OFFICE SPACE ALLOCATION BY USING MATHEMATICAL PROGRAMMING AND META-HEURISTICS

Özgür ÜLKER, BSc., MSc.

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

September 2013

Abstract

Office Space Allocation (OSA) is the task of efficient usage of spatial resources of an organisation. A common goal in a typical OSA problem is to minimise the wastage of space either by limiting the overuse or underuse of the facilities. The problem also contains a myriad of hard and soft constraints based on the preferences of respective organisations. In this thesis, the OSA variant usually encountered in academic institutions is investigated. Previous research in this area is rather sparse. This thesis provides a definition, extension, and literature review for the problem as well as a new parametrised data instance generator.

In this thesis, two main algorithmic approaches for tackling the OSA are proposed: The first one is integer linear programming. Based on the definition of several constraints and some additional variables, two different mathematical models are proposed. These two models are not strictly alternatives to each other. While one of them provides more performance for the types of instances it is applicable, it lacks generality. The other approach provides less performance; however, it is easier to apply this model to different OSA problems. The second algorithmic approach is based on meta-heuristics. A three step process in heuristic development is followed. In the first step, general local search techniques (descent methods, threshold acceptance, simulated annealing, great deluge) traverse within the neighbourhood via random relocation and swap moves. The second step of heuristic development aims to investigate large sections of the whole neighbourhood greedily via very fast cost calculation, cost update, and search for best move procedures within an evolutionary local search framework. The final step involves refinements and hybridisation of best performing (in terms of solution quality) mathematical programming and meta-heuristic techniques developed in prior steps.

This thesis aims to be one of the pioneering works in the research area of OSA. The major contributions are: the analysis of the problem, a new parametrised data instance generator, mathematical programming models, and meta-heuristic approaches in order to extend the state-of-the art in this area.

Acknowledgements

I want to thank my father, Durmuş Ülker first for his never-ending support both emotionally and financially through all these years. Unfortunately, he has to go through a stage 4 lung cancer during the time I have to write this thesis. This thesis is probably my most important way to repay him for his deeds and finishing it before a potential death has become the propelling motivation for me while I struggle to create one by myself. The rest of my family, my mother Vildan and my brother Barış Ülker also get my deepest gratitude. It is great to think at least I have a home back in Turkey with support from my family. I also want to briefly thank all my close relatives because it will take quite a while to list all of them here.

I would like to thank my PhD supervisor Dr. Dario Landa-Silva for his comments and guidance throughout this project, his easy-going attitude and not having a dictatorship over me just because he is my supervisor. Having a supervisor who previously worked exactly in the same combinatorial optimisation problem (office space allocation) certainly helped me a lot in our discussions.

I also want to thank my internal examiner Dr. Rong Qu from University of Nottingham and my external examiner Prof. Julia Bennell from University of Southampton. I believe their valuable suggestions helped this thesis to become stronger and more coherent in scope, content, and presentation.

I want to thank Dr. Ender Özcan without whom I would have probably never been able to step into this PhD in the first place . His valuable suggestions before, during, and hopefully after this PhD have helped me a lot in my academic development.

I am also grateful for University of Nottingham for providing me the funding, office space, and other resources required for this project.

And last but not the least, I want to thank everyone in the ASAP group who both asked and did not ask the dreaded question, "How is the thesis going?". I thank those who have asked because they care. I also thank those who have not too because they already know no one wants to talk about his/her thesis besides work hours!

Dedicated to my father **Durmuş Ülker**
and my mother **Vildan Ülker**

Table of Contents

List of Figures

List of Tables

Introduction

Space is one of the most expensive resources in a typical organisation. The office space allocation (OSA) process aims to make efficient use of these spatial resources such that the misuse of space is minimised. The task of space allocation may include additional constraints and objectives based on the preferences of the respective organisation.

In many institutions, the entities (most commonly people) that use the resources (rooms, machines, spaces, etc.) are organised in structural units like departments. This naturally leads to a clustering problem [Everitt et al., 2009] where each entity within the same organisational unit should be placed in close proximity while different organisational units should not be placed close to each other. This is not an issue when such organisational units are large. In a typical university, it is not usually expected that the rooms associated with an engineering faculty are within the same building as rooms associated with social sciences. This enables us to decompose the office space allocation problem into smaller sub-problems; after all, most buildings within a university are built for specific purposes, and this naturally leads to independent allocations.

However, an office space allocation problem can arise more frequently within the same organisational unit. For example, in the Department of Computer Science at the University of Nottingham, there are eight research groups several of which also perform interdisciplinary research with other departments. While it is desirable to place each entity in the same research group within close proximity to each other away from other research groups, this may not be always possible. After all, in a dynamic environment where sub-organisational units constantly change over time due to arrival or departure of new personnel, it is expected that, in time, these sub-organisational units will be mixed with each other.

Unfortunately, not every office space allocation problem can be solved by scientific optimisation. There is usually a fair amount of organisational politics and bureaucracy

in each institution that govern the actual space allocation process. For example, an automated space allocation process in NASA Langley Research Centre [Ball, 2009] was hampered by budget and other constraints for years and a manual allocation process which was less efficient and more expensive was implemented instead in 2005.

In the office space allocation problem, the most obvious aim is to optimise the efficient usage of space. The space misuse typically involves at least two components. The first one is that each room should not be overused beyond its optimal capacity. The overuse problem happens more and more in organisations with budget restrictions. Since the cost of leasing or owning an office property is rather expensive especially in developed countries, workers are often asked to share the office space resources. This may lead to employees forced to work in limited space and poor work performance.

Another important and unfortunately overlooked problem is the use of the office spaces well below their optimal capacity. In a related teaching space allocation problem, [Beyrouthy et al., 2009] investigated the usage of teaching space in universities, and very low space utilisations (around twenty to thirty percent) were reported. Severe underuse of office space can be a big financial burden due to high leasing costs or even unnecessary new building costs. If such an underuse of rooms cannot be avoided, then the respective space should be used by another related organisational unit. If this is not possible, it might be beneficial to sell or rent the respective rooms instead.

Unlike other resources in most organisations, space can be a very expensive commodity that cannot be easily increased when needed due to the high costs and time involved in construction/renovation of buildings. Space allocation is also a continuous process due to constant changes in an organisation like departure/arrival of new personnel, maintenance/renovation of existing office space, restructuring in organisations etc. Due to the complexities of many organisations, office space allocation may involve many conflicting objectives and constraints. An automated allocation system can usually deal with such conflicting objectives and constraints better than a human decision expert especially if the size of the problem is very large. An automated system can also provide alternative solutions to different scenarios quickly. As a result, rather than tackling a more complex optimisation problem, a decision expert can focus on fine tuning such solutions as required by the scenario conditions.

As a result, the main goal of this thesis is to investigate and propose solution methods that can be used in building an automated office space allocation system. In recent years, several companies developing software related to space planning and organisation process have appeared, so it is quite possible there is an untapped financial market for such automated systems.

2

1.1   Aims and Scope

Previous research on the office space allocation (OSA) problem is rather limited. Hence, the goal of this thesis is to be one of the seminal works in this area by proposing efficient algorithms to solve this problem. The main aims of this thesis are as follows:

- There are several different variants of OSA problems described in the literature. This thesis aims to investigate previous OSA problems in terms of constraints and objectives. It is also desired to add three more constraints to the problem definition which is described in [Landa-Silva, 2003]. A related aim is to create a parametrised data instance generator algorithm that will be used in design, test, and analysis of the mathematical programming and meta-heuristic techniques for the OSA problem.

- Although there are various algorithms proposed for solving OSA problem, there is a lack of research in terms of analysis of the problem. This thesis aims to analyse the nature of the problem from the perspective of space misuse, constraints, and overall objective function. Although a multi-objective analysis (in terms of space misuse and soft constraint penalty) was briefly described in [Landa-Silva, 2003], the analysis was rather lacking in relation to how these aspects affected the difficulty of the problem. This thesis aims to analyse the effects of the incremental changes of four important aspects of the problem: total space misuse, space overuse and underuse, and soft constraint violation penalty. The thesis aims to observe how the difficulty of the problem and some of the key ratios between these parameters are affected subjected to these incremental changes. A further aim is to seek out some algorithm development ideas based on this analysis of parameter effects if possible.

- There are a few mathematical programming approaches such as [Sharpe, 1973], [Ritzman et al., 1980], [Benjamin et al., 1992], and [Giannikos et al., 1995] proposed for several variants of the OSA problem. However, for the OSA problem variant considered in this thesis, there is no mathematical programming model or implementation reported in the literature. This thesis aims to examine the applicability of mathematical programming techniques to solve this specific OSA problem [Landa-Silva, 2003] for the first time in literature. This thesis aims to develop binary integer programming models based on nine types of constraints and a weighted objective function. It will investigate the potential benefit of utilising room and floor relationship while developing the model. The main target in

development of integer programming models is to investigate the performance of the model (finding optimal or near optimal solutions) in long run times.

- In the literature, most common approach for tackling the OSA problem is various meta-heuristics. The thesis aims to build upon some of the ideas proposed in previous heuristic designs. The key aim in this thesis is to seek out some important aspects which can improve the quality of the solutions generated by the heuristic designs. Several of these aspects are the solution replacement strategies and quick objective function value calculation procedures. The thesis aims to investigate meta-heuristics which operate with random or greedy move operators to find high quality solutions for the problem. Although several quick cost calculation procedures based on similar move operators were described in [Landa-Silva, 2003], these techniques only gave approximate cost changes. This thesis aims to provide a very fast cost calculation procedure that will give the *exact* changes in the objective function value based on the application of move operators. The main target in development of these heuristics is to significantly improve the solution quality especially in short run times.

- The final aim of this thesis is to investigate possible combinations of mathematical programming and meta-heuristics for tackling the OSA problem. Best integer programming model and best implementation of local search meta-heuristics will be combined together. The thesis aims to investigate the efficacy of such an approach and tries to analyse the strengths or potential pitfalls of these hybrid combinations. Since the application of of exact approaches can quickly become infeasible with large instances, the thesis will limit its scope to instances of small to medium sizes. This roughly corresponds to the typical size of a three/four storey building commonly encountered in a British university. Another important aim with this size limitation is the desire to make a balanced comparison of meta-heuristics, mathematical programming, and the combinations of these two.

Ultimately, the aim in this thesis is to provide a solid foundation in analysis of the natural components of office space allocation and algorithm development for the problem. Improving the state of art for tackling and providing best results for office space allocation problems is the main goal of this thesis.

In this thesis, the scope is limited to the *optimisation* problem (the initial allocation and space optimisation subject to a set of hard and soft constraints) which is extended from [Landa-Silva, 2003]. Although the *re-allocation* problem (re-optimisation of the solution due to the modifications to the structures of the problem) is briefly discussed,

the analysis and algorithm development of this problem variant are beyond the scope of this thesis.

This thesis will limit the investigated size of the problem to small to medium size buildings. This size of the problem is suitable for making balanced and objective comparisons between mathematical programming and meta-heuristic approaches. Larger office space allocation problems can be decomposed into smaller problems and solved with a bottom-up approach by combining the solutions for sub-problems.

There is a vast amount of algorithmic techniques that can be used to tackle the OSA problem. As a result, the algorithmic focus in this thesis is limited to the investigation of integer programming [Juenger et al., 2010], local search [Glover and Kochenberger, 2003], and genetic algorithms [Goldberg, 1989].

1.2 Overview of the Thesis

The structure of this thesis can be summarised as follows:

- **Chapter 2** gives insights about the nature of the office space allocation problem. Commonly encountered constraints and the objectives are defined here. Several case studies related to the office space allocation process in universities and space agencies are presented. The properties of the real world OSA data instances that have been used throughout this research are described. Also, a new data instance generator algorithm developed for providing new data instances to the research community is described in this chapter.

- **Chapter 3** references the literature related to the office space allocation problem. Some key theoretical problems such as *bin packing*, *generalised assignment*, *multi-dimensional knapsack*, and *clustering* problems that have fundamental ties to the OSA are also described. An extensive literature review of previous research that has been done on the OSA problem is given. Also, some practical problems that resemble the OSA in certain aspects are explained. In the second half of this chapter, some of the methods that can be used for developing OSA algorithms are described. These algorithms include, but are not limited to, local search meta-heuristics, genetic algorithms, linear integer programming models, and hybrid combinations of these methods.

- **Chapter 4** describes the mathematical model for a binary integer programming formulation developed for OSA. The derivations of the mathematical equations

5

for both hard and soft constraints are presented here. Two versions of IP models (based upon the formulations of nine different constraints and space misuse calculation) that are examined in this thesis are explained. Experiments are conducted in order to analyse the nature of the problem by using parametrised data instances. Effects of space misuse and constraint violations on the difficulty of the problem are observed. The model with the additional (floor) variables is compared to the initial model without such variables.

- **Chapter 5** is devoted to the presentation of algorithms that are implemented based upon random *relocate* and *swap* moves within a local search framework. The goal of this approach is the rapid generation of simple local heuristic algorithms that do not need very complex cost calculation procedures, and whether such approaches are effective in generating quality solutions comparable to the ones obtained using mathematical programming. In this framework, hill climbing methods, threshold acceptance, simulated annealing, and great deluge methods are used to develop the algorithm.

- **Chapter 6** presents an evolutionary local search algorithm based upon the combination of a genetic algorithm and local search method that uses greedy *relocate* moves. The details of a very fast cost calculation method based upon a greedy *relocate* move are described. The focus is on the balance between the evolutionary and local search components of the algorithm, and the importance of each sub-component of the local search procedure.

- **Chapter 7** presents the final methods proposed in this thesis: a combination of the mathematical model described in Chapter 4 and the local search methods described in Chapter 6. First, the necessary adjustments and refinements on previous algorithms before the combination are described, then how these techniques can be combined within a framework is explained in detail.

- **Chapter 8** gives an overall review of this thesis and presents some future research directions.

1.3 Contributions of the Thesis

The (original) contributions of this thesis are as follows:

- A description of the office space allocation problem encountered in many institutions is given. An extensive literature survey regarding the office space allocation,

and other theoretical and practical problems is presented. Common constraints and objectives in a typical office space allocation problem are described. Three additional constraints not previously discussed in literature are considered for the first time.

- A new parametrised data instance generator is developed to address the limited number of tests instances in OSA area. This is the first instance generator in OSA research that takes some of important sub-components of a typical OSA problem.

- Two variants of binary integer programming formulation are developed for solving small to medium size instances. These models can be implemented easily by using various off-the-shelf integer linear programming (ILP) solvers. This is the first time mathematical programming models are proposed for the specific OSA variant tackled in this thesis.

- By using the parametrised data instance generator and the binary integer programming models, the nature of the problem is investigated by analysing the space misuse and the soft-constraint violation penalty components. This is the first advanced analysis of changes in difficulty and some key ratios in a typical OSA problem from the perspective of space misuse and constraints.

- New local search algorithms based upon strictly stochastic relocate and swap moves under simulated annealing, threshold acceptance, and great deluge frameworks are provided.

- An evolutionary local search algorithm for solving the OSA is proposed. New problem specific crossover and mutation operators are presented. A new very fast exact cost calculation procedure which yields huge speed-up gains over naive approximate objective value computation approaches is provided.

- Comparisons of developed integer programming and meta-heuristics are provided. These comparisons include the differences in total penalty, minimum penalty, space misuse, and soft constraint violations between the algorithms.

- A mat-heuristic technique which combines integer programming and local search procedure for solving OSA is proposed. The effectiveness of combining the refined versions of both algorithms is investigated for the first time for this problem.

1.4 Academic Papers Related to this Thesis

Published:

- [Ülker and Landa-Silva, 2010] *A 0/1 Integer Programming Model for the Office Space Allocation Problem* Özgür Ülker, Dario Landa-Silva. Electronic Notes in Discrete Mathematics, 36, pp. 575-582, 2010.

- [Ülker and Landa-Silva, 2011] *Designing Difficult Office Space Allocation Problem Instances with Mathematical Programming* Özgür Ülker, Dario Landa-Silva. Experimental Algorithms, Lecture Notes in Computer Science, Vol. 6630, pp. 280-291, Springer-Verlag, 2011.

- [Ülker and Landa-Silva, 2012] *Evolutionary Local Search for Solving the Office Space Allocation Problem* Özgür Ülker, Dario Landa-Silva. Proceedings of the 2012 IEEE Congress on Evolutionary Computation (CEC 2012), pp. 3573-3580, IEEE Press, Brisbane Australia, July 2012.

Papers to be submitted and in preparation:

- [Ülker and Landa-Silva, 2013a] *Analysis of Office Space Allocation Problem using Mathematical Programming* Özgür Ülker, Dario Landa-Silva. planned to be submitted to a journal.

- [Ülker and Landa-Silva, 2013b] *Two Neighbourhood Iterated Local Search Algorithm for Space Allocation Problem* Özgür Ülker, Dario Landa-Silva. planned to be submitted to a conference.

Office Space Allocation

2.1 Introduction

Office space allocation (OSA) is the task of allocating office space (rooms, hallways, etc.) to several entities subject to additional constraints. It is related to the *bin packing*, *multiple knapsack*, and *generalised assignment problems* [Martello and Toth, 1990]. In office space allocation, the primary goal is to maximise the space utilisation by reducing the misuse of the rooms. Misuse of rooms consists of wasting space by under utilisation of the room or overcrowding of the room by placing too many entities into it. Overuse of the rooms is usually considered more undesirable than underuse.

In this chapter, the aim is to introduce the reader to the office space allocation problem variant tackled in this thesis. The information related to the constraints, the objectives, and several real world test instances is provided. A new data instance generator algorithm is also proposed. In the following chapters, most of the experiments will be carried out on the parametrised data instances created by this generator.

This chapter is organised as follows: Some of the approaches in space allocation and planning in organisations are explained in Section 2.2. Several case studies related to the space allocation process in British universities and the NASA LaRC are presented. The problem description, commonly encountered constraints, and objectives are given in Section 2.3. Section 2.4 presents the information related to the real world data instances used in the experiments in this thesis. Section 2.5 is devoted to the data instance generator algorithm developed to provide more test instances to the research community. The conclusions are given in Section 2.6.

2.2 Space Allocation and Planning in Organisations

In this section, we are presenting two case studies related to office space allocation. Universities are among the major institutions that conceptualised the office space allocation process (although not necessarily by using an automated tool). Various guidelines as described in Section 2.2.1 have been posted by universities in their official documents. The OSA case in NASA Langley Research Centre as described in Section 2.2.2 is an important study to explain a real world optimisation tool to tackle the problem.

2.2.1 Case Study: Space Allocation in Universities

A typical google search on office space allocation yields a fair amount of 'guidelines' as described by mostly universities in the world. Unfortunately, most of these are just 'guidelines', there is hardly a reference to an automated system because space allocation in many universities is mainly a manual process. In most universities, the governing body responsible for space allocation is the equivalent of an Estates Department. Requests for office space are usually made officially to these departments, and in a large office space restructuring request, a lengthy and bureaucratic review period is usually required. The estates department usually allocates the space to individual faculties. It is the responsibility of the faculties to allocate the space assigned to them .

However, there are official rules that are still followed for allocating the entities. In most universities, the workers (or people who are eligible for office space) are organised in tiers, and the amount of space required for each tier is predetermined by OSA guidelines. There are also specifications on room structures a person on a specific tier should be allocated to.

As a case study in office space guidelines, this thesis is now going to examine one of the most extensive guidelines in this area: University of Michigan, Ann Arbor [of Michigan, 2012]. This guideline serves a typical example how the office space allocation process is usually handled in a university.

At the top of the responsibility of allocation in University of Michigan is the Provost. The hierarchy from top to bottom is as follows: Provost, Vice President, Deans/Unit Directors, Department Chairs, and Faculty members.

Some of the principles in these guidelines are as follows:

- The space belongs to the institution and the Provost is ultimately responsible for the allocation of all spatial resources of the university.

- Space is allocated based on programmatic needs and priorities as determined by the dean or director or a unit. The decision making may be delegated to chairs and director as long as they have high in-depth knowledge of the activities and the allocations associated with them.

- Specific quantitative metrics should be developed in order to evaluate the research space utilisation. Periodic controls should be made in order to assess whether the current allocation meets the programmatic needs of the university.

- The research space is assigned to research activities, not to individuals, hence it can be taken by the university if the research activity changes.

- Vacant or under-utilised space should be re-claimed, re-assigned or re-purposed.

- Schools are allowed to subsidise research activities that do not generate sufficient indirect costs related to the space usage.

- *Optimal use of research space includes shared use of resources and facilities.*

- Space allocations should be based on maximum utilisation of the existing facilities.

- Space allocation should adhere to health and safety regulations, and procedures.

2.2.2   Case Study: Space Allocation in NASA Langley Research Centre

One of the few reported practical applications of space allocation within a large organisation corresponds to one of the oldest major National Space Agency (NASA) Centres, the Langley Research Center (LaRC) in Virginia, US [Ball, 2009]. LaRC was historically built on a 800 acre land comprising 300 buildings with 6000 rooms totalling 3 million square feet of space usage. LaRC has been used mainly for Aeronautical and wind tunnel research as well as applications in structures and materials, flight electronics, and atmospheric sciences. Such a large scope and different usage scenarios necessitated specific building structures with complex infrastructure.

