



**University of  
Nottingham**

UK | CHINA | MALAYSIA

**Python-Based  
Optimization Software Library  
for the  
Office Space Allocation Problem**

Submitted September 2023, in partial fulfilment of the conditions for the  
award of the degree of MSc Computer Science (Artificial Intelligence)

**Arnav Saxena**

**20444748**

School of Computer Science  
University of Nottingham

I declare that this dissertation is all my own work, except as indicated in  
the text

## Abstract

The Office Space Allocation Problem in an academic setting refers to the assignment of a set of available spaces (such as office rooms or classrooms ) to a set of entities (such as employees, departments, or groups) in a way that reduces space wastage and satisfies additional imposed constraints. In the literature, both exact methods and heuristics have been applied to this problem and it is found that the latter provides competitive solutions in a very short time as compared to the former.

This dissertation provides a software library implementation to tackle this problem. It includes baseline heuristics and procedures that read in test instances and provide an entity-room allocation as output. An analysis regarding the feasibility and infeasibility of the allocations produced by the constructive and exploration heuristics is provided. Furthermore, effects on these allocations by varying parameters like hard constraint penalty and initial allocation quality are also documented in this work. The results reveal that a low hard constraint penalty value allows the exploration metaheuristics to venture out to find better solutions but at the same time the probability of converging and staying in the feasible region decreases. It was also found out that an initial allocation with very low hard constraint violation (feasible/almost feasible) is not a good starting choice for the exploration heuristic as it converges too quickly towards the feasible region and overlooks higher-quality solutions. Overall a balance has to be struck between exploring and converging.

This dissertation aims to equip the readers with the knowledge of selecting the right set of heuristics and parameters to reach the globally optimal allocation in a reasonable amount of time.

Keywords: Office Space Allocation; heuristics; metaheuristics; Hill-Climbing; Simulated Annealing; optimisation; software.

## Acknowledgements

I would like to express my deepest gratitude to my father, Prashant Saxena, and my mother, Divya Saxena for their unwavering love, encouragement and financial support throughout my life. Thanks to my sister, Akshika Saxena for helping me keep my cool and sense of perspective throughout my academic journey. To my uncle, Arun Srivastava and my aunt, Deepa Srivastava, I would like to express my sincere appreciation for providing me with a lovely home away from home here in the UK.

I am extremely grateful to my supervisor Dr. Dario Landa-Silva for his invaluable guidance, mentorship and expertise throughout the length of this project. His patience and knowledge of the subject is unparalleled. Now, every time I come across a problem, I think to myself, "How can it be solved and optimized?". This new way of thinking will help me immensely in my professional career.

Finally, I would like to thank the University of Nottingham for providing the nurturing academic environment and resources that made this journey possible.

# Table of Contents

Abstract.....	2
Acknowledgements.....	3
Table of Contents.....	4
List of Figures .....	6
List of Tables .....	7
Chapter 1 INTRODUCTION .....	8
1.1 Background and Motivation.....	8
1.2 Aims and Contributions.....	9
1.3 Overview of this Dissertation.....	11
Chapter 2 OFFICE SPACE ALLOCATION PROBLEM .....	12
2.1 Introduction .....	12
2.2 Constraints.....	12
2.3 Penalty Evaluation Function .....	15
2.4 Datasets .....	15
2.5 Conclusion .....	18
Chapter 3 LITERATURE REVIEW .....	19
3.1 Introduction .....	19
3.2 Previous Research on Office Space Allocation (Pre-2000) .....	19
3.3 Previous Research on Office Space Allocation (2000-2010) .....	21
3.4 Previous Research on Office Space Allocation (2011-2023) .....	24
3.5 Conclusion .....	28
Chapter 4 IMPLEMENTATION .....	29
4.1 Introduction .....	29
4.2 Solution Representation and Data Structures.....	29
4.3 Penalty Evaluation Function .....	30
4.4 Input Functionality .....	30
4.5 Constructive Heuristics .....	31
4.6 Neighbourhood Exploration Operators.....	36
4.7 Driving Metaheuristics.....	40
4.7.1 Hill Climbing/ Iterative Improvement Algorithm.....	40
4.7.2 Simulated Annealing Algorithm .....	42
4.8 Output Functionality.....	46

4.9 Conclusion .....	46
Chapter 5 RESULTS AND DISCUSSION .....	47
5.1 Introduction .....	47
5.2 Constructive Heuristics Comparison .....	48
5.3 Effect of Hard Constraint Penalty Value.....	53
5.4 Effect of Initial Solution Quality .....	57
5.5 Driving Metaheuristics Comparison.....	61
5.6 Conclusion .....	66
Chapter 6 CONCLUSIONS .....	67
6.1 Limitations and Future Work.....	68
REFERENCES .....	70
APPENDICES.....	73
Appendix A: Dataset Images.....	73
Appendix B: Output Text File Images.....	74

## List of Figures

Figure 4.1 Data structure utilized to represent allocations .....	30
Figure 4.2 Input function of the OSA library .....	31
Figure 4.3 AllocateRnd-Rnd.....	32
Figure 4.4 AllocateRnd-BestRnd.....	33
Figure 4.5 AllocateWgt-BestRnd.....	34
Figure 4.6 AllocateCsrt-BestRnd.....	35
Figure 4.7 AllocateBestAll .....	36
Figure 4.8 Relocate Rnd-Rnd.....	37
Figure 4.9 Relocate Rnd-BestRnd .....	37
Figure 4.10 Mechanism of Relocate operation.....	38
Figure 4.11 Swap Rnd-Rnd.....	38
Figure 4.12 Swap Rnd-BestRnd.....	39
Figure 4.13 Mechanism of Swap operation .....	39
Figure 4.14 IIARnd-Rnd.....	41
Figure 4.15 IIARnd-BestRnd .....	42
Figure 4.16 SAARnd-Rnd.....	44
Figure 4.17 SAARnd-BestRnd.....	45
Figure 5.1 Varying hard constraint penalty value on the nott dataset.....	53
Figure 5.2 Varying hard constraint penalty value on the p000_n025 dataset.....	54
Figure 5.3 Varying hard constraint penalty value on the p025_n000 dataset.....	54
Figure 5.4 Varying hard constraint penalty value on the s000_v000 dataset .....	55
Figure 5.5 Varying hard constraint penalty value on the s000_v100 dataset .....	55
Figure 5.6 Varying hard constraint penalty value on the s100_v000 dataset .....	56
Figure 5.7 Varying hard constraint penalty value on the s100_v100 dataset .....	56
Figure 5.8 Driving Metaheuristics performance on nott dataset .....	62
Figure 5.9 Driving Metaheuristics performance on p000_n025 dataset.....	62
Figure 5.10 Driving Metaheuristics performance on p025_n000 dataset.....	63
Figure 5.11 Driving Metaheuristics performance on s000_v000 dataset.....	63
Figure 5.12 Driving Metaheuristics performance on s000_v100 dataset.....	64
Figure 5.13 Driving Metaheuristics performance on s100_v000 dataset.....	64
Figure 5.14 Driving Metaheuristics performance on s100_v100 dataset.....	65
Appendix Figure 1 Metadata followed by Entities Section.....	73
Appendix Figure 2 The Rooms Section .....	73
Appendix Figure 3 The Constraints Section .....	74
Appendix Figure 4 The entity-room allocation is provided in output.txt .....	74
Appendix Figure 5 Statistics related to the entity-room allocation .....	75
Appendix Figure 6 Satisfaction/Violation of constraints by the allocation .....	75

## List of Tables

Table 2.1 Soft constraint violation penalties for all considered constraints .....	14
Table 2.2 Summary of all benchmark instances considered in this work.....	18
Table 5.1 Average time taken to execute each constructive heuristic.....	47
Table 5.2 Average time taken to execute each driving metaheuristic. ....	47
Table 5.3 Constructive Heuristics performance on nott dataset.....	48
Table 5.4 Constructive Heuristics performance on p000_n025 dataset.....	49
Table 5.5 Constructive Heuristics performance on p025_n000 dataset.....	49
Table 5.6 Constructive Heuristics performance on s000_v000 dataset .....	50
Table 5.7 Constructive Heuristics performance on s000_v100 dataset .....	50
Table 5.8 Constructive Heuristics performance on s100_v000 dataset .....	51
Table 5.9 Constructive Heuristics performance on s100_v100 dataset .....	51
Table 5.10 Effect of initial solution quality on the nott dataset.....	58
Table 5.11 Effect of initial solution quality on the p000_n025 dataset.....	58
Table 5.12 Effect of initial solution quality on the p025_n000 dataset.....	58
Table 5.13 Effect of initial solution quality on the s000_v000 dataset .....	59
Table 5.14 Effect of initial solution quality on the s000_v100 dataset .....	59
Table 5.15 Effect of initial solution quality on the s100_v000 dataset .....	59
Table 5.16 Effect of initial solution quality on the s100_v100 dataset .....	60

# Chapter 1 INTRODUCTION

## 1.1 Background and Motivation

The efficient allocation of office space among various entities is a well-researched problem. This problem is usually subjected to several constraints, both hard (compulsory) and soft (optional), which makes finding quality solutions a difficult task. Its importance is ever increasing and researchers keep coming up with new computational algorithms to tackle this combinatorial optimization problem. Solving this problem increases employee productivity, brings down real estate costs and increases collaboration and communication. Since the COVID-19 pandemic, the hybrid work option (work from home and office) has been provided by many companies. Hence there is renewed interest in this topic. This problem arises in various industries and organizations, large and small. In this work, Office Space Allocation (OSA) problem is looked at in an academic setting. In this setting, entities refer to laboratories, staff, research groups, libraries etc., and the office space refers to the rooms in the university/academic institution owned buildings. The constraints refer to situations that arise in academics, for e.g. research heads have to be placed with their group, heads of departments need their own room etc.

More often than not, the floor space available can't be changed. Hence, in this problem following aspects of office space allocation are tackled:

1. Creating a fresh entity-room mapping from scratch. This is useful when an entire department relocates from one building to another.
2. Reorganising an existing entity-room mapping. This is the most common case as most of the times there are only slight changes in personnel or requirements.

While creating or reorganising this entity-room mapping it is necessary to ensure that space overuse (exceeding the capacity of a room) or space wastage (using less than the capacity of a room) is avoided as much as possible. Furthermore, constraint violations have to be minimised.

For a long time this problem was solved using exact methods such as goal programming (Lee et al., 1972), (Ritzman et al., 1979) and (Giannikos et al.,



1995). While these methods do provide optimal solutions, they take an immense amount of time for their execution which in turn restricts the dataset size and the number of constraints that can be considered. Burke et al., 1998 and (Burke et al., 1999) started the work towards an automated system that solves the OSA problem by using heuristics and meta-heuristics algorithms instead of exact methods. A formal analysis with benchmark instances was given by (Landa-Silva, 2003). This work was extended by the addition of new benchmark datasets and constraints (Ulker and Landa-Silva, 2011). Soon population-based metaheuristics and hybrid metaheuristics (Burke et al., 2001b), (Landa-Silva and Burke, 2007), (Ulker and Landa-Silva, 2012), (Awadallah et al., 2012), (Castillo et al., 2016) and (Bolaji et al., 2017) were customized to tackle the OSA problem. These techniques provide space administrators with multiple near-optimal solutions in a short amount of time. All work done on this topic has a common underlying goal. That goal is to provide the space administrators with good quality allocation(s) which they can then manually adapt (if required) to make their own entity-room mapping.

The amount of work done on this topic can be overwhelming for a beginner who is approaching this problem for the first time. The primary motivation behind this dissertation is to build a bridge between a newcomer and the vast expanse of the OSA literature out there. This is further explained in Section 1.2.

## 1.2 Aims and Contributions

The main aim of this dissertation is to create an optimization software library that includes baseline heuristics that solves the Office Space Allocation Problem as described in (Landa-Silva, 2003). The benchmark instances (Nott, PNe150 and SVe150) used in this work are taken from (Landa-Silva, 2003) and (Ulker and Landa-Silva, 2011). Most of the algorithms implemented in the literature are programmed using C++ (Chapter 3). In this work, Python is used instead of C++. While the latter is known for its speed and memory management, the former provides enhanced readability, rapid development capabilities and extensive data-handling libraries. Due to these advantages, it is an ideal choice for the basis of this library.

Almost all research in the literature focuses on the main driving heuristic and not on the constructive heuristics. For the construction of solutions, usually the Peckish method (Corne et al., 1994) is utilized. Feasibility is guaranteed by the authors at every step of the algorithm. But as the number of constraints increases, this guarantee is not possible. Furthermore, in a bid to maintain feasibility, the exploration process suffers. Hence, it becomes more and more logical to construct a good quality initial solution (feasible or infeasible) and then feed it to the driving metaheuristic as input. This driving metaheuristic then explores the search space under an imposed hard constraint penalty and finds the best feasible solution(s) (Landa-Silva, 2003) and (Ulker, 2013). This is exactly how the algorithms are designed in this library. Several variants of constructive heuristics and driving metaheuristics can be designed by varying the level of randomness in their implementations (Landa-Silva, 2003). Since feasibility is not guaranteed from the beginning, a hard constraint penalty component is added to the evaluation function (Section 4.3). This is done to help the driving metaheuristic converge towards the feasible search space.

Furthermore, it is seen that, even in works that incorporate constructive heuristics and the possibility of infeasible initial solutions, a proper analysis regarding the effect of hard constraint penalty or the effect of initial solution quality on the exploration process, is omitted.

Hence, the novel contributions of this dissertation can be summarised as follows:

1. A code repository in Python containing input functionality, output functionality, standard constructive heuristics and exploration driving metaheuristics (Chapter 4). The library is implemented in Python to aid beginners in the field of OSA.
2. A comparison of the quality of initial solutions provided by the constructive heuristics (Section 5.2). Landa-Silva, 2003 did this for the Nott dataset. In this work, this is extended to the PNe150 and SVe150 benchmark instances as well.

3. For the first time, an in-depth analysis of the effects of varying the hard constraint penalty component on the driving metaheuristics output (Section 5.3) on all the benchmark instances.
4. For the first time, an in-depth analysis of the effects of different initial solutions on the driving metaheuristics output (Section 5.4) on all the benchmark instances. The initial solutions differ based on the number of hard constraints they violate, i.e., their degree of infeasibility.

### 1.3 Overview of this Dissertation

Chapter two outlines the OSA problem in more depth. Constraints considered in the library are defined. The penalty evaluation function is mathematically stated. The format and characteristics of the benchmark instances are explained in detail and are tabulated.

Chapter three provides an extensive chronological literature review of the previous research conducted on the OSA problem. The algorithms, technologies and experiments considered in these works are highlighted. The strengths and weaknesses of certain works are also highlighted.

Chapter four describes the entire implementation of the software library. Pseudocode is provided for all the different processes, i.e. input, initialisation, exploration and finally output.

In chapter five, experiments are carried out on the benchmark test instances. Thereafter, the results and discussions are provided. Proof that the implementation is working correctly is present in this chapter.

Finally, the dissertation concludes with chapter 6. Here, the limitations of this work are outlined. This is followed by suggestions for improvements that can be undertaken in the future.

## Chapter 2 OFFICE SPACE ALLOCATION PROBLEM

### 2.1 Introduction

Office Space Allocation refers to the assignment of several entities or resources such as laboratories, researchers, staff etc., among available rooms or office spaces subjected to additional constraints. These constraints can be hard (mandatory satisfaction) or soft (optional satisfaction). This is an extremely restricted combinatorial optimization problem (due to several constraints) with two conflicting objectives. The first objective is to reduce space misuse, i.e. space overuse and space wastage. The second objective is to reduce the number of violated soft constraints. Furthermore, it is an NP-hard problem which makes the application of heuristics and metaheuristics to solve it advantageous over time-consuming exact methods like integer programming.

This chapter is organised as follows: Section 2.2 looks at the various types of constraints this problem is subjected to. Section 2.3 describes a mathematical formulation of the evaluation function used to calculate the fitness value of a feasible solution. Section 2.4 elaborates on the datasets used in this dissertation in detail. Finally, Section 2.5 concludes this chapter.

### 2.2 Constraints

The constraints involved in this problem are of two types as mentioned above: hard and soft. Landa-Silva, 2003 considered six constraints, namely Allocation ("Be located in"), Same Room ("Be together with"), Not Sharing ("Not sharing"), Adjacency ("Be Adjacent to"), Nearby ("Be grouped with") and Away from ("Be away from"). Three more constraints were added and the model was expanded (Ulker et al., 2011). These are named Not Allocation, Not Same Room, and Capacity. We consider all 9 of these constraints in this dissertation.

Given below is a brief description of each of these constraints.

**Allocation:** This constraint is used to place a particular entity in a particular room. As it operates on two pieces of information, it is a binary constraint. This can be used to allocate specially designed rooms to their roles. For example, the library entity must be placed in a room which contains enough area to house bookshelves and study spaces. In this work, this is considered as a soft constraint.

**Not Allocation:** This constraint is the opposite of the Allocation constraint. It is a binary constraint. It can be used to avoid placing a particular entity in a particular room. For example, certain professors might prefer a majority of classrooms except a few, this constraint can help enforce this requirement. In this work, this is considered as a soft constraint

**Capacity:** This constraint is used to avoid space overuse in a particular room. It is unary in nature as only the room id information is needed to enforce this condition. Overcrowding of entities can lead to a decline in productivity. For example, a computer lab has facilities for a certain number of workstations and hence its capacity can't be exceeded. In this work, this constraint can be hard or soft.

**Same Room:** This constraint is used to place two entities in the same room. It is a binary in nature. This can be used to encourage communication and increase efficiency. For example, two professors who are heads of their respective teams in a research project can share the same room to increase collaboration. In this work, this is considered as a soft constraint.

**Not Same Room:** This constraint is the opposite of the Same Room constraint. It is also binary in nature. It is used to avoid placing two particular entities in the same room. For example, if two staff members require different working conditions then they can't be placed in the same room. This helps in avoiding conflicts. In this work, this is considered as a soft constraint.

**Not Sharing:** This constraint is used to avoid placing a particular entity with any other entity in the same room. It is unary in nature. It is mostly used to allocate senior lecturers, researchers or staff members to their own private offices. In this work, this is considered as a hard constraint.

