	16975.2	38928.4	76477.4
InfeasibileEC - DFS No Backtrack	143530.8	123005.3	108430.6	158701.7	17502.2	**16823.5**	38763.3	76378.8
InfeasibileEC - BFS No Backtrack	143606.9	123177.8	108047.0	158593.2	17587.4	16986.9	**38536.0**	**76321.0**
InfeasibileEC - DFS	143601.9	123102.4	108406.0	158665.4	17501.0	16943.7	38941.7	76559.4
InfeasibileEC - DFS Until Feasibility	143540.0	123010.3	108142.3	158607.6	17529.2	16911.3	38850.3	76252.4
EmulateEC	143261.1	123152.0	108082.1	158698.4	17521.7	17069.1	39117.9	76551.9
RuinRecreateEC	**143141.3**	123065.6	**107963.9**	158588.5	17555.8	17069.1	38658.8	76543.6

Table 6.4: Average penalty of ten 30-minute runs for the solver with different ejection chains on the instances from the second hospital.

Ejection Chain	O4	O5	O6	T4	T5	T6	V4	V6
No Ejection Chain	139399.5	119934.3	104549.0	157916.6	17038.2	16711.1	37521.6	75699.3
InfeasibileEC - DFS No Backtrack	139889.1	119621.9	104871.1	**157911.4**	17043.5	16697.7	**37140.2**	**75403.4**
InfeasibileEC - BFS No Backtrack	139737.0	119671.4	104788.0	157930.5	**16980.6**	16689.9	37349.1	75551.7
InfeasibileEC - DFS	139653.6	119729.2	104890.5	157951.9	17057.0	16728.8	37527.1	75834.8
InfeasibileEC - DFS Until Feasibility	139619.8	119638.9	104826.9	158003.5	17063.1	16763.5	37640.2	75740.6
EmulateEC	**139277.6**	119677.5	104630.3	157914.5	17024.1	16664.0	37538.6	75647.3
RuinRecreateEC	139372.5	**119549.1**	**104412.1**	157940.9	17002.5	**16660.5**	37704.9	75667.7

6.5. Convergence Rate

Settings: This experiment was conducted on instances from the first hospital using ten 30-minute runs.

To answer RQ3, we need to examine the relationship between the penalty of the roster and the solver running time. We have previously looked at the penalties of the chains on the final penalties after 10, and after 30 minutes. But to answer the question, we need to look at the evolution of the roster's penalty throughout the runs, by keeping track of the roster's penalty whenever a swap was accepted. The

progression of the roster's penalty is illustrated for the VPA instance in Figures 6.5 and 6.6, and for the CCU instance in Figures 6.7 and 6.8. Figure 6.6 focuses on penalties for VPA after 100 seconds, while Figure 6.8 shows penalties on CCU starting from 600 seconds onward.

For the VPA instance, the solvers quickly find rosters that satisfy all joker requests (which have an extreme weight), leading to a rapid initial drop in penalty, with convergence occurring near zero. After this early phase, the performance differences between the various ejection chain configurations and the version without an ejection chain, become negligible. The one notable exception is for InfeasibleEC that uses DFSN. Initially, this configuration performs comparably, even slightly better than others, but after around 750 seconds, it begins to reduce the penalty at a slower rate than the other solvers.

In contrast, the change in penalty over time on the CCU instance is substantially different. All solver configurations take considerably longer to produce rosters that satisfy all joker requests. Furthermore, RuinRecreateEC failed to achieve such a roster within the 30-minute runtime in 2 different occasions, and thus RuinRecreateEC was excluded from the visualizations. As shown in Figure 6.8, the penalties among the different configurations remain distinguishable throughout the run. While all methods continue to improve gradually, the rate at which they reduce penalties is relatively consistent. Notably, InfeasibleEC with DFSN (the solver which achieved the worse on VPA) consistently outperforms the other solvers after 750 seconds. Meanwhile, EmulateEC performs the worst overall, maintaining the highest penalty throughout.

Main Takeaways: No ejection chain demonstrated the ability to find improved rosters faster. Furthermore, InfeasibleEC with DFSN appeared to slow down the process of finding better rosters.

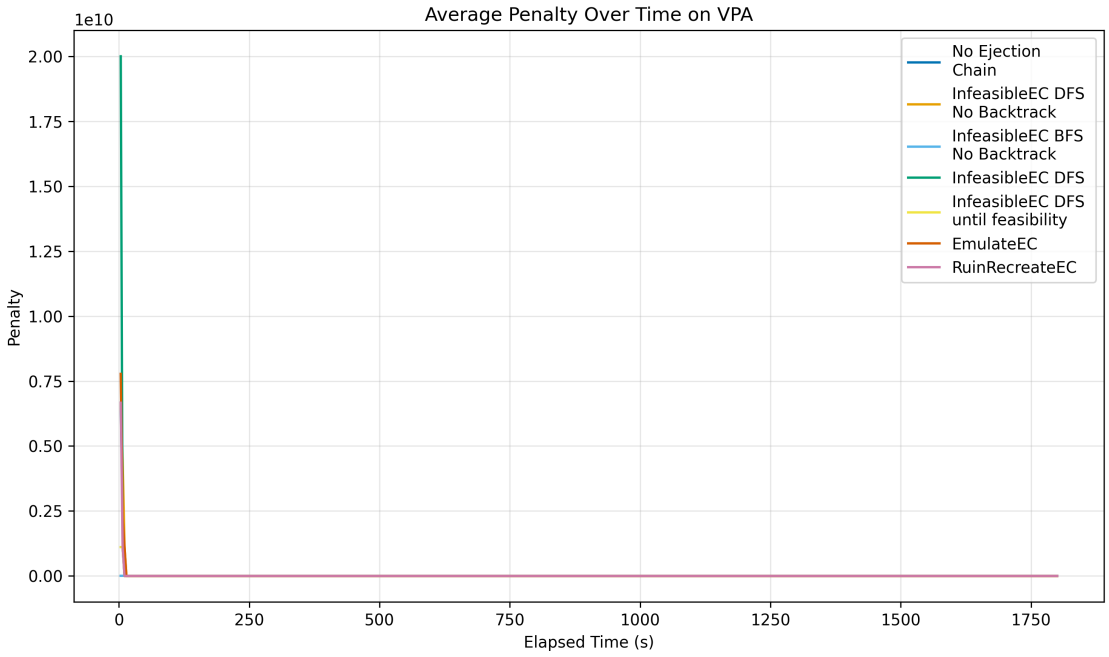

Figure 6.5: Average penalty of ten runs on the VPA instance throughout the running time.

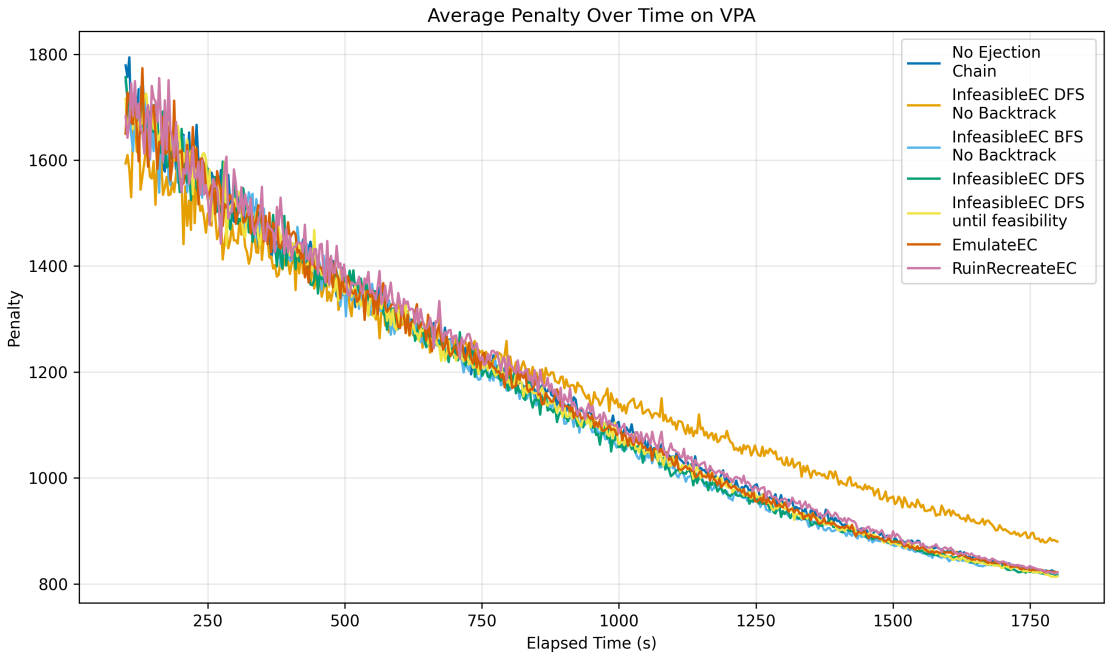

Figure 6.6: Average penalty of ten runs on VPA instance throughout the running time, showing from 100 seconds onwards.