Recently, due to a significant reduction in funding and budget in various operational areas as well as the increasing costs of development of next generation space vehicles, LaRC had to reassess the usage of its facilities and resources. A new dynamic and structured allocation of resources, including the assignment of space within each facility and across LaRC had to be developed. This process was accelerated in late 2004, when LaRC was going through a major reorganisation in which 3000 people were expected to be relocated; the average office space per person was to be reduced from

190+ square feet to 125 while 100 facilities were scheduled for closure and demolition. The Geographic Information System (GIS) team in LaRC was responsible for the development of a series of automated space allocation and re-organisation tools based on *geographic information* and *relational database management systems*. These tools supported the development of various usage scenarios from the perspective of visualisation and analysis of potential space allocation solutions. The GIS followed an evolutionary development strategy with the following areas of focus:

Visualisation: During the early stages of the development, a web browser tool was used for visualising the current personnel location, organisational distribution and space utilisation. The interface employed a dashboard which allowed users to access the unit square diagram, plant level map, building interior layouts and any related tabular data. The user would be able to visualise the current conditions in the allocation such as relative size and proximity of the buildings, rooms and the personnel. A sample visualisation of dashboard is given in Figure 2.1 where a fragmentation of organisations can be seen.
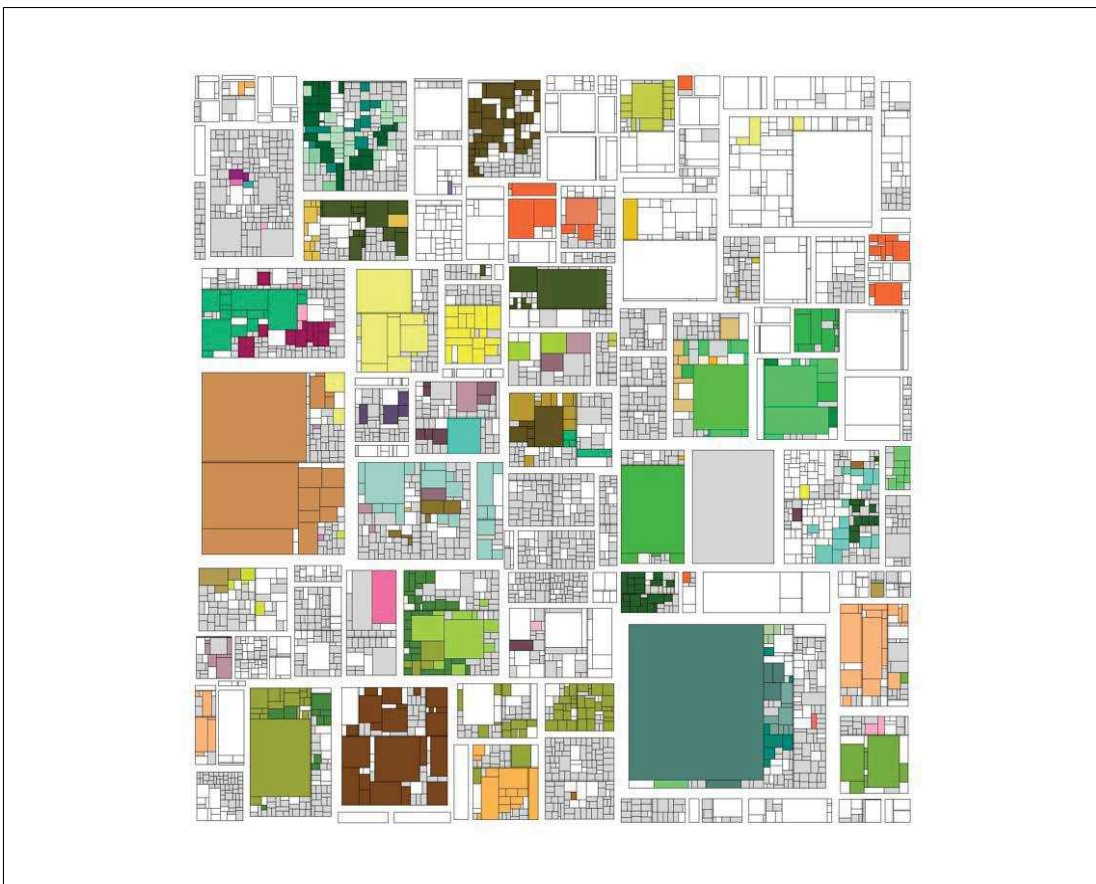
Figure 2.1: Dashboard representation of an allocation in NASA LaRC [Ball, 2009]

The visualisation tools addressed both *plant level* and individual *building level* allocations. For *plant level* visualisation, a *unit square* or *spatial subdivision diagram* were developed. This technique used a rectangle based representation for depicting the buildings. The bold rectangular areas which represented the facilities within a building were oriented based on proximity to buildings near them. The size of a rectangle represented the usable space within the facility. Size proportional sub-blocks represented each room within a building. Colour coding was used in order to symbolise the applicability of each room for general office use or technical usage, the organisations that owned the room, and the closeness between different ones as well as to indicate all the personnel. The aim for colour coding was to represent the close proximity of buildings, rooms, personnel, and their interaction with large and diverse data within a complex facility.

Metrics and Constraints: The space planning and allocation in LaRC is a complex problem and had to address general metrics and constraints. The developed models should handle the critical variables and effects of the problem, should have good generalisability and they should be analysable for the feasibility of generated solutions. The desired solution for the problem is the one that meets all the constraints and rated by a metric. The most obvious metric of a model is the infrastructural, development and application related costs; however, how this cost is calculated can be answered by analysing the components of the problem.

As a typical office space allocation, the problem in LaRC involves entities such as people or functions that consume a space such as laboratories, office and technical areas, etc. The space is most commonly represented in terms of the area of the respective organisational unit. However, additional resources (such as communication jacks, bandwidth, electrical power, etc.) can also be used for defining spatial resources.

The space allocation problem also involves *hard* and *soft* constraints. The *hard* constraints considered can be: utilising enough space for a function or entity (minimum area for different workers), compatibility with adjacent functions (supervisors and workers should not be placed in the same room) and compatibility with features already satisfied by the room. The *soft* constraints can be the improvement of organisational synergy (such as allocating personnel within the same organisational unit to nearby locations), decreasing moves costs, and symmetrical distribution of space.

Data Management: The space allocation system in LaRC has to consider the problems associated with collecting and maintaining accurate, current and dynamic data

that are fetched across different sources. An XML schema was developed for a representation language for the nature of the problem and the models designed. An XQuery language was used to leverage data transformation.

Optimisation Algorithms: The optimisation approach for solving the space allocation problems in LaRC contain two stages. In the first stage, the solver works on an allocation which violates several constraints and tries to find a new solution within the same neighbourhood that satisfies all constraints. In the second stage of the algorithm, a greedy heuristic is applied. This heuristic improves the given constrained solution to a locally optimal solution.

Another approach is not only improving from the current solution but also from a random allocation. While this approach provides good results for smaller organisational units, a lot of problems with compatibility within the larger organisational units are observed when large scale changes are made. The GIS team tries to combine both random and greedy approaches by improving random allocations through constraint solving, re-application of greedy heuristic and a progressive filtering algorithm.

Benefits of Using the Automated GIS System in GIS: The cost benefits of using the proposed automated system was delineated by the GIS team in 2005 office consolidation project at LARC. Due to the time constraints and the lack of funding for GIS, the prototype automated system proposed by GIS was not used. Instead, the final decision process was made by a manual approach. The manual approach reduced the average space utilisation per person from 192 square feet to 149 (the target was 125 square feet). The manual solution relocated 2500 personnel with an average cost of $354 per person. The prototype automated system was able to reduce the average space utilisation to the target 125 square feet while relocating only 1200 personnel and in turn would have saved the centre $460,000 while improving the organisational synergy and minimising the disruption during the relocation. The cost savings only include office space re-allocation process and do not include the annual cost savings associated with additional facility closures accumulated over years and across multiple centres.

2.3   Problem Description and Formulation

In this section, a general description of the OSA problem is given. The majority of the constraints are based on previous research in this area [Landa-Silva, 2003], [Pereira et al., 2010], [Lopes and Girimonte, 2010]. The objective function value is taken simi-

larly to [Landa-Silva, 2003] although it also considers the constraints that are tackled for the first time here in this thesis.

2.3.1 Constraints

Many constraints can arise in different organisations when allocating office space. Some of the commonly encountered constraints are presented in this section. Notice that depending on the specific organisation, any of these constraints can be *hard* (satisfied all the time) or *soft* (desirable but not necessary). Usually the degree of *hardness* or *softness* of an OSA constraint is directly related to the importance of the entities and/or room in that specific constraint.

Allocation: An entity should be allocated to a specific room. This constraint is generally used for allocating high ranked personnel (department heads, professors, etc.) to specific rooms which are selected specially for them. Rooms built for a specific purpose (kitchens, toilets, dining rooms, etc.) also use this constraint which can be referred to as the *room-role* relationship.

Non-Allocation: An entity should not be allocated to a specific room. This constraint is usually used for preventing some rooms being allocated to lower ranked personnel (students, workers, etc.).

Same Room: Two entities should be allocated to the same room. This constraint is usually used to allocate two entities working in the same profession together in the same room.

Not in Same Room: Two entities should not be allocated to the same room. This constraint is usually used to prevent two entities working in different professions or in different organisational groups being allocated to the same room.

Not Sharing: An entity should not share a room with others. This constraint is again usually used for higher-ranked personnel who should not share a room with others. It might also be desirable to use this constraint when the size of an individual entity is very close to the capacity of a room.

One of: An entity has to be allocated to one of the rooms. This constraint is useful when the floor layout contains rooms similar in size and close to each other and an entity can be allocated in any such *similar* rooms.

Capacity: A room should not be over/under used. This constraint is typically used for preventing overuse and/or underuse of some important rooms. This constraint is usually specified with a percentage of the room space that cannot be under/overused. In this thesis, a very strict version of this constraint is used: A room must not be over used beyond its capacity. Any amount of overuse is prohibited. However, the soft constraint violation penalty is still set to fixed value regardless of the amount of space overused.

Adjacency: Two entities should be allocated to adjacent rooms. This constraint is typically used for allocating entities with similar professions or within the same department, to rooms adjacent to each other.

Nearby: Also called the *group by* constraint, this constraint is similar to the adjacency constraint although it is reserved for groups (departments, research groups, etc.). The set of entities within a group should be allocated to the rooms close to the room to which a group head is allocated. The nearby relationship for rooms is larger than the adjacency relationship and typically covers a large section of a floor or the floor itself.

Away from : Two entities should be placed away from each other. This constraint is typically used for allocating two entities away from each other. When used together with nearby constraint, different departmental groups can be allocated away from each other. It is also useful when it is undesirable to have two role-room relationship together in a close area (such as placing a lecture room to a noisy communal area)

Distance minimisation: This constraint (or objective) can be used as an alternative to three proximity constraints (*adjacency*, *nearby*, and *away from*) to represent group relationships between different organisational entities. The goal is to minimise the total distance of entities in the same structural unit to a central point. Euclidean or Manhattan distance can be used for defining the distance metric.

Reorganisation: Also referred to as *re-allocation*, this constraint (or objective) is used when minimal disruption on the current allocation is desired during reassignment of

entities. It is a very common constraint especially if there is already an existing allocation of office space within an institution.

2.3.2 Quality Evaluation of an Allocation

There are two main objectives in the variant of the office space allocation problem [Landa-Silva, 2003] considered in this thesis: the minimisation of the space misuse and the minimisation of the soft constraint violations. The space misuse is the summation of the under and over utilisation of the rooms beyond their capacity. Since it is not desirable for the rooms to be overused beyond their capacity, the overuse of a room is penalised twice as much as under-utilisation of a room. In certain cases, it is not even possible to overuse a room at all. On the other hand, it is not desirable for a large number of rooms being under-utilised. In such allocations, it might be preferable to remove these severely under-utilised rooms from the problem, and redo the allocation task with the remaining rooms.

There are additional constraints that can happen frequently in many organisations. Nine types of such constraints are considered in this thesis. Six of these are taken from the previous work in [Landa-Silva, 2003]. These constraints can be either *hard* (must be satisfied all the time) or *soft* (desirable but not necessary).

The constraints and the soft constraint violation penalty values associated with them are defined in Table 2.1. The penalty values for each soft constraint violation were taken as in [Landa-Silva, 2003]. The penalty weights for the constraints introduced in this thesis (*non-allocation*, *not same room*, and *capacity*) were taken similarly to previous constraints. Notice that for the *nearby* constraint, two different penalty values were used: 11.18 for the *nott*1 instances (described in Section 2.4) and 10 for the instances generated for this thesis (described in Section 2.5.3). During the literature review, we could not trace any reasoning why the penalty value 11.18 was chosen for this constraint type. Hence, in order to have consistency with other types of constraints, the penalty value 10 was chosen for the instances generated for this thesis.

The difference between the *adjacency* and *nearby* constraints is that the *adjacency* constraint is related to rooms that are really close to each other, whereas the *nearby* relation deals with rooms that in a larger neighbourhood (like a floor or a specific large section of a building).

In office space allocation problem, the sets E, R, and C represent entities, rooms and constraints respectively. The set C is further divided into C_{soft} and C_{hard} to differentiate soft and hard constraints.

Type	Description	Weight
allocation	e should be in room r	20
non-allocation	e should not be in room r	10
same room	e_1 and e_2 should be in same room	10
not same room	e_1 and e_2 should not be in same room	10
not sharing	e should not share its room with others	50
adjacency	e_1 and e_2 should be in adjacent rooms	10
nearby	e_1 and e_2 should be in nearby rooms	10, 11.18*
away from	e_1 and e_2 should be away from each other	10
capacity	r should not be overused	10

Table 2.1: Description and penalty values for each constraint

In this thesis, the objective function that has to be minimised is taken as in [Landa-Silva, 2003]: the weighted sum of the space misuse and the soft constraint violations. The space misuse penalty SMP in a solution is given in equation 2.3.1:

$$SMP = \sum_{i=1}^{|R|} max((cap(r_i) - usg(r_i), 2 \times (usg(r_i) - cap(r_i)))) \qquad (2.3.1)$$

where $cap(r_i)$ and $usg(r_i)$ stand for the capacity and the used space of room r_i respectively. $|R|$ refers to the number of rooms in the problem.

The soft constraint violation penalty SCP is the weighted sum of the individual soft constraint violations which is given in equation 2.3.2:

$$SCP = \sum_{i=1}^{|C_{soft}|} (w(c_i) \times v(c_i)) \qquad (2.3.2)$$

where $v(c_i) = 1$ if the soft constraint is violated and $v(c_i) = 0$ if the soft constraint is satisfied. The weight of the soft constraint i is represented by $w(c_i)$. $|C_{soft}|$ refers to the number of soft constraints in the problem.

With SMP and SCP defined, the total penalty TP then becomes as in equation 2.3.3:

$$TP = \alpha \times SMP + \beta \times SCP \qquad (2.3.3)$$

where α and β are selected as 1.0 for this work.

2.4 Test Data Instances from UK Universities

2.4.1 University of Nottingham Dataset

These data instances were generated from the School of Computer Science and Information Technology in University of Nottingham during the 1999-2000 academic year. In the complete data instance (*nott*1) [Landa-Silva, 2003], there are 3 floors and 131 rooms with spaces ranging between 4.2 m^2 and 437.4 m^2. The number of entities is 158 and it is comprised of: 15 research rooms, 11 laboratories, 12 meeting rooms, 16 storage rooms, 6 professors, 1 reader, 5 senior lecturers, 25 lecturers, 16 research staff, 10 secretaries, 1 teaching assistant, 8 technicians, and 32 research students. The space requirements for the smallest and largest entities are 4 m^2 and 437.4 m^2 respectively. From the largest instance *nott*1, four more instances were generated by simplification. In this work, further corrections were done to the instance files because it was observed that some of the previous data instances (*nott*1*b* and *nott*1*c*) contained some invalid constraints. This was due to the fact that the instance data were defined in three separate files (room, entity, and constraint files) and there was some inconsistency between these files. In some of the constraints in the constraint files, there existed some rooms and entities that were not actually defined in the entity and room files. These entities, rooms, and constraints associated with them were removed from the instance files (in this case constraint files).

This thesis will deal with five *nott*1 instances described in Table 2.2. The numbers of entities, rooms, and each specific constraint are given. Notice that, the most important instances are *nott*1, *nott*1*b*, and *nott*1*c* (in this order) while the smaller *nott*1*d*, *nott*1*e* instances are usually used during algorithm implementation and testing.

2.4.2 University of Wolverhampton Dataset

This simple dataset gives the distribution of offices in the SC Building in the Telford campus of University of Wolverhampton during the 1999-2000 academic year. This instance contains 115 rooms, 115 entities and 115 *hard not sharing* constraints, so the final optimal solution is going to be a one-to-one mapping between the entities and rooms where each entity is placed to a single room in order to avoid each *not sharing* constraint.

Instance	nott1		nott1b		nott1c		nott1d		nott1e		Wolver	
Entities	158		104		94		56		86		115	
Rooms	131		77		94		56		59		115	
Constraints	H	S	H	S	H	S	H	S	H	S	H	S
Allocation	0	35	0	9	0	35	0	9	0	26	0	0
Same room	0	20	0	20	0	0	0	0	0	30	0	0
Not sharing	100	0	34	0	84	0	46	0	38	0	115	0
Adjacency	5	15	3	6	5	15	3	6	1	5	0	0
Grouped by	0	10	0	64	0	37	0	24	0	0	0	0
Away from	6	14	0	0	5	8	0	0	4	6	0	0

Table 2.2: Number of *hard* and *soft* constraints in the *nott*1 and *wolver* benchmark instances. [Landa-Silva, 2003]

2.5 Data Instance Generator for Office Space Allocation

The number of test instances in the office space allocation research literature is rather limited. One of the main goals in this research work is to provide new data instances for the research community. Thus, a new parametrised data instance generator algorithm was developed.

The design goals while developing this instance generator algorithm are as follows:

- *Realism*: Creating artificial test instances that do not have any bearing on the real world scenarios will not be beneficial to the research community. The algorithms created based on such unrealistic instances can lead to an erroneous understanding of the problem. Some of the real world instances are analysed in order to derive some relations between the entities, floor layouts, and constraints.

- *Scalability*: Although this research focuses mostly on small-medium type instances, larger instances with higher number of entities, rooms, and constraints may be required in the future. The generator should be able to handle instances from small to large while also retaining an understanding and control over the core fundamentals of the OSA problem.

- *Parametrised*: In order to analyse the nature of the office space allocation, extensive tests on different types of instances are required. The ability to make incremental changes to an instance is important to analyse how such a change affects the difficulty and the nature of instance. A parametrised data generator can handle

these small changes in the amount of core components of an OSA problem, like the space misuse and soft constraint violations.

2.5.1   Generation of Office Space Allocation Structures

The implementation was done to create instances totally randomly or with a optimal or near-optimal bound on the objective function. The generator currently includes nine types of constraints, and generation of entities, rooms, and floor layout. The following subsections deals with the generation of the entities, rooms and constraints.

A common OSA instance contains three set of structures: entities, rooms, and constraints. All of these three sets contain the relationships between each other.

Entities: Each entity is defined as part of a primary and secondary group. The primary group level indicates which structural organisation the entity is in (it could be a department, a research group etc.). Each primary group has one group head which is used for group by constraint. The distributions used for setting the sizes of the entities within a primary group are either uniform, constant or usually a decreasing distribution. The secondary group indicates the level or the profession of the entity (professor, lecturer, research student, etc.). The distributions within a secondary group for the size attributed follow constant distributions (i.e. the sizes of all the entities within a secondary group are the same).

Rooms: The floor layout is designed by dividing each floor into subsections (wings). After that, a specific wing is selected, the neighbouring relationships are defined according to a linear graph based representation. Each room in one wing is numbered in increasing order, and the likelihood of an adjacency relation between rooms numbered closely is higher. Being *near* relationship is either defined on the floors or the subsection of the floors.

Constraints: Currently, nine types of constraints are implemented. Each of these constraints can be set as *hard* or *soft*. The next subsections describe how each constraint is created with the generator.

- *Allocation constraint:* An entity is randomly selected and assigned to a randomly selected room. The likelihood of selecting an entity depends upon the secondary group level (importance) of that entity.

- *Non-allocation constraint:* An entity and a room are randomly selected, and a constraint is placed on them. Entities in lower ranked secondary groups are more likely to be selected.

- *Same Room Constraint:* A primary group is randomly selected, and two individuals are randomly selected from this group. The probability of selecting these two individuals is based upon the sizes of the entities and the secondary group level. It is more likely for individuals in similar lower level (research students, etc.) to be assigned to the same room.

- *Not in Same Room Constraint:* Two primary groups are randomly selected. One individual from each primary group is selected, and a constraint is placed on them. This disables people from different primary groups to be allocated to the same room. Alternatively, only one primary group can be selected, and then two individuals from different secondary groups can be selected. This disables people with different levels of professions to be assigned to the same room.

- *Capacity:* A room is randomly selected.

- *Not sharing:* A secondary group is selected. The priority is given to the higher ranked groups where the entity sizes are naturally larger (professors, group heads, lecturers, etc.).

- *Adjacency:* A primary group is picked, and two individuals from this primary group are selected.

- *Away From:* Two primary groups are selected, and a constraint is assigned between two individuals chosen from each of these two primary groups.

- *Group by:* A primary group is selected then binary constraints between the group head, and the other members of that group are set.

2.5.2 Data Instance Generator Algorithm

One of the goals of the generator is to create instances with known upper bounds on the optimal value. In order to create such instances, two approaches can be considered. The first approach starts with a predetermined floor layout (which holds the neighbourhood relation) with room sizes predetermined and then tries to create constraints, entities, and groups of entities out of it. The second approach starts with the groups, entities, and the constraints and then tries to create or modify the floor layout and/or the room sizes according to them.

In the generator, the second approach is taken because the modifications to the initial random generator are less than the first approach. The first approach is also more complex due to the modifications of three components (groups, entities, and constraints) as opposed to mainly modifying one component (the floor layout). The second approach greatly simplifies the implementation of the data instance generator algorithm.

There is also another important consideration for choosing the second approach. The second approach is more appropriate to simulate scenarios when a building is constructed for a specific purpose in mind. In such cases, the number of entities, groups, and people that may use the building and the relationships between such objects may already be (approximately) determined before the building construction. Even if the building is not specifically constructed for a usage scenario, it is still expected that a building will be rented or bought for a specific usage scenario. Hence, the second approach simulates such situations where the *usage scenario* (the relationships between the entities that will actually use the building) determines what kind of building will be used (rented, bought or constructed). If the usage scenario of the building is not important, then the first approach might be more suitable instead.

The pseudo-code for the *data instance generator algorithm* (DIGA) is presented in Figure 2.2. DIGA starts with the random generation (described in the previous section) of the groups, entities, groups, and initial set of *hard* and *soft* constraints. The algorithm then tries to create a solution out of these objects. Different strategies were considered for handling the placement of entities in the initial constraint set to the rooms, and it was decided on the following strategy: Certain constraint types were given priority over other constraints in the order the generator algorithm evaluates them. While random and no specific constraint biased orderings were also tested, these random orderings usually generated very easy to solve instances. Therefore, after some initial experimentation, the following order of satisfying constraints yielded sufficiently difficult instances: *same room, nearby, not sharing, allocation, adjacency, away from, non-allocation, not same room, capacity*.

The priority of *same room* constraint was due to the rather large percentage gaps between the optimal objective function value and the bound on this value when the instance contains a large number of constraints of this type. There was a large number of *nearby* constraints in a typical OSA instance. In order to satisfy a maximum amount of these constraints, this type was given a large priority as well. It could be very hard for *not sharing* constraint to coexist with other constraint types (due to constraint clashes) even if a large number of entities was initially given these constraint, so its priority

was set high as well. The *capacity* constraint was handled last because it was the only constraint that was associated with rooms; thus, it was not affected by the placement of entities related to other constraints. There was not a significant reasoning behind the ordering of other constraints other than the initial number of such constraints set as in Section 2.5.1.

In order to prevent a future OSA algorithm exploiting this ordering structure, the ordering of constraints in each type is randomised. For example, if there are 30 *allocation* constraints, a random permutation of these constraints is created, and this ordering is then given to the DIGA algorithm. DIGA evaluates these *allocation* constraints according to the random permutation. Future algorithms are not going to have access to this ordering knowledge the generator uses to create the instance. Different orderings in each constraint type can greatly affect the final instance generated by the DIGA algorithm. This randomisation should prevent future OSA algorithms exploiting the design bias of constraint ordering in DIGA because the constraint ordering in the instance file will differ greatly from the ordering the DIGA algorithm operates with.

Input: input file of parameters.
Output: data instance.
 1: — Creation of entities (with sizes) organised within different structural groups
 2: — Generation of floor layout.
 3: — Creation of Initial Set of *hard* and *soft* constraints.
 4: — Placement of entities according to the *hard* constraints.
 5: — Calculate space that must required for each room.
 6: — Placement of entities in the *soft* constraints.
 7: — Room size adjustments via (positive or negative) slack spaces.
 8: — Post processing

Figure 2.2: Data instance generator algorithm (DIGA)

The algorithm first places each group head to a room. Each *hard* constraint is selected one by one and entities in that constraint are placed into rooms without violating the respective constraint. If an entity is present in two different constraints, then following modifications to the constraints may be necessary:

- If an entity in an *allocation* constraint is already placed before, the room in the constraint is replaced with the room the entity is already placed.