**Adjacency:** This constraint is used to place two particular entities in adjacent rooms. It is binary in nature. Information regarding rooms that are adjacent to each other is required to enforce this constraint. For example, module convenors and their teaching assistants can be placed in adjacent rooms to maintain teamwork throughout the term. In this work, this is considered as a soft or hard constraint.

**Nearby:** This constraint is used to place two particular entities on the same floor. It is binary in nature. All rooms on a particular floor are considered near to each other. The number of rooms near to a particular room is greater than the rooms adjacent to it. Hence this constraint can be used to place members of a large research group together with computing facilities. In this work, this is considered as a soft constraint.

**Away from:** This constraint is the opposite of the Nearby constraint. It is also binary in nature. It is used to avoid placing two particular entities on the same floor. For example, a research group performing chemical experiments might disrupt another research group due to heavy fumes and hence needs to be placed on a separate floor. In this work, this is considered as a soft or hard constraint.

Table 2.1 Soft constraint violation penalties for all considered constraints

Constraint Name	Soft Constraint Violation Penalty
ALLOCATION_CONSTRAINT	20
NONALLOCATION_CONSTRAINT	10
CAPACITY_CONSTRAINT	10
SAMEROOM_CONSTRAINT	10
NOTSAMEROOM_CONSTRAINT	10
NOTSHARING_CONSTRAINT	50
ADJACENCY_CONSTRAINT	10
NEARBY_CONSTRAINT	10
AWAYFROM_CONSTRAINT	10

## 2.3 Penalty Evaluation Function

The evaluation or objective function utilized to calculate the fitness value of a feasible allocation is taken from (Landa-Silva, 2003). This function contains two components. These components also represent the two objectives of this problem.

1. Space Misuse Penalty (Overuse and Wastage)
2. Soft Constraint Violation Penalty.

While allocating space it is important that the capacity of a room is not exceeded or wasted. Similarly, it is desirable that most of the soft constraints are satisfied. These two objectives were found to be conflicting (Ek Burke et al., 2001c). Hence, the aggregate of both these components forms the overall evaluation function as described below.

$$\sum_{i=1}^m (WP(i) + OP(i)) + \sum_{r=1}^s SCP(r) \quad (2.1)$$

$m$  = number of rooms

$s$  = number of soft constraints

$WP(i)$  = Penalty due to space wastage in  $i^{th}$  room

$WP(i) = (\text{Capacity of the } i^{th} \text{ room}) - (\text{Total space occupied by entities in } i^{th} \text{ room})$

$OP(i)$  = Penalty due to space overuse in  $i^{th}$  room

$OP(i) = 2 * [(\text{Total space occupied by entities in } i^{th} \text{ room}) - (\text{Capacity of } i^{th} \text{ room})]$

$SCP(r)$  = Penalty due to violating the  $r^{th}$  soft constraint

## 2.4 Datasets

In this dissertation, we examine seven benchmark instances. Each instance contains information on three components. These are described below.

Entities: These are the resources that have to be distributed among the various rooms. The attributes used to provide information regarding entities are EntityID, GroupID and Space.

- EntityID: Unique identifier for each entity.

- GroupID: This is an identifier used for the group in which the entity exists. E.g. Researcher, staff etc.
- Space: Refers to the area required by an entity in  $m^2$ .

Rooms: The whole office space is divided into several floors. Each floor is further broken down into specific areas. The attributes used to provide information regarding rooms are RoomID, Floor, Space, No. of Adjacent Rooms and Adjacent List.

- RoomID: Unique identifier for each room.
- Floor: Provides the floor number on which the room exists.
- Space: Refers to the capacity of the room in  $m^2$ .
- No. of Adjacent Rooms: Provides the amount of adjacent rooms around a particular room.
- Adjacent List: Provides the collection of RoomID's which are adjacent to a particular room.

Constraints: These are the conditions/requirements imposed upon certain entities and rooms. A feasible allocation is an allocation which has all its hard constraints satisfied. The nine types of constraints considered in this work were discussed in Section 2.2. The attributes used to provide information regarding this component are ConstraintID, Constraint Type, Hardness Subject and Target.

- ConstraintID: Unique identifier for each constraint
- Constraint Type: An integer value unique to each of the nine constraints considered.
- Hardness: A value of 0 corresponds to a soft constraint whereas 1 corresponds to a hard constraint.
- Subject: The first entity or room on which the constraint is applied.
- Target: The second entity on which the constraint is applied. If the constraint is unary in nature then this value is -1.

The seven benchmark datasets used in this work are described below:



1. Landa-Silva, 2003 provides us with the **University of Nottingham (nott1)** instance. This was produced from the space allocation process in the School of Computer Science and Information Technology during the 1999-2000 academic year.
2. Ulker and Landa-Silva, 2011 extended the OSA problem description by adding three new constraints and provided us with the **SVe150** and **PNe150** datasets. The formation of these two types of test instances are controlled by four parameters. These parameters are responsible for deciding the complexity of the dataset constructed:
  - i. Slack Space Rate(S): regulates the number of rooms that will have their space modified.
  - ii. Violation Rate(V): determines whether a violated soft constraint remains in the constraints set.
  - iii. Positive Slack Amount(P): responsible for increasing room underuse.
  - iv. Negative Slack Amount(N): responsible for increasing room overuse.

The six datasets created by varying the above parameters are:

- i. **p000\_n025** (P = 0.0, N = 0.25)
- ii. **p025\_n000** (P = 0.25, N = 0.0)
- iii. **s000\_v000** (S = 0.0, V = 0.0)
- iv. **s100\_v000** (S = 1.0, V = 0.0)
- v. **s000\_v100** (S = 0.0, V = 1.25)
- vi. **s100\_v100** (S = 1.0, V = 1.0)

Table 2.2 Summary of all benchmark instances considered in this work

Instance	nott1		p000_n025		p025_n000		s000_v000		s100_v000		s000_v100		s100_v100	
Entities	158		150		150		150		150		150		150	
Rooms	131		92		92		92		92		92		92	
Constraints	272		263		263		252		252		281		281	
Type	H	S	H	S	H	S	H	S	H	S	H	S	H	S
Allocation		35		32		32		32		32		32		32
Not Allocation				10		10		10		10		10		10
Capacity			2	4	2	4	2	4	2	4	2	4	2	4
Same Room		20		25		25		25		25		25		25
Not Same Room				10		10		10		10		10		10
Not Sharing	100		60		60		60		60		60		60	
Adjacency	5	15	1	9	1	9	1	3	1	3	1	15	1	15
Nearby		77		93		93		90		90		103		103
Away From	6	14	4	13	4	13	4	11	4	11	4	15	4	15
Total	111	161	67	196	67	196	67	185	67	185	67	214	67	214

## 2.5 Conclusion

In this chapter, nine types of constraints were defined and classified as hard or soft in the context of this work. Then a mathematical definition of the evaluation function is provided. This is used to calculate the penalty of a feasible solution. A rigorous explanation of the benchmark instances was then outlined and the characteristics were summarised in Table 2.2. The information presented in this chapter will equip the readers with the necessary information required to understand the implementation of the OSA problem in this work. This chapter serves as the basis of the software library that will be constructed in Chapter 4.

## Chapter 3 LITERATURE REVIEW

### 3.1 Introduction

This chapter is organised as follows: Section 3.2 looks at the earliest works done on this problem. These are mostly based on exact methods. In Section 3.3 single-solution, population-based and hybrid heuristic methods are explored to solve the OSA problem. Section 3.4 contains the most recent works on this topic. A good mix of Memetic, Nature-Inspired and Agent-Based algorithms are utilized to solve the OSA problem here. A wide variety of previous work was analysed with a deep emphasis on the solution data structures, heuristic/metaheuristic algorithms used, constraints considered, benchmark datasets, technology used and any provisions implemented to target the feasible region of the search space. Section 3.5 provides a brief conclusion to this chapter.

### 3.2 Previous Research on Office Space Allocation (Pre-2000)

The earliest attempts to solve this problem involved the use of exact methods such as goal programming. Lee et al., 1972 proposes a goal programming model to tackle space allocation in one college with a planning horizon of 1 year. This basic model consists of 7 conflicting objectives where each of them is given a priority. Each of these sub goals has a positive and negative deviation associated with them. These deviations are part of the objective function and need to be minimised. The procedure was executed 3 times under different constraint/goals and priority levels. The authors highlighted the importance of goal programming as an approach as it allows the administrators to vary the input parameters. The limitation of this approach is the pre-requisite knowledge the administrator should have regarding priorities and constraints.

One of the earliest known detailed work on this topic is by (Ritzman et al., 1979). The authors decided to solve the space planning problem at Ohio State University using a mixed-integer goal programming approach coupled with an interactive computer program. This approach was utilised due to the multi-objective nature of the problem (6 conflicting objectives). These objectives were given equal

weightage by the committee. To make the process of computing solutions less expensive, a linear programming code was used as it gave integer solutions to most decision variables. The interactive program was used to evaluate the statistics of the computed solution and allowed the users to change the allocation to their needs. The algorithm was executed 16 times in total after conferring with the department heads after each iteration and was finally accepted by the committee. This work showed that the use of computer optimization methods greatly helps with negotiations and discourse.

Giannikos et al., 1995 implemented an integer pre-emptive goal programming model to solve the space allocation problem faced by the University of Westminster. They first created 3 staff categories and then created 5 conflicting objectives and 2 hard constraints to model this problem. Unlike (Ritzman et al., 1979), priority was assigned to these objectives. Also, unlike (Ritzman et al., 1979), an integer programming optimizer was used instead of a linear programming one. The methodology involved executing the optimizer on 3 different priority structures. The deviations inform the authorities regarding the satisfaction of the sub goals. The results were sent to the accommodation services that were responsible for the reorganization process at Westminster. The author provides two important observations in the conclusion. They stated that an interactive system is necessary to help administrators tackle this problem. Furthermore, they also mentioned the need for implementing heuristics to speed up the computation process.

A need for an automated space allocation system in universities led to a detailed survey by (Burke et al., 1998). This survey was sent to 96 universities of which 38 replied. The answers to the questionnaire showcased the size and diversity of the problem. Clear differences were highlighted in the constraints, number of buildings, guidelines used by both old and new universities etc. This paper also provided the different ways in which space allocation could be applied: (1) Starting from scratch and allocating departmental resources to a set of rooms. (2) Changing the number of rooms that can be occupied and hence changing the current allocation. (3) Shuffling the entities of the current allocation for further

optimization. The most important finding of this work was that a very small percentage of universities use computers for the process of automating space allocation. Finally, the authors concluded by stating that creating such a system is a feasible notion but immense care needs to be taken when considering the quality of the computed solutions, pre-requisite computing knowledge requirements and input test data.

Burke et al., 1999 followed (Burke et al., 1998) by using 3 metaheuristics (Hill Climbing, Simulated Annealing and Genetic Algorithm) on real allocation data from the University of Nottingham. The authors proposed a penalty function for an allocation that takes into account 3 things: the unscheduled resources, the space misuse (exceeding capacity or wasting space) and constraint violations. The constraints considered are of type sharing, grouping, adjacency and proximity. Each of them is provided with a weight (exponent and factor). The data consisted of 83 resources, 52 rooms and 69 constraints. In the Hill Climbing method, 3 functions were implemented: Allocate, Move and Swap. These allocate an unscheduled resource, move an allocated resource to a new room and swaps resources between two rooms respectively. In the Genetic Algorithm, genes represent rooms. Each gene has a linked list attached to it where each element is the resource allocated to it. It was found that Simulated Annealing gave the lowest penalty values while Random Fit Hill Climbing gave the highest. Furthermore, the conflicting nature of space utilization and constraint satisfaction was quite evident from the results.

### 3.3 Previous Research on Office Space Allocation (2000-2010)

Burke et al., 2000 proposed a computer system that uses heuristic/ metaheuristic techniques to automate the Office Space Allocation Problem. This system is a direct result of the research work carried out by the ASAP group (Burke et al., 1998) and (Burke et al., 1999). The system reads the 3 database files: resources, rooms and constraints which contain all the necessary attributes required to carry out the solution search. Then the administrator can choose the algorithm, vary its parameters and select the speed and depth of the search. The system provides the user with the allocation and a graphical output of the layout. It also allows the

user to view the fitness statistics of the allocation. Three data structures are used to implement the heuristics: (1) ResourceGene: Contains information regarding the resources (2) RoomGene: Contains information regarding the rooms (3)ConstraintGene: Contains information regarding the constraints. These are further linked by pointers. The evaluation function used is the same as (Burke et al., 1999). The system was tested on University of Nottingham data and it was found that the solution obtained by the Hill Climbing algorithm was an improvement over the then existing solution.

An in-depth analysis of the use of heuristics (Hill Climbing, Simulated Annealing and Genetic Algorithm) on this problem was carried out by (Burke et al., 2001a). The constraints considered are the same as (Burke et al., 1999) and can be hard or soft. The evaluation function is modified to include a soft constraint violation component instead of a resource conflict component. The data structure that represents an allocation is a list of rooms where each index is a resource. Neighbourhood exploration is done through Allocate, Move and Swap moves. Each of these moves can be random or best fit. The datasets were collected from the University of Nottingham, University of Wolverhampton and Nottingham Trent University. The experiment aimed to find the best algorithm for (1) reorganising an existing allocation (2) optimizing an existing allocation (3) Constructing a new allocation. For (1) and (2) Hill Climbing and Simulated Annealing using best fit gave the best results while for (3) Genetic Algorithm gave the best result but the manual solution was still better.

A need for hybrid metaheuristics to tackle this problem resulted in the paper proposed by (Burke et al., 2001b). The hybrid metaheuristic used here is a culmination of standard techniques used in (Burke et al., 2001a). Hill Climbing is utilized to construct a good allocation while an adaptive cooling schedule (simulated annealing) and mutation operator (genetic algorithm) helps to avoid becoming trapped in a local optimum. The data structure for the allocation consists of a mapping between two vectors representing resources and rooms respectively. The test data is acquired from the School of Computer Science at the University

of Nottingham. The experiment first involves a single-solution hybrid followed by a population based hybrid. Both these variants give better results than the individual standard techniques but have no clear advantage over each other.

Burke et al., 2001c combined the multi-objective nature of the OSA problem with the population based hybrid metaheuristics described above in (Burke et al., 2001b). The paper first shows how space utilisation and constraint satisfaction are conflicting in nature as betterment in one causes a decline of the other. The algorithm was used on the dataset with two fitness different functions: Aggregate Function and Dominance Ranking. Furthermore, two different strategies were used to get the output population: (1) Producing a single good quality allocation (less computation time) and (2) Producing multiple good quality allocations (more computation time). The authors felt that more work needed to be done in this area before such an approach could be efficiently accepted.

Continuing with research related to population based metaheuristics, (Landa-Silva and Burke 2007) implemented an asynchronous cooperative local search component in standard techniques such as Iterative Improvement, Simulated Annealing, Tabu Search and in a Single-Solution Hybrid Metaheuristic (Burke et al., 2001b). By using this component, individual local threads can share information regarding "good" and "bad" aspects of a solution. This information can then be used to improve as well as diversify each of the local threads. The evaluation function used is the same as (Landa-Silva, 2003) and 6 constraints are considered: Not sharing, Be allocated in, Be adjacent to, Be away from, Be together with, Be grouped with. The initial feasible solutions are created using the Peckish algorithm. The Nottingham and Trent datasets were utilized for the experiments. The algorithms were programmed using C++ 6.0. The results show that the asynchronous cooperative population based (PB) variants of the above-mentioned techniques give better solutions than their single solution counterparts. Among them, the Hybrid Metaheuristic(PB) gave the best performance followed by Tabu Search(PB).

An example of applying heuristics in a large organization in a non-specific industry is given by (Pereira et al., 2010). The authors used Greedy Search and Tabu Search in their work. Initial solutions are randomly constructed. Allocate and swap functions were used to explore the neighbourhood. The evaluation function consists of three objectives: (1) minimizing the distance between employees (2) minimizing the office space misuse (3) maximizing the number of empty rooms. These subobjectives were given a weight to prioritise them. Four test instances with varying sizes were generated for the experiment. Results show that Tabu Search performed better than Greedy Search in terms of solution quality as well as computational efficiency. Furthermore, the absence of an aspiration criteria in TS causes a slight worsening of solution quality.

A work that furthered the understanding of exact methods in the field of OSA was proposed by (Ulker and Landa-Silva, 2010). The authors proposed a 0/1 integer programming model. The objective function, constraints and penalty values are the same as (Landa-Silva, 2003) to assist in the comparison of the results. Six datasets were used (Nott a - Nott e, Wolver) and CPLEX 11 was deployed to solve the model. The aim of the experiments was to investigate the quality of solutions whilst varying the hardness/softness of the constraints involved. The authors reported their findings: (1) Keeping all constraints soft leads to high computation time. (2) Keeping all constraints hard leads to infeasible solutions. (3) Same room constraint is harder to optimise as compared to the Allocation constraint.

### 3.4 Previous Research on Office Space Allocation (2011-2023)

Ulker and Landa-Silva, 2011 formulated an improved 0/1 integer programming model as compared to (Ulker and Landa-Silva, 2010). This work extends the OSA problem as defined by (Landa-Silva, 2003) as the IP formulation provided here contains three new constraints: Non allocation, Not in same room and Capacity. All constraints are then mathematically modelled as hard and soft. The authors also built a test instance generator. This generator takes the required input parameters and then systematically creates entities, rooms, constraints and their respective attributes. The first experiment was aimed at investigating the best solution found by using the IP model described in this work. The Gurobi solver and



six real-world datasets (Nott1- Nott1e, Wolver) were used. The best solutions found had the lowest penalty in the literature so far. Furthermore, the analysis showed that the constraint penalty is much lower than the space misuse penalty. The second experiment focussed on gauging the complexity of the test instances by varying four parameters: (1) Slack Space Rate(S): regulates the number of rooms that will have their space modified. (2) Violation Rate(V): determines whether a violated soft constraint remains in the constraints set. (3) Positive Slack Amount: responsible for increasing room underuse. (4) Negative Slack Amount: responsible for increasing room overuse. The difficulty of the instances is highest when the percentage gap between the optimal solution penalty and the lower bound is large. This occurs when S and V values are high as well as when P and N values are near to each other.