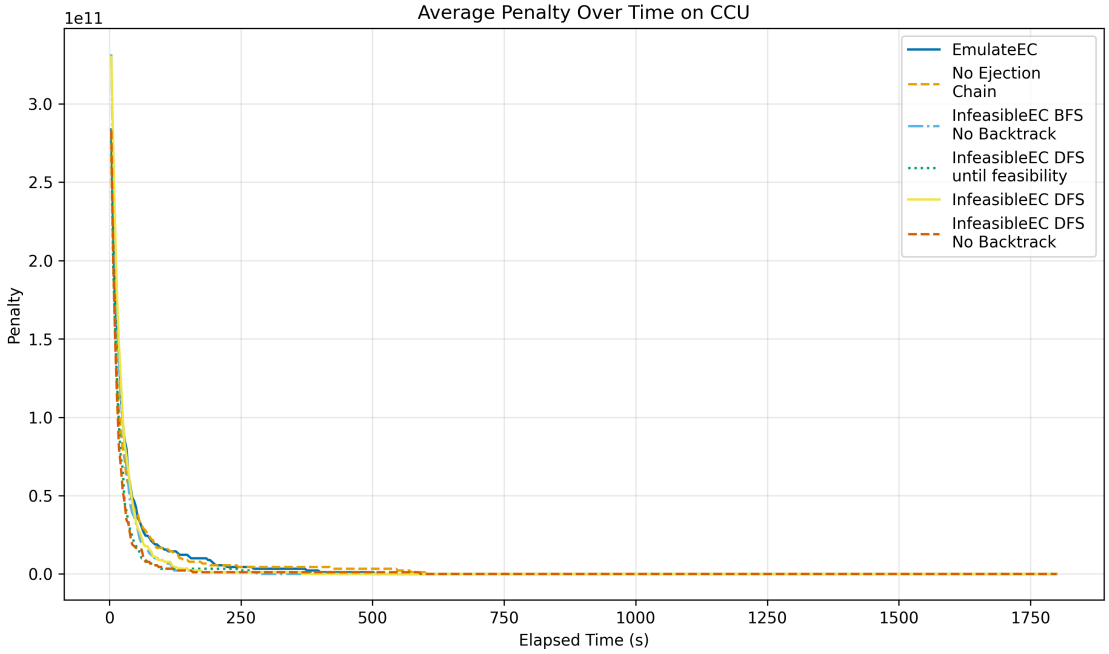

Figure 6.7: Average penalty of ten runs on the CCU instance throughout the running time.

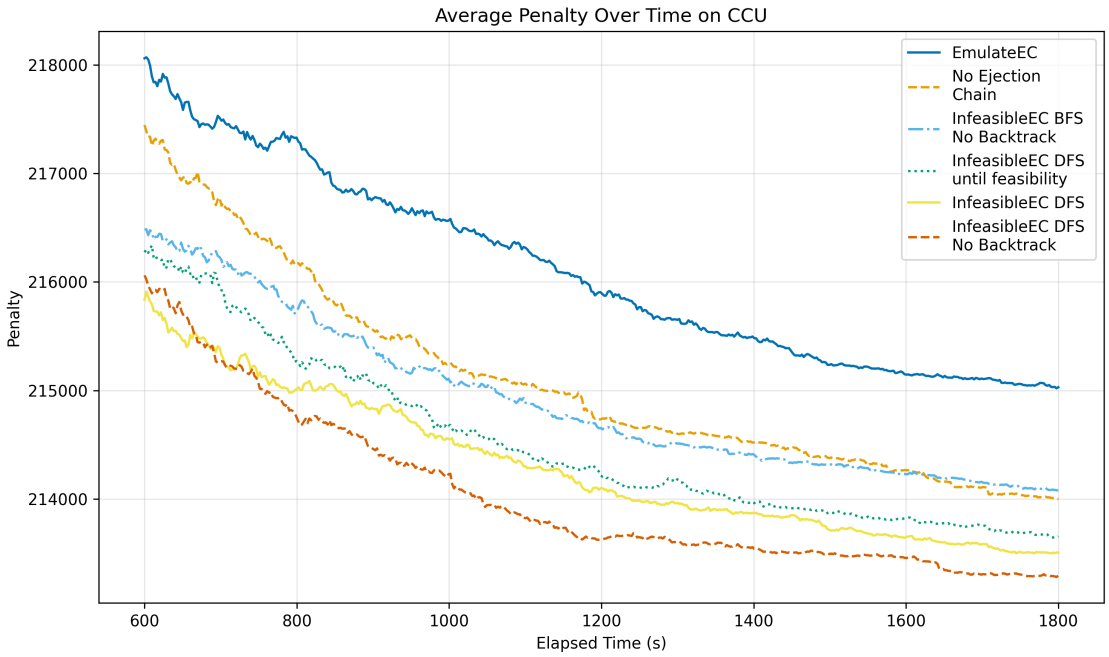

Figure 6.8: Average penalty of ten runs on the CCU instance throughout the running time, showing from 600 seconds onwards.

6.6. Significance Test

To assess the impact of adding ejection chains to the solver, we fitted a linear mixed-effects model to compare the average penalty obtained on the 8 instances from the second hospital. We have opted for linear mixed effect model, as it allows to account for all runs and instances while incorporating the random effect of instance variability.

The alpha level was set to $\alpha = 0.05$ We decided to test the significance only on the additional instances and on the 30 minute runs, since the tuning was not done on those instances. This approach ensures that we evaluate whether any of the ejection chains significantly improves performance on unseen instances. Furthermore, testing on 30 minute runs is a more realistic usage scenario.

The penalty is modelled as follows:

$$\text{penalty}_{ijk} = \beta_0 + \beta_i \cdot \text{EC}_i + b_j + \epsilon_{ijk} \tag{6.1}$$

Where i is the index of the ejection chain method used. j is the index of the instance. k is the index of the run. β_0 is the intercept. β_i is the estimated fixed effect of the solver with ejection chain i. $b_j \sim N(0, \sigma_b^2)$ is the random intercept capturing variation between instances. $\epsilon_{ijk} \sim N(0, \sigma^2)$ is the residual error.

The null Hypothesis is:

$$H_0 : \beta_i \geq 0 \quad \text{for all solvers } i$$

and the alternative hypothesis is:

$$H_1 : \exists i \quad \beta_i < 0$$

This tests whether any ejection chain results in a statistically significant reduction in the average penalty compared to the baseline solver.

The results of the test are found in Table 6.5. None of the ejection chains have a p-value below 0.05, therefore we cannot reject the null hypothesis. This means there is insufficient evidence to conclude that any of the ejection chains significantly reduce the average penalty compared to the baseline solver.

Table 6.5: Results of the significance test using a linear mixed effects model to assess whether any of the ejection chains lead to a significant improvement on instances from the second hospital on 30-minute runs.

Ejection Chain	p-value	Is it Significant?
InfeasibileEC - DFS No Backtrack	0.302	No
InfeasibileEC - BFS No Backtrack	0.423	No
InfeasibileEC - DFS	0.949	No
InfeasibileEC - DFS Until Feasibility	0.923	No
EmulateEC	0.142	No
RuinRecreateEC	0.106	No

6.7. Addressing the Research Questions

This section will answer the three sub research questions in Sections 6.7.1 to 6.7.3, as well as the main research question in Section 6.7.4.

6.7.1. Research Question 1

How do different types of ejection chains differ in their impact on solution quality for nurse rostering?
Considering a 10 minute run time on the VPA and CCU instances (the settings which the chains were tuned) the addition of some ejection chains resulted in a modest penalty reduction. Specifically, InfeasibleEC combined with DFSN and DFSF demonstrated substantial decrease in penalty, having a lower 75th percentile penalty than the 25th percentile of the baseline solver. Other ejection chains showed variable results across the two instances or average penalties similar to the baseline solver.

EmulateEC achieved a higher average penalty for both VPA and CCU, while RuinRecreateEC produced a slightly lower penalty on CCU but a higher penalty on VPA. Analysis of penalties across soft constraints revealed a similar breakdown across all solvers, with none demonstrating substantial improvement on any specific soft constraints. Notably, InfeasibleEC with DFS was the only algorithm to achieve the lowest penalty on a soft constraint for both VPA and CCU instances, namely the coverage constraint. However, even in this case, it was a minute reduction compared to the other solvers.

Answer: Overall, none of the ejection chains with their various search methods yielded substantial improvements in solution quality for nurse rostering.

6.7.2. Research Question 2

How robust is the impact of ejection chains across different nurse rostering instances?
The primary limitation of InfeasibleEC is its lack of generalizability, both across additional instances and even when extending the solver's runtime. This limitation becomes clear when examining the average penalties for additional instances at both the 10 and 30 minute runs: no chain substantially or consistently reduced penalties and no chain yielded a lower penalty than the baseline solver for all eight instances.

In contrast, both EmulateEC and RuinRecreateEC demonstrated a better capacity to generalise to new instances. For example, after 30 minutes, both solvers achieved lower average penalties than the baseline solver on 6 out of the 8 instances from the second hospital, although they both performed worse on CCU and VPA. However, a statistical significance test indicated insufficient evidence to confirm that any of the chains significantly reduced penalties across the instances.

Answer: InfeasibleEC lacks the ability to generalise to new instances, whereas both EmulateEC and RuinRecreateEC show potential to generalise to new unseen instances.

6.7.3. Research Question 3

How do ejection chains influence the progression of the solution quality as the solver runs?

For EmulateEC, RuinRecreateEC and InfeasibleEC, with the exception of DFSN, there is no substantial difference in how the ejection chains affect the speed of finding better rosters. On the VPA instance, differences in roster penalties over time across the different chains are difficult to distinguish. Similarly, on the CCU instance, all solvers display a very similar rate of finding better rosters.