- If both entities in a *same room*, *not same room*, *adjacency*, *near*, *away from hard* constraint are placed before, the constraint is deleted unless the room placement of the entities satisfies the respective constraint. If the constraint is *soft*, then the

constraint is deleted stochastically (with *violation* rate *V*). If only one entity is placed, the other entity in that constraint is placed according to the constraint specifications.

After all the entities in the *hard* constraints are placed, the minimum space (*hard capacity*) required for each room to hold these entities is calculated. After this step, a permutation is created using *soft* constraints with the aforementioned priority ordering and all the related entities are placed into rooms as in *hard* constraints.

After all the entities related to the constraints are placed to the rooms, the remaining entities (if any) are placed to randomly selected rooms. At this stage, the capacity for each room should be readjusted. Two different methods are used to create the final instances. In *tight fit* instances, the capacity for each room is set to the total space required for all the entities allocated to that specific room. These instances have zero space misuse and no *soft* or *hard* constraint violations, thus the optimal objective value for these *tight fit* instances is zero. Another approach is to adjust these final room capacities for adjusting the space misuse. Some rooms are randomly selected, and either additional space is added to the capacity (*positive slack*) or some space is subtracted from the room capacity (*negative slack*). When *negative slack* is used, the generator controls whether the adjusted room capacity is still valid for satisfying the *hard* constraints. After these final adjustments, an instance with an upper bound of *positive slack* $+2 \cdot$ *negative slack* on the optimal objective function value is created.

The initial desired goal while developing the algorithm was to create instances with known optimal values on the objective function value, the space misuse components or soft constraint violation penalty. However, it was quickly observed that such an approach was extremely difficult with the current algorithm. The only optimal values that the algorithm could guarantee were for the *tight fit* instances (which is described in the paragraph above). It was quickly observed that the moment *positive* and *negative* slack amounts were added to individual rooms, the control on the optimal objective function value, the space misuse components, and the soft constraint violation was quickly lost. In our preliminary tests with the algorithms for office space allocation (which will be described in Chapters 4, 5, 6, and 7), a lot of higher quality solutions (solutions that were better than the ones given by the DIGA algorithm) could be found if the instances begin to differ from the initial *tight fit* instances that they were derived from. Hence, it became excessively difficult to find tight upper bounds on the optimal objective function value for these generated instances. The upper bound generated by the algorithm (*positiveslack* $+ 2 \cdot$ *negativeslack*) was unfortunately not tight enough to be useful.
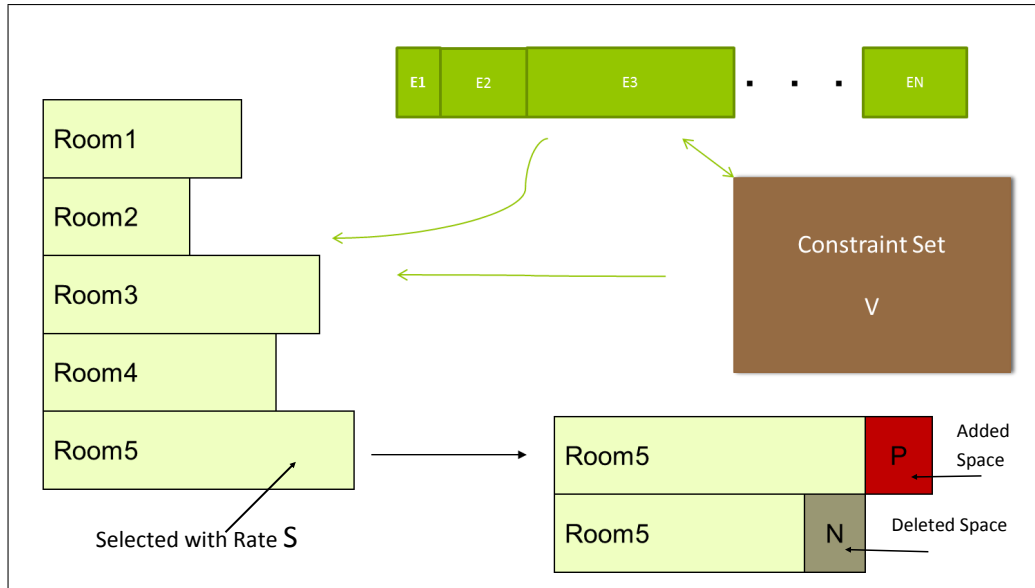
Figure 2.3: Depiction of S, V, P and N parameters on entity, room, and constraint sets within the data instance generator algorithm for OSA.

DIGA Parameters: There are four basic different parameters for adjusting the difficulty of the instance created. The relationships between these parameters and entity, room, and constraint sets are depicted in Figure 2.3. These four parameters are:

- *Slack Space Rate (S):* After all the entities are placed into the rooms, this rate determines whether a room will have more or less space than required by the entities already within it. This parameter adjusts the amount of space misuse.

- *Negative Slack Amount (N):* Determined by a percentage of the total sizes of the entities already placed into the room; the room capacity is reduced by this amount. This parameter adjusts the amount of space overuse.

- *Positive Slack Amount (P):* Determined by a percentage of the total sizes of the entities already placed into the room; the room capacity is increased by this amount. This parameter adjusts the amount of space underuse.

- *Violation Rate (V):* When the entities in the soft constraints are to be placed into the rooms, there may be some violations between constraints. A constraint is removed from the constraint set probabilistically if a conflict occurs.

We can explain these parameters on Figure 2.3. Let $S = 0.20$, $P = 0.20$, $N = 0.10$, and $V = 0.80$. In this case, it is expected that on the average, one out of five rooms will be selected for space misuse manipulations ($S = 0.20$). The higher the value of S, the more rooms will be probabilistically selected for space misuse manipulation.

After a room is selected for space manipulation, we have to decide whether the room capacity should be increased or decreased and the amount of such increase or decrease. The room capacity can either be increased or decreased (not both). The probabilities of the decisions to increase or decrease the room capacity are equal (which is fifty percent). The parameter P probabilistically determines the percentage of space that will be added to the room capacity. For example, if the room capacity is initially 25 square meters, the room capacity will be increased by a value drawn from uniform distribution $(0, ..., 5)$ because $P = 0.20$ (twenty percent of 25 is 5). Conversely, the room capacity can be decreased by a value drawn from uniform distribution $(0, ..., 2.5)$ because $N = 0.10$ (ten percent of 25 is 2.5).

The V parameter controls the number of soft constraints violations. Whenever the data instance generator algorithm faces conflicts in soft constraints, it has to decide whether it should keep one of the conflicting soft constraints in the instance or remove it. For example, if $V = 0.80$, eighty percent of the soft constraints that the algorithm determines a conflict with other soft constraints will be kept in the instance. The remaining twenty percent of such conflicting soft constraints will be removed from the instance.

2.5.3   SVe150 and PNe150 Datasets

To be used in experiments in this thesis, two data sets were created using the generator. These data sets were generated to investigate the effects of incremental parameter changes on the difficulty of the problem. They were also used to conduct performance comparison between various algorithms and any further analysis.

There are two datasets that were created specifically for this thesis: *SVe*150 and *PNe*150.

Attribute	Value
Number of entities:	150
Entity size:	5.5 - 30.5
Number of Groups:	10
Number of Floors:	3
Number of Rooms:	92
Number of Hard Constraints:	67
Number of Soft Constraints:	185 - 214

Table 2.3: Attributes of the instances in *SVe*150 and *PNe*150 datasets

In each of these data instances, the generator starts with the same entity set (that is

the size of the entities, the number of groups, and the groups the entities belong to are the same initially). All the *hard* constraints in these instances are the same. The generator starts with the same initial *soft constraint set* as well. Some of the *soft* constraints are pruned based upon the *V* parameter. Also, the initial floor layout (the neighbourhood relation between rooms) is the same across all data instances. The aim for this incremental build is due to the intent of exploring the effects of incremental changes of the parameters on the instances.

These parameter values were decided after preliminary experimentation with generation of data instances. In *SVe*150 dataset, the *S* and *V* parameters were varied between 0.00 and 1.00 with 0.20 increments to provide 36 instances. *P* and *N* parameters were set to 0.10 for this dataset. In *PNe*150 dataset, the *P* and *N* parameters were varied between 0.00 and 0.25 with 0.05 increments. This provided another 36 instances.

In Table 2.3, the attributes of each data set are summarised. The fixed parameters in both datasets were decided after preliminary experimentation with integer programming models (described in Chapter 4). The main goal for setting these fixed parameters was to create sufficiently difficult instances (that were not solved rather quickly). Another aim was to create a consistent basis for the instance analysis based on the four parameters. This analysis will be described in Section 4.6 in more detail.

While creating these two data instance sets, the *nott*1 dataset was used an inspiration to determine several attributes of the instances. The number of entities, the size range for these entities, and the number of groups and floors were taken similarly to the case as in *nott*1 instance (which was the largest real world considered in this thesis and in the literature). The number of rooms was determined during the data instance generator algorithm in a way *all* rooms in the instance were expected to be used. The number of each specific constraint was taken similarly to *nott*1 instance. However, the number of *not sharing* constraints was not as numerous as in *nott*1 because it became excessively difficult to increase the number of hard *not sharing* constraints in the instance while still keeping the instance solvable. In certain cases, it was not even possible to generate the instance in the first place with such a large number of hard *not sharing* constraints in addition to other constraints.

Although any of the constraints in office space allocation problem can be *hard* or *soft*, in our test instances, not all cases were considered. For example, the *allocation*, *non-allocation same room*, *not same room*, and *nearby* constraints were always set as *soft* constraints. On the other hand, *not sharing* constraints were always *hard*. These were set as such in *SVe*150 and *PNe*150 datasets in order to have consistency with *nott*1 instances. Also, the relatively higher number of *soft* constraints compared to *hard* constraints was

Constraint	SVe150		PNe150	
	Hard	Soft	Hard	Soft
allocation	0	32	0	32
non-allocation	0	10	0	10
same room	0	25	0	25
not same room	0	10	0	10
not sharing	60	0	60	0
adjacency	1	3-15	1	9
nearby	0	90-103	0	93
away from	4	11-15	4	13
capacity	2	4	2	4
Total	67	185-214	67	196

Table 2.4: Number of constraints in $SVe150$ and $PNe150$ datasets

due to the desire to make the objective function more complicated and hence, make the instance more difficult to solve. By making the soft constraint violation penalty part in the objective function more complicated, it was also desired that the problem solving was not dominated simply by the minimisation of space misuse components. This was due to the conflicting nature of soft constraint violation penalty and the space misuse penalty (which was reported in [Landa-Silva, 2003]). The numbers of *hard* and *soft* constraints for each type in $SVe150$ and $PNe150$ instances are given in Table 2.4.

2.6   Conclusion

In this section, a background information regarding the office space allocation was provided. Two case studies for office space allocation in NASA LaRC Research Center and universities were explained. The constraints, the objective function, and the data sets were explained. Also, the implementation of a data parametrised data instance generator algorithm was provided.

During the rest of the thesis, nine types of constraints (*allocation, non-allocation, same room, not same room, not sharing, adjacency, nearby, away from, and capacity*) will be used during algorithm implementation in Chapters 4, 5, 6, and 7. The *nott*, $SVe150$, and $PNe150$ instances will be heavily investigated using these proposed algorithms.

For the rest of the thesis, OSA problem variant that is given in Section 2.3.2 will be used. The objective function given in equations 2.3.1, 2.3.2, and 2.3.3 will be used throughout the thesis.

Literature Review

3.1 Introduction

This chapter is devoted to the literature review related to office space allocation (OSA) and several exact and heuristic approaches. Some theoretical problems related to office space allocation and how these problems can be thought of as variants of OSA are also explained. These problems are multi-dimensional knapsack, generalised assignment, bin packing and clustering problems. The office space allocation problem variant undertaken in this thesis can be considered as a combination and modification of these problems in some aspects.

The theoretical or practical research on the office space allocation problem is rather limited. One of the aims of this chapter is to give detailed information about the past work in this area. Various mathematical models and meta-heuristic approaches to solve different variants of OSA are explained. Information about other practical space allocation problems is also given in this chapter.

Since the office space allocation problem is a combinatorial optimisation problem, preliminary concepts of computational complexity are explained. Solving many versions of office space allocation systems may consume heavy amount of processing power. However, they can usually be efficiently implemented in terms of memory requirements. Hence, the focus of this chapter related to computational complexity will be issues in time-complexity. These concepts include *decision* and *optimisation* problems, the classes of *P*, *NP*, *NP-Complete*, *NP-Hard*, and the *no-free-lunch theorem*.

There are various solution approaches that can be used for OSA. This chapter includes a review of some of the mathematical models and meta-heuristics that have been proposed previously in the literature and that are revisited in this thesis. Since the focus of this thesis is on the application of mathematical programming and meta-

heuristics, the literature review in this chapter targets *local search* meta-heuristics, *genetic algorithms*, and *integer programming* solution techniques.

This chapter is organised as follows: some of the theoretical problems related to the office space allocation are described in Section 3.2. Section 3.3 presents previous research performed in the area of office space allocation. Some of the practical problems similar to office space allocation are presented in Section 3.4. Section 3.5 deals with some of the important topics in computational complexity (P, NP, NP-Complete, NP-Hard problems, and No free lunch theorem). Section 3.6 describes a range of algorithms that can be considered for solving office space allocation problems. Section 3.7 concludes the literature review.

3.2 Theoretical Problems Related to Office Space Allocation

In this section, some of the theoretical problems related to the office space allocation problem are formulated. OSA can be thought of a generalisation or combination of these problems with additional constraints and objectives.

3.2.1 Bin Packing Problem

Bin packing problem (BPP) is a difficult combinatorial optimisation problem in which items of different sizes have to be packed into a minimal number of bins of fixed capacity. Bin packing has many variants with respect to the number of dimensions used, the arrival of bins, and the distribution of the size of the items. The one dimensional bin packing problem can be formulated in equations 3.2.1, 3.2.2, 3.2.3, 3.2.4, and 3.2.5 as in [Martello and Toth, 1990]:

$$minimise \quad \sum_{i=1}^{n} y_i \tag{3.2.1}$$

$$s.t. \quad \sum_{j=1}^{n} s_j x_{ij} \leq c y_i \qquad i = \{1, \ldots, n\} \tag{3.2.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad j = \{1, \ldots, n\} \tag{3.2.3}$$

$$y_i \in \{0, 1\} \qquad i = \{1, \ldots, n\} \tag{3.2.4}$$

$$x_{ij} \in \{0, 1\} \quad i = \{1, \ldots, n\}, j = \{1, \ldots, n\} \tag{3.2.5}$$

where binary decision variable y_i indicates whether the bin i holds an item or not, and each binary decision variable x_{ij} indicates whether the item j is placed in the bin i, and

n stands for the number of items being packed. The size of each item *j* is represented by s_j, and the capacity of a bin is represented by *c* (each bin has the same capacity). In this formulation, there are *n* items, and as a result the maximum number of bins (whether they are used or not) is again *n* as the size (s_j) of a single item cannot exceed the capacity (*c*) of a bin. The goal of BPP then becomes to minimise the number of bins that are used (where $y_i = 1$) as in equation 3.2.1 while packing all items to the bins (equation 3.2.3) without going over the capacity *c* in any used bin (equation 3.2.2).

BPP can be thought of a limited variant of OSA when all the rooms (bins) in the problem have a *capacity* constraint on them (that no room can be overused beyond its capacity), and all rooms are of equal size. The items to be packed into the rooms are the entities of various sizes, and the goal is to allocate all entities to minimal amount of rooms subject to additional constraints.

3.2.2 Multi-dimensional Knapsack Problem

Multi-dimensional knapsack problem (MDKP) is a generalisation of the traditional knapsack problem [Kellerer et al., 2004] by adding more constraints (dimensions) into it. In MDKP, there are more than one knapsack each with their own capacities. Each item that is to be placed into the knapsacks has a profit and size value associated with it. The goal is to place the items into the knapsacks such that the total profit is maximised while not overusing the knapsacks beyond its capacity. The mathematical definition of the problem as in [Puchinger et al., 2009] is given in equations 3.2.6, 3.2.7, and 3.2.8:

$$maximise \quad \sum_{j=1}^{n} p_j x_j \tag{3.2.6}$$

$$s.t. \quad \sum_{j=1}^{n} w_{ij} x_j \leq c_i \quad i = \{1, \ldots, m\} \tag{3.2.7}$$

$$x_j \in \{0, 1\} \quad j = \{1, \ldots, n\} \tag{3.2.8}$$

where there is a set of *n* items with profits $p_j > 0$ and *m* resources with capacities $c_i > 0$. Each item *j* requires an amount of $w_{ij} \geq 0$ from each resource *i*. The binary decision variables x_j indicates which items are selected. The goal of the MDKP is to select a subset of items with maximum total profit while not exceeding the resource constraints of the problem.

The MDKP resembles the OSA where each room has a hard *capacity* on it and cannot be overused, in this case, each room can be thought of as a knapsack. Each item (entity) has an associated size with it and is to be placed into the knapsacks (items).

The difference between MDKP and OSA is that MDKP has an explicit profit values associated with each entity while OSA decides the cost of placing an entity to a room dynamically based upon the other constraints of the room and the current allocation. MDKP is also a maximisation problem whereas OSA minimises the objective function.

3.2.3 Generalised Assignment Problem

Generalised assignment problem (GAP) is the task of assigning n jobs to m agents such that each job is uniquely assigned to one agent. The goal is to distribute all the jobs to agents to minimise the total cost while not exceeding the capacity of an agent. GAP is similar to MDKP. However, in MDKP, the profit of assigning an item to any knapsack is the same whereas in GAP, each assignment of a job to a particular agent has a different cost value. The mathematical definition as used in [Cattrysse and Wassenhove, 1992] is given in equations 3.2.9, 3.2.10, 3.2.11, and 3.2.12 :

$$minimise \quad \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} \qquad (3.2.9)$$

$$s.t. \quad \sum_{j=1}^{n} a_{ij}x_{ij} \leq b_i \qquad i = \{1,\ldots,m\} \qquad (3.2.10)$$

$$\sum_{i=1}^{m} x_{ij} = 1 \qquad j = \{1,\ldots,n\} \qquad (3.2.11)$$

$$x_{ij} = \{0,1\} \qquad i = \{1,\ldots,m\}, j = \{1,\ldots,n\} \qquad (3.2.12)$$

where c_{ij} is the cost of assigning job j to agent i, a_{ij} is the capacity absorption when j is assigned to agent i, b_i the available capacity of agent i. The decision variable x_{ij} is equal to 1 if agent i performs job j, 0 otherwise.

The GAP can be thought of OSA problem that each job (entity) has a cost (size) associated with it, all the agents (rooms) have a hard *capacity* constraint and the goal is to distribute each job (entity) to agents (rooms) such that the cost of allocation is minimised. The difference is that while GAP uses explicit cost values for assigning each job to an agent, OSA sets the cost of allocation of an entity to a room based upon the constraints of the problem and the current allocation.

3.2.4 Clustering

Cluster analysis [Everitt et al., 2009] is the task of clustering (groups) a set of objects such that the objects that are similar to each other (to a predefined metric) are placed in the

same cluster while dissimilar objects are grouped in other clusters. Clustering analysis is a very important area in data mining [Witten and Frank, 2005].

In clustering analysis, the models are derived based on the definition of the similarity of the objects (that which objects form a cluster and which do not). There are various models based upon different similarity metric, such as hierarchical clustering, centroid models such as k-means, statistical distribution and density models, and graph based structures.

A very commonly researched problem in clustering analysis is *k-means clustering*. In this problem, the goal is to create K clusters while minimising the euclidean distance of each entity to the cluster centre. The mathematical definition of the problem is given in equations 3.2.13, 3.2.14, 3.2.15, and 3.2.16.

$$minimise \quad \sum_{i=1}^{n}\sum_{k=1}^{K} z_{ki}||x_i - c_k||^2 \tag{3.2.13}$$

$$s.t. \quad c_k = \frac{\sum_{x_i \in C_k}}{N_k} \qquad\qquad i = \{1,\ldots,n\} \tag{3.2.14}$$

$$N_k = \sum_{i=1}^{n} z_{ki} \qquad\qquad k = \{1,\ldots,K\} \tag{3.2.15}$$

$$z_{ki} = \{0,1\} \qquad i = \{1,\ldots,n\}, k = \{1,\ldots,K\} \tag{3.2.16}$$

where there are K clusters and n items. N_k represents the number of items within a cluster. The c_k is the centre of each cluster k, and z_{ki} is 1 if the item i is associated to cluster k, 0 otherwise. The coordinate of item i is represented by x_i. Each cluster centre c_k of cluster C_k is determined by the arithmetic mean of the items in the cluster C_k.

The clustering problem can represent some of the constraints in OSA such as the proximity constraints (*adjacency*, *nearby*, *away from*). In OSA, it is often desired that the entities within the same organisational unit are placed (clustered) close to each other with *adjacency* and *nearby* constraints. On the other hand, certain units or rooms should not be allocated within close proximity to each other as defined in *away from* constraints. In this case, one goal can be to create clusters of entities where the distances between each entity in an organisational unit are minimised while the distances between different and unrelated units are maximised.

3.3 Previous Research on Office Space Allocation

Sharpe: The earliest study of optimum space allocation within buildings that the author of this thesis can trace back to belongs to [Sharpe, 1973]. In this article, Sharpe pro-

posed a mathematical model technique for optimising floor layouts within buildings with two main objectives: the minimisation of circulation and establishment costs. The model was an extension of a quadratic assignment model as in [Beckman and Koopmans, 1957] with zero-one variables replaced by continuous variables. The model was applied to a single-storey industrial building comprised of 6x6=36 grid locations. 19 different activities (the role of a room) were allocated based upon the interaction between each activity level. The level of interaction between each activities depended on staff-contact hours (the interaction level) which were calculated according to the number of hours a staff member of activity A spent with activity B. In addition to the interaction of different activities, benefits or penalties of allocating some activities to certain regions or the zones within a building were also considered.

Ritzman: One of the earliest works on the optimisation of office space utilisation is in [Ritzman et al., 1980] who developed a linear programming model for the distribution of academic offices at the Ohio State University. The objectives they considered were assigning offices to departments with enough space for each member, minimising the space deviation of each department from the space requirements, distribution of the offices with air conditioning evenly among departments, minimising the distance between the rooms in a department and the administrative offices, even distribution of high quality offices between different departments, and minimising the reassignments of offices. The authors focused on producing multiple layouts interactively for enabling the administrators to have control of the solution stage and for allowing them to choose from a set of solutions.

Benjamin: A linear goal programming model was proposed for planning the layout of floor space in a manufacturing laboratory in University of Missouri Rolla in [Benjamin et al., 1992]. Five different conflicting objectives were considered: increasing the usage of laboratory facilities, development of new courses, stimulation of research, increasing the awareness of industry, and image of the university. The authors pointed out that prioritisation of each constraint required a significant amount of time and expert knowledge of the administrators was needed especially due to the different preferences of each administrator about the goals of the project.

Giannikos: A goal programming [Tamiz et al., 1998] approach to automate the distribution of offices among staff in an academic institution was proposed in [Giannikos et al., 1995]. The objectives considered were: assigning offices with specific type evenly

among the schools, assigning specific rooms to the usage of only one school, minimising the relocation of people after the allocation, and minimising the distance of each room in one school to its administrative centre. Additionally, hard constraints included the allocation of school heads to a office conforming to the standards, and the usability of each office by the members of the staff in the same level or category. The authors compared their results to the actual usage of the office spaces and concluded that the space was allocated inefficiently with a manual method.