Evolutionary Local Search Algorithms (ELS) or Memetic Algorithms are a class of metaheuristics that combine the diversification and intensification provided by Genetic Algorithms and Local Search Algorithms respectively. Ulker and Landa-Silva, 2012 was responsible for applying a variant of this algorithm to the OSA problem. The evolutionary part includes using crossover and mutation operators whereas the local search part includes applying the relocate operator to the solutions obtained from the first part. Arrays are used as data structures for the solution. The algorithm is programmed in C++ and the SVe150 and PNe150 datasets are used for the experiments (Ulker and Landa-Silva, 2011). The experiments focussed on finding the best parameter values (mutation rate, local search iteration, neighbourhood size and population size) for the ELS algorithm and then compared ELS performance with the IP model proposed (Ulker and Landa-Silva, 2011). The authors concluded by stating that low mutation rates and local search iterations gave better solutions. Furthermore, ELS performs better than IP on the majority of the datasets.

Awadallah et al., 2012 tackled the OSA problem by using a popular population based algorithm called Harmony Search Algorithm (HSA). The constraints and the evaluation function are taken from (Landa-Silva and Burke, 2007). The algorithm consists of 5 steps: (1) The information variables, objective function and the HSA

parameters (Harmony Memory Consideration Rate, Harmony Memory Size, Pitch Adjustment Rate and No. of Iterations) are initialised. (2) The initial feasible solution is constructed using the peckish method and stored in the Harmony Memory (HM). (3) The next move involves creating a new solution through three different operators: (i) Memory Consideration: chosen decision variable is assigned to the corresponding value present in the best solution in HM (ii) Random Consideration: chosen decision variable is assigned to a random room. (iii) Pitch Adjustment: The solution is locally improved using Move, Swap or Interchange functions. (4) If the constructed solution is an improvement over the worst solution in HM then it replaces it, otherwise it is rejected. (5) Stop the algorithm if NI is reached. The algorithm is programmed in C#. The experiment was aimed at finding good quality solutions alongside the best set of HSA parameters. The algorithm gave better results on smaller datasets (Nott1d and Wolver1) and performed satisfactorily on Nott1.

In recent years (Bolaji et al., 2017) proposed the Artificial Bee Colony algorithm to solve the OSA problem. This is a population based swarm intelligence algorithm. The constraints, their penalties and the objective function are taken from (Landa-Silva, 2003). The algorithm follows 5 steps: (1) All variables and parameters related to the ABC algorithm and the OSA problem are initialised. (2) A food source memory which consists of feasible solutions is initialised. (3) The employed bee operator selects each solution from the memory and applies the relocate, swap and interchange functions to explore their neighbourhood. Replacement occurs if a better solution is found. (4) The onlooker bee operator does the same but for only probabilistically selected solutions. (5) If the exploration fails, the scout bee operator randomly generates a new replacement solution. The algorithm was programmed using Basic .Net and the Nottingham and Wolverhampton datasets were used for the experiments. Firstly, the best values for the ABC parameters were determined using 9 scenarios. This was followed by comparing the performance of this algorithm with other algorithms found in the literature. The ABC algorithm produced new best results for the Nott1 and Nott1e datasets.

Dediu et al., 2018 described an agent based model to solve the OSA problem. The constraints and their penalties are in (Ulker and Landa-Silva, 2011) and the objective function is from (Landa-Silva, 2003). Here the entities are termed as active agents whereas the constraints and rooms are passive agents/objects. Each entity/agent has a satisfaction state associated with it which it decides through different strategies. If the agent is unsatisfied, it then moves to another room based on another set of strategies. These strategies check the amount of space misuse and constraint violation and are a mix of random and greedy approaches. Once these two steps are completed, the environment brings all the agents together to create a solution and calculates the fitness value. Since each step occurs simultaneously, it can be thought of as "distributed optimization". The work utilizes a subset of the SVe150 and PNe150 datasets (Ulker and Landa-Silva, 2011). The authors found that the combination of strategies that involved randomness worked better to find a good allocation. The same was seen when comparing neighbourhood exploration heuristics. This was attributed to the changing nature of the environment at each step and the lack of communication between the agents. An asynchronous approach was suggested to correct this in the future.

Bolaji et al. 2019 described a Late Acceptance Hill Climbing Algorithm to tackle the OSA problem. The constraints, penalties and objective function are from (Landa-Silva, 2003). This is a one-point search metaheuristic that keeps a memory list of the fitness values of past solutions. First, an initial feasible solution is created using the Peckish Algorithm. Then Relocate, Swap and Interchange operators are used to explore the neighbourhood of this solution. If the fitness of the candidate solution is greater than that of a solution  $t$  iterations ago ( $t = \text{iteration number mod memory list length}$ ) then it is accepted. This solution now becomes the current solution and the memory list is updated accordingly. This process is repeated till the stopping criterion is met. The Nottingham, Wolverhampton and Trent benchmark datasets are used for the experiments. The experiments concluded that using less than two neighbourhood search operators results in poor results. Additionally, LAHC achieves two new better results for the Nott1 and Wolver1 instances.

### 3.5 Conclusion

In this chapter, we saw the development of several methods used to confront the OSA problem. Starting from exact methods like mixed goal programming and integer goal programming to standard local search techniques followed by population-based metaheuristics and hybridizations. We also saw the problem statement become more detailed and complex with the addition of new constraints. The common aim of the researchers was to provide the space administration committee with a selection of good quality allocations which they can then use as a starting point.

## Chapter 4 IMPLEMENTATION

### 4.1 Introduction

In this chapter, we examine the implementation of the entire software library from the beginning. All programming is done using Python 3.8.3 and Jupyter Notebook. The algorithms implemented here are derived from (Landa-Silva, 2003). Certain aspects have been modified based on information presented in (Ulker, 2013), which will be explained later on. The main aim here is to provide a beginner in the field of office space allocation a good starting point, in terms of a code repository filled with baseline heuristics used to tackle this problem.

This chapter is organised as follows: Section 4.2 examines the data structures used to represent an allocation/solution. Section 4.3 describes the penalty function used in the implemented algorithms. Section 4.4 looks at the way input parameters of the OSA problem are extracted from the datasets. Section 4.5 is on the various constructive heuristics used to initialise a solution. Section 4.6 describes the neighbourhood exploration operators used in this work. Section 4.7 describes the exploration driving algorithms implemented in this library. Section 4.8 examines the output functionality implemented. Finally, Section 4.9 concludes this chapter.

### 4.2 Solution Representation and Data Structures

Python's Pandas Dataframe object is used to contain information regarding Entities, Rooms and Constraints as described in Section 2.4. These dataframes have a row-column structure and are 2-D in shape.

Several data structures are used in the literature for solutions as outlined in Chapter 3 (vectors, linked lists, arrays etc.). The data structure used to represent allocations of rooms and entities in this work is a Python NumPy 1-D Array. Each index represents an entity and each element at that index represents the room allocated to that entity.

E1	E2	E3	E4	E5	E6	E7
R5	R1	R2	R3	R1	R6	R4

Figure 4.1 Data structure utilized to represent allocations

### 4.3 Penalty Evaluation Function

The objective function or penalty evaluation function or cost function is described in Section 2.3. This function is used to calculate the penalty associated with a feasible allocation. However, in this implementation, feasibility is not guaranteed at every stage of the process. Sometimes a better infeasible solution has to be chosen during exploration as a way to reach the feasible search space. To calculate the penalty associated with infeasible solutions, a hard constraint penalty component is added to this evaluation function. For feasible solutions, this component becomes zero.

$$\sum_{i=1}^m (WP(i) + OP(i)) + \sum_{r=1}^s SCP(r) + \sum_{r=1}^h HCP(r) \quad (4.1)$$

$h$  = number of hard constraint penalties

$HCP(r)$  = Penalty due to violating the  $r^{th}$  hard constraint

The definitions of  $m$ ,  $s$ ,  $WP$ ,  $OP$  and  $SCP$  are mentioned in Section 2.3.

### 4.4 Input Functionality

Before initialising solutions and exploring neighbourhoods, the information regarding entities, rooms and constraints has to be read in the program. All Information regarding 1 test instance is given in 1 file. There are 7 files in total. Figures of how the data looks are presented in Appendix A. The input code is explained in more detail in Section D.4 of the documentation provided in the supplementary material.

Input: Text file

Output: Three Python Dataframes (entities, rooms and constraints)

1. Open file using File Handling options
2. Remove whitespace from line
3. If line under Entities Section
  - 3.1. Add row data to Entities list
4. If line under Rooms Section
  - 4.1. Add row data to Rooms list
5. If line under Constraints Section
  - 5.1. Add row data to Constraints list
6. Go to 2 until whole file is traversed
7. Initialize the dataframes using the lists

Figure 4.2 Input function of the OSA library. Data is traversed section-by-section and the rows are appended to the appropriate dataframes.

The following characteristics were kept in mind while creating this functionality:

1. File Format: The test instances are in text format.
2. Delimiters: There are spaces between columns to denote separation between attributes.
3. Headers: Although section headers are present, no column headers are present. Hence they were manually added in afterwards.
4. Data Types: Python reads all information in and gives it the object datatype. Hence object to float datatype conversion had to be carried out.
5. Additional Data: Data regarding constraint penalty is added to the Constraints Dataframe.

## 4.5 Constructive Heuristics

The construction of an initial solution can occur in many ways. The methods used here are derived from (Landa-Silva, 2003). These methods vary in terms of the level of greediness at which they operate. Many works in the literature used the greedy Peckish Heuristic (Landa-Silva and Burke, 2007), (Awadallah et al., 2012), (Bolaji et al., 2017) and (Bolaji et al., 2019) to construct initial solutions. Periera et al., 2010 used random initialisation while (Burke et al., 2001b) used a hill climbing heuristic to construct initial solutions.

As the number of hard constraints rises, it becomes more and more difficult to guarantee the feasibility of initial solutions. It becomes more logical to construct a good initial solution and allow the driving metaheuristic to find the best feasible solution rather than feeding it a feasible solution from the beginning. This also helps in widening the search space and increases the diversity of the explored allocations.

The methods implemented in this software library are given below. The code is explained in more detail in Section D.5 of the documentation provided in the supplementary material.

**AllocateRnd-Rnd:** An unallocated entity is selected at random and then a random room is allocated to it.

Input: Empty 1-D NumPy Array

Output: Initial Allocation, Penalty

1. Initialize list of unallocated entities
2. Select random entity from list
3. Select random room
4. Allocate entity to room
5. Remove entity from unallocated list
6. Go to step 2 till Length(unallocated list) = 0

Figure 4.3 AllocateRnd-Rnd is a random initialisation heuristic that places a random entity in a random room iteratively till allocation is constructed.

**AllocateRnd-BestRnd:** An unallocated entity and a subset of rooms are chosen at random. This entity is placed in each room and the penalty is calculated. The room which provides the least solution penalty is selected for allocation.



Input: Empty 1-D NumPy Array

Output: Initial Allocation, Penalty

1. Initialize list of unallocated entities
2. Select random entity from list
3. Construct random room subset
4. Best Penalty = +inf
5. For each room in subset
  - 5.1. Temporarily allocate entity to room
  - 5.2. Calculate Penalty
  - 5.3. If Penalty < Best Penalty
    - 5.3.1. Best Penalty = Penalty
6. Allocate entity to best room
7. Remove entity from unallocated list
8. Go to step 2 till Length(unallocated list) = 0

Figure 4.4 In AllocateRnd-BestRnd entity is chosen randomly. Then the algorithm iterates through a subset of rooms and calculates the penalty. The combination which provides the least Best Penalty is selected. Repeat till constructed.

**Allocate Wgt-BestRnd:** The unallocated entities are sorted in decreasing order based on the area they cover. The largest unallocated entity and a subset of rooms is chosen at random. This entity is placed in each room and the penalty is calculated. The room which provides the least solution penalty is selected for allocation.

Input: Empty 1-D NumPy Array

Output: Initial Allocation, Penalty

1. Initialize and sort list of unallocated entities on space
2. Select first entity from list
3. Construct random room subset
4. Best Penalty = +inf
5. For each room in subset
  - 5.1. Temporarily allocate entity to room
  - 5.2. Calculate Penalty
  - 5.3. If Penalty < Best Penalty
    - 5.3.1. Best Penalty = Penalty
6. Allocate entity to best room
7. Pop first element from unallocated list
8. Go to step 2 till Length(sorted unallocated list) = 0

Figure 4.5 In AllocateWgt-BestRnd the unallocated entity with the highest space is selected. The algorithm iterates through a subset of rooms and calculates the penalty. The combination which provides the least Best Penalty is selected. Repeat till constructed.

**AllocateCsrt-BestRnd:** In the datasets, among the hard constraints, the number of Not Sharing constraints is high. In this constructive heuristic, the entities subjected to these constraints are allocated first to satisfy them. The rest of the unallocated entities are allocated the same way as AllocateRnd-BestRnd. This increases the chance of constructing an initial feasible solution.

Input: Empty 1-D NumPy Array

Output: Initial Allocation, Penalty

1. Initialize list of unallocated entities
2. Initialize list of rooms
3. For each unallocated entity affected by Not Sharing constraint
  - 3.1. Construct random room subset
  - 3.2. Best Penalty = +inf
  - 3.3. For each room in subset
    - 3.3.1. Temporarily allocate entity to room
    - 3.3.2. Calculate Penalty
    - 3.3.3. If Penalty < Best Penalty
      - 3.3.3.1. Best Penalty = Penalty
  - 3.4. Allocate entity to best room
  - 3.5. Remove entity from unallocated list
  - 3.6. Remove room from room list
4. Select random entity from list
5. Construct random room subset from room list
6. Best Penalty = +inf
7. For each room in subset
  - 7.1. Temporarily allocate entity to room
  - 7.2. Calculate Penalty
  - 7.3. If Penalty < Best Penalty
    - 7.3.1. Best Penalty = Penalty
8. Allocate entity to best room
9. Remove entity from unallocated list
10. Go to step 4 till Length(unallocated list) = 0

Figure 4.6 In AllocateCsrt-BestRnd the entities affected by the Not Sharing constraints are first allocated. The entities and rooms are removed from circulation. Then the unallocated entities are placed in the same manner as AllocateRnd-BestRnd algorithm. Repeat till constructed.

**AllocateBestAll:** Among all the unallocated entities and rooms, the combination that provides the least solution penalty is implemented. This is repeated till all entities are allocated.

Input: Empty 1-D NumPy Array

Output: Initial Allocation, Penalty

1. Initialize list of unallocated entities
2. Initialize list of rooms
3.  $len = \text{Length}(\text{unallocated entities})$
4. While  $len$  not equal to 0
  - 4.1. Best Penalty =  $+\infty$
  - 4.2. For entity in unallocated list
    - 4.2.1. For room in room list
      - 4.2.1.1. Temporarily allocate entity to room
      - 4.2.1.2. Calculate Penalty
      - 4.2.1.3. If  $\text{Penalty} < \text{Best Penalty}$ 
        - 4.2.1.3.1. Best Penalty = Penalty
  - 4.3. Allocate best entity to best room
  - 4.4. Remove entity from unallocated list
  - 4.5.  $len = len - 1$

Figure 4.7 AllocateBestAll is a greedy initialisation algorithm. Penalties for all entity-room combinations are calculated. The one which provides the least Best Penalty value is implemented. Repeat till constructed.

## 4.6 Neighbourhood Exploration Operators

Once a solution is initialised through a constructive heuristic, the next step is to explore the neighbourhood of that solution. This is done by varying the ordering or changing the elements of the existing solution through various operators. This is a necessary step in any search heuristic as the main aim is to find a solution that gives the lowest penalty. The only way to do so is through a trial and error neighbourhood search process.

The Relocate, Swap and Interchange operators are used in many past works on this topic (Burke et al., 1999), (Burke et al., 2001a), (Landa-Silva, 2003), (Landa-Silva and Burke, 2007), (Pereira et al., 2010), (Ulker and Landa-Silva, 2012), (Awadallah et al., 2012), (Ulker, 2013) and (Dediu et al., 2018). In this work, we only consider the Relocate and Swap operator along with their respective random and greedy variants.

1. **Relocate**: Responsible for changing the room allocated to an entity.

a. **Relocate Rnd-Rnd**: Selects an entity and a room at random and then proceeds to allocate the entity to this new room.

Input: Current Allocation, Current Penalty

Output: Candidate Allocation, Candidate Penalty

1. Select random entity
2. Select random room
3. CandidateAllocation = CurrentAllocation
4. CandidateAllocation[random entity] = random room
5. Calculate Penalty (CandidateAllocation)

Figure 4.8 Relocate Rnd-Rnd algorithm is used as part of the IIRnd-Rnd and SAARnd-Rnd heuristics. A copy of the current allocation with the relocate changes is created and its penalty is calculated. This candidate allocation is sent to the calling heuristic.

b. **Relocate Rnd-BestRnd**: Selects an entity and a subset of rooms at random and then proceeds to allocate the entity to the room in the subset which provides the most decrease in the evaluation penalty function.

Input: Current Allocation, Current Penalty

Output: Candidate Allocation, Candidate Penalty

1. Select random entity
2. Construct random room subset
3. For each room in subset
  - 3.1. CandidateAllocation = CurrentAllocation
  - 3.2. CandidateAllocation[random entity] = room
  - 3.3. Calculate Penalty (CandidateAllocation)
  - 3.4. Save (room, Penalty) in list
4. Select element from list with minimum penalty
5. CandidateAllocation[random entity] = best room
6. Calculate Penalty(CandidateAllocation)

Figure 4.9 Relocate Rnd-BestRnd algorithm is used as part of the IIRnd-BestRnd and SAARnd-BestRnd heuristics. A copy of the current allocation with the relocate changes is created for each room in a subset. The candidate allocation with the lowest penalty is selected and sent to the calling heuristic.