0ne exception occurs when running the InfeasibleEC with DFSN on the VPA instance: improved roster are found slower. Since this configuration spends the most time executing InfeasibleECs, this suggests that spending excessive time on InfeasibleECs worsens the solver, perhaps because it reduces the time spent on other parts of the solver. While this configuration achieved the lowest penalty after 100 seconds, its speed of finding better rosters was slower for the remainder of the run, resulting in the worse average penalty after 1800 seconds.

Answer: InfeasibleEC with DFSN, appears to slow down the pace at which better rosters are found, whereas the other ejection chains did not demonstrate any improvement in solution progression.

6.7.4. Main Research Question

How do ejection chains impact the quality of rosters produced by a commercial nurse scheduling solver?

InfeasibleEC showed an ability to improve the quality of rosters when tuned on the instances and solver run time it was tested on. However, this approach is impractical in real world scenarios, where solvers need to generalise to unseen instances. Due to its limited generalisability, InfeasibleEC did not significantly reduces the penalties compared to the baseline solver across the new instances. Furthermore, when substantial time was dedicated to the chains, the results showed a risk of lowering the speed of finding better roster. Lastly, InfeasibleEC did not show substantial impact on penalties related to specific soft constraints.

EmulateEC failed to improve roster quality in the settings it was tuned on. On the other hand, it showed a better potential to generalise, as its performance, relative to the baseline, was better on the unseen instances than on the seen instances. However, its reduction of penalty was not statistically significant, nor did it substantially improve any specific soft constraint penalties.

RuinRecreateEC exhibited similar conclusions to EmulateEC . It did not show improvements under its tuning conditions, but demonstrated an ability to achieve better results on unseen instances. Like the others, RuinRecreateEC did not impact specific soft constraint penalties or accelerate the progression toward better rosters. A notable drawback was its occasional failure to satisfy all joker requests, which occurred twice after running the solver for 30 minutes.

6.8. Research Limitations

This study examined the impact of adding different ejection chains to a simulated annealing solver. The parameter configuration chosen for the study was based on limited experimentation, involving only two instances and testing only a small number of parameter values. Additionally, each configuration was evaluated using just 10 runs, reducing the confidence in the superior parameter value. Furthermore, the sequential greedy tuning strategy employed did not explore the relationship between different parameters, although the parameters are interconnected. All these limitations resulted in potentially not identifying optimal parameter configurations.

The results clearly illustrate considerable variability in solver performance from one run to the next. Even when using the same parameter configuration and instance, there was a large variance in the final penalty. Conducting more than 10 runs per configuration would have given a better understanding of the impact by reducing the influence of randomness. This could be further improved by testing on additional instances. The current study only examined 10 instances from only 5 departments and two hospitals. Given the notable differences between instances from different hospitals and departments, using more hospitals and departments would result in a more comprehensive evaluation of the effects of the ejection chains.

Although significance testing indicated no statistical significant reduction in penalty, small differences in penalties were observed. These small changes in penalties also made it difficult to tune correctly and find good ejection chain configurations. The differences could be attributed to randomness of the solver, or that some ejection chains are more effective when used on certain instances. This may depend on the

specific hard and soft constraints, and overall instance characteristics. Meaning there is a limitation regarding if ejection chain works better on some scenarios compared to others.

Another limitation is that the experiments were conducted only on 10- and 30-minute solver runtimes. Since the solver needs to run once a month for each department, allowing it to run for longer periods is reasonable. Therefore, experimenting with extended runtimes could yield valuable insights.

7

Conclusion

This chapter starts with a summary of the research in Section 7.1. Afterwards, Section 7.2 describes possible future research.

7.1. Summary

Nurse rostering is a NP-hard problem that directly influences staff satisfaction and as a result service quality [16]. Although prior research has demonstrated the potential of ejection chains, their effectiveness in real-world applications remains underexplored. It is important to investigate whether these benefits extend to practical improvements of a commercial nurse scheduling solver.

This study addresses the research question: *How do ejection chains impact the quality of rosters produced by a commercial nurse scheduling solver?*

To this end, we implemented and evaluated three different ejection chains: InfeasibleEC, EmulateEC, and a novel approach, RuinRecreateEC.

InfeasibleEC, introduced by Kingston [12], explores the search space that includes infeasible rosters by allowing temporary violations of hard constraints. It activates when a low-penalty roster contains a single hard constraint violation. It attempts to repair this violation through a sequence of local moves. Each move resolves the current violation but may introduce another. To effectively navigate the infeasible search space, we developed four search strategies: DFSN, BFSN, DFSF, and DFS. Our implementation differs from Kingston's original work by targeting a distinct nurse rostering problem formulation and extends it through the integration of multiple search strategies, and synthetic constraints.

EmulateEC, proposed by Cortuis et al. in [8], replicates human schedule planners' strategy. It improves the penalty associated with one nurse by performing a vertical swap that worsens another's penalty. This chain proceeds through a sequence of swaps where each swap improves the penalty of the nurse adversely affected by the previous swap, while potentially increasing the penalty of a different nurse. A swap is accepted only if the resulting roster penalty is lower than the best found so far, disregarding the penalty increase experienced by the nurse just affected. We extended EmulateEC by adapting it to a different problem formulation.

Our study advances both the works of Curtois et al. [22] and Kingston [12], by exploring the effects of various parameter settings on ejection chain functioning, analysing their influence on different soft constraint violations, and assessing their influence on finding better rosters faster.

Lastly, we proposed a novel ejection chain approach called RuinRecreateEC. This method applies multiple small ruin and recreate operations. Each ruin operation greedily unassigns five shifts, which is followed by the recreate operation which greedily assigns shifts. Both operations choose shifts based on penalty change. To the best of our knowledge, no previous research has explored chaining multiple smaller ruin and recreate steps for NRP in this manner.

We evaluated the impact of these chains when integrated to a simulated annealing solver. Initial experiments on tuning instances with 10- and 30-minute runtimes showed that some InfeasibleEC search strategies were capable of reducing average penalties. We analysed the penalty components corresponding to each soft constraint violation, but no clear pattern emerged. Moreover, there was no improvement observed in the speed of finding better rosters by adding ejection chains. Lastly, testing on additional instances demonstrated minor penalty reductions for certain chains, but these improvements were not statistically significant.

In conclusion, incorporating ejection chains into a commercial nurse scheduling solver did not significantly improve the quality of the rosters produced.

7.2. Future Work

This study examined three different ejection chains, however there are many additional approaches to further develop and analyse these chains. The following examples highlight potential areas of future research regarding these ejection chains:

- **Additional repair types for InfeasibleEC:** One area where this and previous research on InfeasibleEC is limited, concerns the generation of repairs. Conducting systematic experiments on repair generation could improve our understanding of their workings. This includes experimenting with different types of swaps for the repairs, rather than only vertical swaps, as well as using swaps involving longer block lengths instead of restricting swaps to block size one. Moreover, considering repairs that address combinations of two or more hard constraint violations could provide further insight.
- **Additional parameters:** There were several additional parameters for each ejection that could have been tuned, but were not due to time limitations. For InfeasibleEC, these include parameters such as the neighbourhood structures that can start a chain and a maximum number of repairs attempted on each infeasible roster before concluding the chain as unsuccessful. Potential parameters for EmulateEC include the maximum block lengths for swaps both at the start of the chain and during the chain. Similarly, for RuinRecreateEC, possible parameters for tuning include the sorting criteria used to select shifts, such as the criteria used by Faasse [9] for creating the initial roster.
- **Different ruin and recreate approaches:** There exist multiple different approaches to apply ruin and recreate, such as the approaches described in Section 3.2. For RuinRecreateEC, exploring these different approaches would be valuable. For instance, the ruin step can unassign shifts from a set of nurses or days, or based on the number of steps passed since that shift was assigned. For the recreate step, reassignment could be done though mathematical optimization or sequentially by considering them in an order found by Faasse [9]. Further research should also investigate the isolated impact of ruin and recreate operations when integrated into commercial nurse rostering solvers, when it is not part of an ejection chain.
- **Different criteria for executing RuinRecreateEC:** One limitation of RuinRecreateEC is not consistently satisfying all joker requests on the CCU instance during the 30-minute runs. One possible explanation is that the chain did not allow the solver to intensify sufficiently, as it was executed too frequently and diverted the search to different regions of the search space. Thus, direction for future research is to experiment with alternative criteria for executing the chain, aiming to better balance intensification and diversification. For example, the chain could be triggered only after a certain number of consecutive moves fail to improve the best roster, or only after all joker requests have been satisfied.
- **Combining the ejection chains:** An interesting avenue for future research is analyzing the impact of incorporating multiple types of ejection chains simultaneously into the solver. Since each of the three ejection chains executes under different circumstances, it is possible to combine two or even all three chains within the solver. Investigating these combinations reveal whether they can collectively improve solver performance.
- **More comprehensive tuning:** As discussed in Section 6.8, one constraint of this study was the limited parameter tuning. Due to time restrictions, a sequential greedy tuning algorithm was employed. Consequently, we have limited insight into the interactions between the different parameters, neglecting these interdependencies may have prevented us from identifying parameter

combinations that perform well together. Thus, it is beneficial to further research these interactions between the parameters.

- **Implementation efficiency:** Another limitation concerns the efficiency of the ejection chains implementation. Improving the implementation could provide a clearer assessment of their impact, as the observed lack of performance may be partly due to inefficiencies. This is particularly evident for RuinRecreateEC, where each chain execution requires substantial computation time. Finding faster methods to select shifts for assignment or unassignment, or improving the speed of calculating penalty changes, would meaningfully accelerate the chain.