Burke et al.:   A questionnaire by [Burke and Varley, 1998] was reported on the space allocation process in 38 British universities. The emphasis was on the scope of the problem, computing tools to solve it, and the constraints in each university. It was reported that very few universities used a fully automated allocation system although most of them still used computers during the allocation process. The survey also indicated that almost half of the institutions did not use *staff should not share the rooms* constraints, particularly the older universities. It was also reported that most universities extensively used the positioning and locality constraints to group resources belonging to the same group together and away from other groupings. It was also suggested that although the *matching the room sizes* constraint could be used to reduce the *reorganisation* process, it was rarely used in universities. The authors concluded that any automated allocation system should cover all aspects of the space allocation process and should be easy to use. They also emphasised that the system's usage of the real world data and ability to cope up with missing information (especially the floor layouts) was paramount.

Research on developing heuristic approaches for the office space allocation problem began with [Burke et al., 2001b]. The authors applied hill climbing, simulated annealing [Kirkpatrick et al., 1983], and a genetic algorithm [Goldberg, 1989] to solve the *optimisation* (task of creating a complete solution from scratch) and *reorganisation* (task of reallocating entities in a given solution) problem. Three different move operators were used: *Allocate* move chose an unassigned resource and allocated it to a room. *Relocate* move changed the room for an already allocated resource. *Swap* move interchanged the rooms of the two selected resources. For searching which resources and rooms should be used in these moves, random and greedy methods were used. By applying their algorithm to the real world data from three universities, the authors observed that the hill climbing using greedy search for room allocation worked the best for the initial optimisation of the problem. However, simulated annealing and genetic algorithm using greedy room search yielded better results during the *reorganisation* problem.

[Burke et al., 2001a] later investigated a hybridisation of their previous approaches under a population based framework. The initial solutions were created using a hill climbing operator and were improved using a simulated annealing approach using adaptive cooling technique. A mutation operator was also implemented for disrupting the solution if the search could not find a feasible move to improve the solution quality. The authors further augmented a population into their algorithm where the goal was to bring an individual into a *stable state* by using hill climbing and simulated annealing. A *stable state* occurred when either an individual reached a local optima or no further improvement was achieved using the simulated annealing stage. The population based approach was further modified to produce an overall high quality population or a single higher quality solution. The authors compared the population based algorithm to its single solution counterpart and no real advantage in terms of solution quality was observed.

[Burke et al., 2001c] further reported the application of multi-objective optimisation [Coello et al., 2006] to the office space allocation problem. The algorithm described above was modified to make a comparison between a weighted aggregating objective function versus a bi-criteria multi-objective optimisation using pareto domination. Two objectives were considered: the total space misuse due to under/over usage of the rooms, and the weighted sum of the other constraints. It was observed that these two objectives were conflicting in nature. The bi-criteria method was further tailored to produce a population with high diversity of good solutions or a population with a single high quality solution.

Landa-Silva: Prior to this thesis, the most recent results on benchmark instances of the office space allocation problem investigated here, were reported in the paper by [Landa-Silva and Burke, 2007] who developed an asynchronous cooperative local search method. In this work, local search threads in a population co-operated with each other asynchronously to improve solution quality. In order to improve the global search strategy, a pool of genes which contained segments of the solutions was shared between the local search threads (*information sharing strategy*). This shared information was used in two ways: parts of the solutions which were marked as good were used to improve the solutions in individual threads whereas the parts of the solutions marked as "bad" were ignored during a *diversification strategy* by reconstruction with a disruption operator. The authors compared four different single-solution heuristics (hill-climbing, simulated annealing, tabu search, and a hybrid meta-heuristic) with their population based variants using their asynchronous cooperative mechanism. It was

pointed out that their population based asynchronous algorithm which was extended from a single-solution hybrid meta-heuristic provided the best results while a population based tabu search algorithm provided competitive results. Based upon the experiments, due to the high number of violations, *group by* constraint was regarded as the most difficult one in the heuristics approaches implemented.

Lopes: [Lopes and Girimonte, 2010] extended the work by [Landa-Silva and Burke, 2007] in an office space allocation problem in European Space Agency (ESA). They implemented four types of meta-heuristics: Hill climbing, Simulated Annealing, Tabu Search, a Hybrid meta-heuristic by [Burke et al., 2001a]. To improve the performance of these algorithms, variations to the local search, and new constraints management algorithms were designed by the authors. They also pointed out that in ESA, there was a strong hierarchy amongst the entities, therefore their entity structures took what they called the 'standard code' (category of the entity based upon the size and sharing) and the 'address code' (for the hierarchy of the units, and grouping information for the entities) into consideration. The OSA problem contained *allocation*, *adjacency*, *not sharing* constraints as described in [Landa-Silva, 2003]. Additionally, the problem contained two types of *grouping* constraints (at department and division level) and *avoid spreading* (to avoid the split of departments over several buildings/floors) which was similar to the *nearby* constraint in [Landa-Silva, 2003] and *avoid share across* (to avoid the sharing of offices between different departments) constraint. The authors made a clear distinction between constraints by dividing them into two sets: the *particular constraints* (*allocation* and *adjacency* constraints) which applied to a specific subject and/or target and *global constraints* (the rest of the constraints) which applied to every entity. When the algorithm was tested on the ESA instance by using *swap*, *relocate*, and *interchange* moves as in [Landa-Silva, 2003], the authors reported the *Hybrid meta-heuristic* performed the best when the run length was sufficiently long, and *Tabu Search* converged faster when the run length limit was short. The authors then modified the *swap* and *relocate* move operators with a property called *length*. In *fixed length* operators, the number of entities and resources handled by the operator were fixed, whereas in *variable length* operators, the number of entities and resources were randomly generated between 1 and *length* at each iteration. The experimentations pointed out the performance of *fixed length swap and relocate* operators were very poor whereas the *variable length* operator slightly improved the performance of the algorithms.

Zahiri: An application of particle swarm optimisation [Poli et al., 2007] was implemented in [Zahiri, 2009] to solve OSA. An adaptive fuzzy system integrate with

a multi-objective particle swarm optimiser (Fuzzy-MPSO) was proposed to tackle the same OSA problems described in [Landa-Silva, 2003]. Three design goals were considered: good generalization for the proposed algorithm, maximisation of the number of non-dominated solutions in the Pareto front, and the maximisation of the spread of non-dominated solutions as smooth and uniform as possible. Two performance metrics (named aggregation factor and minimal spacing) were considered in this algorithm. There were three fuzzy inputs in the algorithm: the aggregation factor, *minimal spacing* for minimising the distance between the solutions, and the number of iterations where the non-dominated points on the pareto front were not changed. Seven fuzzy rules were used based on the linguistic description based on the parameters of the algorithm. In this Fuzzy-MPSO algorithm, the values of the swarm size (the number of swarms in the population, constriction coefficient, and neighbourhood size (fuzzy outputs) were dynamically adjusted for achieving trade-offs between the better-coverage, uniformity, and closeness to the optimal pareto-front. The experiments on some of the *nott1* instances yielded acceptable results.

Trung: In [Trung et al., 2009], the authors tackled the problem of dorm room assignment in HCMC University of Technology by using an improved version of simulated annealing (SA) called *informed SA* (ISA). The task was to assign 2500 students to the dorm rooms subject to *hard* and *soft constraints* and to minimise the objective function value (the *unhappiness* of the students). The *hard* constraints included gender restrictions to keep male and female students separate, room capacity restrictions, satisfaction of privileged students, and the room change requests from the senior students. The *soft constraints* were divided into two categories: *room-student* and *student-student* constraints. The *room-student* constraints included dissatisfaction levels related to the preferences for the hall, room type, beds, rent price, floor, reassignment and room change requests, and partial fill (underuse) of the rooms. The *student-student* constraints included dissatisfaction levels related to the preferences for music, study habit, computer game, smoking, friend, room-mates of one student for another. The objective function to minimise was the weighted sum of these *soft* constraint violations (*unhappiness* value).

The ISA algorithm contained two stages: the first stage satisfied all the *hard* constraints while the second stage tried to minimise the number of *soft* constraint violations. The neighbourhood operator was swapping the rooms of two students of the same gender. The algorithm used *utility* values for estimating the probability that a variable-value pair (the student-room assignment pair in this case) appeared in an op-

timal solution. The utility was considered significant when it was deemed important for improving the solution of the SA. A *threshold* value was set in order to separate significant utilities from non-significant ones. Another important component of the algorithm was the *repair* procedure which chose a variable (student) randomly and replaced the current value (room) assigned to this variable by randomly choosing a new value in order to remove the conflicts.

The authors compared their ISA algorithm with the regular SA algorithm in [Trung et al., 2009] and for the dataset of HCMC University of Technology, a speed-up of twice the previous performance was observed. The authors later compared their ISA algorithm with SA with non-monotonic reheating and a very-fast SA algorithm with re-annealing in [Trung and Anh, 2009] and [Anh and Trung, 2011] and claimed their ISA algorithm was the most time efficient one.

Pereira: [Pereira et al., 2010] applied a greedy local search and tabu search algorithm to investigate an OSA problem where the goals were to minimise the distance between the employees in the same organisation, minimise the office space misallocation, and maximise the office space allocation. The authors used a weighted objective function containing these three sub-objectives. An instance generator was created, and then a greedy local search and tabu search algorithm were used by utilizing two move operators: either the room assignment of two entities were swapped or an entity was moved to an empty available position. Tabu search performed better on the instances created by the authors.

Adewumi: A recent work by [Adewumi and Ali, 2010] was done in multi-stage hostel space allocation problem based on data on a tertiary institution. The problem involved satisfaction of the constraints based upon the accommodations of certain categories of students (foreign, freshman, senior, scholar, health, sports, discretionary) and maximization of the space utilisation. The first stage of the algorithm tried to ascertain the number of students in each category while the capacity in available hostels was not being overused. In this stage, all students in certain categories were given high priority in allocation. Remaining students were allocated in greedy fashion based upon their priority level. In the second stage, students were distributed to the hostels under specific constraints. Both of these stages used a genetic algorithm (GA) which incorporated representations based on multidimensional array structures for hall and block/floor allocations. One point crossover and random mutations which increased or decreased the number of allocations at hall or block level were used in the algorithm.

Awadallah: Awadallah et al. [Awadallah et al., 2012] applied harmony search algorithm (HSA) [Geem et al., 2001], [Geem, 2008] to solve the same office space allocation problems as in [Landa-Silva, 2003], [Landa-Silva and Burke, 2007]. The harmony search algorithm was a population based meta-heuristic inspired from musical improvisation. There were three operators to generate a new harmony at each iteration: memory consideration, random reconsideration, and pitch adjustment. In this algorithm, first, the variables (information about the entities and the rooms) of an office space allocation problem were extracted. The second step included setting up some of the HSA parameters: The harmony memory consideration rate was for selecting the value that would go through memory or random consideration procedures. The harmony memory size determined the number of solutions stored in harmony memory. The pitch adjustment rate determined the rate of local improvement. Each row in the harmony memory corresponded to a solution of the space allocation problem, and these rows were constructed using the peckish method described in [Corne and Ross, 1996]. Each feasible new harmony was improvised at each iteration using memory consideration (selecting the value of the current decision variable from the best solution stochastically), random consideration (assignment of decision variables to rooms), and pitch adjustment (improvement procedures that employed *move, swap, interchange* operators similar to the ones in [Burke et al., 2001b] and [Burke et al., 2001c]). The algorithm was tested on some instances in the *nott1* and *wolver* datasets, and performance of the algorithm was adequate.

3.4 Other Practical Problems Related to Office Space Allocation

This section is devoted to exploring some other practical space allocation problems tackled in the literature.

3.4.1 Retail Shelf Space Allocation

The retail shelf space allocation (RSSA) [Yang and Chen, 1999], [Bai, 2005] is the task of efficient usage of shelf spaces in retail sector. Due to the scarce shelf space and extremely varied amount of products to sell, retail companies need to reduce the cost of shelf usage while maximising the operational profit by presenting the goods to the customers in such a way to increase the sales.

The most basic management unit in RSSA is the *stock-keeping unit* (SKU). A SKU

is a unique identifier of a specific product. The quantity of each SKU currently held by a retailer is called the *inventory*. While keeping a large inventory may increase the storage costs considerably, a limited inventory on the other hand can lead to lack of products to sell. A collection of products with similar attributes forms a *category*. A category is usually formed by a number of different brands having several SKUs. The *facing* of a SKU is the number of an item which is visible on the shelves. A location refers to the placement of a SKU. A highly profitable or marketed SKU is usually given a higher facing with a better location for view by the customers.

The main goal of a RSSA system is to maximise the operational profit by finding an efficient allocation of products to the shelves. The problem involves many different constraints most of which depend on the corresponding retail company. However, several physical constraints are quite common: the products should be placed to the shelves where they can actually fit and stay (dimensional and weight requirements). The problem is usually strictly integral as well, because each SKU should be given facings which are integer numbers. There are also lower and upper bounds on the facings of a SKU to provide a minimal or maximum exposure of a product. Certain products should be placed close to each other while several other types should be kept away from each other.

3.4.2 Teaching Space Allocation

Teaching space allocation (TSA) [Beyrouthy, 2008] is a special case of space management problem encountered in many educational institutions. The teaching space includes lecture halls as well as rooms dedicated for tutorials, seminars, workshops, etc. The efficiency of teaching space management is measured by the utilisation of spatial resources. In the most common case, the utilisation is measured as the fraction of used space over the total available space. Contrary to common perception, the utilisation of teaching space in many universities is quite low. In fact, the percentage of practical utilisation was reported as low as twenty to thirty percent in [Beyrouthy et al., 2009].

A typical TSA problem involves the task of assigning events (usually a set of students taking a class) while satisfying two hard constraints: The capacity of the room an event is assigned should not be exceeded and the number of events associated with a room should not be larger than the available number of time slots. The target goal is to improve the utilisation which can be quantified by using *seat-hours* [Beyrouthy, 2008]. A seat-hour can be defined as the summation over all timeslots and rooms of the students allocated to that room-slot. The utilisation in this case can be measured as the

fraction of the actually used seat-hours to total available seat hours.

3.5   Algorithm Complexity

The complexity theory of algorithms deals with difficulty of problems in terms of computational resources required to solve them. The most common complexity analysis is performed over time and space. The *time complexity* deals with the growth of computational time to solve a problem as the size of the problem is increased while *space complexity* deals with the growth of memory required to solve the problem. In this thesis, only the *time complexity* will be considered because the memory requirement for a typical office space allocation problem is usually not a concern in modern computers. For a more in depth investigation, please refer to several books in this area such as [Sipser, 1996] and [Hopcroft et al., 2006].

The time complexity of an algorithm is measured by the amount of time required for a given size of input (*problem size*) and the growth of this time with respect to the problem size. Most commonly observed growth functions are *constant*, *logarithmic*, *polynomial*, and *exponential*. It is desirable to develop algorithms that are more efficient than the ones with *exponential* growth due to the quick and explosive growth of this function which makes a lot of problems extremely difficult to solve in practice. A problem is called *tractable* if there is an algorithm that can solve it in polynomial time, and it is called *intractable* if there is not such an algorithm. If there can be no algorithm that can solve the problem regardless of time complexity, the problem is referred to as *undecidable*.

P versus NP: In terms of time complexity for *decidable* problems, these can be mainly classified into two categories. The class *P* includes problems that can be solved with a *deterministic* algorithm in polynomial time. The class *NP* (Non-deterministic polynomial) includes problems that can be solved polynomially by a *non-deterministic* algorithm (more formally by a *non-deterministic Turing machine*). A non-deterministic algorithm includes two steps: in the first step, the algorithm guesses a solution for the problem. Due to the nature of non-determinism, each solution structure of the problem can be guessed at a given state. The second stage involves verification of the current structure. In this stage, the verification of whether this structure is the actual solution to the problem must be done in polynomial time if the problem is to be called as in *NP*.

One of the most important problems in mathematics is whether $P = NP$ or not. Most common conjecture while still not proven is that $P \neq NP$. This practically disal-

lows any efficient algorithm with polynomial time from being implemented for problems that are in the difference between two sets ($NP - P$).

Another important classification related to NP problem is the classes of *NP-Complete* and *NP-Hard*. A problem (or language) L is *NP-Complete* if and only if the two following conditions are satisfied:

- $L \in NP$

- All problems $L' \in NP$ are reducible to L in polynomial time.
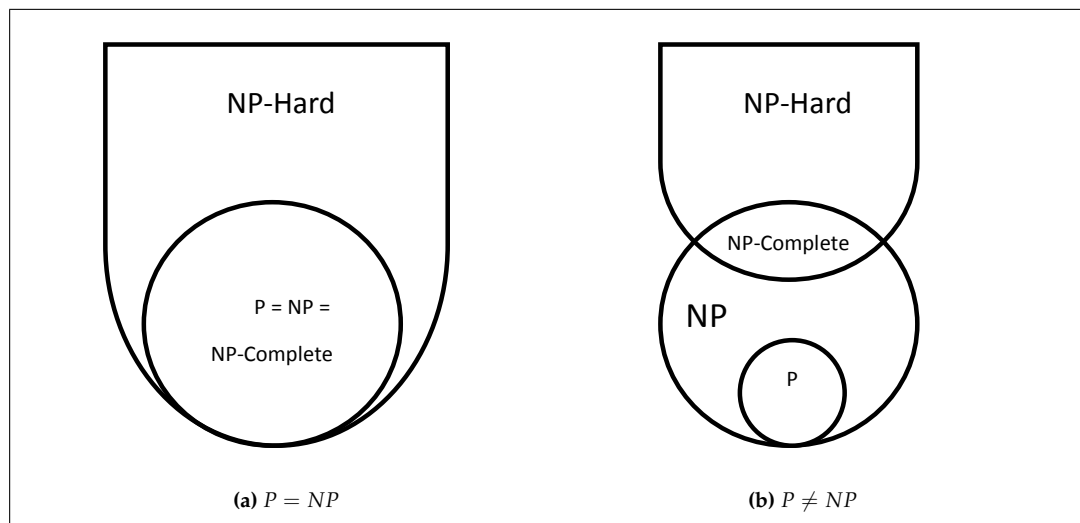
Figure 3.1: The classes *P*, *NP*, *NP-Complete* and *NP-Hard* under the conjectures $P = NP$ and $P \neq NP$

If only the second condition is satisfied, then L is in *NP-Hard*. The second property forms a relationship between all the problems in NP. Any problem in NP can be reduced to any other NP problem in polynomial time, which allows algorithms designed for a specific NP problem being used for another NP problem by means of a polynomial time reduction. If the conjecture $P = NP$ is true, then a polynomial algorithm that can solve a specific P problem can be reduced to solve any other problem in NP. Unfortunately, this conjecture while not proven is believed to be false, so any possibility of an efficient deterministic polynomial time algorithm for NP problems is slim. The diagram of classes P, NP, *NP-Complete* and *NP-Hard* is given in Figure 3.1 for conjectures $P = NP$ and $P \neq NP$.

Space Allocation as a Decision and Optimisation Problem

In complexity theory, problems are usually described in terms of decision problems. A *decision problem* is described as a problem whose solution can be given in terms of a

"yes-no" type answer. For example, given three integer numbers x, y, and z the problem ''does z divide both x and y'' is a decision problem. An *optimisation problem* is the task of finding the best solution among all feasible solutions. For example, the problem of finding the ''minimum z that divides both x and y'' is an optimisation problem.

The office space allocation problem can also be described as a decision problem as below:

Given an entity set E of size $|E|$, a room set R of size $|R|$, a set of hard constraints HC, and a set of soft constraints SC, is there a mapping from E to R (an allocation of entities to rooms) such that all hard constraints are satisfied and the total penalty (defined as the sum of the penalty for the space misuse and soft constraint violation) is less than a fixed value TP (total penalty)? The answer to this problem is 'yes' if such a mapping (allocation) exists, 'no' otherwise.

Any given solution to this decision problem of office space allocation can be verified in polynomial time, because it is possible to check each constraint violation and space misuse with polynomial time algorithms and hence, this decision problem for OSA is said to be in NP.

However, the office space allocation problem can also be described as an optimisation problem. In this case, the problem becomes given the sets E, R, SC, and HC; finding the allocation which yields the minimum total penalty. It is not possible to verify a solution to this problem in polynomial time, so the optimisation problem of office space allocation is not in NP.

The proof of whether there is a polynomial time reduction of every problem L in NP to the office space allocation problem and hence the proof of *NP-Completeness* or *NP-Hardness* is beyond the scope of this thesis. Assuming there is a polynomial time reduction of every problem L in NP to OSA, then the decision problem of the OSA will be in *NP-Complete* while the optimisation version of the problem will be in *NP-Hard*.

3.5.1   No Free Lunch Theorem

No free lunch theorem (NFL) is theorised by Wolpert and Macready [Wolpert and Macready, 1995], [Wolpert and Macready, 1997] to address the possibility of a general-purpose universal optimisation algorithm. According to the NFL theorem, the average performances of all algorithms over all types of problems are equal to each other. That is, no algorithm is essentially better than random search when the average performance is considered over all problems. A natural outcome of the NFL as described by Ho and Pepyne [Ho and Pepyne, 2002] is that ''a general-purpose universal optimization strat-

egy is theoretically impossible, and the only way one strategy can outperform another is if it is specialised to the specific problem under consideration".

However, NFL theorem should not be interpreted as an all-or-nothing proposition since it encompasses all problems. There can still be certain problem types and categories where an algorithm a_1 can give superior results over algorithm a_2. The originators of the NFL theorem, Wolpert and Macready showed the possibility of free lunches in co-evolutionary optimisation [Wolpert and Macready, 2005].

3.6 Review of Solution Approaches

This section describes some of the solution techniques that can be considered to tackle OSA. These methods are local search heuristics, genetic algorithms, integer programming, and hybridisations of these approaches. Heuristics can be necessary if the size of the problem grows larger than an exact algorithm can handle. Local search heuristics are chosen to supplement the integer programming method due to the ease of simple and efficient implementations. Finally, hybrid algorithms can be used to draw synergy between these various solution methods for development of state of the art solution methodologies for office space allocation problem.

This section describes some of the meta-heuristics and integer programming techniques used in this thesis. Note that the following survey is not an exhaustive list of algorithms, the specific techniques that are considered in this thesis will be described only to provide background information to the reader to follow the remaining chapters. The following heuristics are well researched in the literature and they are suitable for rapid prototyping and fast, efficient implementations. Integer programming is solution technique to provide exact results for combinatorial optimisation problems. Significant commercial investment has been allocated to develop state of the art general purpose integer programming solvers in the past decade. Consequently, this thesis aims to investigate the potential of using exact-inexact solution methods (and hybridisations of these) to tackle the office space allocation problem.