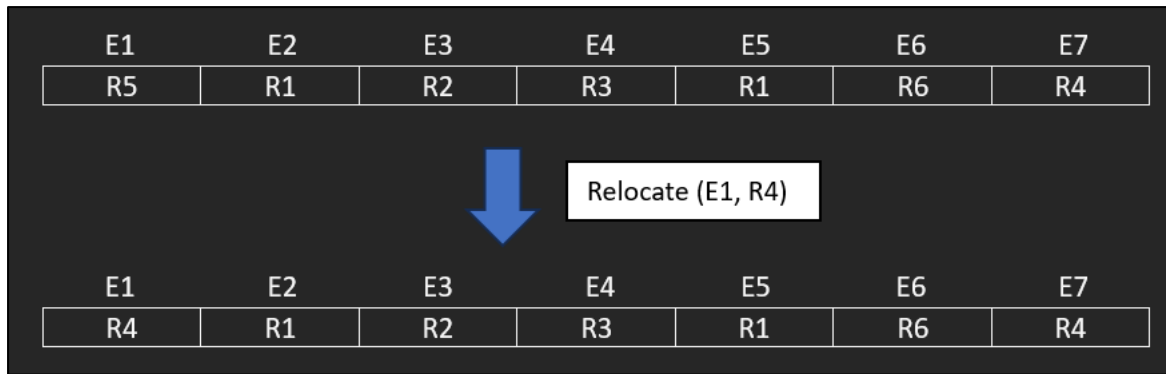


Figure 4.10 Mechanism of Relocate operation

2. **Swap**: Responsible for selecting two entities and then proceeds to swap the rooms allocated to them.

a. **Swap Rnd-Rnd**: Selects two entities at random and then swaps the rooms allocated to them.

Input: Current Allocation, Current Penalty

Output: Candidate Allocation, Candidate Penalty

1. Select first random entity
2. Select second random entity
3. CandidateAllocation = CurrentAllocation
4. CandidateAllocation[random entity1], CandidateAllocation[random entity2] = CandidateAllocation[random entity2], CandidateAllocation[random entity1]
5. Calculate Penalty (CandidateAllocation)

Figure 4.11 Swap Rnd-Rnd algorithm is used as part of the IIARnd-Rnd and SAARnd-Rnd heuristics. A copy of the current allocation with the swap changes is created and its penalty is calculated. This candidate allocation is sent to the calling heuristic.

b. **Swap Rnd-BestRnd**: The first entity is selected at random and then the second entity needed for the swap is chosen from a random subset. This second entity would be allocated a room that when swapped will provide the most decrease in the penalty function among all other entities in the subset.

Input: Current Allocation, Current Penalty

Output: Candidate Allocation, Candidate Penalty

1. Select first entity randomly
2. Construct a second entity random subset
3. For each entity in subset
  - 3.1. CandidateAllocation = CurrentAllocation
  - 3.2. CandidateAllocation[random entity1], CandidateAllocation[entity2] = CandidateAllocation[entity2], CandidateAllocation[random entity1]
  - 3.3. Calculate Penalty (CandidateAllocation)
  - 3.4. Save (entity2, Penalty) in list
4. Select element from list with minimum penalty
5. CandidateAllocation[random entity1], CandidateAllocation[best entity2] = CandidateAllocation[best entity2], CandidateAllocation[random entity1]
6. Calculate Penalty(CandidateAllocation)

Figure 4.12 Swap Rnd-BestRnd algorithm is used as part of the IIARnd-BestRnd and SAARnd-BestRnd heuristics. A copy of the current allocation with the relocate changes is created for each room in a subset. The candidate allocation with the lowest penalty is selected and sent to the calling heuristic.

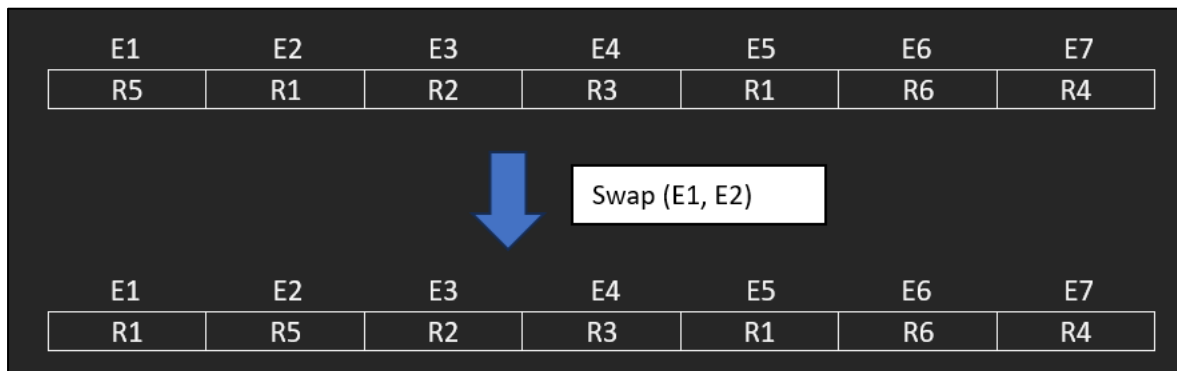


Figure 4.13 Mechanism of Swap operation

The Interchange operator selects two rooms and then swaps all entities allocated to these rooms. This operator is not considered in this work as the other two neighbourhood operators are enough to cause disruptions and explore the search space (Ulker, 2013) and (Bolaji et al., 2019). An important role for such operators is to help the driving metaheuristic avoid local optimal solutions. This is achieved by using only Relocate and Swap.

## 4.7 Driving Metaheuristics

In this work, the Hill Climbing Algorithm, Simulated Annealing Algorithm and their random and greedy variants are examined in detail. These are standard techniques that work well with combinatorial optimization problems. The code is explained in more detail in Section D.6 of the documentation provided in the supplementary material.

### 4.7.1 Hill Climbing/ Iterative Improvement Algorithm

Once an initial solution is constructed, its neighbourhood is explored. A candidate solution found is accepted by this algorithm only if it is an improvement over the previous solution. In the context of this work, an improvement refers to a lesser penalty value. Variety can be incorporated into this algorithm by changing the neighbourhood exploration operator.

#### 4.7.1.1 Iterative Improvement Algorithm Rnd-Rnd

Here the randomised version of neighbourhood operators were used (Relocate Rnd-Rnd and Swap Rnd-Rnd) to explore the search space. The pseudo code is given below.



Input: Current Allocation, Current Penalty

Output: Best Feasible Allocation, Best Penalty

1. CurrentAllocation = Initial Solution, CurrentPenalty = Initial Solution Penalty
2. Best Penalty = +inf
3. While iter not equal to MaxIter:
  - 3.1. Choose randomly (RelocateRnd\_Rnd(), SwapRnd\_Rnd())
  - 3.2. If choice = RelocateRnd\_Rnd():
    - 3.2.1. RelocateRnd\_Rnd(CurrentAllocation, CurrentPenalty)
    - 3.2.2. If Candidate Penalty < Best Penalty:
      - 3.2.2.1. Best Penalty = Candidate Penalty
      - 3.2.2.2. CurrentAllocation = CandidateAllocation
  - 3.3. Else:
    - 3.3.1. SwapRnd\_Rnd(CurrentAllocation, CurrentPenalty)
    - 3.3.2. If Candidate Penalty < Best Penalty:
      - 3.3.2.1. Best Penalty = Candidate Penalty
      - 3.3.2.2. CurrentAllocation = CandidateAllocation
  - 3.4. If CurrentAllocation is Feasible:
    - 3.4.1. Save (CurrentAllocation, Best Penalty) in list
  - 3.5. iter = iter + 1
4. Return element from list with minimum penalty

Figure 4.14 In IIRnd-Rnd algorithm the neighbourhood is explored by using RelocateRnd-Rnd or SwapRnd-Rnd. The candidate allocation returned by them is tested against the best allocation obtained by IIRnd-Rnd until then. If it is better, then the candidate allocation becomes the best allocation. Parallely, the best feasible allocation is being noted too.

#### 4.7.1.2 Iterative Improvement Algorithm Rnd-BestRnd

Here the greedy version of neighbourhood operators were used (Relocate Rnd-BestRnd and Swap Rnd-BestRnd) to explore the search space. The pseudo code is given below.

Input: Current Allocation, Current Penalty

Output: Best Feasible Allocation, Best Penalty

1. CurrentAllocation = Initial Solution, CurrentPenalty = Initial Solution Penalty
2. Best Penalty = +inf
3. While iter not equal to MaxIter:
  - 3.1. Choose randomly (RelocateRnd\_BestRnd(), SwapRnd\_BestRnd())
  - 3.2. If choice = RelocateRnd\_BestRnd():
    - 3.2.1. RelocateRnd\_BestRnd(Current Allocation, Current Penalty)
    - 3.2.2. If Candidate Penalty < Best Penalty:
      - 3.2.2.1. Best Penalty = Candidate Penalty
      - 3.2.2.2. CurrentAllocation = CandidateAllocation
  - 3.3. Else:
    - 3.3.1. SwapRnd\_BestRnd(CurrentAllocation, CurrentPenalty)
    - 3.3.2. If Candidate Penalty < Best Penalty:
      - 3.3.2.1. Best Penalty = Candidate Penalty
      - 3.3.2.2. CurrentAllocation = CandidateAllocation
  - 3.4. If CurrentAllocation is Feasible:
    - 3.4.1. Save (CurrentAllocation, Best Penalty) in list
  - 3.5. iter = iter + 1
4. Return element from list with minimum penalty

Figure 4.15 In IIARnd-BestRnd algorithm the neighbourhood is explored by using RelocateRnd-BestRnd or SwapRnd-BestRnd. The candidate allocation returned by them is tested against the best allocation obtained by IIARnd-BestRnd until then. If it is better, then the candidate allocation becomes the best allocation. Parallely, the best feasible allocation is being noted too.

#### 4.7.2 Simulated Annealing Algorithm

An issue that arises from the standard hill climbing algorithm is that it is prone to get stuck in a local optimum as it only accepts better solutions. Simulated Annealing (Kirkpatrick et al., 1983) is a way around this problem. In this algorithm, a better solution is accepted while a worse solution is also accepted based on a calculated probability value (acceptance probability). This probability depends on a temperature component

Acceptance Probability =  $\exp(-\Delta F/T_i)$  where  $\Delta F = \text{fitness}(x') - \text{fitness}(x)$  and  $T_i$  is the current temperature

Initially, the temperature is high and hence the probability of accepting a worse solution is also high. An arithmetic cooling schedule is implemented which gradually reduces the temperature which in turn reduces the probability of accepting worse solutions. By the end, the temperature becomes zero and this algorithm operates in the same way as the iterative improvement algorithm.

#### 4.7.2.1 Simulated Annealing Algorithm Rnd-Rnd

Here the randomised version of neighbourhood operators were used (Relocate Rnd-Rnd and Swap Rnd-Rnd) to explore the search space. The pseudo code is given below.

Input: Current Allocation, Current Penalty

Output: Best Feasible Allocation, Penalty

1. CurrentAllocation = Initial Solution, CurrentPenalty = Initial Solution Penalty
2. T = Initial Temperature
3. While iter not equal to MaxIter:
  - 3.1. Choose randomly (RelocateRnd\_Rnd(), SwapRnd\_Rnd())
  - 3.2. If choice = RelocateRnd\_Rnd():
    - 3.2.1. RelocateRnd\_Rnd(CurrentAllocation, CurrentPenalty)
    - 3.2.2. If Candidate Penalty < Current Penalty:
      - 3.2.2.1. Current Penalty = Candidate Penalty
      - 3.2.2.2. CurrentAllocation = CandidateAllocation
    - 3.2.3. If Candidate Penalty > Current Penalty:
      - 3.2.3.1.  $\Delta P = \text{Candidate Penalty} - \text{Current Penalty}$
      - 3.2.3.2.  $AP = \exp(-\Delta P/T)$
      - 3.2.3.3. If  $AP > \text{random}[0,1]$ :
        - 3.2.3.3.1. Current Penalty = Candidate Penalty
        - 3.2.3.3.2. CurrentAllocation = CandidateAllocation
  - 3.3. Else:
    - 3.3.1. SwapRnd\_Rnd(Current Allocation, Current Penalty)
    - 3.3.2. If Candidate Penalty < Current Penalty:
      - 3.3.2.1. Current Penalty = Candidate Penalty
      - 3.3.2.2. CurrentAllocation = CandidateAllocation
    - 3.3.3. If Candidate Penalty > Current Penalty:
      - 3.3.3.1.  $\Delta P = \text{Candidate Penalty} - \text{Current Penalty}$
      - 3.3.3.2.  $AP = \exp(-\Delta P/T)$
      - 3.3.3.3. If  $AP > \text{random}[0,1]$ :
        - 3.3.3.3.1. Current Penalty = Candidate Penalty
        - 3.3.3.3.2. CurrentAllocation = CandidateAllocation
  - 3.4. If CurrentAllocation is Feasible:
    - 3.4.1. Save (CurrentAllocation, Current Penalty) in list
  - 3.5. Decrease T (arithmetic cooling schedule)
  - 3.6. iter = iter + 1
4. Return element from list with minimum penalty

Figure 4.16 In SAARnd-Rnd algorithm the neighbourhood is explored by using RelocateRnd-Rnd or SwapRnd-Rnd. The candidate allocation returned by them is tested against the current allocation obtained by SAARnd-Rnd until then. If it is better or if it is worse but the acceptance probability calculated is sufficient, then the candidate allocation becomes the current allocation. Parallely, temperature is being lowered and the best feasible allocation is being noted too.

#### 4.7.2.2 Simulated Annealing Algorithm Rnd-BestRnd

Here the randomised version of neighbourhood operators were used (Relocate Rnd-BestRnd and Swap Rnd-BestRnd) to explore the search space. The pseudo code is given below.

```
Input: Current Allocation, Current Penalty

Output: Best Feasible Allocation, Penalty

1. CurrentAllocation = Initial Solution, CurrentPenalty = Initial Solution Penalty
2. T = Initial Temperature
3. While iter not equal to MaxIter:
    3.1. Choose randomly (RelocateRnd_BestRnd(), SwapRnd_BestRnd())
    3.2. If choice = RelocateRnd_BestRnd():
        3.2.1. RelocateRnd_BestRnd(CurrentAllocation, CurrentPenalty)
        3.2.2. If Candidate Penalty < Current Penalty:
            3.2.2.1. Current Penalty = Candidate Penalty
            3.2.2.2. CurrentAllocation = CandidateAllocation
        3.2.3. If Candidate Penalty > Current Penalty:
            3.2.3.1.  $\Delta P = \text{Candidate Penalty} - \text{Current Penalty}$ 
            3.2.3.2.  $AP = \exp(-\Delta P/T)$ 
            3.2.3.3. If  $AP > \text{random}[0,1]$ :
                3.2.3.3.1. Current Penalty = Candidate Penalty
                3.2.3.3.2. CurrentAllocation = CandidateAllocation

    3.3. Else:
        3.3.1. SwapRnd_Rnd(CurrentAllocation, CurrentPenalty)
        3.3.2. If Candidate Penalty < Current Penalty:
            3.3.2.1. Current Penalty = Candidate Penalty
            3.3.2.2. CurrentAllocation = CandidateAllocation
        3.3.3. If Candidate Penalty > Current Penalty:
            3.3.3.1.  $\Delta P = \text{Candidate Penalty} - \text{Current Penalty}$ 
            3.3.3.2.  $AP = \exp(-\Delta P/T)$ 
            3.3.3.3. If  $AP > \text{random}[0,1]$ :
                3.3.3.3.1. Current Penalty = Candidate Penalty
                3.3.3.3.2. CurrentAllocation = CandidateAllocation

    3.4. If CurrentAllocation is Feasible:
        3.4.1. Save (CurrentAllocation, CurrentPenalty) in list
    3.5. Decrease T (arithmetic cooling schedule)
    3.6. iter = iter + 1
4. Return element from list with minimum penalty
```

Figure 4.17 In SAARnd-BestRnd algorithm the neighbourhood is explored by using RelocateRnd-BestRnd or SwapRnd-BestRnd. The candidate allocation returned by them is

tested against the current allocation obtained by SAARnd-BestRnd until then. If it is better or if it is worse but the acceptance probability calculated is sufficient, then the candidate allocation becomes the current allocation. Parallely, temperature is being lowered and the best feasible allocation is being noted too.

## 4.8 Output Functionality

After successfully executing the search heuristics mentioned above, the best feasible solution is recorded. Post-optimality analysis is a very important process. This is implemented in this work through an output function that takes the recorded feasible allocation as input and prints out the following information to a text file:

1. The iteration number at which the best feasible allocation was encountered. (-1 if algorithm fails to find the feasible region)
2. The values of the different penalty components for the recorded best feasible solution i.e. total penalty, soft constraint violation penalty and space misuse penalty.
3. The Entity-Room mapping of the recorded best feasible solution.
4. The amount of space utilized and left for each room in the dataset.
5. The satisfaction/violation status for each constraint in the dataset.

This information allows the user to look at the allocation in more detail and hence can make manual changes to it if required. Figures of contents of this output text file are placed in Appendix B. The code is explained in more detail in Section D.7 of the documentation provided in the supplementary material.

## 4.9 Conclusion

In this chapter, the entire structure of the OSA library was explained. In total five constructive heuristics and two exploration driving metaheuristics were implemented. Variations are introduced in these methods based on the level of randomness/greediness. Pseudo code for each of them is given in the figures above. The input and output functionality helps the user to read in the dataset and perform post-optimality analysis respectively.

## Chapter 5 RESULTS AND DISCUSSION

### 5.1 Introduction

In this chapter, experiments are conducted by executing the algorithms implemented above on the datasets (see Section 2.4) to generate a variety of results. The results are then discussed in detail and are compared with past literature works on this problem.

This chapter is organised as follows: Section 5.2 looks at a comparison of solutions generated by the constructive heuristics. Section 5.3 looks at the effect of hard constraint penalty value on the driving metaheuristic output. Section 5.4 looks at the effect of quality of the initial solution on the driving metaheuristic output. Section 5.5 looks at the comparison between the solutions generated by the random and greedy variant of the driving metaheuristics implemented in Section 4.7. A brief conclusion is provided in Section 5.6.