Another promising area for further research is the relaxation of hard constraints. This technique allows the solver to treat hard constraints as soft constraints with large penalties, enabling the exploration of infeasible rosters, while still guiding the solver toward feasibile solutions. Previously, this approach was applied to the ruin and recreate operation in [17], but effectively adapting it for NRP solvers in real-world settings requires additional investigation.

Given that InfeasibleEC showed some improvements (on the instances and run time it was tuned on) but lacked generalizability, relaxing hard constraints might offer a more robust alternative. Both approaches facilitate exploration through infeasible rosters. However, unlike InfeasibleEC, which requires a repair generator for each hard constraint, hard constraint relaxation only requires an appropriate penalty for each violation. Consequently, this method may generalize better, as it is less dependent on the specifics of each individual constraint.

An additional technique that represents a worthwhile research topic is periodic restarts. Although, the lack of impact of the initial solution on the final solution found in the thesis by Faasse [9] poses challenges for restarts, clever strategies may still improve the solver's performance. For example, retaining information learned previously, such as estimating a fitness function that maps assignments to utility, could help guide the search towards high-quality search regions more quickly after restarts.

This work presents a detailed study on the use of ejection chains for solving a real-world nurse rostering problem (NRP). We adapted and optimised two ejection chains from literature and proposed a novel ejection chain approach. Our empirical evaluation on real data from two Dutch hospitals, indicated that adding ejection chains into the ORTEC solver improves roster quality, although we did not observe a statistically significant decrease in penalty. Given that InfeasibleEC showed promise in finding better rosters by exploring infeasible solutions but lacked generalisability, I recommend further research into hard constraint relaxation. This alternative also explores infeasible regions of the search space but is less dependent on specific instances and hard constraints.

References

[1] *Arbeidstijdenwet*. §5.2 [Accessed on June 3, 2024]. 2022.

[2] M. J. Bester, I. Nieuwoudt, and Jan H. Van Vuuren. "Finding good nurse duty schedules: a case study". In: *Journal of Scheduling* 10.6 (Oct. 2007), pp. 387–405. ISSN: 1099-1425. DOI: 10.1007/s10951-007-0035-7. URL: https://doi.org/10.1007/s10951-007-0035-7.

[3] Edmund K. Burke et al. "A hybrid heuristic ordering and variable neighbourhood search for the nurse rostering problem". In: *European Journal of Operational Research* 188.2 (2008), pp. 330–341. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2007.04.030. URL: https://www.sciencedirect.com/science/article/pii/S0377221707004390.

[4] Edmund K. Burke et al. "A Time Predefined Variable Depth Search for Nurse Rostering". In: *INFORMS Journal on Computing* 25.3 (2013), pp. 411–419. DOI: 10.1287/ijoc.1120.0510. eprint: https://doi.org/10.1287/ijoc.1120.0510. URL: https://doi.org/10.1287/ijoc.1120.0510.

[5] Sara Ceschia et al. "Second International Nurse Rostering Competition (INRC-II) - Problem Description and Rules -". In: *CoRR* abs/1501.04177 (2015). arXiv: 1501.04177. URL: http://arxiv.org/abs/1501.04177.

[6] Kathryn A. Dowsland. "Nurse scheduling with tabu search and strategic oscillation". In: *European Journal of Operational Research* 106.2 (1998), pp. 393–407. ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(97)00281-6. URL: https://www.sciencedirect.com/science/article/pii/S0377221797002816.

[7] Gielen Driven. *Werkdruk en arbeidstevredenheid in de zorg*. Tech. rep. The Hague, Netherlands: Centraal Bureau voor de Statistiek (CBS), 2022. URL: https://www.cbs.nl/nl-nl/longread/statistische-trends/2022/werkdruk-en-arbeidstevredenheid-in-de-zorg?onepage=true.

[8] Tim Curtois Edmund K. Burke. *An ejection chain method and a branch and price algorithm applied to the instances of the first international nurse rostering competition*. Accessed: 2024-11-27. 2010. URL: https://nrpcompetition.kuleuven-kulak.be/wp-content/uploads/2020/06/l2.pdf.

[9] Jonathan Faasse. "Comparing the performance of state-of-the-art algorithms for the nurse rostering problem". MA thesis. Delft University of Technology, 2024.

[10] Stefaan Haspeslagh et al. *First International Nurse Rostering Competition*. https://nrpcompetition.kuleuven-kulak.be/wp-content/uploads/2020/06/nrpcompetition_description.pdf. Accessed: 2025-06-03. 2010.

[11] Conny Helder. *Arbeidsmarktprognose zorg en welzijn 2023*. Official Open Letter. Available at: https://open.overheid.nl/documenten/bc2de994-6c54-4660-9d69-0ef33a8fba06/file. Dec. 2023.

[12] Jeffrey H Kingston. "KHE24: Towards a Practical Solver for Nurse Rostering". In: *Proceedings of the 14th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2024)*. 2024.

[13] Holmes Miller, William Pierskalla, and Gustave Rath. "Nurse Scheduling Using Mathematical Programming". In: *Operations Research* 24 (Oct. 1976), pp. 857–870. DOI: 10.1287/opre.24.5.857.

[14] Ministry of Social Affairs and Employment. *Arbeidsduur*. Accessed: 2025-05-20. 2021. URL: https://cao-ziekenhuizen.nl/cao/arbeidsduur-en-arbeids-en-rusttijden.

[15] Florian Mischek and Nysret Musliu. "Integer programming model extensions for a multi-stage nurse rostering problem". In: *Annals of Operations Research* 275.1 (2019), pp. 123–143. ISSN: 1572-9338. DOI: 10.1007/s10479-017-2623-z. URL: https://doi.org/10.1007/s10479-017-2623-z.

[16] Chong Man Ngoo et al. "A Survey of the Nurse Rostering Solution Methodologies: The State-of-the-Art and Emerging Trends". In: *IEEE Access* 10 (2022), pp. 56504–56524. DOI: 10.1109/ACCESS.2022.3177280.

[17] Erfan Rahimian, Kerem Akartunalı, and John Levine. "A hybrid Integer Programming and Variable Neighbourhood Search algorithm to solve Nurse Rostering Problems". In: *European Journal of Operational Research* 258.2 (2017), pp. 411–423. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2016.09.030. URL: https://www.sciencedirect.com/science/article/pii/S0377221716307822.

[18] Razamin Ramli et al. "A tabu search approach with embedded nurse preferences for solving nurse rostering problem". In: *International Journal for Simulation and Multidisciplinary Design Optimization* 11.10 (2020), pp. 1–10. DOI: 10.1051/smdo/2020010. URL: https://doi.org/10.1051/smdo/2020002.

[19] Cesar Rego, Tabitha James, and Fred Glover. "An ejection chain algorithm for the quadratic assignment problem". In: *Networks* 56 (Oct. 2010), pp. 188–206. DOI: 10.1002/net.20360.

[20] Kenneth N. Reid et al. "Shift Scheduling and Employee Rostering: An Evolutionary Ruin & Stochastic Recreate Solution". In: *2018 10th Computer Science and Electronic Engineering (CEEC)*. 2018, pp. 19–23. DOI: 10.1109/CEEC.2018.8674200.

[21] Martin Stølevik et al. "A Hybrid Approach for Solving Real-World Nurse Rostering Problems". In: *Principles and Practice of Constraint Programming – CP 2011*. Ed. by Jimmy Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 85–99. ISBN: 978-3-642-23786-7.

[22] Rong Qu Tim Curtois. *Computational results on new staff scheduling benchmark instances*. 2014. URL: https://www.schedulingbenchmarks.org/papers/computational_results_on_new_staff_scheduling_benchmark_instances.pdf.

[23] Aykut Melih Turhan and Bilge Bilgen. "A hybrid fix-and-optimize and simulated annealing approaches for nurse rostering problem". In: *Computers & Industrial Engineering* 145 (2020), p. 106531. ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2020.106531. URL: https://www.sciencedirect.com/science/article/pii/S0360835220302655.

[24] D. Michael Warner. "Scheduling Nursing Personnel according to Nursing Preference: A Mathematical Programming Approach". In: *Operations Research* 24.5 (1976), pp. 842–856. ISSN: 0030364X, 15265463. URL: http://www.jstor.org/stable/169810 (visited on 11/29/2024).

[25] Ziran Zheng, Xiyu Liu, and Xiaoju Gong. "A simple randomized variable neighbourhood search for nurse rostering". In: *Computers & Industrial Engineering* 110 (2017), pp. 165–174. ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2017.05.027. URL: https://www.sciencedirect.com/science/article/pii/S0360835217302310.

A

Additional Parameter Tuning Results

This appendix contains the results of the parameter tuning on EmulateEC and InfeasibleEC with the search methods of BFSN, DFSF and DFS.