3.6.1 Meta-heuristics

Heuristic, a word with a Greek origin means 'discover' or 'find'. In optimisation, a heuristic refers to solution method applied to a difficult optimisation problem where an exact algorithm will consume too many resources. Meta-heuristics are extensions and formalisms to heuristics. We now paraphrase two formal definitions to point out

the perspective of this thesis in regards to meta-heuristics:

[Osman and Laporte, 1996] *"A meta-heuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions."*

[Voss et al., 1999] *"A meta-heuristic is an iterative master process that guides and modifies the operations of subordinate heuristics to efficiently produce high-quality solutions. It may manipulate a complete (or incomplete) single solution or a collection of solutions at each iteration. The subordinate heuristics may be high (or low) level procedures, or a simple local search, or just a construction method."*

In [Blum and Roli, 2003], several classifications of meta-heuristics according to some of their properties were made. These classifications are as follows:

- *Nature inspired vs. non-nature inspired*: The meta-heuristics are classified based on their origin. Algorithms such as *genetic algorithms* [Holland, 1975] and *simulated annealing* [Kirkpatrick et al., 1983] are inspired from several natural phenomena (theory of evolution and thermo-dynamical systems respectively), while several of them like *iterated local search* [Lourenco et al., 2002] are not.

- *Population based vs. single solution search*: This classification is made based on whether the search is performed on a single solution or using a population of different solutions. The single solution search methods (also called *trajectory methods*) such as *tabu search* [Glover and Laguna, 1997] or *variable neighbourhood search* [Mladenovic and Hansen, 1997] operate on a trajectory in the search space while population based algorithms simulate an evolution of a set of solutions in the search space.

- *Dynamic vs static objective functions*: While many meta-heuristics usually operate with a static predetermined objective function, others like *guided local search* [Voudouris and Tsang, 1999] can modify the objective function during the search in order to escape from local minima points.

- *Single vs multiple neighbourhoods*: While most meta-heuristic techniques operate on a single neighbourhood structure, others like *variable neighbourhood search* [Mladenovic and Hansen, 1997] can utilise different neighbourhoods to diversify the search.

- *Memory usage vs. memory-less*: This classification is based on the application of search history during the search. A memory-less meta-heuristic can be thought

of as a Markov process where the next outcome is determined strictly based on the current stage in the search when the number of possible outcomes is finite. Some meta-heuristics on the other hand can employ some memory structures to keep track of past information that can be used to guide the search.

Different types of meta-heuristics can be used together (hybridised) to solve difficult combinatorial optimisation problems. Several classifications of hybrid meta-heuristics [Cotta-Porras, 1998], [Talbi, 2002], [Dumitrescu and Stützle, 2003], [Blum and Roli, 2003], [Raidl, 2006] were proposed in the literature. In this section, the taxonomy proposed by [Talbi, 2002] is presented. This taxonomy is given in Figure 3.2.
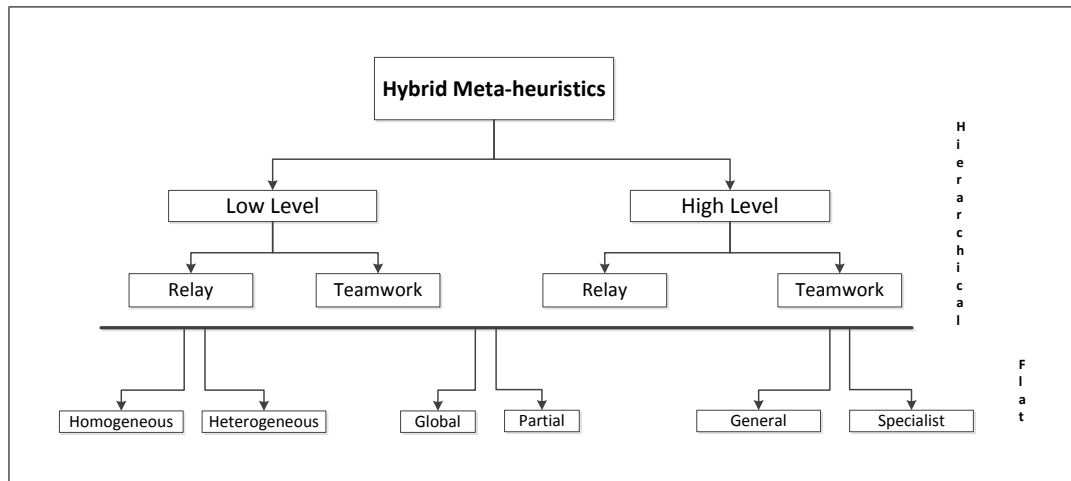
Figure 3.2: A taxonomy of hybrid meta-heuristics [Talbi, 2002]

Talbi categorises hybrid algorithms in two stages: hierarchical and flat. In hierarchical categorisation, the top level is divided into low and high level hybrids. In *low-level* hybridisation, a function of a meta-heuristic is replaced by another meta-heuristic while in *high-level* hybridisation, meta-heuristics are independent; there is no relationship between the internals of different meta-heuristics. The second level is divided based on relay or teamwork. In *relay* hybridisation, a set of meta-heuristics works in tandem by taking the output of the previous one as its input. *Teamwork* hybridisations include cooperation mechanism with different parallel agents operating on the search space.

By using the notions *low*, *high*, *relay*, and *teamwork*, four hierarchical classes can be derived:

- *Low-level relay (LRH):* A given meta-heuristic is embedded into a single-solution meta-heuristic.

- *Low-level teamwork (LTH):* This hybrid utilises exploration and exploitation by embedding a meta-heuristic in another.

- *High-level relay (HRH):* Several independent meta-heuristics are executed in sequence.

- *High-level teamwork (HTH):* Independent meta-heuristics perform in parallel and cooperate to find an optimal solution.

The flat classification is performed on hierarchical classification. In *homogeneous* hybrids, all the algorithms in the overall framework use the same meta-heuristic. *Heterogeneous* hybrids on the other hand are comprised of different meta-heuristics. If all the meta-heuristics in the overall algorithm operate in the same complete search space, then this hybrid is referred to as *global*. In *partial* hybrids, the complete problem is divided into sub-problems each with its own search space, and each algorithm is responsible for analysing this search space. Another classification is specialist vs general hybrids. The previously mentioned hybrids are all *general* hybrids, all the algorithms in the complete meta-heuristic framework try to solve the same problem. The *specialist* hybrids combine different algorithms to solve different sub-problems within a larger problem.

Local Search

Local search (LS) [Aarts and Lenstra, 2003] is a general purpose framework to solve difficult combinatorial optimisation problems. A local search algorithm works on a single incumbent solution and a cost function which defines the quality of the solution at hand. The goal is to find a solution which maximises (or minimises) the cost function. The local search algorithms work on a *neighbourhood structure* of a given solution and traverse from one solution to another by means of *move* operators. The *neighbourhood* of a present solution is defined the solutions that can be reached by a move operator from the current solution. For the case of office space allocation problem, a move operator (and hence the neighbourhood relation) can be defined as moving one entity from the room it is allocated in to another one in the current solution. The neighbourhood of a solution is defined as all the solutions that can be reached from the current solution by the move operator. The move operators can be complex, for example in *k-opt* neighbourhoods, k components of the current solution are changed to produce a neighbouring solution.

The local search usually terminates until no further improvement to the objective

```
1: Initialise a starting solution x
2: repeat
3:    Generate a new solution x∗ within the neighbourhood of x
4:    if Acceptance criterion is met then
5:       x ← x∗
6: until Termination criterion is met
```

Figure 3.3: General local search framework

function value is possible (that a *local optima* is reached). The pseudo-code for the general *local search* framework is given in Figure 3.3. For the sake of consistency in this thesis, from now on, the objective function value (which will be abbreviated as *obj*) is always going to be minimised. There have been many improved versions of local search which are built upon the basic framework. A few of the common local search algorithms that are used in this thesis are presented.

Iterated Local Search

The *iterated local search* Lourenco et al. [2002] algorithm extends the local search meta-heuristic with a perturbation operator. The algorithm creates an initial solution and improves it usually until a local optimum is reached. At this stage, a perturbation operator alters the solution. Local search is reapplied to this perturbed solution until another local optimum is found. At this stage, an acceptance criterion is used to decide whether to accept or reject the current solution over the previous one. The most commonly used acceptance criterion is to accept the new solution if it is better than the old one.

```
1: Initialise a starting solution x
2: x^1 ← Apply local search on x until a local optimum is found
3: repeat
4:    x∗ ← Perturb x^1
5:    x^2 ← Apply local search on x∗ until a local optimum is found
6:    if x^2 is better than x^1 based on the acceptance criterion used then
7:       x^1 ← x^2
8: until Termination criterion is met
```

Figure 3.4: Iterated local search meta-heuristic

Although the *iterated local search* algorithm is simple to implement, attention should be given while designing the perturbation operator and the acceptance criterion of the new solution. The extent of the perturbation operator should be just enough to es-

cape from local optima points to guide the search to other regions of the neighbourhood. Very large perturbations are discouraged due to the possibility of (near) random restarts.

The pseudo-code for *iterated local search* meta-heuristic is given in Figure 3.4.

Tabu Search

Tabu search (TS) [Glover, 1989] [Glover, 1990] is a local search method in which the decision to move to a neighbouring solution depends on short or long term memory structures. The pseudo-code for *tabu search* meta-heuristic is given in Figure 3.5.

1: Initialise a starting solution x

2: Initialise an empty *tabu list*

3: **repeat**

4:     Generate a set of solutions X' from x based upon a neighbourhood structure N

5:     Eliminate solutions from X' that are set as tabu

6:     **if** solution $x' \in X'$ is reached through a tabu move and the objective value of x' is best so far **then**

7:         Reinsert x' into X' (aspiration criterion)

8:     Select the best solution $x' \in X'$.

9:     Insert the move $x' \longrightarrow x$ into the *tabu list* with tabu tenure $tt + 1$

10:     $x \longleftarrow x'$

11:     Decrement the tabu tenure tt of each move in tabu list by 1

12:     **if** the tabu tenure tt of a move in the tabu list becomes 0 **then**

13:         remove the respective move from tabu list

14: **until** Termination criterion is met

Figure 3.5: Tabu search meta-heuristic

At each iteration, the TS method generates a large subset of solutions within a search neighbourhood. This subset of solutions does not include the ones designated as *tabu* (forbidden) in the respective *tabu list*. TS usually utilises this tabu list to prevent returning back to recently visited solutions and hence cycles in the search. The tabu list can be implemented in various ways, it may contain the representation of whole solution. However, for a more efficient implementation, usually only moves, attributes or conditions related to the tabu solution are stored in the tabu lists.

TS usually is implemented as best improvement local search algorithm, the move that leads to the best solution in the current neighbourhood is selected. In most implementations, the reverse of the most recent move is stored in the tabu list to prevent immediate cycles. There are two issues in handling the tabu lists: the *size* of the tabu

list and the *tabu tenure* which determines the duration a tabu condition is going to be kept in the tabu list. A huge tabu list which contains a lot of tabu moves or a long tabu tenure can lead to a lot of promising moves (moves that can lead to a better local optima eventually) being rejected. A small tabu list or a short tabu tenure on the other hand may not be enough to prevent the search from getting stuck in local optima. The other important issue is that it is sometimes possible to accept a solution even if it is designated as tabu. These solutions might not be visited during the search or might be of high quality but set as tabu due to a condition in the tabu list. In order to overcome this problem, an *aspiration criterion* can be set to revoke the tabu conditions. Most commonly used aspiration criterion is to accept a tabu move if it leads to the best encountered solution during the whole search.

Apart from this tabu list based framework, more advanced TS algorithms try to utilise the information gathered during the search. This leads to four general long term memory principles : *recency*, *frequency*, *quality*, and *influence*. Recency-based memory keeps track of the most recent operations for a given solution. In frequency-based memory, the number of times each solution (attribute or condition) is visited is kept. Quality principle aims to identify good solution components from the accumulated information during the search. The fourth principle, *influence* tries to analyse the past decisions for determining the most critical ones during the search history.

Greedy Randomised Adaptive Search Procedure

Greedy randomised adaptive search procedure (GRASP) [Resende and Ribeiro, 2003] is a multi-start meta-heuristic in which each iteration of the search consists of two stages: *construction* and *local search*. A feasible solution is generated in the *construction* stage and this solution is improved to a local optima by means of a local search operator. The pseudo-code for the *greedy randomised adaptive search procedure* meta-heuristic is given in Figure 3.6.

The *construction* stage of GRASP randomly chooses elements from a data structure called the *restricted candidate list* (RCL). In GRASP, the elements that will be used in generation of a candidate solution are stored in a list of elements E. The incremental cost of addition of element $e \in E$ into the solution is denoted by $c(e)$. The RCL is formed by elements with the smallest incremental costs $c(e)$. In a basic construction stage, the minimum and maximum incremental costs (c^{min} and c^{max} respectively) are found first. RCL is formed by the elements $e \in C$ that fit the condition given at line 7 of Figure 3.7. Parameter α determines the greediness and randomness of the algorithm. By varying α

```
1: Set best solution x∗ as ∅
2: Set best objective value of x∗ as ∞ (for minimisation problems)
3: repeat
4:     Generate a greedy randomised solution x using construction stage (Figure 3.7)
5:     Find local optimum x′ with local search starting from solution x
6:     if obj(x′) is better than obj(x∗) then
7:         x∗ ← x′
8: until Termination criterion is met
```

Figure 3.6: GRASP meta-heuristic

from 0 to 1, the construction algorithm can shift from a pure greedy approach to a pure random one. The element chosen randomly from the RCL is then incorporated into the current partial solution x to result in the smallest incremental cost. This chosen element is also removed from the list C. After the addition of this element, all incremental costs $c(e)$ are recalculated. The construction algorithm ends when all the elements $e \in C$ are incorporated into x. A general outline of the construction stage is given in Figure 3.7 [Resende and Ribeiro, 2003].

```
Input: α
Output: Solution x
 1: x ← ∅
 2: Initialise candidate set C ← E
 3: Calculate incremental costs c(e) ∀e ∈ C
 4: repeat
 5:     c^min ← minimum c(e)
 6:     c^max ← maximum c(e)
 7:     RCL ← {e ∈ C|c(e) ≤ c^min + α(c^max − c^min)}
 8:     Select an element rcl_i from RCL randomly
 9:     x ← x ∪ rcl_i
10:     Update C by removing rcl_i from C
11:     Recalculate c(e) ∀e ∈ C
12: until C = ∅
```

Figure 3.7: Construction stage in GRASP meta-heuristic

Simulated Annealing

Simulated annealing (SA) [Kirkpatrick et al., 1983] is a local search optimisation algorithm inspired from the statistical mechanics algorithm of Metropolis [Metropolis et al., 1953]. SA starts with an initial solution, and the moves which lead to a better so-

lution are always accepted. The probability of accepting a non-improving move depends upon the cooling schedule. The algorithm starts with a high initial temperature (the probability of accepting a non-improving move is higher), and the temperature is gradually lowered (cooling procedure) during the algorithm (that the probability of accepting a non-improving move gets less and less) . Various cooling algorithms were proposed in the literature [Hajek, 1988], [Strenski and Kirkpatrick, 1991].

1: Initialise a starting solution x

2: Initialise a starting temperature T

3: **repeat**

4:     Generate a solution x' from x based upon a neighbourhood structure N

5:     **if** objective function value $obj(x')$ is better than the $obj(x)$ **then**

6:         $x \longleftarrow x'$

7:     **else if** $random(0,1) <= e^{-\frac{obj(x')-obj(x)}{T}}$ **then**

8:         $x \longleftarrow x'$

9:     **else**

10:        reject x'

11:     **if** temperature reduction criteria is met **then**

12:        reduce T according to the cooling procedure

13: **until** Termination criterion is met

Figure 3.8: Simulated annealing meta-heuristic

Figure 3.8 represents the pseudo-code for the *simulated annealing* meta-heuristic. At line 7, the annealing function is taken as in [Kirkpatrick et al., 1983].

Threshold Acceptance

Threshold acceptance is first described in [Dueck and Scheuer, 1990] and [Moscato and Fontanari, 1990]. This algorithm is another update on the traditional local search algorithm and simplification of the simulated annealing by replacing the probabilistic acceptance criteria with a deterministic one. At each iteration of a *threshold acceptance* algorithm, a new solution is randomly generated based on a neighbourhood structure and a given solution,. If the new solution is better than the old one, the new solution is accepted. The algorithm also accepts moves that are within a threshold of the current solution. The threshold is usually set larger for large neighbourhoods and smaller for small neighbourhoods. Threshold management is crucial in *threshold acceptance* algorithms.

The pseudo-code for *threshold acceptance* meta-heuristic is given in Figure 3.9.

1: Initialise a starting *threshold T*
2: Initialise a starting solution *x*
3: **repeat**
4: Generate a solution x' from x based upon a neighbourhood structure N
5: **if** objective function value $obj(x')$ is better than the $obj(x)$ **then**
6:   $x \longleftarrow x'$
7: **else if** $|obj(x') - obj(x)| \leq$ *threshold T* **then**
8:   $x \longleftarrow x'$
9: **else**
10:   reject x'
11: **if** No improvement to the solution is observed or a number of iterations is passed **then**
12:   Reduce *threshold T*
13: **until** Termination criterion is met

Figure 3.9: Threshold acceptance meta-heuristic

Great Deluge

The *great deluge* (GD) algorithm [Dueck, 1993] is an extension of *threshold acceptance*. GD can be explained with a rain and flooding metaphor. Assuming it is raining constantly, and there is no drainage mechanism, the water-level is expected to increase over time. In order for a person to survive, he has to move a higher altitude above the current water level, otherwise he will drown.

The GD operates similarly to other local search algorithms. At each iteration, the current solution *x* is perturbed within a neighbourhood structure. The decision to accept or reject such a move is made based on the objective function value of the solution and the current water level. For a minimisation problem, in order for a move to be accepted, the objective function value should be below the current water level, otherwise the move is rejected. The water level is decreased over time with a rate called rain speed *rs*. Gradually, the acceptance of uphill moves (moves that increase the objective function value) becomes more difficult due to the decreased water level. For an efficient performance of the algorithm, the initial water level *W* and the rate of increase of the water level (rain speed) *rs* should be adjusted very carefully.

The pseudo-code for the *great deluge* meta-heuristic is given in Figure 3.10.

```
 1: Initialise the starting solution x
 2: Initialise the rain speed rs
 3: Initialise the starting water level w
 4: repeat
 5:     Generate a solution x′ from x based upon a neighbourhood structure N
 6:     if obj(x′) is better than the water level w then
 7:         x ⟵ x′
 8:         w ← w + rs
 9:     else
10:         reject x′
11:     if no improvement to the solution is observed or other water level reduction
        criteria are met then
12:         reduce w
13: until Termination criterion is met
```

Figure 3.10: Great deluge meta-heuristic

Variable Neighbourhood Search

Variable neighbourhood search (VNS) [Mladenovic and Hansen, 1997] is based upon the idea of using multiple neighbourhood structures during the local search. VNS algorithm tries to exploit the notion that a local optimum with respect to a single neighbourhood structure may not necessarily be optimal from the point of another neighbourhood structure. However, a global optimum of a problem is locally optimal in all neighbourhoods irrespective of the neighbourhood structure used. In a VNS, a fair amount of different neighbourhood structures which can represent different regions in the search space is desired. The intensification of the search can be satisfied via a local search within a single neighbourhood while diversification can be achieved by systematically switching to different neighbourhood structures.

The basic VNS algorithm is usually divided into three stages: *shaking*, *local search*, and *move* stages. The neighbourhood structure list N (which contains each neighbourhood N_k) is sorted in order from index $k = 1$ to $k = k_{max}$. In the *shaking* stage, a random solution is created by using the neighbourhood structure N_k starting with $k = 1$. The solution in *shaking* stage is improved by *local search* operators by using this neighbourhood structure N_k. If the new local optimum solution is better than the incumbent solution, then this new solution is accepted (*move* stage). If the solution cannot be improved with this neighbourhood N_k, the algorithm switches to the next neighbourhood (N_{k+1}) in the neighbourhood list. If none of the neighbourhood structures can improve the solution x, then the neighbourhood index is reset back to $k = 1$, and the algorithm

```
 1: Initialise the starting solution x
 2: Generate k_max amount of neighbourhood structures in N
 3: repeat
 4:     k ← 1
 5:     repeat
 6:         Shaking: Generate new solution x′ from x using the neighbourhood N_k
 7:         x∗ ← Apply Local search to x′
 8:         if obj(x∗) is better than obj(x′) then
 9:             Move to new local optimum x ← x∗
10:         else
11:             switch to next neighbourhood by k ← k + 1
12:     until k = k_max
13: until Termination criterion is met
```

Figure 3.11: Variable neighbourhood search meta-heuristic

continues with a new random solution generated by the *shaking* stage. The VNS algorithm continues until a pre-determined termination criterion is met.

The pseudo-code for *variable neighbourhood search* meta-heuristic is given in Figure 3.11.

Ruin and Recreate

The *ruin and recreate* (R&R) [Schrimpf et al., 2000] meta-heuristic is a departure from many local search algorithms such as *threshold acceptance* and *simulated annealing* which choose to make small disruptions to the current solution. (R&R) instead applies rather large disruptions to the current solution and tries to rebuild the heavily disrupted solution with the goal of finding a promising new solution.

```
 1: Initialise a solution x
 2: repeat
 3:     x′ ← Ruin solution x
 4:     x∗ ← Recreate x′
 5:     if Acceptance criterion is met then
 6:         x ← x∗
 7: until Termination criterion is met
```

Figure 3.12: Ruin and recreate meta-heuristic

In [Schrimpf et al., 2000], the proponents of the *ruin and recreate* algorithm claim that complex problems are usually discontinuous in nature, and small moves and dis-

ruptions within the neighbourhood may not be sufficient to find increasingly better solutions during the search. The search landscape can be uneven, and promising solutions can be far apart from each other due to many constraints in a complex combinatorial optimisation problem. Therefore, without making drastic changes to the current solution, it may not be possible to reach promising solutions that might be far away from the current best solution.

A typical *ruin and recreate* algorithm has two states: *ruin* and *recreate*. In the *ruin* stage, the current solution is heavily disrupted by removal or mutation of the components of the solution. The recreate stage operates on the ruined solution and tries to re-optimise it by restructuring of the deleted or mutated sections of the previous solution. When the *recreate* stage is finished, a new solution is obtained. The acceptance or rejection of the new solution over the previous one is decided similar to other local search algorithms like *simulated annealing* or *threshold acceptance*.

The pseudo-code for *ruin and recreate* meta-heuristic is given in Figure 3.12.

Genetic Algorithms

A genetic algorithm (GA) [Holland, 1975] is a search method to find approximate solutions to optimization problems. Genetic algorithms use techniques like crossover, mutation, and natural selection that are inspired from the theory of evolution. In a typical genetic algorithm, a population of chromosomes (abstract representation of candidate solutions, also referred to as individual) goes through an evolutionary process. The most common representation scheme is binary encoding. The evolution usually starts from a population initialised randomly or heuristically. In each generation, the quality of the solutions is evaluated by a objective function value; individuals are randomly selected and are modified by crossover and mutation operators to form a new population. This process continues until the optimal solution is found or a termination criteria set by the user is met. The pseudo-code for a general genetic algorithm is given in Figure 3.13.

In a genetic algorithm, the representation refers to the encoding of the chromosomes in the population.The most common approach is using fixed size binary strings as in Holland's original encoding method [Holland, 1975]. However, variable size and non-binary encodings may also be used depending on the problem. The efficiency of the encoding method is problem dependent, thus an encoding method should be adjusted according to the needs of the problem at hand.