All experiments are conducted on Windows 11 laptop with Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz and 8 GB RAM.

Table 5.1 Average time taken across all datasets to execute each constructive heuristic. Calculation is done under same experimental conditions as Section 5.2.

Constructive Heuristics	AllocateBestAll	AllocateRnd-Rnd	AllocateRnd-BestRnd	AllocateWgt-BestRnd	AllocateCsrt-BestRnd
Average Execution Time	~6h	160ms	16.5s	16.2s	18.4s

Table 5.2 Average time taken across all datasets to execute each driving metaheuristic. Calculation is done under same experimental conditions as Section 5.5.

Driving Metaheuristics	IIARnd-Rnd	SAARnd-Rnd	IIARnd-BestRnd	SAARnd-BestRnd
Average Execution Time	15min 19s	16min 52s	56min 31s	58min 10s

## 5.2 Constructive Heuristics Comparison

### Setup:

In this first set of experiments, the various initialisation heuristics as described in Section 4.5 are executed. Each heuristic is programmed to output 50 initial solutions. The minimum, average and maximum penalty (space misuse + soft constraint violation) are recorded and compared. Furthermore, the average number of hard constraints violated by each method is also noted. The results are provided in Tables 5.1 to 5.7. The hard constraint penalty per violation is 500. The length of the subset considered for the greedier variants is  $n/20$  where  $n$  is the number of entities in the dataset.

Table 5.3 Constructive Heuristics performance on nott dataset

Constructive Heuristics	Space Misuse + Soft Constraint Violation			No. of unsatisfied hard constraints (Average)	Average Hard Constraint Penalty
	Maximum	Average	Minimum		
AllocateBestAll	1221.0	1221.0	1221.0	6	3000
AllocateRnd-Rnd	8959.9	8256.1	6597.15	76.6	38300
AllocateRnd-BestRnd	6754.75	6240.59	4982.15	22.78	11390
AllocateWgt-BestRnd	5765.45	5079.075	3475.45	18.1	9050
AllocateCsrt-BestRnd	6572.5	5985.7	4435.55	6.96	3480



Table 5.4 Constructive Heuristics performance on p000\_n025 dataset

Constructive Heuristics	Space Misuse + Soft Constraint Violation			No. of unsatisfied hard constraints (Average)	Average Hard Constraint Penalty
	Maximum	Average	Minimum		
AllocateBestAll	3395.6	3395.6	3395.6	0	0
AllocateRnd-Rnd	5708.8	5288.96	4836.8	51.28	25640
AllocateRnd-BestRnd	4216.8	3560.79	2944.5	18.66	9330
AllocateWgt-BestRnd	3352.3	2968.96	2616.6	21.32	10660
AllocateCsrt-BestRnd	5371.4	5031.1	4439.6	2.05	1030

Table 5.5 Constructive Heuristics performance on p025\_n000 dataset

Constructive Heuristics	Space Misuse + Soft Constraint Violation			No. of unsatisfied hard constraints (Average)	Average Hard Constraint Penalty
	Maximum	Average	Minimum		
AllocateBestAll	3228.9	3228.9	3228.9	1	500
AllocateRnd-Rnd	6229.2	5221.03	4765.1	51.2	25600
AllocateRnd-BestRnd	4123.5	3484.48	3019.4	19.2	9600
AllocateWgt-BestRnd	3379.3	2847.57	2488.1	21.64	10820
AllocateCsrt-BestRnd	5373.4	4829.17	4031.1	1.78	890

Table 5.6 Constructive Heuristics performance on s000\_v000 dataset

Constructive Heuristics	Space Misuse + Soft Constraint Violation			No. of unsatisfied hard constraints (Average)	Average Hard Constraint Penalty
	Maximum	Average	Minimum		
AllocateBestAll	3028.5	3028.5	3028.5	1	500
AllocateRnd-Rnd	5728	5188.44	4567.5	50.94	25470
AllocateRnd-BestRnd	4233	3495.24	3072.5	19.3	9650
AllocateWgt-BestRnd	3275	2784.13	2355.5	21.54	10770
AllocateCsrt-BestRnd	5285	4675.45	3991	1.9	950

Table 5.7 Constructive Heuristics performance on s000\_v100 dataset

Constructive Heuristics	Space Misuse + Soft Constraint Violation			No. of unsatisfied hard constraints (Average)	Average Hard Constraint Penalty
	Maximum	Average	Minimum		
AllocateBestAll	3208.5	3208.5	3208.5	1	500
AllocateRnd-Rnd	5806	5366.66	4608	51.38	25690
AllocateRnd-BestRnd	4271.5	3666.55	3039	18.76	9380
AllocateWgt-BestRnd	3474.5	3020.55	2588	21.82	10910
AllocateCsrt-BestRnd	5353.5	4858.39	4191	1.98	990

Table 5.8 Constructive Heuristics performance on s100\_v000 dataset

Constructive Heuristics	Space Misuse + Soft Constraint Violation			No. of unsatisfied hard constraints (Average)	Average Hard Constraint Penalty
	Maximum	Average	Minimum		
AllocateBestAll	3138.1	3138.1	3138.1	1	500
AllocateRnd-Rnd	5709.3	5143.23	4493.5	51.18	25590
AllocateRnd-BestRnd	4040.1	3474.83	2901.6	19.58	9790
AllocateWgt-BestRnd	3195.3	2825.59	2434.1	22.02	11010
AllocateCsrt-BestRnd	5285.49	4858.97	4199.7	1.88	940

Table 5.9 Constructive Heuristics performance on s100\_v100 dataset

Constructive Heuristics	Space Misuse + Soft Constraint Violation			No. of unsatisfied hard constraints (Average)	Average Hard Constraint Penalty
	Maximum	Average	Minimum		
AllocateBestAll	3308.1	3308.1	3308.1	1	500
AllocateRnd-Rnd	6155.1	5320.35	4577.5	51.5	25750
AllocateRnd-BestRnd	4259.5	3631.67	2900.3	18.78	9390
AllocateWgt-BestRnd	3352.5	3000.71	2624.5	21.82	10910
AllocateCsrt-BestRnd	5387.2	5035.41	4322.9	1.66	830

## Discussion:

When comparing the average solution penalty of these constructive heuristics, differences between the nott dataset and the six s and p datasets can be seen.

For the (space misuse + soft constraint violation) component:

1. Nott: AllocateRnd-Rnd > AllocateRnd-BestRnd > AllocateCsrt-BestRnd > AllocateWgt-BestRnd
2. SVe150 and PNe150: AllocateRnd-Rnd > AllocateCsrt-BestRnd > AllocateRnd-BestRnd > AllocateWgt-BestRnd

For the hard constraint violation component:

1. Nott: AllocateRnd-Rnd > AllocateRnd-BestRnd > AllocateWgt-BestRnd > AllocateCsrt-BestRnd
2. S and P: AllocateRnd-Rnd > AllocateWgt-BestRnd > AllocateRnd-BestRnd > AllocateCsrt-BestRnd

It is no surprise that the AllocateRnd-Rnd algorithm provides the worst quality of solutions across all datasets. The solutions it provides also have the highest number of hard constraint violations. The lowest number of hard constraint violations comes from the AllocateBestAll heuristic across all datasets. [For the p000\_n025 dataset it guarantees feasibility]. But it only provides us with one solution (lowest diversity) and hence isn't an optimal input choice for the neighbourhood exploration procedures. Furthermore, it has a huge time complexity of  $O(m \cdot n^2)$ . AllocateCsrt-BestRnd provides good quality solutions with low hard constraint violations. This is because the hard constraints are tackled first in this algorithm (Section 4.5)

We see that for all datasets, there doesn't exist a single heuristic that can guarantee feasibility. The aim is then to select such a constructive heuristic that provides a solution that:

1. Prevents the driving metaheuristic from becoming trapped in a local optima
2. Able to reach the feasible search space in a reasonable amount of iterations.

3. Able to stay in the feasible search space without bouncing back into infeasibility.
4. Provides good quality (low penalty) feasible solutions.

### 5.3 Effect of Hard Constraint Penalty Value

#### Setup:

In Section 4.3, a hard constraint penalty for an infeasible solution was discussed. Varying this penalty value can cause changes in the solution quality of the best feasible solution acquired. In this second set of experiments, AllocateRnd-BestRnd is selected as the constructive heuristic (length of subset =  $n/20$  where  $n$  is number of entities) and the random variant of the Iterative Improvement Algorithm (IIARnd-Rnd) is chosen as the driving metaheuristic. The experiment is repeated 20 times and the best feasible solution penalty is recorded and compared for each hard constraint penalty value. The stopping criterion for each run is 20000 iterations. The results are provided in Figures 5.1 to 5.7.