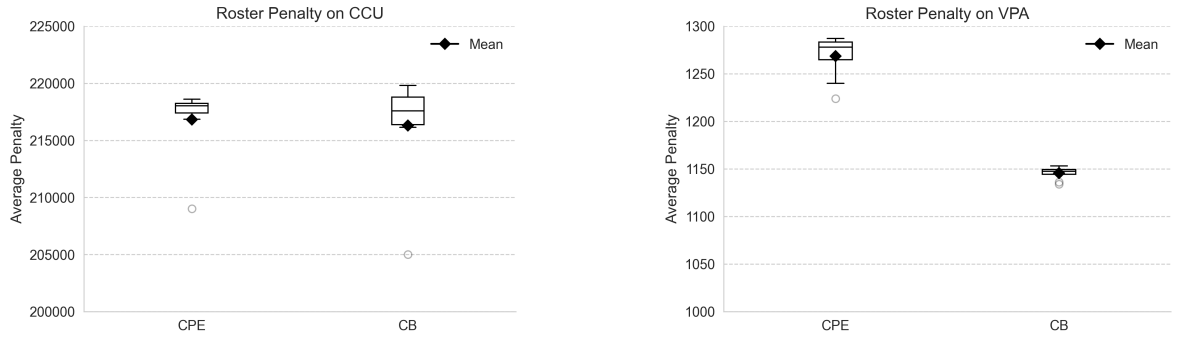

Figure A.1: Parameter tuning EmulateEC on which roster to compare with. CPE indicates the runs where the current roster was compared to pre-ejection chain roster. CB indicates the runs where the current roster was compared to best roster found so far.

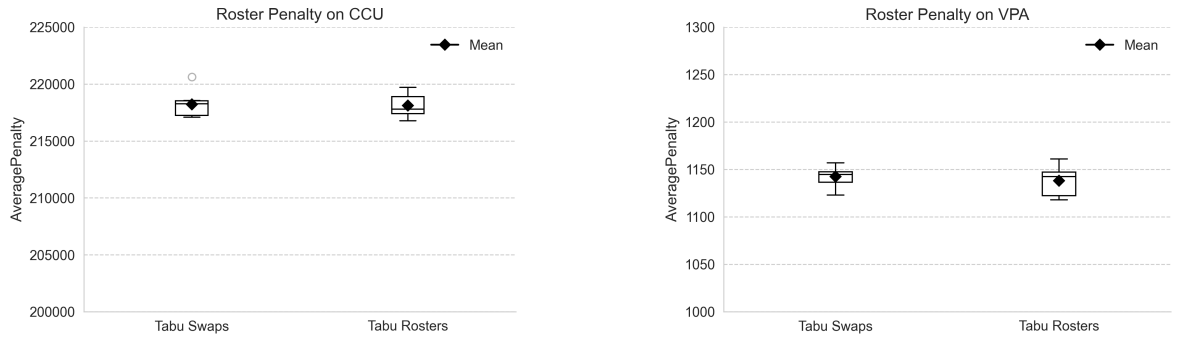

Figure A.2: Parameter tuning EmulateEC on which tabu list to use.

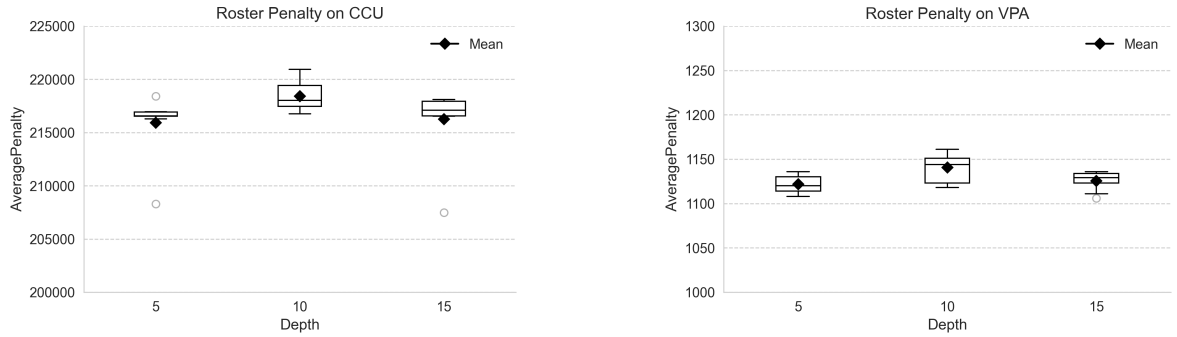

Figure A.3: Parameter tuning EmulateEC on the maximum allowed depth.

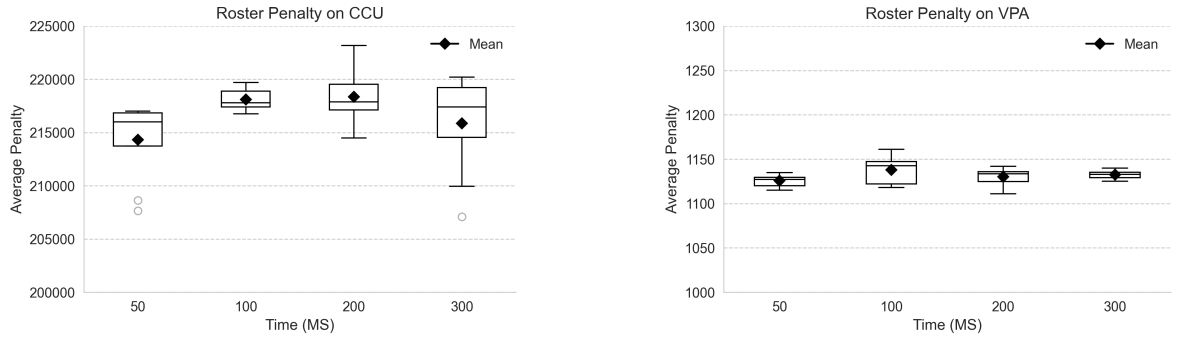

Figure A.4: Parameter tuning EmulateEC on the maximum allowed time per ejection chain.

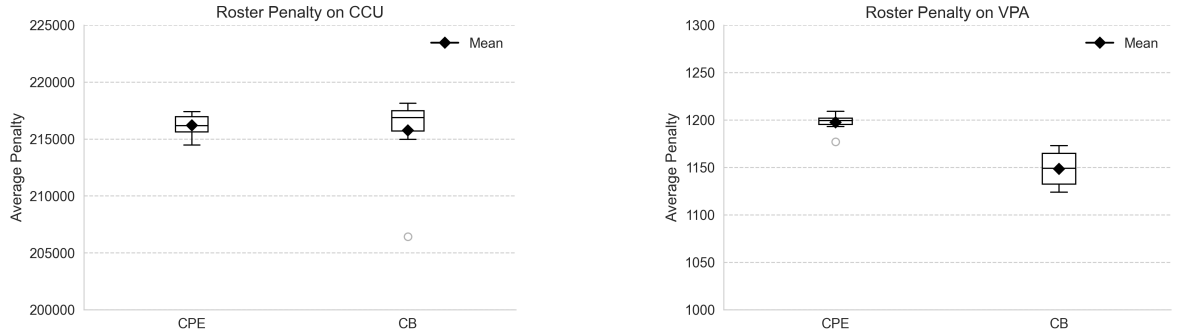

Figure A.5: Parameter tuning InfeasibleEC with BFSN on which roster to compare to. CPE indicates the runs where the current roster was compared to pre-ejection chain roster. CB indicates the runs where the current roster was compared to best roster found so far.

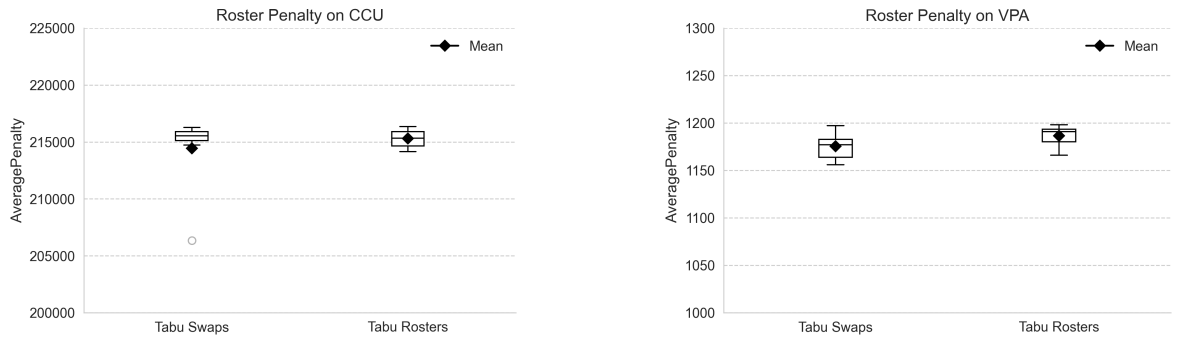

Figure A.6: Parameter tuning InfeasibleEC with BFSN on which tabu list to use.

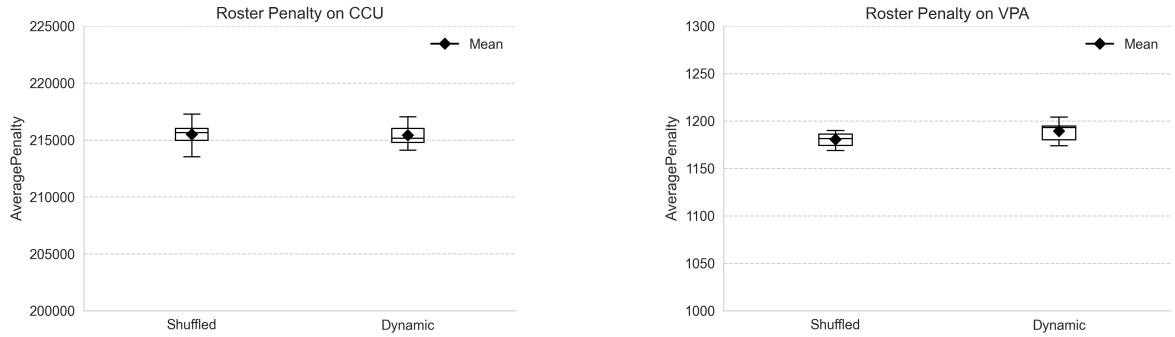

Figure A.7: Parameter tuning InfeasibleEC with BFSN, on how to order employees.