At each iteration of a GA, several chromosomes are selected to generate the next

```
1: Initialization a population of chromosomes (individuals)
2: Evaluate the objective function values of the chromosomes
3: repeat
4:     Selection of chromosomes for crossover
5:     Crossover
6:     Mutation
7:     Objective function value evaluation
8:     Population Update
9: until Termination criterion is met
```

Figure 3.13: Outline of a genetic algorithm

population. The new population is created by means of crossover and mutation operators. However, a selection operator is needed to decide which chromosomes in the population will undergo these operations. The selection operator mimics the *natural selection* in theory of evolution. Some of the commonly used selection methods are as follows:

- *Roulette wheel selection (RWS)*: The selection chance of the chromosome is directly proportional to its objective function value. In this method, each chromosome represents a pocket on the wheel, and the size of each pocket is directly proportional to the probability of selection. The efficiency of this selection depends greatly on the range of the objective function values of the chromosomes.

- *Rank based selection (RBS)*: In this method [Baker, 1985], the chromosomes in the population is sorted according to their objective function values, and each chromosome is given a rank. The probability of selection is determined according to this rank value, not to its actual objective function value.

- *Tournament based selection*: In this method [Brindle, 1981], k chromosomes chosen randomly enter into a tournament where a winner is selected (which is usually the chromosome with the best objective function value). This selection has several implementation advantages such as no objective function value scaling as in RWS or sorting as in RBS. Also, it is easy to adjust the selection pressure by changing the size of the tournament.

After the selection, a typical GA uses a crossover to generate children solutions from the parent chromosomes selected. The crossover is a recombination operator which exchanges segments of solutions between chromosomes. Some of the most commonly used crossovers are as follows [Goldberg, 1989]:

- *One Point Crossover (1-PTX):* A crossover point is randomly selected on the parent chromosome, and the values beyond that point are swapped between the two parent chromosomes.

- *N Point Crossover (N-PTX): n* crossover points are randomly selected on the parent chromosome, and the values between odd and even crossover positions are swapped.

- *Uniform Crossover (UX):* Each gene of the first parent has a fixed probability (usually 0.5) of swapping with the respective gene of the second parent.

Mutation is another important genetic operator whose purpose is to maintain the genetic diversity between successive generations of population. With just crossover and without mutation, it is usually impossible to generate new information not present before in the population unless crossover operator also includes some mutational characteristics. Local optima are avoided by using a mutation operator which prevents the over-similarity of the chromosomes in the population. The most traditional mutation in many binary coded representations is to simply flip the value of an arbitrarily chosen gene. In many implementations, mutation is given very low rates usually proportional to the length of the chromosome as too much mutation is inherently destructive to the solution quality.

After new solutions have been generated by genetic operators such as selection, crossover, and mutation, these solutions (referred to as children) have to be inserted into the population. Some replacement strategies are as follows:

- *Pure Reinsertion:* Generate as many children as possible and replace all parents by this children.

- *Uniform Reinsertion:* Generate less children than parents and replace them with a uniform distribution.

- *Elitist Reinsertion:* Generate less children than parents and replace the worst parents.

- *Fitness Based Reinsertion:* Generate more children than needed and reinsert the best children only.

Memetic Algorithms

Similar to genetic algorithms, a memetic algorithm (MA) [Moscato, 1989] is a population based meta-heuristic for solving optimization problems. Because MAs usually

combine local search with genetic operators, they are also referred to as hybrid genetic algorithms. Memetic algorithms can be categorised into two types based on where the local search operators are applied. In a Baldwinian memetic algorithm [Baldwin, 1996], the local search is applied before the objective function value is evaluated. The improvements of the local search are not saved in the individual; therefore acquired traits of the parents are not inherited to the children. In a Lamarckian memetic algorithm [Whitley et al., 1994], the local search is applied after the objective function value is evaluated therefore acquired traits of the parents influence the children. Lamarckian MA algorithms are usually faster in finding quality solutions with the risk of premature convergence. A Baldwinian MA would be more resistant to diversity loss in the population but it is usually much more slower than its Lamarckian counterpart.

It is assumed that a genetic algorithm is able to cover a broad range in the search landscape due to its population based nature, and a local search is able to find optimal solutions in promising parts of the landscapes. Due to a local search component, a memetic algorithm can also incorporate domain specific knowledge better than a blind genetic algorithm. Therefore memetic algorithms are usually much more efficient in terms of computing resources and tend to give state of the art results in many problems.

3.6.2 Integer Programming

A linear program is an optimisation problem consisting of decision variables, a linear objective function, and a set of linear constraints defined as inequalities. Mathematically, a minimisation version of integer programming can be represented as in equations 3.6.1, 3.6.2, and 3.6.3 as defined in [Lübbecke and Desrosiers, 2005]:

$$minimise \quad z \leftarrow \sum_{n \in N} c_n \lambda_n \tag{3.6.1}$$

$$s.t. \quad \sum_{n \in N} a_n \lambda_n \geq b \tag{3.6.2}$$

$$\lambda_n \geq 0 \qquad n \in N \tag{3.6.3}$$

where $c \in R^n$ is an n-dimensional column vector of coefficients and λ is an n-dimensional column vector of variables. The n-th element in vectors c and λ are represented by c_n and λ_n respectively. The objective function z to be minimised is given in equation 3.6.1 which is the dot product of c and λ. The dot product of the matrix $a \in R^{m \times n}$ and $b \in R^m$ define the m inequality constraints. Each column in matrix a is represented by a_n. Each a_n is m sized vector. If only some of the variables in x are integers, the problem is called *mixed integer programming* problem whereas *pure integer programming problem*

refers to cases when x is purely integer. If all the variables in x are binary, the problem is regarded as *0/1* or *binary integer programming problem*.

Every integer linear program, referred to as *primal problem* can be converted into a *dual problem* which provides a bound to the optimal value of the *primal problem*. The *dual problem* of z (which is given in equations 3.6.1, 3.6.2 and 3.6.3) is depicted in y (which is given in equations 3.6.4, 3.6.4 and 3.6.6). The dual variables are given in vector π.

$$maximise \quad y \leftarrow \sum_{n \in N} b_n \pi_n \tag{3.6.4}$$

$$s.t. \quad \sum_{n \in N} a_n \pi_n \leq c \tag{3.6.5}$$

$$\pi_n \geq 0 \qquad n \in N \tag{3.6.6}$$

The minimisation version of a linear problem can be performed by simply changing the sign of c. Greater than constraints can again be converted into less than and equal to constraints by changing the sign of corresponding coefficients and equality constraints can be handled by using pairs of less and greater than inequalities.

A linear programming model with no integer variables can be solved efficiently with *simplex* [Dantzig, 1963] and *interior point* based algorithms [Karmarkar, 1984]. However, the linear programming model becomes intractable or NP Complete [Garey and Johnson, 1979] when integer variables are introduced into the model. Common methods currently used in integer programming are *branch and bound* [Land and Doig, 1960] and *cutting planes* [Marchand et al., 2002].

Branch and Bound: The *branch and bound* algorithm [Land and Doig, 1960] is a divide-and-conquer tree algorithm which recursively partitions (branching) the problem into smaller sub-problems. It is an exact algorithm, in the sense that it provides a provable upper and lower bound on the global optimal value of the problem. Branch and bound algorithms are unfortunately generally very slow, in the worst case, the time required for solving the problem grows exponentially with the increased problem sizes.

The branch and bound algorithm is based on the enumeration of integer solutions in a tree structure [Chinneck, 2012]. Figure 3.14 depicts a possible enumeration of the three integer variables x_1, x_2, and x_3 where $1 \leq x_1 \leq 3$ and $0 \leq x_2, x_3 \leq 1$. The goal of the branch and bound algorithm is not to generate the tree all at once but in stages, the algorithm tries to find the most suitable node by estimating a bound on the objective

value that can be obtained by expanding the tree from that node. A bounding function is the estimate on the best value of the objective function by expanding a non-leaf node. Since the leaf nodes represent complete solutions, they have actual objective function values not estimates. The design of the bounding function is very important because it directly affects the efficiency of the algorithm.
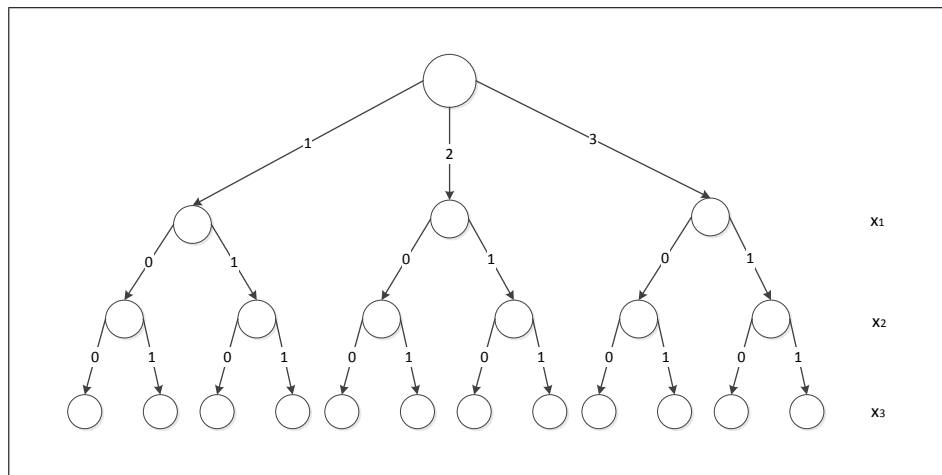
Figure 3.14: Enumeration tree of three integer variables x_1, x_2, and x_3

At any stage, the algorithm has to decide which node to select and branch on it. Some of the most commonly used techniques for selecting the node for branching are as follows:

- *Best-first of global best node selection:* The node with the best value on the bounding function in the branch and bound tree is chosen.

- *Depth-first:* Select the node with the best value on the bounding function among the ones created by the branching. Depth-first aims to go deep within the tree and achieve an early incumbent (best encountered so far) solution.

- *Breadth-first:* This technique expands the nodes in the exact order they are created.

Another important aspect of the algorithm is to decide to prune certain non-leaf nodes. The non-leaf nodes are pruned if it can be proved that no descendent of that non-leaf node will be infeasible or non-optimal. If the bounding function on the non-leaf node (the estimate of the objective function value on that non-leaf node) is worse than the objective function value of the incumbent solution, then the non-leaf node can be pruned due to non-optimality. The expansion of a node can sometimes be stopped due to the best possible value on the objective function being seen directly (fathoming).

The decision to prune a non-leaf node due to the infeasibility of the descendants of a non-leaf node is problem dependent.

The branch and bound algorithm can terminate when the objective function value of the incumbent solution is better than or equal to the bounding function value of all the non-leaf nodes in the tree. In this case, the algorithm reaches the optimal solution of the problem.

Cutting Planes: *Cutting plane* algorithms are based on adding more inequalities into a problem. The algorithm starts with a small set of inequalities (cuts), and tries to find more cuts which do not violate the original problem but still not satisfy the current solution at hand. More cuts are added into the problem while it is still being resolved using linear relaxation. The algorithm terminates when no further cuts can be generated. Although there exists generic types of cuts, like *Gomory cuts* [Gomory, 1958] which guarantee optimality, algorithms using these cuts converge very slowly to an optimal solution, therefore problem specific cuts are usually employed in the state of the art algorithms. For more information about different cutting plane method, one can refer to [Marchand et al., 2002].

Column Generation: *(Delayed) column generation* (CG) [Wilhelm, 2001], [Lübbecke and Desrosiers, 2005] is another technique to undertake integer programming problems. Instead of formulating the problem with all the constraints and objectives (a *master problem*), the *column generation* approach starts with an initial smaller problem with a subset of variables specifically chosen (*restricted master problem* - RMP). The first stage in an iteration of column generation consists of the optimisation of the RMP and determination of the current optimal objection function values and dual variables. In the second stage, the algorithm tries to find a not yet considered variable to include into the current problem for improving the solution. This stage is called the *pricing problem*. For a minimisation problem, the variable can be incorporated into the solution if it has negative reduced costs. After the *pricing* of the variable, new *restricted master problem* is resolved. This process is repeated until it is no longer possible to incorporate new variables with negative reduced costs. An optimal solution for the *master problem* is then obtained.

A more formal and detailed definition of column generation as described by [Lübbecke and Desrosiers, 2005] is given below. For a linear program, the master problem MP is defined as in equations 3.6.1, 3.6.2, and 3.6.3 in Section 3.6.2:

A non-basic variable is searched for pricing and for being entered into the basis in

each iteration of the simplex method. Given a non-negative vector π of dual variables, the aim is to find a $n \in N$ which minimises $c_n - \pi^t a_n$. Unfortunately, this pricing operation is too expensive when $|N|$ is high. A restricted master problem (RMP) is created with a small subset of variables ($N' \subseteq N$ columns) and is evaluated for reduced costs. Let λ and π be the primal and dual optimal solutions to the current RMP, respectively. When columns a_n are the elements of set A, cost coefficient c_n is computed from a_n by a function c, then the pricing is performed as in equation 3.6.7:

$$\bar{c}* \leftarrow minimum(c(a) - \pi^t a | a \in A) \tag{3.6.7}$$

If $\bar{c}* \geq 0$, the solution λ of the RMP is also the optimal solution of the master problem. If $\bar{c}*$ is negative, then the column derived from the optimal sub-problem solution is added to the RMP, and the RMP is re-optimised. Each $a \in A$ is generated at most once because no variable in an optimal RMP solution can have negative reduced cost. The column generation algorithm gives an exact optimal solution if the set A is finite. It also allows the generation of bounds on the objective function value. When an upper bound $k \geq \sum_{n \in N} \lambda_n$ is found for the optimal solution $\bar{z}$ of the master problem, a lower bound is also found because $\bar{z}$ cannot be reduced more than k times the smallest reduced cost $\bar{c}*$ as in equation 3.6.8:

$$\bar{z} + k\bar{c}* \leq z*_{MP} \leq \bar{z} \tag{3.6.8}$$

where in the optimal case, $\bar{c}* = 0$ for the basic variables, and $\bar{z} = z*_{MP}$.

Hybrid Methods: *Branch and cut* algorithms are hybridizations of the *branch and bound* and *cutting planes* where additional cuts are added into the sub-trees in the branch and bound tree. Combining *branch and bound* and *column generation* into one framework produces the *branch and price* algorithms. *Branch and cut* and *branch and price* are frequently used in state of the art integer programming solvers. For more information, please refer to Wolsey [Wolsey, 1998] for integer programming and Williams [Williams, 1999] for mathematical modelling

Mat-heuristics: A mat-heuristic [Maniezzo et al., 2009] is an optimisation method of combining *mathematical programming* and *heuristics* techniques to solve combinatorial optimisation problems. Puchinger and Raidl [Puchinger and Raidl, 2005] categorised the combination of the exact algorithms and heuristics under two branches: In *collaborative combinations*, independent algorithms exchange information with each other. The

algorithms can be applied sequentially, intertwined or in parallel. In *integrative combinations*, one method (heuristic or exact approach) is integrated into another (*master algorithm*) as a subcomponent. Further classifications can be found in Talbi [Talbi, 2002] or [Dumitrescu and Stützle, 2003].

Mat-heuristics have been used in following areas in solving combinatorial optimisation problems as listed in [Raidl and Puchinger, 2008]:

- finding high-quality incumbent (best) solutions and bounds in the branch-and-bound process.

- finding relaxations for the meta-heuristic search; utilising the primal-dual relationships to generate tighter bounds.

- utilising local search in branch-and-bound process.

- using integer linear programming techniques for exploring large scale neighbourhoods.

- merging solutions generated from heuristics and mathematical programming.

- using integer linear programming techniques for decoding indirect and missing representations.

- multi-stage approaches of heuristics and integer programming.

- additional cuts and column generation by the use of meta-heuristics; heuristics can be used to solve the *pricing problem* in *column generation* and they can be used to generate additional *cuts* in a *cutting plane* or a *branch and cut* algorithm.

- guidance of search and collaboration.

3.7 Conclusion

This chapter described the previous research performed in the office space allocation area, different variants of the OSA problem tackled and various types of algorithms to solve each type of problem. Some of the theoretical and practical problems related to OSA were also explained. An overview of time complexity in computation theory was given. Finally, several meta-heuristic and mathematical programming techniques that could be utilised for solving the OSA were also explained.

The aim of this chapter was to inform the reader about the previous work in OSA problem and algorithm techniques that were investigated in this thesis. This chapter

serves as an introductory material to Chapter 4 for integer programming models, to Chapter 5 for local search heuristics that work on acceptance-rejection of random move neighbourhoods, to Chapter 6 for an evolutionary greedy local search algorithm that traverses the neighbourhood greedily and to Chapter 7 for the combination of different type of algorithms considered in Chapter 4, 5 and 6 in a single framework.

Integer Programming Formulations

4.1 Introduction

In this chapter, the application of mathematical programming on the office space allocation problem is investigated. The model of choice for tackling OSA is binary integer programming formulations. Binary formulations are chosen over other representations due to the ease of formulation of constraints and objectives in the problem and to reduce redundant equations in the model.

The first half of this chapter is devoted to derivations of the integer programming models. The equations in the OSA mathematical models are derived using basic equations described in [Williams, 1999]. By using these four basic equations, each constraint (whether it is *hard* or *soft*) is derived. Based upon different variants of some of the constraints, two binary integer programming models are developed. The first model does not use specific variables to describe the floor relationships while the second model does. These two models are not strictly alternatives to each other; one of them is a more generalised but less effective version while the other one has more performance in OSA instances where it is applicable.

There is also an investigation of OSA problem using the test samples created by the parametrised data instance generator described in Section 2.5. The investigation focuses on how different aspects of the OSA problem (space misuse, i.e. *overuse* and *underuse*) affect the difficulty of the problem. By observing some of the ratios between these sub-components, inquiries about the differences between the desired (while the instance is created by the generator) and the obtained values (after the instance is solved by a solver) are made. Also, the effect of different weights associated with total soft constraint violation penalty on the difficulty of solving an OSA problem is investigated in this chapter.

This chapter is organised as follows: The proofs and derivations for each *hard* and *soft* constraint and the objective function in the binary mathematical model are given in Section 4.2. Section 4.3 briefly describes a model for the *re-allocation* problem for OSA. Section 4.4 presents the reasoning why general integer programming models are not preferred over binary models in this study. Two binary models that are generated by using or not using the floor variables are presented in Section 4.5. Section 4.6 presents the experimental results on the parametrised tests instances by using both mathematical models. The conclusions for this chapter are given in Section 4.7.

4.2 Binary Mathematical Programming Model

The set of entities is denoted by E. The set of rooms is denoted by R. The set of floors is denoted by F. The size of entity e is S_e and the capacity of room r is C_r. There is a matrix x of $|E| \times |R|$ binary decision variables where each $x_{er} = 1$ if entity e is allocated to room r, otherwise $x_{er} = 0$. Let A be the adjacency list of $|R|$ adjacency vectors each one denoted by A_r Each A_r holds the list of rooms adjacent to room r. Similarly, let N be the nearby list of $|R|$ nearby vectors each one denoted by N_r. Each N_r holds the list of rooms near to room r. The adjacency vector A_r for a room r is usually smaller compared to the nearby vector N_r. More rooms are considered to be *near* to room r than to be *adjacent* to the same room.

The binary representation described in the paragraph above can be extended with additional variables which define entity and floor relationships. We will call these variables *floor variables*. The fl is a $|E| \times |F|$ size matrix which holds the floor each entity is placed in. If the entity e is placed in floor f then, $fl_{ef} = 1$, otherwise $fl_{ef} = 0$. Since each room (and hence the entity) can only be in one floor, the summation over the rows of F is equal to 1 which is given in equation 4.2.1:

$$\forall e \in E \quad \sum_{f \in F} fl_{ef} = 1 \tag{4.2.1}$$

The connection between the x and fl matrices is as follows (equation 4.2.2):

$$\forall e \in E \quad \forall f \in F \quad fl_{ef} = \sum_{r \in f}^{F} x_{er} \tag{4.2.2}$$

The relationship between the x and fl binary matrices is depicted in Figure 4.1 with a sample allocation of entities to rooms.

There are ten requirements or constraints handled in this thesis. Most of these constraints can be set as *hard* (must be satisfied) or *soft* (desirable to satisfy) in our

	r_1	r_2	r_3	r_4	r_5	r_6	r_7			fl_1	fl_2	fl_3
e_1	0	0	0	1	0	0	0		e_1	0	1	0
e_2	0	0	0	0	0	0	1		e_2	0	0	1
e_3	0	0	1	0	0	0	0		e_3	0	1	0
e_4	0	1	0	0	0	0	0		e_4	1	0	0
e_5	0	0	0	0	0	0	1		e_5	0	0	1
e_6	0	0	0	0	1	0	0		e_6	0	1	0
e_7	1	0	0	0	0	0	0		e_7	1	0	0
e_8	0	0	0	0	0	1	0		e_8	0	0	1
e_9	0	0	0	0	1	0	0		e_9	0	1	0
e_{10}	1	0	0	0	0	0	0		e_{10}	1	0	0
e_{11}	0	0	0	0	0	0	1		e_{11}	0	0	1
e_{12}	0	1	0	0	0	0	0		e_{12}	1	0	0
e_{13}	0	0	0	0	1	0	0		e_{13}	0	1	0
e_{14}	0	0	0	0	0	0	1		e_{14}	0	0	1
			x variable matrix								fl variable matrix	
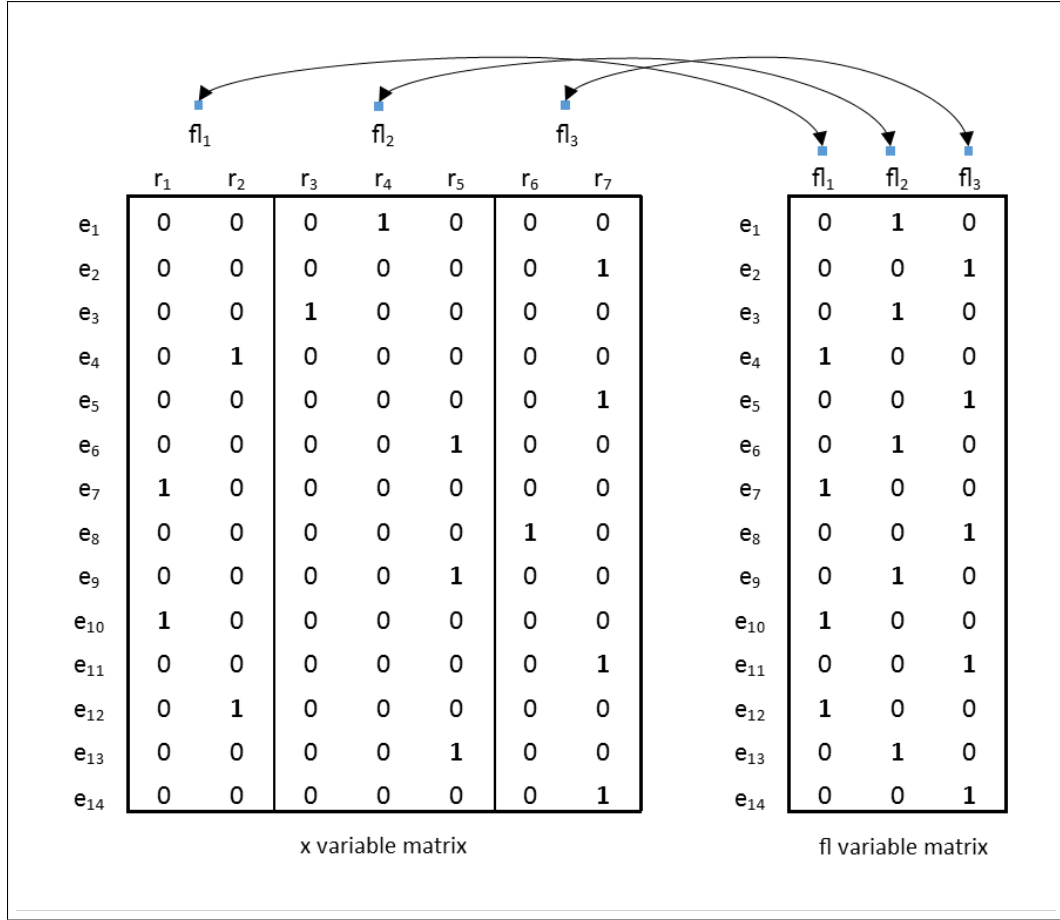
Figure 4.1: Relationship between x and fl binary matrices

formulations. In other words, when a constraint is set as soft, minimising its violation becomes an objective in the problem formulation. The exception here is the *All allocated* constraint (all entities must be allocated) which is always enforced as hard. Section 4.2.1 will present these alternative hard or soft formulations in the constraint set and in the objective function.