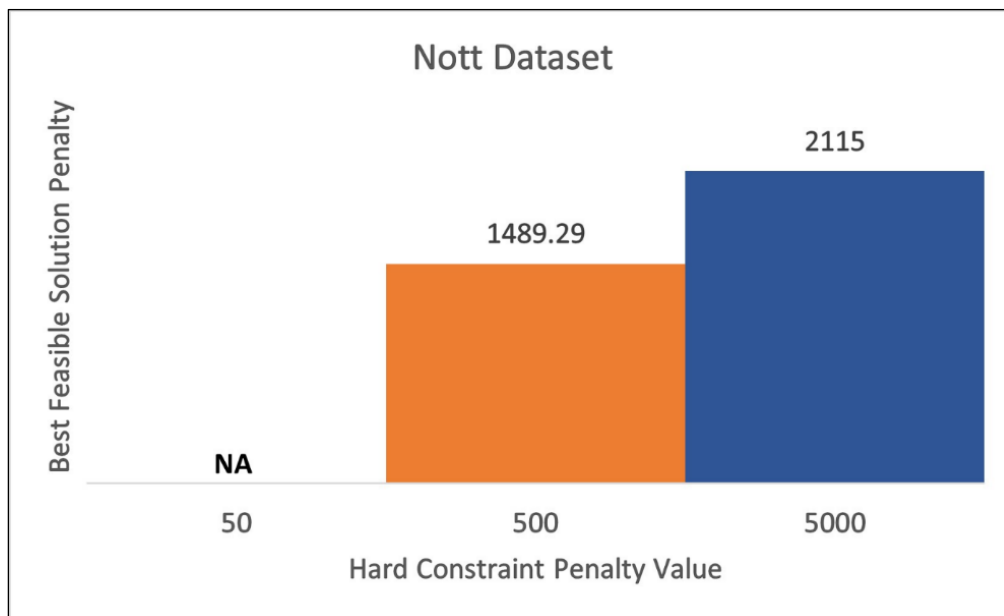


Figure 5.1 Varying hard constraint penalty value on the nott dataset

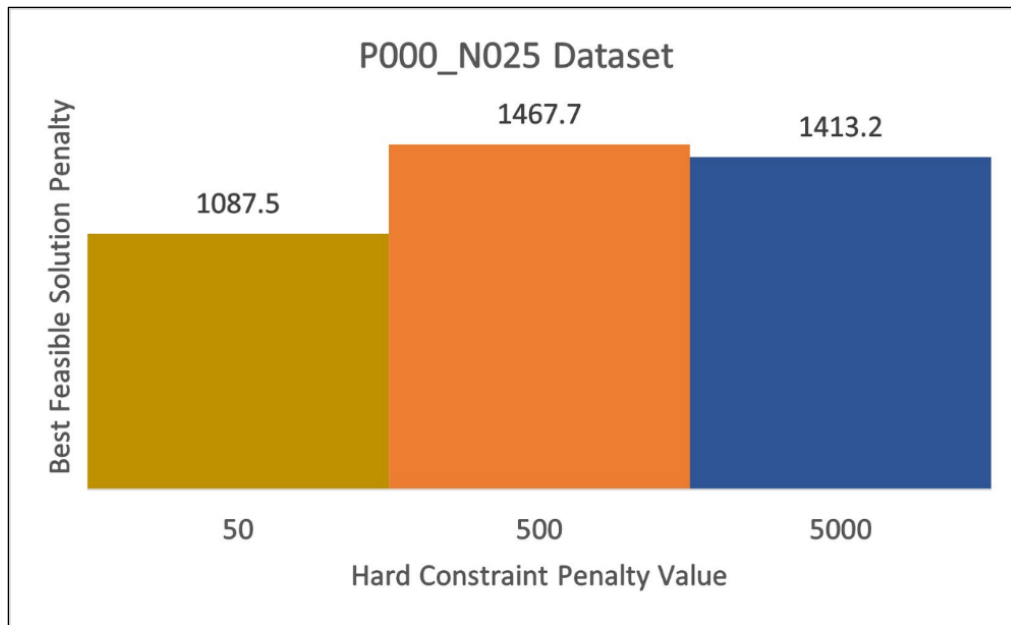


Figure 5.2 Varying hard constraint penalty value on the p000\_n025 dataset

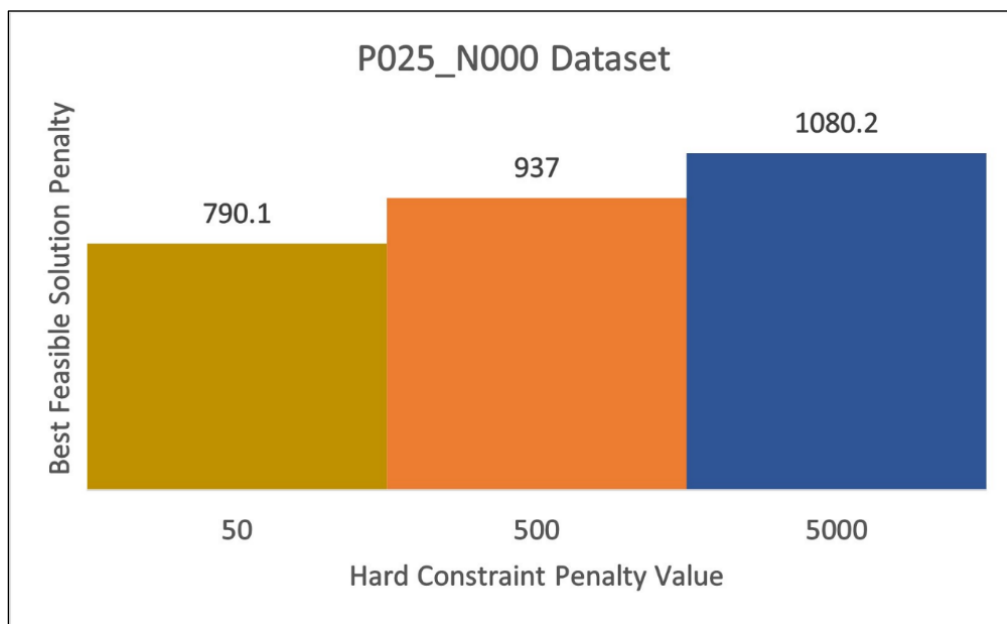


Figure 5.3 Varying hard constraint penalty value on the p025\_n000 dataset

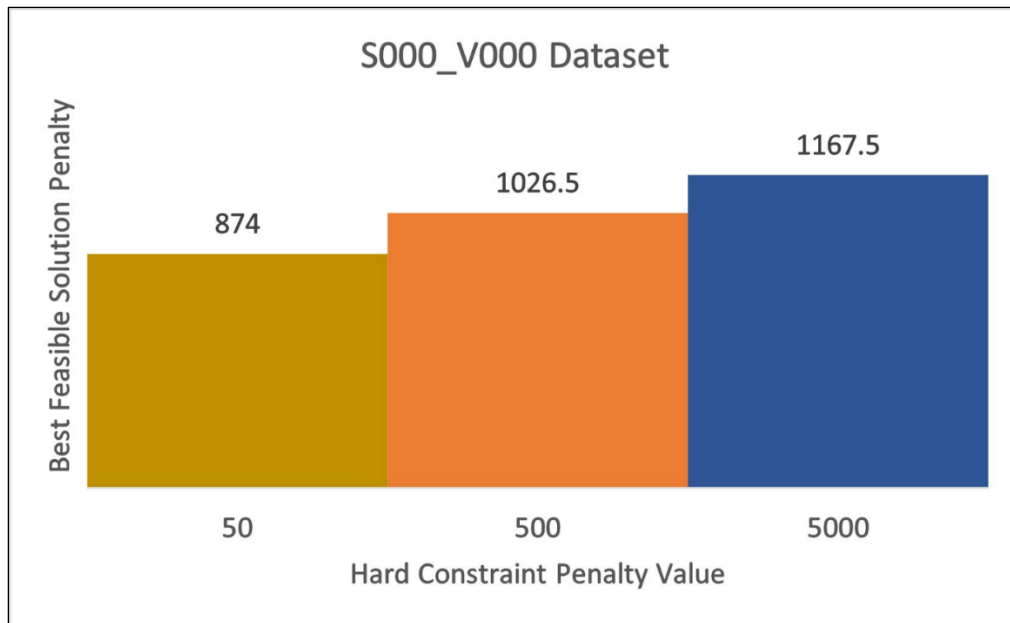


Figure 5.4 Varying hard constraint penalty value on the s000\_v000 dataset

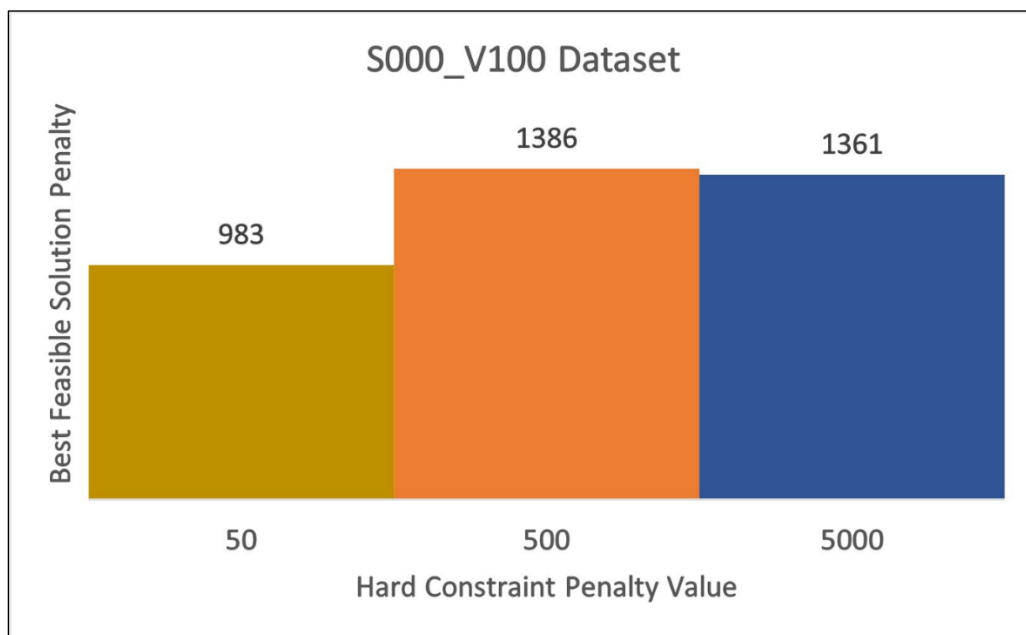


Figure 5.5 Varying hard constraint penalty value on the s000\_v100 dataset

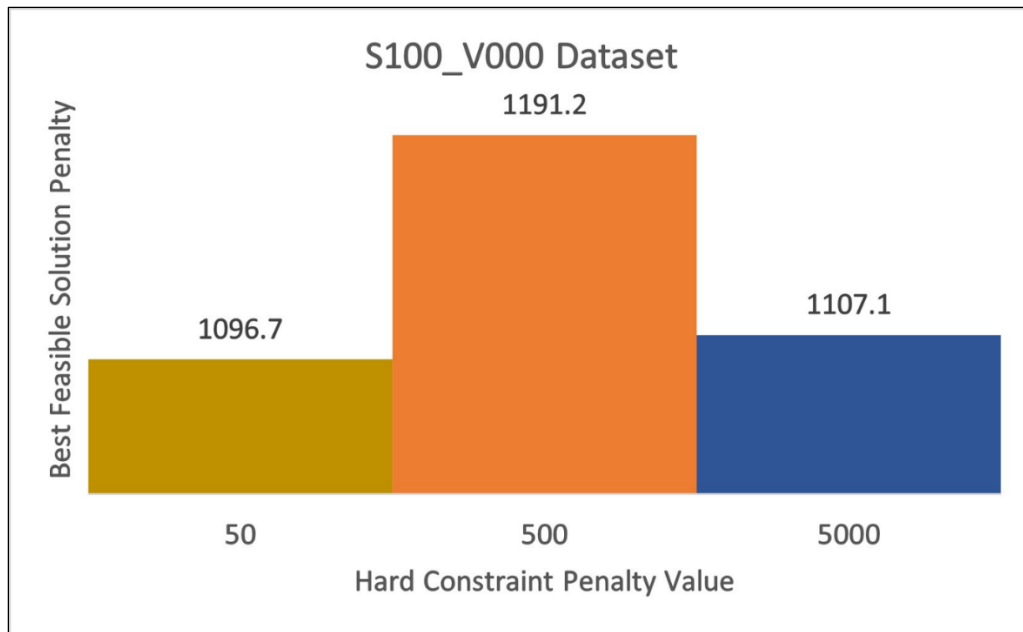


Figure 5.6 Varying hard constraint penalty value on the s100\_v000 dataset

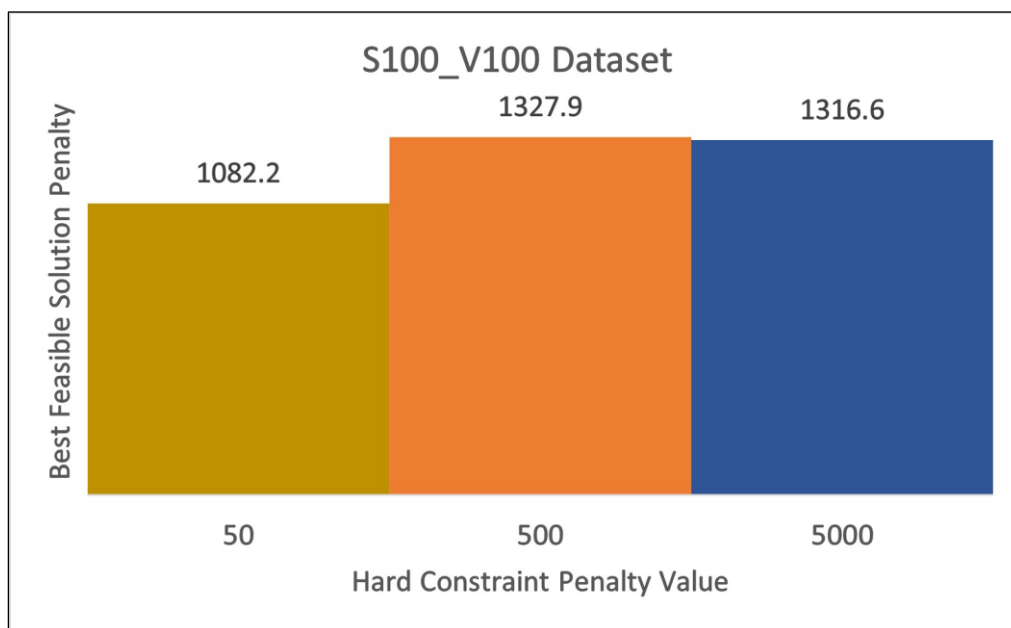


Figure 5.7 Varying hard constraint penalty value on the s100\_v100 dataset

### Discussion:

When the hard constraint penalty is low (near to the soft constraint penalties as seen in Figures 5.1 to 5.7), exploration is encouraged by the driving metaheuristics. The IIRnd-Rnd algorithm doesn't prioritise the elimination of



hard constraint violations. Rather it reduces all three components (space misuse, soft constraint violation and hard constraint violation) almost equally. Hence we see that when the hard constraint penalty is 50, we get the lowest feasible solution penalty. Although this value has its advantages, a glaring disadvantage is that around 60% of the time the algorithm is not able to converge towards the feasible search space. For the Nott dataset (Figure 5.1), there was no convergence for all 20 runs. Furthermore, when it does, it can easily bounce back into infeasibility due to the low hard constraint penalty values.

When the hard constraint penalty is 500, priority is given to the elimination of the hard constraint violations. Exploration still occurs, as seen from the decent quality of the best feasible solution in the figures above, but not as well as in the first case. A huge advantage here is that reaching feasibility is guaranteed and there is no chance of bouncing back towards infeasibility.

When the hard constraint penalty is 5000, no significant change is seen in the best feasible solution penalty achieved for each of the datasets. Hence, it seems that after a certain value of hard constraint penalty, increasing it will not affect the algorithms performance.

## 5.4 Effect of Initial Solution Quality

### **Setup:**

In this third set of experiments, the random variant of the iterative improvement algorithm (Section 4.7.1.1) is executed by using the output of two different constructive heuristics: AllocateRnd-BestRnd and AllocateCsrt-BestRnd (length of subset is  $n/20$  where  $n$  is the number of entities in the dataset). The hard constraint penalty per violation is 500. This experiment is repeated 20 times for each dataset. The stopping criterion for each run is 20000 iterations. The information regarding the first feasible solution iteration, best feasible solution iteration and best feasible solution penalty are recorded and compared. The aim of this experiment is to look at the effects of different qualities of initial solutions on the driving metaheuristics output. The results are provided in Tables 5.8 to 5.14.

Table 5.10 Effect of initial solution quality on the nott dataset

Constructive Heuristic	Driving Metaheuristic	First Feasible Solution Iteration	Best Feasible Solution Iteration	Best Feasible Solution Penalty
AllocateRnd- BestRnd	IIA Rnd-Rnd	12153	19793	1489.29
AllocateCsrt- BestRnd	IIA Rnd-Rnd	9286	19226	1285.6

Table 5.11 Effect of initial solution quality on the p000\_n025 dataset

Constructive Heuristic	Driving Metaheuristic	First Feasible Solution Iteration	Best Feasible Solution Iteration	Best Feasible Solution Penalty
AllocateRnd- BestRnd	IIA Rnd-Rnd	4994	19643	1467.7
AllocateCsrt- BestRnd	IIA Rnd-Rnd	393	19583	3576.1

Table 5.12 Effect of initial solution quality on the p025\_n000 dataset

Constructive Heuristic	Driving Metaheuristic	First Feasible Solution Iteration	Best Feasible Solution Iteration	Best Feasible Solution Penalty
AllocateRnd- BestRnd	IIA Rnd-Rnd	5859	19998	937
AllocateCsrt- BestRnd	IIA Rnd-Rnd	3709	19886	3482.3

Table 5.13 Effect of initial solution quality on the s000\_v000 dataset

Constructive Heuristic	Driving Metaheuristic	First Feasible Solution Iteration	Best Feasible Solution Iteration	Best Feasible Solution Penalty
AllocateRnd- BestRnd	IIA Rnd-Rnd	9639	19725	1026.5
AllocateCsrt- BestRnd	IIA Rnd-Rnd	121	19927	2673.5

Table 5.14 Effect of initial solution quality on the s000\_v100 dataset

Constructive Heuristic	Driving Metaheuristic	First Feasible Solution Iteration	Best Feasible Solution Iteration	Best Feasible Solution Penalty
AllocateRnd- BestRnd	IIA Rnd-Rnd	7215	19191	1386
AllocateCsrt- BestRnd	IIA Rnd-Rnd	0	18976	3216

Table 5.15 Effect of initial solution quality on the s100\_v000 dataset

Constructive Heuristic	Driving Metaheuristic	First Feasible Solution Iteration	Best Feasible Solution Iteration	Best Feasible Solution Penalty
AllocateRnd- BestRnd	IIA Rnd-Rnd	15961	19916	1191.2
AllocateCsrt- BestRnd	IIA Rnd-Rnd	0	19687	2946.2

Table 5.16 Effect of initial solution quality on the s100\_v100 dataset

Constructive Heuristic	Driving Metaheuristic	First Feasible Solution Iteration	Best Feasible Solution Iteration	Best Feasible Solution Penalty
AllocateRnd- BestRnd	IIA Rnd-Rnd	6958	19919	1327.9
AllocateCsrt- BestRnd	IIA Rnd-Rnd	1710	19621	2274.6

### Discussion:

In Section 5.2 we saw that AllocateCsrt-BestRnd provides solutions with a low number of hard constraint violations. Hence, while the solutions provided by this algorithm are infeasible, it is the closest we can get to feasibility at the initialisation stage. AllocateRnd-BestRnd provides solutions with a relatively high number of hard constraint violations. In the Tables 5.8 to 5.14, there are two important observations:

1. The first feasible solution iteration refers to the iteration number at which the driving metaheuristic (IIARnd-Rnd) reaches the first feasible solution. This number is much lower when AllocateCsrt-BestRnd is chosen as the constructive heuristic instead of AllocateRnd-BestRnd.
2. The best feasible solution penalty values are lower when AllocateRnd-BestRnd is chosen as the constructive heuristic instead of AllocateCsrt-BestRnd. An exception is seen for the Nott dataset.

From these two observations, we understand that reaching the feasible search space earlier does not guarantee finding a good quality final feasible solution. It is seen that initial solutions with a slightly higher number of hard constraint violations provide a better quality final feasible solution. This is because exploring certain infeasible moves before reaching the feasible moves helps in avoiding local

optima situations. If the number of hard constraint violations is very low, then the driving metaheuristic reaches the feasible solutions faster but has a high possibility of getting stuck on a worse allocation for the remaining iterations. For the Nott dataset, the AllocateCsrt-BestRnd provides solutions with low hard constraint violations. But not as low as that provided by the same constructive heuristic on the SVe150 and PNe150 datasets (Table 5.1). This could be a possible reason as to why AllocateCsrt-BestRnd is the better choice for the Nott dataset.

## 5.5 Driving Metaheuristics Comparison

### **Setup:**

In this fourth and final set of experiments, AllocateCsrt-BestRnd is selected as the constructive heuristic for the Nott dataset while AllocateRnd\_BestRnd is selected as the constructive heuristic for the SVe150 and PNe150 datasets. Then this initial allocation is used as input for the driving metaheuristics implemented in Section 4.7. The length of the subset considered is  $n/20$  where  $n$  is the number of entities in the dataset. Simulated Annealing conditions: Initial Temperature = 500, Decrement = 0.125, Cooling Schedule = Arithmetic. The hard constraint penalty per violation is 500. This experiment is repeated 20 times and the best feasible solution penalty acquired is recorded and compared. The stopping criterion for each run is 20000 iterations. This experiment aims to confirm that the implementation is working correctly. Furthermore, it shows the difference between the solution quality of random and greedy variants of these metaheuristics. The results are provided in Figures 5.8 to 5.14.