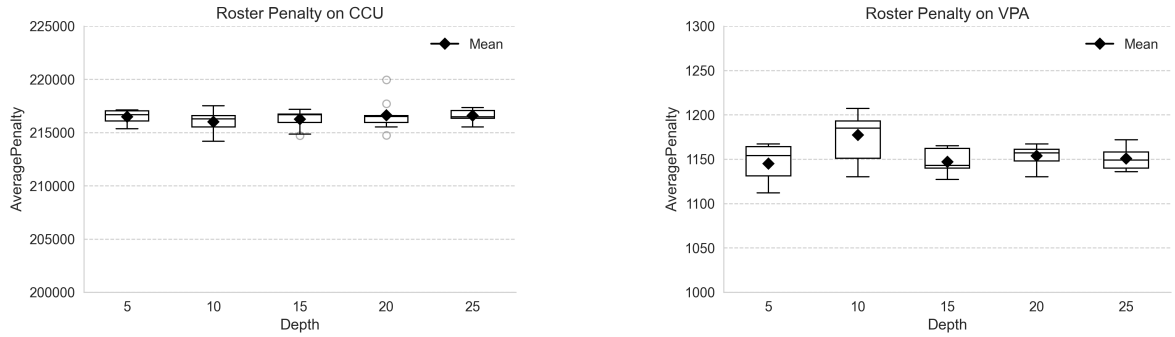

Figure A.8: Parameter tuning InfeasibleEC with BFSN on the maximum allowed depth.

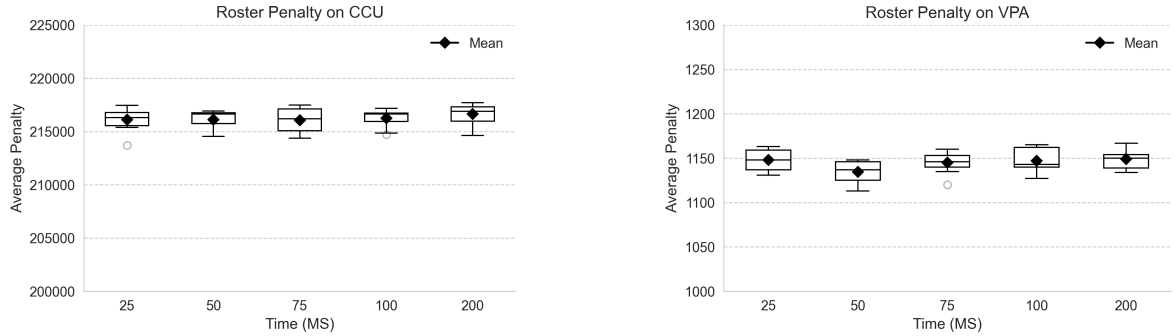

Figure A.9: Parameter tuning InfeasibleEC with BFSN on the maximum allowed time per ejection chain.

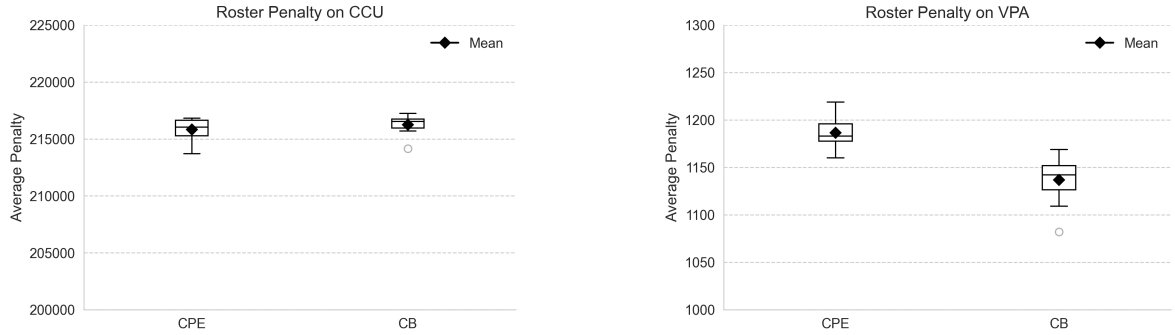

Figure A.10: Parameter tuning InfeasibleEC with DFS on which roster to compare with. CPE indicates the runs where the current roster was compared to pre-ejection chain roster. CB indicates the runs where the current roster was compared to best roster found so far.

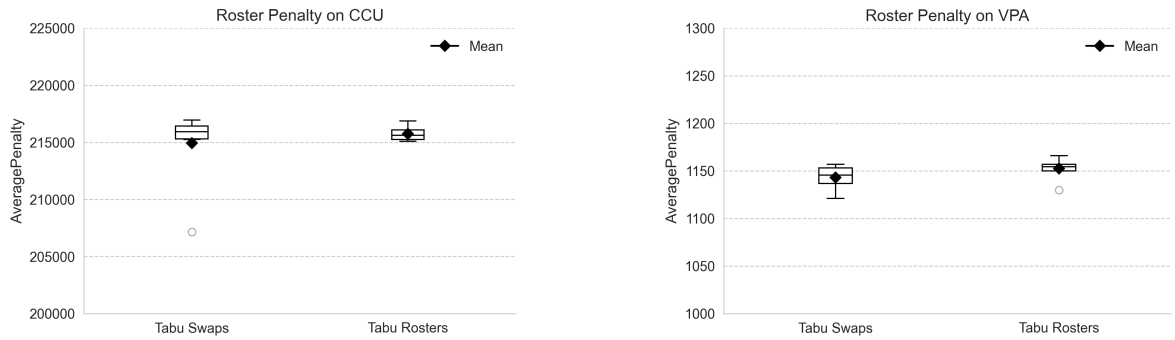

Figure A.11: Parameter tuning InfeasibleEC with DFS on which tabu list to use.

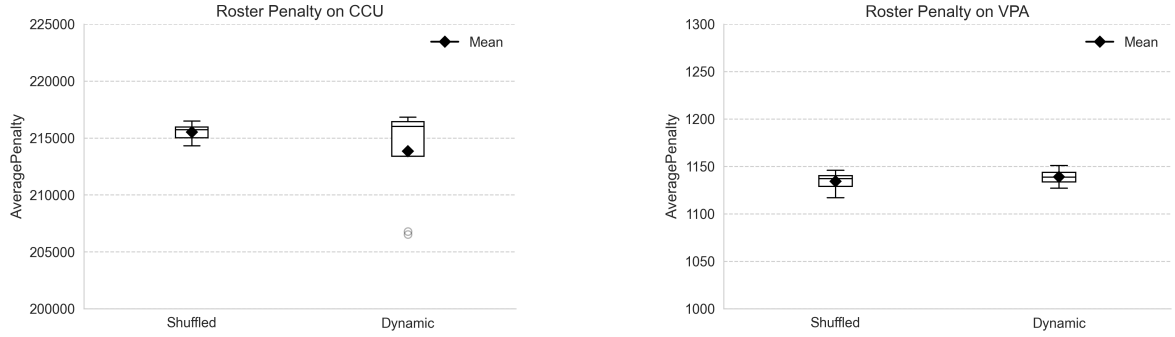

Figure A.12: Parameter tuning InfeasibleEC3 on how to order employees.

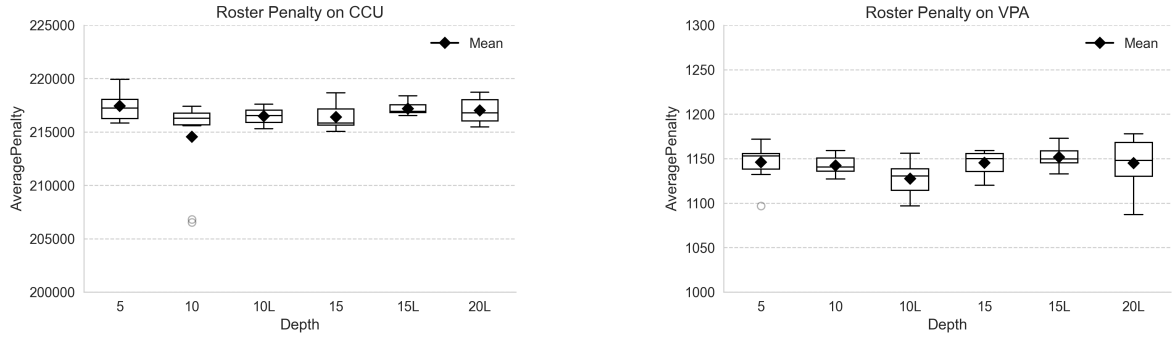

Figure A.13: Parameter tuning InfeasibleEC with DFS on the maximum allowed depth.