For each constraint type (defined below), HC^{al}, HC^{na}, HC^{sr}, HC^{nsr}, HC^{nsh}, HC^{ad}, HC^{nr}, HC^{aw}, HC^{cp} denote the corresponding constraint as *hard* while SC^{al}, SC^{na}, SC^{sr}, SC^{nsr}, SC^{nsh}, SC^{ad}, SC^{nr}, SC^{aw}, SC^{cp} denote the corresponding constraint as *soft*. Note that each *soft* constraint is associated with a binary indicator variable y^{cst} which is set to 1 if the respective *soft* constraint is violated. Some constraint types require additional binary variables (y_r^{cst}) over $r \in R$.

In order to derive the following mathematical models, four different basic equations described in [Williams, 1999] were used. These are given in equations 4.2.3, 4.2.4, 4.2.5, and 4.2.6:

$$\delta = 1 \longrightarrow \sum_j a_j \lambda_j \leq b \Longrightarrow \sum_j a_j \lambda_j + M\delta \leq M + b \tag{4.2.3}$$

$$\delta = 1 \longrightarrow \sum_j a_j \lambda_j \geq b \Longrightarrow \sum_j a_j \lambda_j + m\delta \geq m + b \tag{4.2.4}$$

$$\sum_j a_j \lambda_j \leq b \longrightarrow \delta = 1 \Longrightarrow \sum_j a_j \lambda_j - (m - \epsilon)\delta \geq b + \epsilon \tag{4.2.5}$$

$$\sum_j a_j \lambda_j \geq b \longrightarrow \delta = 1 \Longrightarrow \sum_j a_j \lambda_j - (M + \epsilon)\delta \leq b - \epsilon \tag{4.2.6}$$

In these equations m and M are lower and upper bounds on $\sum_j (a_j \lambda_j) - b$ i.e. $m \leq \sum_j (a_j \lambda_j) - b \leq M$ and ϵ is an arbitrarily small number. Notice that in these basic equations, a summation $\sum_j (a_j \lambda_j)$ is compared to a constant b value, and the δ variable is set according to this comparison. If the summation $\sum_j (a_j \lambda_j)$ is equal to b, then a combination of equations 4.2.3 and 4.2.4 or 4.2.5, and 4.2.6 have to be used together. However, due to the nature of office space allocation constraints, it is sometimes possible to make some simplifications/adjustments to these equations. In a typical OSA constraint formulation, a range of rooms is searched to check if a specific entity is placed into one of these rooms; a summation of the form $\sum_j (a_j \lambda_j)$ as in above equations can be used to represent such checks.

Ideally, the bounds m and M should be taken as tight as possible in order to make the resulting formulation as strict as possible. There are mainly two reasons for trying to make m and M as tight as possible. If m is taken too small or M is taken too large with respect to $(a_j \lambda_j) - b$, the simplex algorithm may run into numerical stability issues due to not exact representation of small floating point numbers. Also, without tight m and M values, it may take longer to solve the linear relaxation of the problem with simplex algorithm due to slow convergence.

4.2.1 Formulation of the Constraints

All Allocated

Hard Constraint: each entity $e \in E$ must be allocated to exactly one room $r \in R$.

In all allocation, the summation over all the rooms for all entities e should be equal to 1 as given in equation 4.2.7. This constraint is always taken as hard.

$$\sum_{r \in R} x_{er} = 1 \quad \forall e \in E \tag{4.2.7}$$

72

Allocation Constraint

An entity should be allocated to a specific room.

Hard Constraint: entity e to be placed into room r ($(e, r) \in HC^{\text{al}}$).

The formulation is simply derived by an equation which checks if the corresponding location in the x matrix is equal to 1 as in equation 4.2.8:

$$x_{er} = 1 \tag{4.2.8}$$

Soft Constraint: indicator variable $y^{\text{al}}(i)$ is set to 1 if $SC^{\text{al}}(i)$ is not satisfied.

Since the constraint is violated when the corresponding location x_{er} is equal to 0 and satisfied when it is equal to 1 as in equations 4.2.9 and 4.2.10:

$$x_{er} = 0 \longrightarrow y^{\text{al}}(i) = 1 \tag{4.2.9}$$

$$x_{er} = 1 \longrightarrow y^{\text{al}}(i) = 0 \tag{4.2.10}$$

The formulation for the soft constraint then becomes as in equation 4.2.11:

$$y^{\text{al}}(i) = 1 - x_{er} \tag{4.2.11}$$

Non-Allocation Constraint

An entity should not be allocated to a specific room.

Hard Constraint: entity e not to be placed into room r ($(e, r) \in HC^{\text{na}}$).

The formulation is simply derived by an equation which checks if the corresponding location in the x matrix is equal to 0 as in equation 4.2.12:

$$x_{er} = 0 \tag{4.2.12}$$

Soft Constraint: indicator variable $y^{\text{na}}(i)$ is set to 1 if $SC^{\text{na}}(i)$ is not satisfied.

This constraint is the opposite of the allocation constraint. It is violated when the corresponding location x_{er} is equal to 1 and satisfied when it is equal to 0 as in equations 4.2.13 and 4.2.14

$$x_{er} = 0 \longrightarrow y^{\text{na}}(i) = 0 \tag{4.2.13}$$

$$x_{er} = 1 \longrightarrow y^{\text{na}}(i) = 1 \tag{4.2.14}$$

The formulation for the soft constraint then becomes as in equation 4.2.15:

$$y^{\text{na}}(i) = x_{er} \tag{4.2.15}$$

Same Room Constraint

Two entities should be allocated to the same room.

Hard Constraint: entities e_1 and e_2 to be placed into the same room $((e_1, e_2) \in HC^{\text{sr}})$.

This formulation would require the column rows of x_{e_1} and x_{e_2} to be equal at each location which is given in equations 4.2.16 and 4.2.17:

$$x_{e_1 r} = 0 \longleftrightarrow x_{e_2 r} = 0 \quad \forall r \in R \tag{4.2.16}$$

$$x_{e_1 r} = 1 \longleftrightarrow x_{e_2 r} = 1 \quad \forall r \in R \tag{4.2.17}$$

The formulation for the hard constraint then becomes as in 4.2.18:

$$x_{e_1 r} - x_{e_2 r} = 0 \quad \forall r \in R \tag{4.2.18}$$

Soft Constraint: indicator variable $y^{\text{sr}}(i)$ is set to 1 if $SC^{\text{sr}}(i)$ is not satisfied.

Since *soft same room* constraint is non-linear at its default case, we are going to linearise this constraint. Therefore, unlike the following constraints which start with applying the basic equations 4.2.3, 4.2.4, 4.2.5, and 4.2.6 on the *hard* constraint version, we are going to apply different logical functions for the *soft same room* constraint. This should give a slightly different form in the final equation unlike the following constraints. Two alternate ways to formulate the soft constraint version of *same room* are considered:

Absolute Value Formulation: In this formulation, the *same room* relationship is defined as in equations 4.2.19 and 4.2.20:

$$|x_{e_1 r} - x_{e_2 r}| = 0 \longrightarrow y_r^{\text{sr}}(i) = 0 \tag{4.2.19}$$

$$|x_{e_1 r} - x_{e_2 r}| = 1 \longrightarrow y_r^{\text{sr}}(i) = 1 \tag{4.2.20}$$

This absolute value formulation can be represented linearly by bounding $x_{e_1 r} - x_{e_2 r}$ and $x_{e_2 r} - x_{e_1 r}$ with $y_r^{\text{sr}}(i)$ as in equations 4.2.21 and 4.2.22:

$$x_{e_1 r} - x_{e_2 r} \leq y_r^{\text{sr}}(i) \tag{4.2.21}$$

$$x_{e_2 r} - x_{e_1 r} \leq y_r^{\text{sr}}(i) \tag{4.2.22}$$

Then, a summation of $y_r^{\text{sr}}(i)$ variables over r can be written. Notice that in this summation, the violation will be counted twice (because two locations in $y_r^{\text{sr}}(i)$ are going to 1 if the constraint is violated), that is why the final summation is multiplied by 0.5 instead to count the violation only once. The final form then becomes as in

74

equation 4.2.23:

$$y^{\text{sr}}(i) = 0.5 \sum_{r \in R} y_r^{\text{sr}}(i) \tag{4.2.23}$$

Not IF formulation: Consider the following equations 4.2.24, 4.2.25, 4.2.26 and 4.2.27 which define the *same room* conditions:

$$x_{e_1 r} = 0 \quad AND \quad x_{e_2 r} = 0 \longrightarrow y_r^{\text{sr}}(i) = 0 \tag{4.2.24}$$

$$x_{e_1 r} = 0 \quad AND \quad x_{e_2 r} = 1 \longrightarrow y_r^{\text{sr}}(i) = 0 \tag{4.2.25}$$

$$x_{e_1 r} = 1 \quad AND \quad x_{e_2 r} = 0 \longrightarrow y_r^{\text{sr}}(i) = 1 \tag{4.2.26}$$

$$x_{e_1 r} = 1 \quad AND \quad x_{e_2 r} = 1 \longrightarrow y_r^{\text{sr}}(i) = 0 \tag{4.2.27}$$

In these cases, there is a check to determine whether the binary variables $x_{e_1 r}$ and $x_{e_2 r}$ are equal or not. If both are equal to 0 or 1 (as in equations 4.2.24 and 4.2.27), then the constraint is not violated so the corresponding sub-indicator variable $y_r^{\text{sr}}(i)$ can be set to 0. The constraint will be violated as in equations 4.2.25 and 4.2.26. However, for simplification, the violation in equation 4.2.25 can be safely ignored and it is only counted once as in equation 4.2.26 instead. Overall, this relationship can be represented as the negation of IF function with two variables $x_{e_1 r}$ and $x_{e_2 r}$.

Note that this formulation will only work if the *all allocated* constraint is enforced as *hard* (which is the default case in this thesis). Since an entity can only be allocated to a single room and all entities have to be allocated, this formulation when summed over all rooms will count the violation only once. As a result, the sub-indicator variables $y_r^{\text{sr}}(i)$ can only be 1 at one specific location where the constraint is violated. The equation 4.2.26 is rewritten based upon this condition and then is expressed as in 4.2.28:

$$x_{e_1 r} - x_{e_2 r} \geq 1 \longleftrightarrow y_r^{\text{sr}}(i) = 1 \tag{4.2.28}$$

If equations 4.2.4 and 4.2.6 are applied to 4.2.28, we obtain equations 4.2.29 and 4.2.30:

$$y_r^{\text{sr}}(i) = 1 \longrightarrow x_{e_1 r} - x_{e_2 r} \geq b \implies x_{e_1 r} - x_{e_2 r} + m y_r^{\text{sr}}(i) \geq m + 1 \tag{4.2.29}$$

$$x_{e_1 r} - x_{e_2 r} \geq b \longrightarrow y_r^{\text{sr}}(i) = 1 \implies x_{e_1 r} - x_{e_2 r} - (M + \epsilon) y_r^{\text{sr}}(i) \leq 1 - \epsilon \tag{4.2.30}$$

where $b = 1$ and $m \leq x_{e_1 r} - x_{e_2 r} - b \leq M$, then the tightest values for m and M become $m = -2$ and $M = 0$. By applying b, m and M, and rewriting equations 4.2.29 and 4.2.30, and summing $y_r^{\text{sr}}(i)$ over R, the final formulation of the *same room* constraint is obtained as in equations 4.2.31 and 4.2.32.

$$2 y_r^{\text{sr}}(i) - 1 \leq x_{e_1 r} - x_{e_2 r} \leq 1 - \epsilon + \epsilon y_r^{\text{sr}}(i) \quad \forall r \in R \tag{4.2.31}$$

$$y^{\text{sr}}(i) = \sum_{r \in R} y_r^{\text{sr}}(i) \tag{4.2.32}$$

Not Same Room Constraint

Two entities should not be allocated to the same room.

Hard Constraint: entities e_1 and e_2 to be placed into different rooms $((e_1, e_2) \in HC^{nsr})$.

This formulation would require the row vectors of x_{e_1} and x_{e_2} to differ in two locations as in equations 4.2.33 and 4.2.34:

$$x_{e_1 r} = 1 \longleftrightarrow x_{e_2 r} = 0 \quad \forall r \in R \tag{4.2.33}$$

$$x_{e_1 r} = 0 \longleftrightarrow x_{e_2 r} = 1 \quad \forall r \in R \tag{4.2.34}$$

This equation can be rewritten by using a NAND function which leads to equation 4.2.35:

$$x_{e_1 r} + x_{e_2 r} \leq 1 \quad \forall r \in R \tag{4.2.35}$$

Soft Constraint: indicator variable $y^{nsr}(i)$ is set to 1 if $SC^{nsr}(i)$ is not satisfied.

Starting with the *hard constraint* version as in 4.2.35 and by assigning a sub-indicator variable $y_r^{nsr}(i) = 1$ if the constraint is satisfied at that specific room $r \in R$, $y_r^{nsr}(i) = 0$ otherwise, the following equation 4.2.36 is derived:

$$x_{e_1 r} + x_{e_2 r} \leq 1 \longleftrightarrow y_r^{nsr}(i) = 1 \quad \forall r \in R \tag{4.2.36}$$

The basic equations 4.2.3 and 4.2.5 are applied to 4.2.36

$$y_r^{nsr}(i) = 1 \longrightarrow x_{e_1 r} + x_{e_2 r} \leq b \implies x_{e_1 r} + x_{e_2 r} + M y_r^{nsr}(i) \leq M + b \tag{4.2.37}$$

$$x_{e_1 r} + x_{e_2 r} \leq b \longrightarrow y_r^{nsr}(i) = 1 \implies x_{e_1 r} + x_{e_2 r} - (m - \epsilon) y_r^{nsr}(i) \geq b + \epsilon \tag{4.2.38}$$

where $b = 1$ and $m \leq x_{e_1 r} + x_{e_2 r} - b \leq M$. The tightest values are m and M as $m = -1$ and $M = 1$. By applying b, m and M to equations 4.2.37 and 4.2.38 and rewriting the equations the final formulation is obtained as in equations 4.2.39 and 4.2.40. Notice that in the final summation in 4.2.40, the formulation $(1 - y_r^{nsr}(i))$ is used instead. In this way (unlike the *same room soft constraint*), the sub-indicator variables $y^{nsr}(i)$ will be set to 1 if the constraint is violated. This type of summation is going to be used for the rest of the constraints from now on because the derivations of *not same room* and the following *soft* constraints start with the *hard* constraint formulation.

$$(1 + \epsilon) - (1 + \epsilon) y_r^{nsr}(i) \leq x_{e_1 r} + x_{e_2 r} \leq 2 - y_r^{nsr}(i) \quad \forall r \in R \tag{4.2.39}$$

$$y^{nsr}(i) = \sum_{r \in R} (1 - y_r^{nsr}(i)) \tag{4.2.40}$$

Not Sharing (Formulation 1)

An entity should not share a room with others.

Hard Constraint: entity e should not share a room with any other entity ($e \in HC^{\text{nsh}}$).

In this variant, the row vector of e is compared with the other row vectors $e' \in E - e$ one by one. If any row vector of e is equal to any e', then the constraint is violated. The comparison function is implemented using a NAND function similar to *not same room* constraint:

$$x_{er} + x_{e'r} \leq 1 \quad \forall e' \in E - e \quad \forall r \in R \tag{4.2.41}$$

Soft Constraint: indicator variable $y^{\text{nsh}}(i)$ is set to 1 if $SC^{\text{nsh}}(i)$ is not satisfied.

Starting with the equation 4.2.41, the sub-indicator variable $y^{\text{nsh}}(i) = 0$ if the constraint is violated, $y^{\text{nsh}}(i) = 1$ otherwise.

$$x_{er} + x_{e'r} \leq 1 \longleftrightarrow y^{\text{nsh}}(i) = 1 \quad \forall e' \in E - e \quad \forall r \in R \tag{4.2.42}$$

By applying the basic equations 4.2.3 and 4.2.5 on 4.2.42, following equations 4.2.43 and 4.2.44 are obtained:

$$y^{\text{nsh}}_{e'r}(i) = 1 \longrightarrow x_{er} + x_{e'r} \leq b \implies x_{er} + x_{e'r} + M y^{\text{nsh}}_{e'r}(i) \leq M + b \tag{4.2.43}$$

$$x_{er} + x_{e'r} \leq b \longrightarrow y^{\text{nsh}}_{e'r}(i) = 1 \implies x_{er} + x_{e'r} - (m - \epsilon) y^{\text{nsh}}_{e'r}(i) \geq b + \epsilon \tag{4.2.44}$$

where $b = 1$ and $m \leq x_{er} + x_{e'r} - 1 \leq M$. The tightest bounds are $m = -1$ and $M = 1$. Plugging the appropriate values in equations 4.2.43 and 4.2.44, final equations 4.2.45 and 4.2.46 are obtained.

$$(1 - \epsilon) - (1 - \epsilon) y^{\text{nsh}}_{e'r}(i) \leq x_{er} + x_{e'r} \leq 2 - y^{\text{nsh}}_{e'r}(i) \quad \forall e' \in E - e \quad \forall r \in R \tag{4.2.45}$$

$$y^{\text{nsh}}(i) = \sum_{e' \in E - e} \sum_{r \in R} (1 - y^{\text{nsh}}_{e'r}(i)) \tag{4.2.46}$$

During our experiments, it was observed that this row based comparison formulation used too much memory. This was due to the need of additional variables for each row comparison between the row corresponding to entity e that is not to be shared with others. As a result, the following formulation that requires a lot less memory is going to be described next is used in our experiments.

Not Sharing (Formulation 2)

An entity should not share a room with others.

Hard Constraint: entity e should not share a room with any other entity ($e \in HC^{\mathrm{nsh}}$).

In this formulation, the room r where the entity e is placed is searched ($x_{er} = 1$). When the corresponding row vector r is found, the summation over the whole vector should be 1 (or in our adjusted formulation the summation over the whole vector minus the location x_{er} should be 0) which is in equation 4.2.47:

$$x_{er} = 1 \longrightarrow \sum_{f \in E-e} x_{fr} = 0 \quad \forall r \in R \tag{4.2.47}$$

Applying the basic equations 4.2.3 and 4.2.4 to 4.2.47, following conditions are derived as in equations 4.2.48 and 4.2.49

$$x_{er} = 1 \longrightarrow \sum_{f \in E-e} x_{fr} \leq b \implies \sum_{f \in E-e} x_{fr} + Mx_{er} \leq M + b \tag{4.2.48}$$

$$x_{er} = 1 \longrightarrow \sum_{f \in E-e} x_{fr} \geq b \implies \sum_{f \in E-e} x_{fr} + mx_{er} \geq m + b \tag{4.2.49}$$

where $b = 0$ and $m \leq \sum_{f \in E-e} x_{fr} \leq M$. The tightest values for m and M are $m = 0$ and $M = |E| - 1$. By applying the values, rewriting and simplifying equations 4.2.48 and 4.2.49, the final equation 4.2.50 becomes:

$$\sum_{f \in E-e} x_{fr} \leq (|E| - 1) - (|E| - 1)x_{er} \quad \forall r \in R \tag{4.2.50}$$

Soft Constraint: indicator variable $y^{\mathrm{nsh}}(i)$ is set to 1 if $SC^{\mathrm{nsh}}(i)$ is not satisfied.

Starting with the equation 4.2.50, the sub-indicator variable $y_r^{\mathrm{nsh}}(i) = 1$ if the constraint is satisfied, $y_r^{\mathrm{nsh}}(i) = 0$ otherwise; the condition then becomes as in equation 4.2.51.

$$\sum_{f \in E-e} x_{fr} \leq (|E| - 1) - (|E| - 1)x_{er} \longleftrightarrow y_r^{\mathrm{nsh}}(i) = 1 \quad \forall r \in R \tag{4.2.51}$$

Applying the basic equations 4.2.3 and 4.2.5 on 4.2.51, equations 4.2.52 and 4.2.53 are obtained respectively.

$$y_r^{\mathrm{nsh}}(i) = 1 \longrightarrow \sum_{f \in E-e} x_{fr} \leq b \implies \sum_{f \in E-e} x_{fr} + My_r^{\mathrm{nsh}}(i) \leq M + b \tag{4.2.52}$$

$$\sum_{f \in E-e} x_{fr} \leq b \longrightarrow y_r^{\mathrm{nsh}}(i) = 1 \implies \sum_{f \in E-e} x_{fr} - (m - \epsilon)y_r^{\mathrm{nsh}}(i) \geq b + \epsilon \tag{4.2.53}$$

where $b = (|E| - 1) - (|E| - 1)x_{er}$ and $m \leq \sum_{f \in E-e} x_{fr} - ((|E| - 1) - (|E| - 1)x_{er}) \leq M$. By setting $\sum_{f \in E-e} x_{fr}$ to at least 0 and at most $(|E| - 1)$ and setting x_{er} as 0 or 1, m and M become $m = -(|E| - 1)$ and $M = (|E| - 1)$. Plugging these values and with

reorganisation of equations 4.2.52 and 4.2.53, the final formulation for the *not sharing soft constraint* becomes as in equations 4.2.54, 4.2.55 and 4.2.56.

$$(|E| - 1)(2 - x_{er} - y_r^{\text{nsh}}(i)) \geq \sum_{f \in E - e} x_{fr} \quad \forall r \in R \tag{4.2.54}$$

$$\sum_{f \in E - e} x_{fr} \geq (|E| - 1)(1 - x_{er}) + \epsilon - (|E| - 1 + \epsilon)y_r^{\text{nsh}}(i) \quad \forall r \in R \tag{4.2.55}$$

$$y^{\text{nsh}}(i) = \sum_{r \in R}(1 - y_r^{\text{nsh}}(i)) \tag{4.2.56}$$

Adjacency Constraint

Two entities should be allocated to adjacent rooms.

For this constraint, the adjacency lists A are used for each room. Each A_r list holds the rooms adjacent to room r.