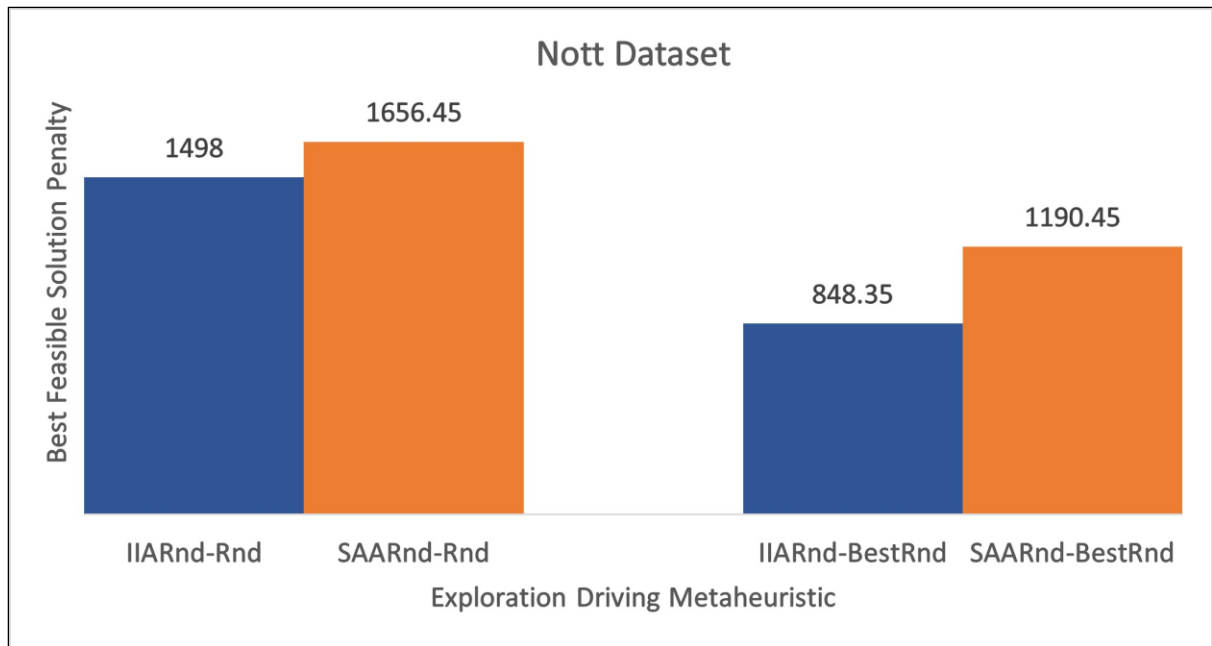


Figure 5.8 Driving Metaheuristics performance on nott dataset

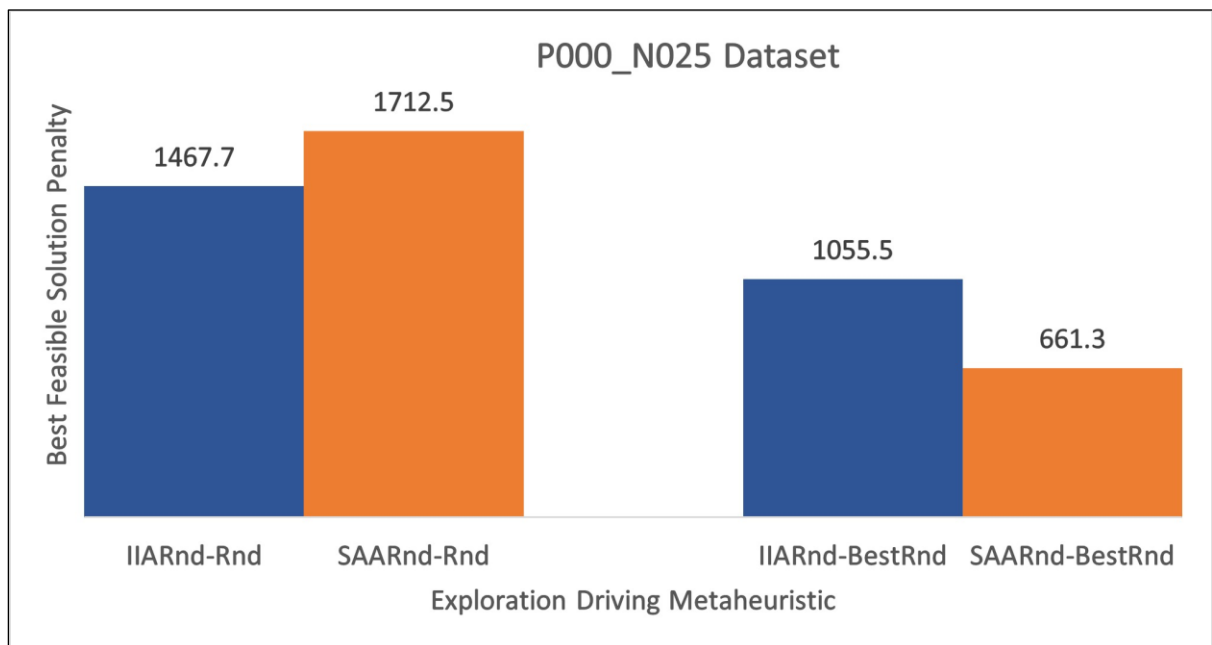


Figure 5.9 Driving Metaheuristics performance on p000\_n025 dataset

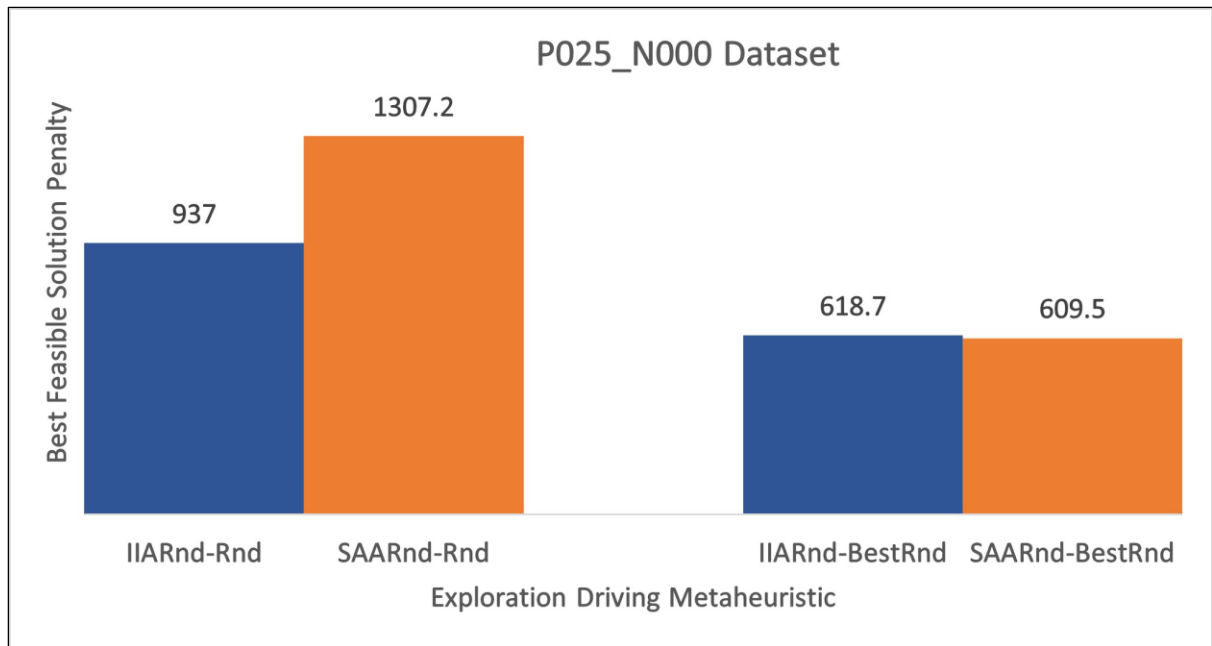


Figure 5.10 Driving Metaheuristics performance on p025\_n000 dataset

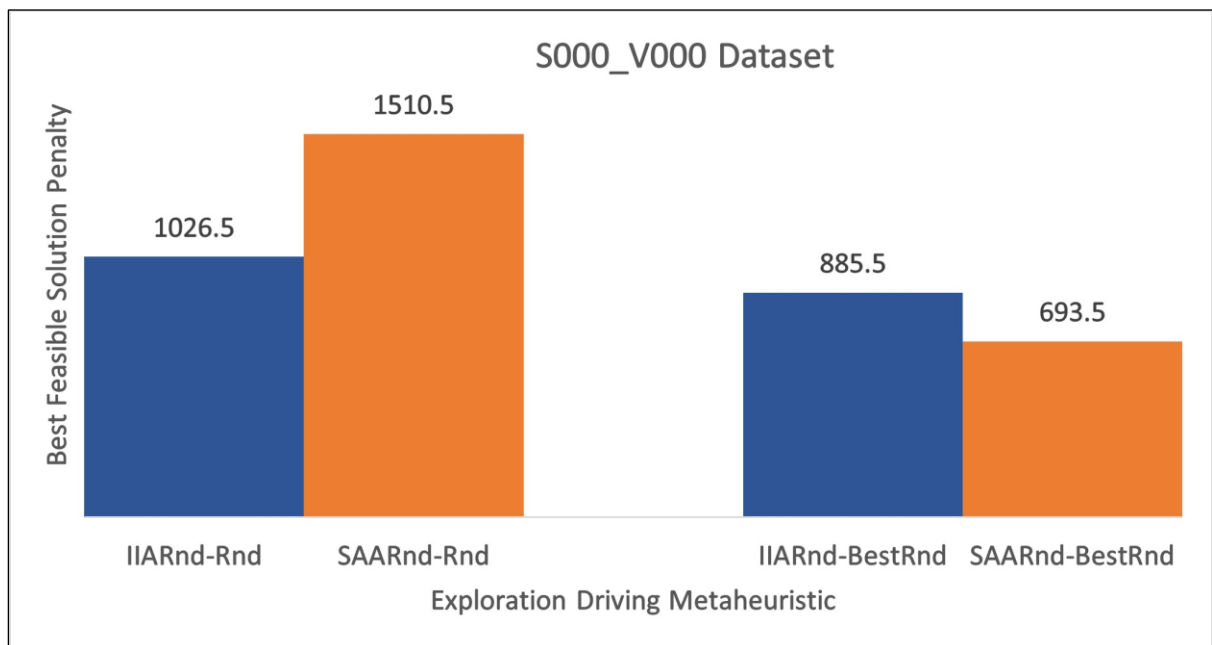


Figure 5.11 Driving Metaheuristics performance on s000\_v000 dataset

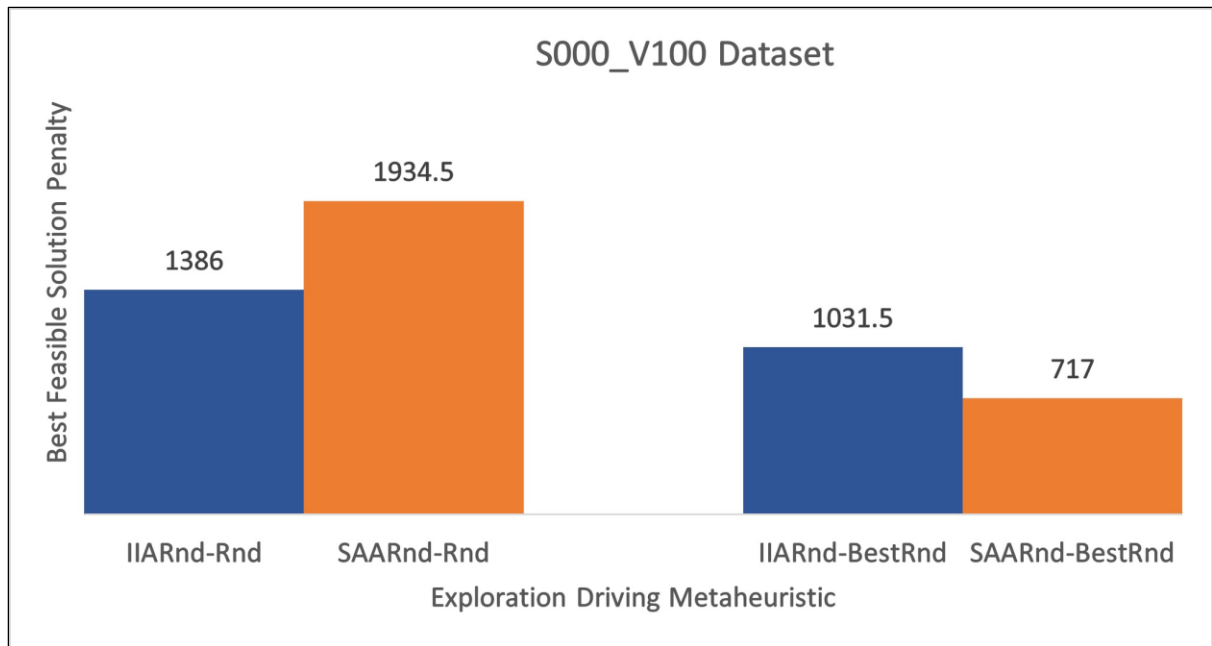


Figure 5.12 Driving Metaheuristics performance on s000\_v100 dataset

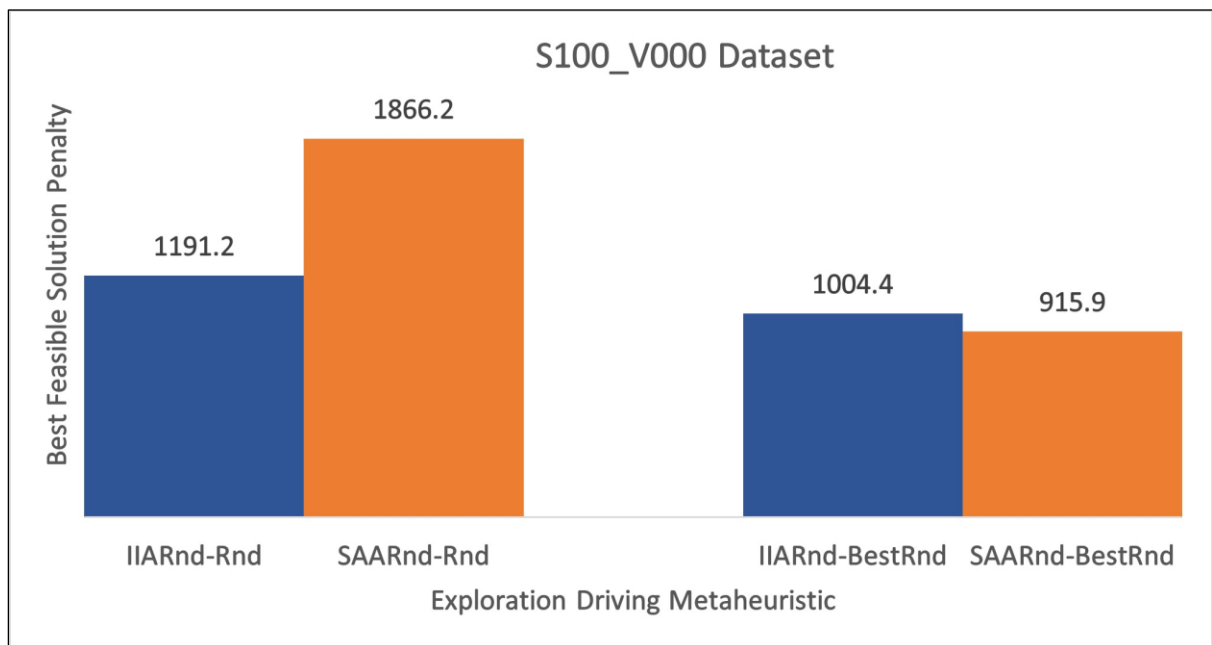


Figure 5.13 Driving Metaheuristics performance on s100\_v000 dataset



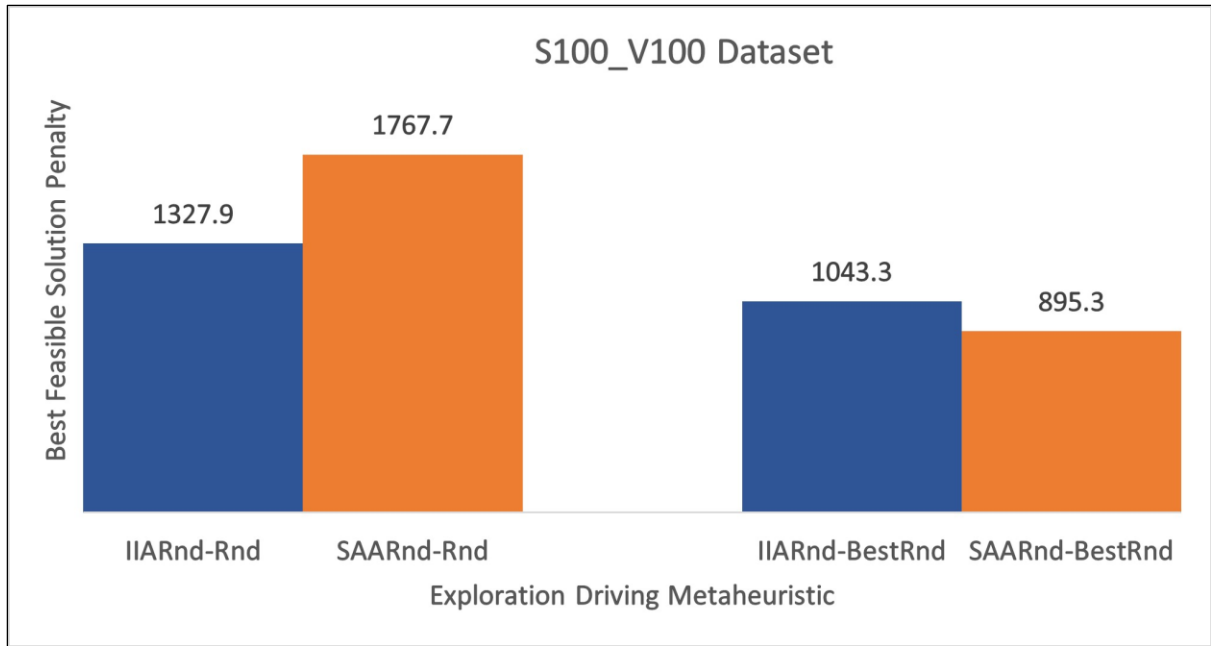


Figure 5.14 Driving Metaheuristics performance on s100\_v100 dataset

### Discussion:

Based on the results obtained above, the following is observed:

1. The greedy variants of both the iterative improvement and the simulated annealing algorithms outperform their random variants.
2. The IIRnd-Rnd algorithm outperforms SAARnd-Rnd for every dataset.
3. The SAARnd-BestRnd algorithm outperforms IIRnd-Rnd for every dataset except nott (Figure 5.8)

In the literature, the SVE150 and PNE150 datasets are used by a few works as mentioned before in Chapter 3. It is observed that the greedy implementations of the iterative improvement algorithm and simulated annealing algorithm give better quality solutions than that provided by the Agent Based Modelling algorithm applied to these datasets (Dediu et al., 2018). On the other hand, the local search method (Ulker et al., 2013) outperforms the implementations in this work.

## 5.6 Conclusion

In this chapter, four sets of experiments were run using the 7 datasets (Nott, S and P). In the first experiment, the various constructive algorithms were compared and a ranking was established between them in the discussion. These rankings are different for the different components that make up the penalty evaluation function as described in Section 4.3. It was seen that AllocateRnd-Rnd gives the worst quality of solutions while AllocateBestAll and AllocateCsrt-BestRnd give solutions with low hard constraint violations. Furthermore, the characteristics of a good constructive heuristic were outlined.

In the second experiment, the hard constraint penalty value was varied and its effect was observed. At low values, the IIARnd-Rnd reaches better feasible solutions but fails to converge on multiple runs. At high values, neighbourhood exploration is a bit restricted but the driving metaheuristic reaches feasibility and doesn't regress out of it. The solution obtained is of good enough quality and hence the advantages outweigh the disadvantages.

In the third experiment, initial solutions were generated by two different constructive heuristics. One provides initial solutions with a low number of hard constraint violations while the other provides initial solutions with a slightly higher number of hard constraint violations. It was established that while it is necessary to reach the feasible search space, it is also necessary to not get stuck in a local optima. Hence, initial solutions with a slightly higher number of hard constraint violations are preferred over those with almost none or no hard constraint violations.

In the final experiment, the iterative improvement and simulated annealing algorithms (random and greedy) were executed and compared on all the datasets.

## Chapter 6 CONCLUSIONS

To properly discuss conclusions, attention is drawn towards the aims and contributions section in the introduction chapter.

This paper started with a background into one of the most important optimization problems in today's world, i.e. the Office Space Allocation problem. The research work already conducted on this topic is vast and complex (Chapter 3). Many new heuristics have been proposed to tackle this topic. These heuristics can be single-solution based, population-based or hybrid.

The primary objective of this work was to implement an optimization software library with standard techniques that can help a new researcher navigate the complexities of the OSA problem. This was successful as evidenced by the detailed steps taken to design the algorithms in Chapter 4 followed by testing them on the benchmark test instances in Section 5.5.

Any exploration of the search space requires first an initial solution. A contribution promised in Chapter 1 was the comparison between the initial solution quality provided by the different constructive heuristics implemented on the benchmark datasets. This was provided in Section 5.2. The results showed that the fully randomised variant (AllocateRnd-Rnd) provided the worst solutions with very hard constraint violations. On the other hand, greedier variants such as AllocateCsrt-BestRnd and AllocateBestAll constructed initial solutions with very low hard constraint violations.

The feasibility of initial solutions was also discussed. It was decided that instead of trying to maintain feasibility, it is easier to allow the driving metaheuristics (iterative improvement algorithm or simulated annealing algorithm) to find feasible solutions on their own through exploration under a hard constraint penalty. The effect of varying this penalty was analysed in Section 5.3 and to the best knowledge of the author, this hasn't been performed on these datasets. The results showed that while a low hard constraint penalty encouraged exploration, it also discouraged convergence towards the feasible search space.

The next contribution of this dissertation was the analysis of the feasible solution quality achieved when the initial solution has a low vs. high number of hard constraint violations. To the best knowledge of the author, this hasn't been performed on these datasets. This is outlined in Section 5.4. Here, the results showed that in certain cases it is better to have an initial solution with a slightly higher number of hard constraint violations since it aids in the process of neighbourhood exploration.

Overall, the whole procedure of finding a feasible allocation was analysed from start to finish. Starting from reading the test instances to then finding the initial solutions, followed by exploring the neighbourhood search space, to finally reaching the best feasible solution. Post-optimality interpretation is carried out on these solutions and the results are written out to a text file for the benefit of the user/space administrators.

## 6.1 Limitations and Future Work

In the software library, there are five constructive heuristics and two exploration driving metaheuristics (each with a random and greedy variant). While the number of constructive heuristics is enough, the number of driving metaheuristics can be increased. This is one of the most important points of improvement for this dissertation project. The two methods implemented are both single-solution algorithms. In the future, the library can be extended by introducing population-based and hybrid metaheuristics. This would give more options to a new researcher on the OSA problem. These new methods would also improve the quality of feasible solutions.