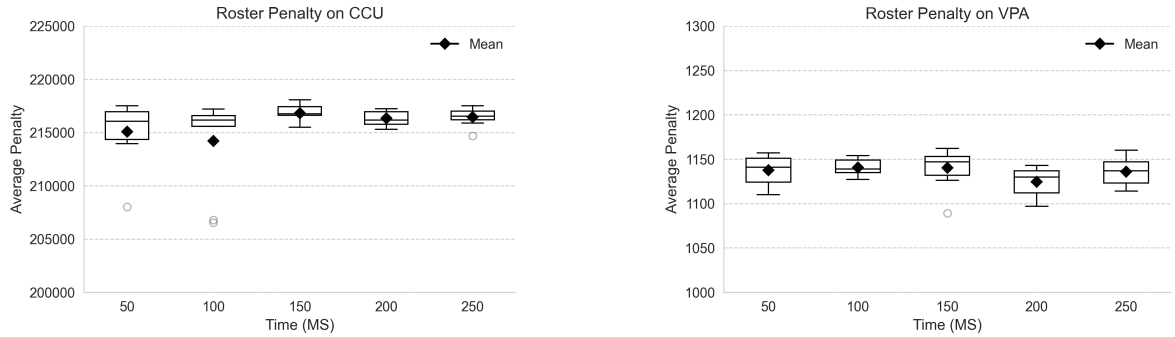

Figure A.14: Parameter tuning InfeasibleEC with DFS on the maximum allowed time per ejection chain.

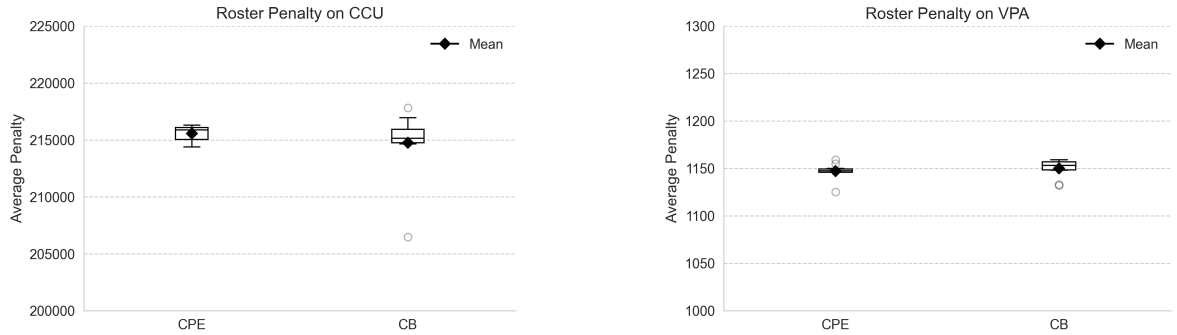

Figure A.15: Parameter tuning InfeasibleEC with DFSF on which roster to compare with. CPE indicates the runs where the current roster was compared to pre-ejection chain roster. CB indicates the runs where the current roster was compared to best roster found so far.

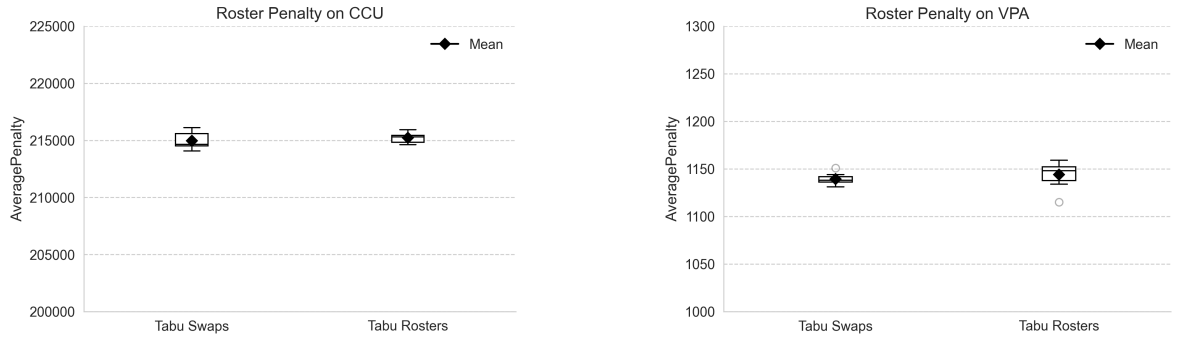

Figure A.16: Parameter tuning InfeasibleEC with DFSF on which tabu list to use.

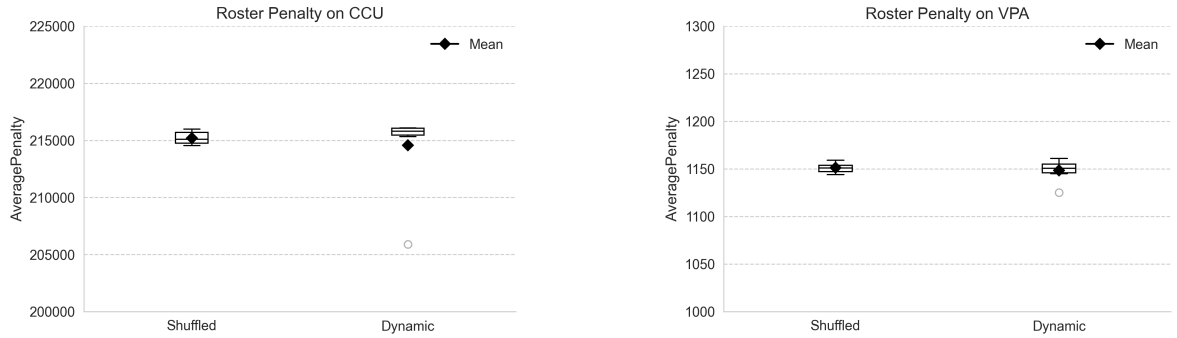

Figure A.17: Parameter tuning InfeasibleEC with DFSF on how to order employees.

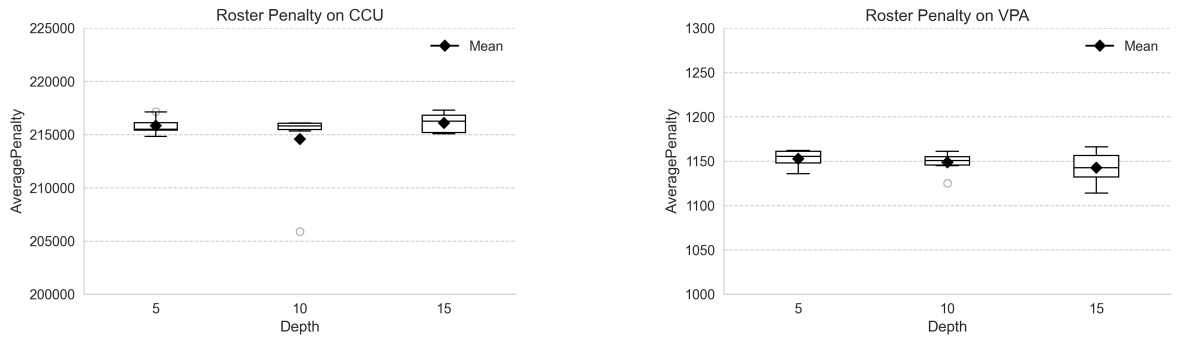

Figure A.18: Parameter tuning InfeasibleEC with DFSF on the maximum allowed depth.

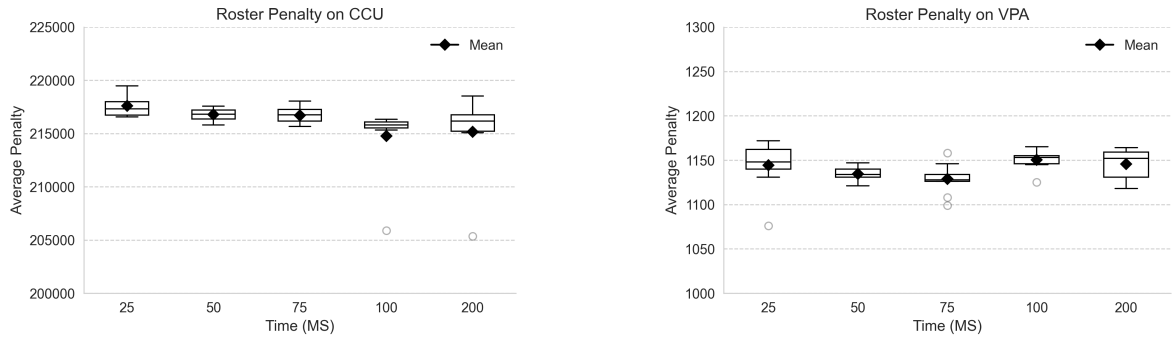

Figure A.19: Parameter tuning InfeasibleEC with DFSF on the maximum allowed time per ejection chain.

B

Repair Generators Pseudo Code

The following are pseudo codes of the repair generators. For sake of simplicity, the code is not complete, and the repair generators themselves generate more specific repairs that generate repairs from more methodologies. For example the pseudo code for the repair generator for weekly rest, does not consider the options of increasing the break by a couple hours by changing to an earlier shift, in the last shift before the break. Or later shift in the first shift after the break. As well as it does not consider the option of having two rests of 32 hours. While the real repair generator considers both. Furthermore, the repair generators do not show the addition of unfixable constraints, due to the synthetic fixed shifts explained in Section 4.1.3.

Algorithm 4 GenerateAllSwapsWeeklyRest

```
1:  Input: Employee employee, Constraint constraint
2:  days ← empty list
3:  firstDay ← weeklyRest.firstDay
4:  for day ← firstDay to min(endDay, firstDay + 14) do
5:      shift ← Shifts[day]
6:      if not isFixedShift(shift) and not isRestShift(shift) then
7:          requiredBreakInHours ← 72
8:          if day < firstDay + 7 then
9:              requiredBreakInHours ← 36
10:         end if
11:         if day − 1 ≥ 0 then
12:             newRest ← employee.RestAfterShiftInHours[day − 1] +
13:                 employee.RestAfterShiftInHours[day] +
14:                 employee.shift[day].validTimeInMins /60.0
15:             if Dbl.GE(newRest, requiredBreakInHours) then
16:                 add day to days
17:             end if
18:         end if
19:     end if
20: end for
21: Order(days)
22: employeesToConsider ← getAllOtherEmployeesWithCommonSkills()
23: yield GenerateAllVerticalSwaps(employeeA = employee, daysToConsider = days, otherEmployees =
24:     employeesToConsider, newShiftsNeedsToBeOffDays = true)
```

Algorithm 5 GenerateRepairsMaxNightShiftConstraint

```
1: Input: Employee employee, Constraint constraint
2: days ← empty list
3: for all day ∈ schedulingperiod do
4:     shift ← employee.ShiftOnDay[day]
5:     if not shift.isRestShift() and shift.isNightShift() and not isFixedShift(shift) then
6:         add day to days
7:     end if
8: end for
9: if constraint.minNumberNightShiftsNeededToChange ≤ 1 then
10:    Order(days)
11:    employeesToConsider ← AllOtherEmployeesWithCommonSkills()
12:    yield return GenerateAllSingleVerticalSwaps(eployeeA = employeeIndex, daysToConsider =
   days, otherEmployees =
13:     employeesToConsider, newShiftShouldBeNonNightShift = true)
14: end if
```

Algorithm 6 GenerateRepairsMaxSundaysConstraint

```
1: Input: Employee employee, Constraint constraint
2: sundayDays ← empty list
3: nonSundayDays ← empty list
4: for all day ∈ schedulingperiod do
5:     shift ← employee.ShiftOnDay[day]
6:     if not isRestShift(shift) and not isFixedShift(shift) then
7:         if dayOfTheWeek(day) = Sunday then
8:             add day to sundayDays
9:         else if isNightShift(shift) and dayOfTheWeek(day + 1) == Sunday then
10:            add day to sundayDays
11:        else
12:            add day to nonSundayDays
13:        end if
14:    end if
15: end for
16: if constraint.minNumberSundayShiftsNeededToChange ≤ 1 then
17:    Order(sundayDays)
18:    Order(nonSundayDays)
19:    employeesToConsider ← getAllOtherEmployeesWithCommonSkills()
20:    yield GenerateAllVerticalSwaps(employeeA = employeeIndex, daysToConsider = sundayDays,
   otherEmployees =
21:     employeesToConsider, newShiftNeedsToBeDayOff = true)
22:    yield GenerateAllHorizontalSwaps(employeeA = employeeIndex, daysSetA = sundayDays,
   daysSetB =
23:     nonSundayDays)
24: end if
```

Algorithm 7 GenerateAllRepairsRestAfterConsecutiveNightShifts

Input: Employee *employee*, Constraint *constraint*
daysToMakeOffDay ← empty list
daysToMakeNightShifts ← empty list
daysToMakeNonNightShifts ← empty list
dayOfLastNightShift ← constraint.dayOfLastNightShift
$shift0$ ← Shifts[dayOfLastNightShift]
$shift1$ ← Shifts[dayOfLastNightShift + 1]
$shift2$ ← Shifts[dayOfLastNightShift + 2]
if not isRestShift($shift2$) **and** isNightShift($shift2$) **then**
    add $dayOfLastNightShift + 1$ to *daysToMakeNightShifts*
end if
if isRestShift($shift1$) **then**
    add $dayOfLastNightShift$ to *daysToMakeOffDay*
    if $dayOfLastNightShift + 2 \leq endDay$ **then**
        add $dayOfLastNightShift + 2$ to *daysToMakeOffDay*
    end if
end if
if isRestShift($shift2$) **then**
    add $dayOfLastNightShift + 1$ to *daysToMakeOffDay*
end if
if consecutiveNightShifts = 3 **then**
    add $dayOfLastNightShift$ to *daysToMakeOffDay*
    if $dayOfLastNightShift - 2 \geq 0$ **then**
        add $dayOfLastNightShift - 2$ to *daysToMakeNonNightShifts*
    end if
    if $dayOfLastNightShift - 1 \geq 0$ **then**
        add $dayOfLastNightShift - 1$ to *daysToMakeNonNightShifts*
    end if
end if
Order(*daysToMakeOffDay*)
Order(*daysToMakeNightShifts*)
employeesToConsider ← getAllOtherEmployeesWithCommonSkills()
yield GenerateAllVerticalSwaps(employeeA = employee, daysToConsider = daysToMakeOffDay, otherEmployees =
    employeesToConsider, newShiftsNeedsToBeDayOff = **true**)
yield GenerateAllVerticalSwaps(employeeA = employee, daysToConsider = daysToMakeNightShifts, otherEmployees =
    employeesToConsider, newShiftsNeedsToBeNightShifts = **true**)
yield GenerateAllVerticalSwaps(employeeA = employee, daysToConsider = daysToMakeNonNightShifts, otherEmployees
    = employeesToConsider, newShiftsNeedsToBeNonNightShifts = **true**)

Algorithm 8 GenerateAllRepairsMinWeekendsOff

```
1: Input: Employee employee, Constraint constraint
2: makeEarlierList ← minWeekendsOff.FridaysAndMondays ▷ Only includes weeks where both Saturday and Sunday
3:     are off. Also contains whether it is a Friday and needs to become an earlier shift, or a Monday and needs to become a
4:     later shift
5: makeOffDayDays ← minWeekendsOff.SaturdayAndSundayDays ▷ Only includes weeks where either only Saturday or
6:     only Sunday are worked
7: Order(makeEarlierList)
8: Order(makeOffDayDays)
9: employeesToConsider ← getAllOtherEmployesWithCommonSkillsAndCanWorkWeekend()
10: yield GenerateAllVerticalSwaps(employeeA = employee, daysToConsider = makeOffDayDays, otherEmployees
11:     = employeesToConsider, newShiftNeedsToBeOfDay = true)
12: yield GenerateAllVerticalSwaps(employeeA = employee, daysToConsider = makeEarlierList.days, otherEmployees
13:     = employeesToConsider, newShiftsNeedsToBeEarlier = makeEarlierList.makeEarlier)
```

Algorithm 9 GenerateAllMaxWorkloadRepairs

```
1: Input: Employee employee, Constraint constraint
2: days ← empty list
3: for all day ∈ schedulingperiod do
4:     shift ← employeeA.shiftOnDay[day]
5:     if not isFixedShift(shift) and and not isRestShift(shift) then
6:         add day to days
7:     end if
8: end for
9: Order(days)
10: employeesToConsider ← empty list
11: availableTimeA ← employee.GetAvailableWorkingTime()
12: for all employeeB in employees do
13:     if employee ≠ employeeB then
14:         availableTimeB ← employeeB.GetAvailableWorkingTime()
15:         if availableTimeA + availableTimeB ≥ 0 then
16:             add employeeB to employeesToConsider
17:         end if
18:     end if
19: end for
20: Order(employeesToConsider)
21: yield GenerateAllVerticalSwaps(employeeA = employee, days = days, otherEmployees
22:     = employeesToConsider, newShiftsNeedsToHaveLessWorkingHours = true)
```

Algorithm 10 GenerateAllDailyRestRepairs

```
1: Input: Employee employee, Constraint constraint
2: days ← List(constraint.day, constraint.day + 1)
3: if DailyRestExceptionUsedPreviously() and DailyRestExceptionUsedCurrently() then
4:     days.add(previousDayExceptionUsed)
5:     days.add(previousDayExceptionUsed + 1)
6: end if
7: Order(days)
8: employeesToConsider ← getAllOtherEmployeesWithCommonSkills()
9: Order(employeesToConsider)
10: repairList ← List(verticalSingleRepair, horizontalSingleRepair)
11: Shuffle(repairList)
12: yield GenerateAllVerticalSwaps(employeeA = employee, daysToConsider = days, otherEmployees =
13:       employeesToConsider, newShiftsNeedsToIncreaseBreak = true)
```

Algorithm 11 GenerateConshiftsWithNightRepairs

```
1: Input: Employee employee, Constraint constraint
2: days ← empty list
3: consecutiveBlock ← constraint.block
4: if length(consecutiveBlock) = 7 then
5:     days.add(consecutiveBlock[0])
6:     days.add(consecutiveBlock[6])
7: end if
8: for all day in consecutiveBlock do
9:     shift ← employee.shiftOnDay[day]
10:    if not isFixedShift(shift) and and not isRestShift(shift) then
11:        if restIfUnshifting(day) ≥ 32 then
12:            days.add(day)
13:        end if
14:    end if
15: end for
16: Order(days)
17: employeesToConsider ← getAllOtherEmployeesWithCommonSkills(employee)
18: Order(employeesToConsider)
19: yield GenerateAllVerticalSwaps(employeeA = employee, daysToConsider = days, otherEmployees =
20:       employeesToConsider, newShiftsNeedsToIncreaseBreak = true)
```