In recent times (Section 3.4), Nature-Inspired (Bolaji et al., 2017) and Phenomenon-Mimicking (Awadallah et al., 2012) metaheuristics have been applied to the OSA problem. Both of these have a memory component that guides the search by taking into account the history of solutions acquired. Furthermore, they are highly customizable due to the number of parameters that they take as input. They have provided excellent results on the Nott dataset but haven't been implemented on the PNe150 and SVe150 datasets. A future direction could be to incorporate algorithms from this class of metaheuristics in the software library.

While Python has distinct advantages in terms of readability, fast development capabilities and extensive library support, a major disadvantage it faces is its slow runtime. It is estimated that Python is 10-100 times slower than C++ due to its dynamically typed nature. So although a beginner student will find it easier to develop or adapt heuristics in this language, it comes at a cost in terms of speed and memory. Due to this reason, the subset size considered in the experiments in Chapter 5 is kept low. Several solutions exist to improve this. Firstly, code can always be refactored to reduce time and space complexities. Secondly, a package exists in Python named PyPy. When installed, it provides Just-In-Time (JIT) compilation that can increase the execution speed by almost 5 times. Finally, Rust and Julia are two programming languages that combine the readability of Python with the performance of C++. These are still under development but are excellent alternatives and can be used to implement heuristics.

Finally, a graphical user interface can be developed to further enhance the experience of a newcomer/beginner. This interface would help the user visualise the appropriate code snippets. The user would input the name of the test instance text file, constructive heuristic and exploration heuristic he/she wants to use to find a feasible allocation. Then the program would paste the appropriate code snippets together and render it to the user screen. The user can then copy this code and run it in a Python IDE.

## REFERENCES

- Awadallah, M.A., Khader, A.T., Al-Betar, M.A. and Woon, P.C. (2012). Office-space-allocation problem using harmony search algorithm. In *Neural Information Processing: 19th International Conference, ICONIP 2012, Doha, Qatar, November 12-15, 2012, Proceedings, Part II* 19 (pp. 365-374). Springer Berlin Heidelberg.
- Bolaji, A.L.A., Michael, I. and Shola, P.B. (2017). Optimization of office-space allocation problem using artificial bee colony algorithm. In *Advances in Swarm Intelligence: 8th International Conference, ICSI 2017, Fukuoka, Japan, July 27–August 1, 2017, Proceedings, Part I* 8 (pp. 337-346). Springer International Publishing.
- Bolaji, A.L.A., Michael, I. and Shola, P.B. (2019). Adaptation of late acceptance hill climbing algorithm for optimizing the office-space allocation problem. In *Hybrid Metaheuristics: 11th International Workshop, HM 2019, Concepción, Chile, January 16–18, 2019, Proceedings* 11 (pp. 180-190). Springer International Publishing.
- Burke, E.K., Varley, D.B. (1998). Space allocation: An analysis of higher education requirements. In: Burke, E., Carter, M. (eds) *Practice and Theory of Automated Timetabling II. PATAT 1997. Lecture Notes in Computer Science*, vol 1408. Springer, Berlin, Heidelberg.
- Burke, E.K. and Varley, D.B. (1999). Automating space allocation in higher education. In *Simulated Evolution and Learning: Second Asia-Pacific Conference on Simulated Evolution and Learning, SEAL'98 Canberra, Australia, November 24–27, 1998 Selected Papers* 2 (pp. 66-73). Springer Berlin Heidelberg.
- Burke, E.K., Cowling, P., Landa, J.D., McCollum, B. and Varley, D. (2000, October). A computer based system for space allocation optimisation. In *Proceedings of the 27th International Conference on Computers and Industrial Engineering*.
- Burke, E.K., Cowling, P., Landa Silva, J.D. and McCollum, B. (2001a). Three methods to automate the space allocation process in UK universities. In *Practice*

and Theory of Automated Timetabling III: Third International Conference, PATAT 2000 Konstanz, Germany, August 16–18, 2000 Selected Papers 3 (pp. 254-273). Springer Berlin Heidelberg.

Burke, E.K., Cowling, P. and Silva, J.L. (2001b). Hybrid population-based metaheuristic approaches for the space allocation problem. In Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546) (Vol. 1, pp. 232-239). IEEE.

Burke, E.K., Cowling, P., Landa Silva, J.D. and Petrovic, S. (2001c). Combining hybrid metaheuristics and populations for the multiobjective optimisation of space allocation problems. In Proceedings of the 2001 Genetic and Evolutionary Computation Conference (GECCO 2001) (pp. 1252-1259).

Castillo, F., Riff, M.C. and Montero, E. (2016, July). New Bounds for Office Space Allocation using Tabu Search. In Proceedings of the Genetic and Evolutionary Computation Conference 2016 (pp. 869-876).

Corne, D., Ross, P. and Fang, H.L. (1994, April). Fast practical evolutionary timetabling. In AISB Workshop on Evolutionary Computing (pp. 250-263). Berlin, Heidelberg: Springer Berlin Heidelberg.

Dediu, A., Landa-Silva, D. and Siebers, P.O. (2018). An agent based modelling approach for the office space allocation problem.

Giannikos, I., El-Darzi, E. and Lees, P. (1995). An integer goal programming model to allocate offices to staff in an academic institution. *Journal of the Operational Research Society*, 46(6), pp.713-720.

Kirkpatrick, S., Gelatt Jr, C.D. and Vecchi, M.P. (1983). Optimization by simulated annealing. *science*, 220(4598), pp.671-680.

Landa Silva, J.D. (2003). Metaheuristic and multiobjective approaches for space allocation (Doctoral dissertation, University of Nottingham).

Landa-Silva, D. and Burke, E.K. (2007). Asynchronous cooperative local search for the office-space-allocation problem. *INFORMS Journal on Computing*, 19(4), pp.575-587.

Lee, S.M. and Clayton, E.R. (1972). A goal programming model for academic resource allocation. *Management Science*, 18(8), pp.B-395.

Pereira, R., Cummiskey, K. and Kincaid, R. (2010, April). Office space allocation optimization. In *2010 IEEE Systems and Information Engineering Design Symposium* (pp. 112-117). IEEE.

Ritzman, L., Bradford, J. and Jacobs, R. (1979). A multiple objective approach to space planning for academic facilities. *Management Science*, 25(9), pp.895-906.

Ülker, Ö. and Landa-Silva, D. (2010). A 0/1 integer programming model for the office space allocation problem. *Electronic Notes in Discrete Mathematics*, 36, pp.575-582.

Ülker, Ö. and Landa-Silva, D. (2011). Designing difficult office space allocation problem instances with mathematical programming. In *Experimental Algorithms: 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings 10* (pp. 280-291). Springer Berlin Heidelberg.

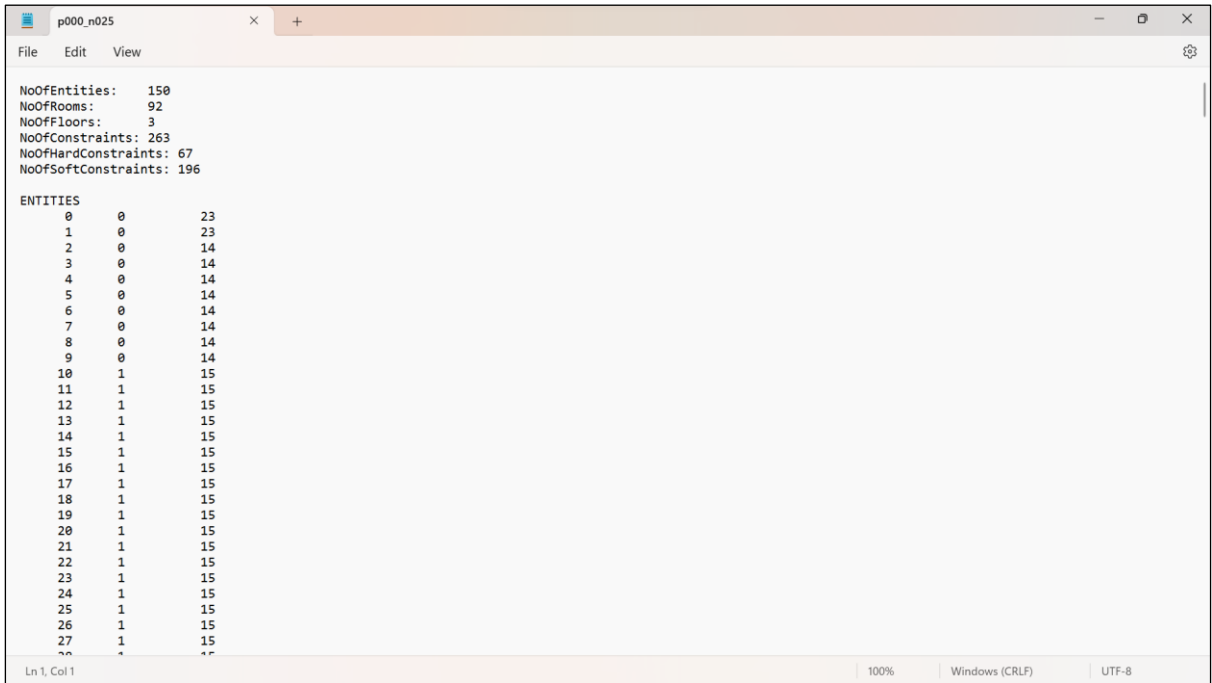
Ülker, Ö. and Landa-Silva, D. (2012, June). Evolutionary local search for solving the office space allocation problem. In *2012 IEEE Congress on Evolutionary Computation* (pp. 1-8). IEEE.

Ulker, O. (2013). Office space allocation by using mathematical programming and meta-heuristics (Doctoral dissertation, University of Nottingham).



# APPENDICES

## Appendix A: Dataset Images



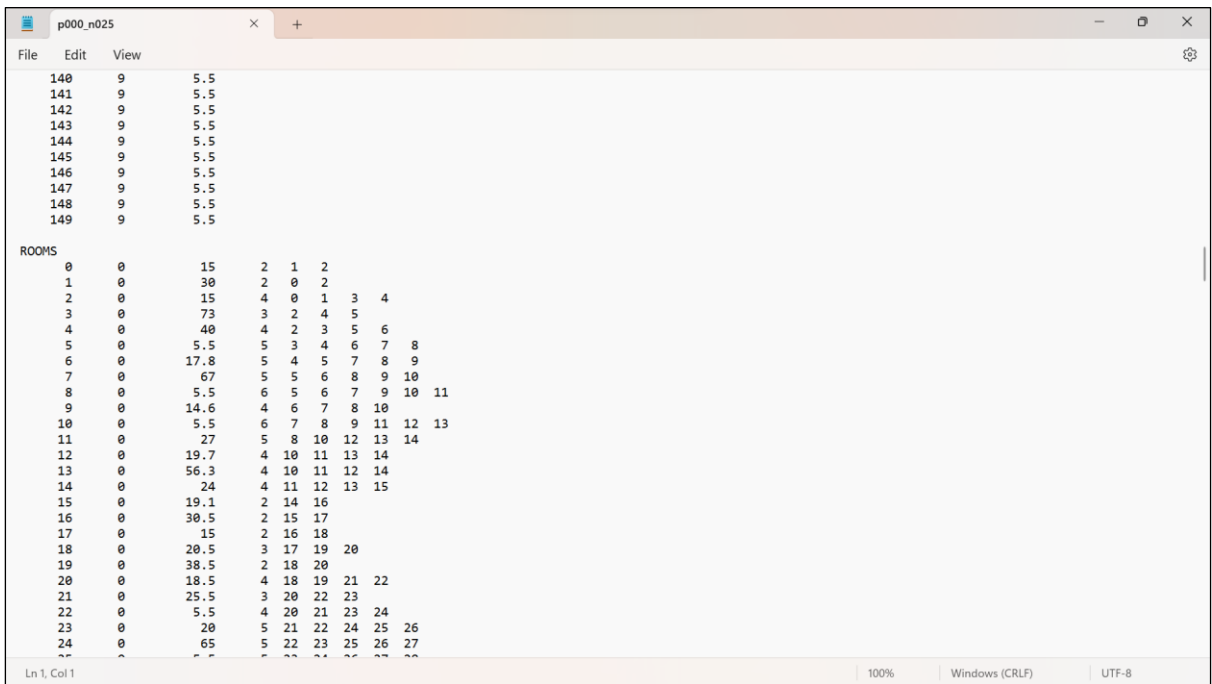
The screenshot shows a text editor window titled 'p000\_n025'. The menu bar includes 'File', 'Edit', and 'View'. The content of the file is as follows:

```
NoOfEntities: 150
NoOfRooms: 92
NoOfFloors: 3
NoOfConstraints: 263
NoOfHardConstraints: 67
NoOfSoftConstraints: 196

ENTITIES
0 0 23
1 0 23
2 0 14
3 0 14
4 0 14
5 0 14
6 0 14
7 0 14
8 0 14
9 0 14
10 1 15
11 1 15
12 1 15
13 1 15
14 1 15
15 1 15
16 1 15
17 1 15
18 1 15
19 1 15
20 1 15
21 1 15
22 1 15
23 1 15
24 1 15
25 1 15
26 1 15
27 1 15
~
```

The status bar at the bottom indicates 'Ln 1, Col 1', '100%', 'Windows (CRLF)', and 'UTF-8'.

Appendix Figure 1 The dataset contains metadata at the top followed by information on the entities.



The screenshot shows the same text editor window, now displaying the 'ROOMS' section of the dataset. The content is as follows:

```
140 9 5.5
141 9 5.5
142 9 5.5
143 9 5.5
144 9 5.5
145 9 5.5
146 9 5.5
147 9 5.5
148 9 5.5
149 9 5.5

ROOMS
0 0 15 2 1 2
1 0 30 2 0 2
2 0 15 4 0 1 3 4
3 0 73 3 2 4 5 6
4 0 40 4 2 3 5 6
5 0 5.5 5 3 4 6 7 8
6 0 17.8 5 4 5 7 8 9
7 0 67 5 5 6 8 9 10
8 0 5.5 6 5 6 7 9 10 11
9 0 14.6 4 6 7 8 10
10 0 5.5 6 7 8 9 11 12 13
11 0 27 5 8 10 12 13 14
12 0 19.7 4 10 11 13 14
13 0 56.3 4 10 11 12 14
14 0 24 4 11 12 13 15
15 0 19.1 2 14 16
16 0 30.5 2 15 17
17 0 15 2 16 18
18 0 20.5 3 17 19 20
19 0 38.5 2 18 20
20 0 18.5 4 18 19 21 22
21 0 25.5 3 20 22 23
22 0 5.5 4 20 21 23 24
23 0 20 5 21 22 24 25 26
24 0 65 5 22 23 25 26 27
~
```

The status bar at the bottom indicates 'Ln 1, Col 1', '100%', 'Windows (CRLF)', and 'UTF-8'.

Appendix Figure 2 The entities section is then followed by the rooms section

	90	2	14	3	88	89	91
	91	2	14	1	90		
CONSTRAINTS							
0	0	0	94	23			
1	0	0	4	64			
2	0	0	38	35			
3	0	0	146	7			
4	0	0	32	44			
5	0	0	35	51			
6	0	0	16	62			
7	0	0	83	1			
8	0	0	53	44			
9	0	0	2	86			
10	0	0	40	78			
11	0	0	78	7			
12	0	0	133	16			
13	0	0	74	9			
14	0	0	143	19			
15	0	0	87	24			
16	0	0	6	76			
17	0	0	122	56			
18	0	0	128	40			
19	0	0	90	31			
20	0	0	103	69			
21	0	0	121	53			
22	0	0	66	79			
23	0	0	15	81			
24	0	0	30	54			
25	0	0	0	72			
26	0	0	30	54			
27	0	0	60	79			
28	0	0	70	33			
29	0	0	85	24			
30	0	0	115	35			
31	0	0	130	12			
32	1	0	3	69			

Appendix Figure 3 The dataset contains the constraints section after the rooms section

Appendix B: Output Text File Images

entities	rooms
0	72
1	66
2	86
3	80
4	64
5	91
6	76
7	88
8	73
9	90
10	27
11	4
12	65
13	17
14	77
15	81
16	62
17	4
18	63
19	74
20	82
21	9
22	0
23	49
24	57
25	85
26	6
27	57
28	39
29	74
30	54
31	46
32	44
33	55
34	35

Appendix Figure 4 The entity-room allocation is provided in output.txt

```
output
File Edit View
147 10
148 22
149 7

Iteration: 19930
Total Penalty: 867.7
Space Misuse Penalty: 457.7
Soft Constraint Violation Penalty: 410
No. of hard constraints violated: 0.0

room id space used space left
0 15.0 0.0
1 35.5 -5.5
2 15.0 0.0
3 74.5 -1.5
4 47.5 -7.5
5 5.5 0.0
6 15.0 2.800000000000007
7 80.0 -13.0
8 5.5 0.0
9 15.0 -0.4000000000000036
10 5.5 0.0
11 24.5 2.5
12 25.5 -5.800000000000001
13 70.0 -13.700000000000003
14 20.5 3.5
15 20.5 -1.3999999999999986
16 29.5 1.0
17 15.0 0.0
18 20.5 0.0
19 20.5 18.0
20 18.5 0.0
21 25.5 0.0
22 5.5 0.0
23 18.5 1.5
24 58.5 6.5
25 5.5 0.0
26 20.5 0.0

Ln 1, Col 1 100% Windows (CRLF) UTF-8
```

Appendix Figure 5 Statistics related to the entity-room allocation are provided next. This includes different types of penalties and space used/left in rooms.

```
output
File Edit View
90 14.0 0.0
91 14.0 0.0

constraint id status
0 violated
1 satisfied
2 violated
3 satisfied
4 satisfied
5 satisfied
6 satisfied
7 satisfied
8 satisfied
9 satisfied
10 violated
11 satisfied
12 violated
13 violated
14 violated
15 satisfied
16 satisfied
17 violated
18 violated
19 satisfied
20 satisfied
21 satisfied
22 satisfied
23 satisfied
24 satisfied
25 satisfied
26 satisfied
27 violated
28 satisfied
29 violated
30 violated
31 satisfied
32 satisfied
33 satisfied

Ln 1, Col 1 100% Windows (CRLF) UTF-8
```

Appendix Figure 6 Information related to the satisfaction/violation of constraints is provided at the end of output.txt