***Hard Constraint*:** entities e_1 and e_2 must be placed into adjacent rooms ($(e_1, e_2) \in HC^{\text{ad}}$).

The formulation will check whenever the entity e_1 is placed into room r, the summation over the locations $x_{e_2 s} \quad \forall s \in A_r$ should be equal to 1 which is equation 4.2.57:

$$x_{e_1 r} = 1 \longrightarrow \sum_{s \in A_r} x_{e_2 s} = 1 \quad \forall r \in R \tag{4.2.57}$$

By applying the basic equations 4.2.3 and 4.2.4 using 4.2.57, the following equations 4.2.58 and 4.2.59 are obtained respectively:

$$x_{e_1 r} = 1 \longrightarrow \sum_{s \in A_r} x_{e_2 s} \leq b \Longrightarrow \sum_{s \in A_r} x_{e_2 s} + M x_{e_1 r} \leq M + b \tag{4.2.58}$$

$$x_{e_1 r} = 1 \longrightarrow \sum_{s \in A_r} x_{e_2 s} \geq b \Longrightarrow \sum_{s \in A_r} x_{e_2 s} + m x_{e_1 r} \geq m + b \tag{4.2.59}$$

By setting $b = 1$, $m \leq \sum_{s \in A_r} x_{e_2 s} - 1 \leq M$ which yields $m = -1$ and $M = 0$ are obtained as tightest bounds. Plugging these values into equations 4.2.58 and 4.2.59, the final equation 4.2.60 becomes:

$$x_{e_1 r} \leq \sum_{s \in A_r} x_{e_2 s} \leq 1 \quad \forall r \in R \tag{4.2.60}$$

***Soft Constraint*:** indicator variable $y^{\text{ad}}(i)$ is set to 1 if $SC^{\text{ad}}(i)$ is not satisfied.

Starting with the equation 4.2.60, the sub-indicator variable $y_r^{\text{ad}}(i) = 1$ if the constraint is satisfied at specific room r as in equation 4.2.61. Notice that the equation is

simplified by the removal of $\sum_{s \in A_r} x_{e_2 s} \leq 1$ since this summation can never be greater than 1 and this inequality will hence be satisfied in all conditions.

$$x_{e_1 r} \leq \sum_{s \in A_r} x_{e_2 s} \longleftrightarrow y_r^{\text{ad}}(i) = 1 \quad \forall r \in R \tag{4.2.61}$$

Using the basic equations 4.2.4 and 4.2.6 on 4.2.61, equations 4.2.62 and 4.2.63 are derived respectively.

$$y_r^{\text{ad}}(i) = 1 \longrightarrow \sum_{s \in A_r} x_{e_2 s} \geq b \Longrightarrow \sum_{s \in A_r} x_{e_2 s} + m y_r^{\text{ad}}(i) \geq m + b \tag{4.2.62}$$

$$\sum_{s \in A_r} x_{e_2 s} \geq b \longrightarrow y_r^{\text{ad}}(i) = 1 \Longrightarrow \sum_{s \in A_r} x_{e_2 s} - (M + \epsilon) y_r^{\text{ad}}(i) \leq b - \epsilon \tag{4.2.63}$$

where $b = x_{e_1 r}$ and $m \leq \sum_{s \in A_r} x_{e_2 s} - x_{e_1 r} \leq M$. The tightest bounds in this case are $m = -1$ and $M = 1$. By plugging the appropriate values in equations 4.2.62 and 4.2.63, and with reorganisation, the final form as in equations 4.2.64 and 4.2.65 is obtained:

$$y_r^{\text{ad}}(i) + x_{e_1 r} - 1 \leq \sum_{s \in A_r} x_{e_2 s} \leq x_{e_1 r} - \epsilon + (1 + \epsilon) y_r^{\text{ad}}(i) \quad \forall r \in R \tag{4.2.64}$$

$$y^{\text{ad}}(i) = \sum_{r \in R} (1 - y_r^{\text{ad}}(i)) \tag{4.2.65}$$

Nearby Constraint

Two entities should be allocated to adjacent rooms. For this constraint, the nearby lists N are used for each room. Each N_r list holds the rooms adjacent to room r.

Hard Constraint: entities in a group placed near to the group head e_2 $((e_1, e_2) \in HC^{\text{nr}})$.

The formulation will check whenever the entity e_1 is placed into the room r, the summation over the locations $x_{e_2 s}$ $s \in N_r$ should be equal to 1 which is equation 4.2.66:

$$x_{e_1 r} = 1 \longrightarrow \sum_{s \in N_r} x_{e_2 s} = 1 \quad \forall r \in R \tag{4.2.66}$$

By applying basic equations 4.2.3 and 4.2.4 on 4.2.66, following equations 4.2.67 and 4.2.68 are obtained:

$$x_{e_1 r} = 1 \longrightarrow \sum_{s \in N_r} x_{e_2 s} \leq b \Longrightarrow \sum_{s \in N_r} x_{e_2 s} + M x_{e_1 r} \leq M + b \tag{4.2.67}$$

$$x_{e_1 r} = 1 \longrightarrow \sum_{s \in N_r} x_{e_2 s} \geq b \Longrightarrow \sum_{s \in N_r} x_{e_2 s} + m x_{e_1 r} \geq m + b \tag{4.2.68}$$

where $b = 1$ and $m \leq \sum_{s \in N_r} x_{e_2 s} - 1 \leq M$. The tightest values are $m = -1$ and $M = 0$. By plugging values in equations 4.2.67 and 4.2.68, and combining them, the

final equation 4.2.69 becomes:

$$x_{e_1 r} \leq \sum_{s \in N_r} x_{e_2 s} \leq 1 \quad \forall r \in R \tag{4.2.69}$$

Soft Constraint: indicator variable $y^{\text{nr}}(i)$ is set to 1 if $SC^{\text{nr}}(i)$ is not satisfied.

Starting with the equation 4.2.69, we set the sub-indicator variable $y_r^{\text{nr}}(i) = 1$ if the constraint is satisfied at specific room r as in equation 4.2.70. Notice that the equation is simplified by the removal of $\sum_{s \in N_r} x_{e_2 s} \leq 1$ since this summation can never be greater than 1 and this inequality will hence be satisfied in all conditions.

$$x_{e_1 r} \leq \sum_{s \in N_r} x_{e_2 s} \longleftrightarrow y_r^{\text{nr}}(i) = 1 \quad \forall r \in R \tag{4.2.70}$$

Using the basic equations 4.2.4 and 4.2.6 on 4.2.70, the equations 4.2.71 and 4.2.72 are derived:

$$y_r^{\text{nr}}(i) = 1 \longrightarrow \sum_{s \in N_r} x_{e_2 s} \geq b \implies \sum_{s \in N_r} x_{e_2 s} + m y_r^{\text{nr}}(i) \geq m + b \tag{4.2.71}$$

$$\sum_{s \in N_r} x_{e_2 s} \geq b \longrightarrow y_r^{\text{nr}}(i) = 1 \implies \sum_{s \in N_r} x_{e_2 s} - (M + \epsilon) y_r^{\text{nr}}(i) \leq b - \epsilon \tag{4.2.72}$$

where $b = x_{e_1 r}$ and $m \leq \sum_{s \in N_r} x_{e_2 s} - x_{e_1 r} \leq M$. The tightest bounds in this case are $m = -1$ and $M = 1$. By plugging the appropriate values in equations 4.2.71 and 4.2.72, and with reorganisation, the final form becomes as in equations 4.2.73 and 4.2.74:

$$y_r^{\text{nr}}(i) + x_{e_1 r} - 1 \leq \sum_{s \in N_r} x_{e_2 s} \leq x_{e_1 r} - \epsilon + (1 + \epsilon) y_r^{\text{nr}}(i) \quad \forall r \in R \tag{4.2.73}$$

$$y^{\text{nr}}(i) = \sum_{r \in R} (1 - y_r^{\text{nr}}(i)) \tag{4.2.74}$$

Nearby Constraint with Floor Variables

In this formulation, the floor variable matrix fl is utilised. This formulation can only work in instances when the allocation is clearly organised in terms of floor structures. If the neighbourhood relationship is defined in a way different than a floor relationship, then the *nearby* constraint version given previously has to be used.

Hard Constraint: entities e_1 and e_2 must be placed in the same floor $((e_1, e_2) \in HC^{\text{nr}})$.

This formulation is practically very similar to the *same room* constraint that two entities have to be placed in the same floor instead of in the same room as in equations 4.2.75 and 4.2.76:

$$fl_{e_1 f} = 0 \longleftrightarrow fl_{e_2 f} = 0 \quad \forall f \in F \tag{4.2.75}$$

$$fl_{e_1 f} = 1 \longleftrightarrow fl_{e_2 f} = 1 \quad \forall f \in F \tag{4.2.76}$$

then the formulation for the hard constraint becomes:

$$fl_{e_1r} - fl_{e_2r} = 0 \quad \forall f \in F \tag{4.2.77}$$

Soft Constraint: indicator variable $y^{nr}(i)$ is set to 1 if $SC^{nr}(i)$ is not satisfied.

We rewrite the equation 4.2.77 using a similar reasoning in deriving the *same room* constraint as in equation 4.2.78:

$$fl_{e_1f} - fl_{e_2f} \geq 1 \longleftrightarrow y^{nr}_f(i) = 1 \tag{4.2.78}$$

If equations 4.2.4 and 4.2.6 are applied to 4.2.78, then equations 4.2.79 and 4.2.80 become:

$$y^{nr}_f(i) = 1 \longrightarrow fl_{e_1f} - fl_{e_2f} \geq b \implies fl_{e_1f} - fl_{e_2f} + my^{nr}_f(i) \geq m+1 \tag{4.2.79}$$

$$fl_{e_1f} - fl_{e_2f} \geq b \longrightarrow y^{nr}_f(i) = 1 \implies fl_{e_1f} - fl_{e_2f} - (M+\epsilon)y^{nr}_f(i) \leq 1-\epsilon \tag{4.2.80}$$

where $b = 1$ and $m \leq fl_{e_1f} - fl_{e_2f} - b \leq M$, then the tightest values for m and M become $m = -2$ and $M = 0$. By applying b, m and M and combining, and rewriting equations 4.2.79 and 4.2.80, and summing the $y^{nr}_r(i)$ over R, the final formulation for *nearby* constraint is obtained in equations 4.2.81 and 4.2.82.

$$2y^{nr}_f(i) - 1 \leq fl_{e_1f} - fl_{e_2f} \leq 1 - \epsilon + \epsilon y^{nr}_f(i) \quad \forall f \in F \tag{4.2.81}$$

$$y^{nr}(i) = \sum_{f \in F} y^{nr}_f(i) \tag{4.2.82}$$

Away From Constraint

Two entities should be placed away from each other.

Hard Constraint: entities e_1 and e_2 must be placed in rooms away from each other ($(e_1, e_2) \in HC^{aw}$).

The formulation will check whenever the entity e_1 is placed into the room r, the summation over the locations x_{e_2s} $\quad \forall s \in N_r$ should be equal to 0 which is:

$$x_{e_1r} = 1 \longrightarrow \sum_{s \in N_r} x_{e_2s} = 0 \quad \forall r \in R \tag{4.2.83}$$

By applying basic equations 4.2.3 and 4.2.4 on 4.2.83, equations 4.2.84 and 4.2.85 are obtained respectively:

$$x_{e_1r} = 1 \longrightarrow \sum_{s \in N_r} x_{e_2s} \leq b \implies \sum_{s \in N_r} x_{e_2s} + Mx_{e_1r} \leq M + b \tag{4.2.84}$$

$$x_{e_1r} = 1 \longrightarrow \sum_{s \in N_r} x_{e_2s} \geq b \implies \sum_{s \in N_r} x_{e_2s} + mx_{e_1r} \geq m + b \tag{4.2.85}$$

where $b = 0$ and $m \leq \sum_{s \in N_r} x_{e_2 s} \leq M$. In this case, the tightest bounds are $m = 0$ and $M = 1$. Applying these on equations 4.2.84 and 4.2.85, the final equation 4.2.86 becomes:

$$0 \leq \sum_{s \in N_r} x_{e_2 s} \leq 1 - x_{e_1 r} \quad \forall r \in R \tag{4.2.86}$$

Soft Constraint: indicator variable $y^{\text{aw}}(i)$ is set to 1 if $SC^{\text{aw}}(i)$ is not satisfied.

Starting with the equation 4.2.86, we set the sub-indicator variable $y_r^{\text{aw}}(i) = 1$ if the constraint is satisfied at specific room r, $y_r^{\text{aw}}(i) = 0$ otherwise, as in equation 4.2.87. Notice that the equation is simplified by the removal of $0 \leq \sum_{s \in N_r} x_{e_2 s}$ since the summation can never be less than 0 and this inequality will hence be satisfied in all conditions.

$$\sum_{s \in N_r} x_{e_2 s} \leq 1 - x_{e_1 r} \longleftrightarrow y_r^{\text{aw}}(i) = 1 \quad \forall r \in R \tag{4.2.87}$$

By applying basic equations 4.2.3 and 4.2.5 on 4.2.87, equations 4.2.88 and 4.2.88 are derived respectively:

$$y_r^{\text{aw}}(i) = 1 \longrightarrow \sum_{s \in N_r} x_{e_2 s} \leq b \implies \sum_{s \in N_r} x_{e_2 s} + M y_r^{\text{aw}}(i) \leq M + b \tag{4.2.88}$$

$$\sum_{s \in N_r} x_{e_2 s} \leq b \longrightarrow y_r^{\text{aw}}(i) = 1 \implies \sum_{s \in N_r} x_{e_2 s} - (m - \epsilon) y_r^{\text{aw}}(i) \geq b + \epsilon \tag{4.2.89}$$

In this case, $b = 1 - x_{e_1 r}$ and $m \leq \sum_{s \in N_r} x_{e_2 s} - (1 - x_{e_1 r}) \leq M$. The tightest bounds are $m = -1$ and $M = 1$. Plugging the appropriate values to equations 4.2.88 and 4.2.89, final equations 4.2.90 and 4.2.91 become:

$$1 - x_{e_1 r} + \epsilon - (1 + \epsilon) y_r^{\text{aw}}(i) \leq \sum_{s \in N_r} x_{e_2 s} \leq 2 - x_{er} - y_r^{\text{aw}}(i) \quad \forall r \in R \tag{4.2.90}$$

$$y^{\text{aw}}(i) = \sum_{r \in R} (1 - y_r^{\text{aw}}(i)) \tag{4.2.91}$$

Away From Constraint with Floor Variables

Hard Constraint: entities e_1 and e_2 must be placed in rooms away from each other $((e_1, e_2) \in HC^{\text{aw}})$.

This formulation, similar to the nearby variant, uses the floor variable matrix fl and can only be used if the allocation is organised in terms of floor relationships. This formulation is derived similar to the *not same room* constraint that the row vectors of e_1 and e_2 in fl should not be equal to each other as in equations 4.2.92 and 4.2.93:

$$fl_{e_1 f} = 0 \longleftrightarrow fl_{e_2 r} = 1 \quad \forall f \in F \tag{4.2.92}$$

$$fl_{e_1 f} = 1 \longleftrightarrow fl_{e_2 r} = 0 \quad \forall f \in F \tag{4.2.93}$$

The formulation for the *hard* constraint then becomes as in equation 4.2.94:

$$fl_{e_1f} + fl_{e_2f} \leq 1 \quad \forall f \in F \tag{4.2.94}$$

Soft Constraint: indicator variable $y^{\text{aw}}(i)$ is set to 1 if $SC^{\text{aw}}(i)$ is not satisfied.

Starting with the equation 4.2.94, the same reasoning used in deriving the *not same room* constraint can be applied as in equation 4.2.95:

$$fl_{e_1f} + fl_{e_2f} \leq 1 \longleftrightarrow y_f^{\text{aw}}(i) = 1 \tag{4.2.95}$$

If equations 4.2.3 and 4.2.5 are applied to 4.2.95, then equations 4.2.96 and 4.2.97 are derived respectively:

$$y_f^{\text{aw}}(i) = 1 \longrightarrow fl_{e_1f} + fl_{e_2f} \leq b \implies fl_{e_1f} + fl_{e_2f} + My_f^{\text{aw}}(i) \leq M + b \tag{4.2.96}$$

$$fl_{e_1f} + fl_{e_2f} \leq b \longrightarrow y_f^{\text{aw}}(i) = 1 \implies fl_{e_1f} + fl_{e_2f} - (m - \epsilon)y_f^{\text{aw}}(i) \geq b + \epsilon \tag{4.2.97}$$

where $b = 1$ and $m \leq fl_{e_1f} + fl_{e_2f} - 1 \leq M$. In this case, the tightest bounds are $m = -1$ and $M = 1$. Placing b, m and M into equations 4.2.96 and 4.2.97, final equations 4.2.98 and 4.2.99 are derived.

$$(1 + \epsilon) - (1 + \epsilon)y_f^{\text{aw}}(i) \leq fl_{e_1f} + fl_{e_2f} \leq 2 - y_f^{\text{aw}}(i) \quad \forall f \in F \tag{4.2.98}$$

$$y^{\text{aw}}(i) = \sum_{f \in F} (1 - y_f^{\text{aw}}(i)) \tag{4.2.99}$$

Capacity Constraint

A room should not be overused.

Hard Constraint: Room r must not be overused ($r \in HC^{\text{cp}}$).

For this formulation, we multiply the entity size vector S with the column vector of the specific room r to find the space used in room r. The result should be less than or equal to the room capacity C_r as in equation 4.2.100:

$$\sum_{e \in E} S_e x_{er} \leq C_r \tag{4.2.100}$$

Soft Constraint: indicator variable $y^{\text{cp}}(i)$ is set to 1 if $SC^{\text{cp}}(i)$ is not satisfied.

Starting with the equation 4.2.100, the indicator variable $y^{\text{cp}}(i) = 0$ if the constraint is satisfied, $y^{\text{cp}}(i) = 1$ otherwise. In order to simplify the derivation process a temporary indicator variable $z^{\text{cp}}(i) = 1 - y^{\text{cp}}(i)$ is used.

$$\sum_{e \in E} S_e x_{er} \leq C_r \longleftrightarrow z^{cp}(i) = 1 \tag{4.2.101}$$

By applying basic equations 4.2.3 and 4.2.5 on 4.2.101, equations 4.2.102 and 4.2.103 are derived respectively:

$$z^{cp}(i) = 1 \longrightarrow \sum_{e \in E} S_e x_{er} \leq b \implies \sum_{e \in E} S_e x_{er} + M z^{cp}(i) \leq M + b \tag{4.2.102}$$

$$\sum_{e \in E} S_e x_{er} \leq b \longrightarrow z^{cp}(i) = 1 \implies \sum_{e \in E} S_e x_{er} - (m - \epsilon) z^{cp}(i) \geq b + \epsilon \tag{4.2.103}$$

where $b = C_r$ and $m \leq \sum_{e \in E} S_e x_{er} - C_r \leq M$. In this case, the tightest bounds are $m = -C_r$ and $M = (\sum_{e \in E} S_e - C_r)$. Plugging the appropriate values and replacing $z^{cp}(i)$ with $1 - y^{cp}(i)$, the final equations are as in 4.2.104 and 4.2.105 respectively:

$$\sum_{e \in E} S_e x_{er} + (\sum_{e \in E} S_e - C_r)(1 - y^{cp}(i)) \leq \sum_{e \in E} S_e \tag{4.2.104}$$

$$\sum_{e \in E} S_e x_{er} + (C_r + \epsilon)(1 - y^{cp}(i)) \geq C_r + \epsilon \tag{4.2.105}$$

4.2.2 Objective Function

The objective function is the weighted sum of the space misuse ($underuse + 2 \cdot overuse$) and the soft constraints violation penalty. The penalties associated to each soft constraint type are: w^{al}, w^{na}, w^{sr}, w^{nsr}, w^{nsh}, w^{ad}, w^{nr}, w^{aw}, and w^{cp}. The objective function Z to minimise is given in equation 4.2.106:

$$Z = \sum_{r \in R} max \left(C_r - \sum_{e \in E} x_{er} S_e \,,\, 2(\sum_{e \in E} x_{er} S_e - C_r) \right) \tag{4.2.106}$$

$$+ w^{al} \sum_{i=1}^{|SC^{al}|} y^{al}(i) + w^{na} \sum_{i=1}^{|SC^{na}|} y^{na}(i) + w^{sr} \sum_{i=1}^{|SC^{sr}|} y^{sr}(i)$$

$$+ w^{nsr} \sum_{i=1}^{|SC^{nsr}|} y^{nsr}(i) + w^{nsh} \sum_{i=1}^{|SC^{nsh}|} y^{nsh}(i) + w^{ad} \sum_{i=1}^{|SC^{ad}|} y^{ad}(i)$$

$$+ w^{nr} \sum_{i=1}^{|SC^{nr}|} y^{nr}(i) + w^{aw} \sum_{i=1}^{|SC^{aw}|} y^{aw}(i) + w^{cp} \sum_{i=1}^{|SC^{cp}|} y^{cp}(i)$$

The first line in equation 4.2.106 refers to the space misuse and the following lines represent individual soft constraint penalties. This formulation is the most general form for the office space allocation problem tackled in this thesis. However, the instances in the data sets considered in this thesis do not contain any *soft* constraint for the *not sharing* constraint. Therefore, the respective part in 4.2.106 ($w^{nsh} \sum_{i=1}^{|SC^{nsh}|} y^{nsh}(i)$)

can be omitted when the datasets *nott*1, *wolver* (described in Section 2.4), and *SVe*150 or *PNe*150 (described in Section 2.5.3) are considered.

The space misuse component is formulated with the maximum function. However, not every integer programming solver supports this operation. This maximum function in this formulation can be formulated using the absolute value representation which is similar to the absolute value formulation of the *same room* constraint. A list of k_r non-integer variables for each room space misuse can be used as in equation 4.2.107, and the summation over k_r variables will give the whole space misuse. These k_r variables are bounded by the underuse and overuse as in equations 4.2.108 and 4.2.109.

$$SMP = \sum_{r \in R} k_r \tag{4.2.107}$$

$$\forall r \in R \quad k_r \geq C_r - \sum_{e \in E} x_{er} S_e \tag{4.2.108}$$

$$\forall r \in R \quad k_r \leq 2(C_r - \sum_{e \in E} x_{er} S_e) \tag{4.2.109}$$

A further optimisation on the k_r variables can be performed by using a constant upper bound value K to restrict the overuse and underuse in each room as in equation 4.2.110. This can be considered as the addition of *hard* constraints (cuts) to the model. This K upper bound can be set as ∞ (or sufficiently high number) if no explicit upper bound is desired.

$$\forall r \quad k_r \leq K \tag{4.2.110}$$

4.3 Model for Re-Allocation Problem

The mathematical formulation described in Section 4.2 is for the initial allocation problem for OSA. In that case, there is not a prior allocation of entities to room, and the formulation creates an allocation from scratch. In this section, an extension of the *allocation* model is presented in order to adapt to the *re-allocation* problem of OSA.

In the *re-allocation* (or re-organisation) variant of the OSA problem, the algorithm starts with an initial partial or complete allocation of entities to rooms. The goal is to re-optimise the solution with minimal amount of disruption to the current solution. This problem is mostly encountered due to the arrival or departure of new entities; removal, addition or restructuring of the office space; and removal or addition of constraints.

The formulation for the *re-allocation* model is based on the optimisation problem as defined in this chapter. The solution in the optimisation problem is represented by

a binary matrix x as described in Section 4.2. In the new formulation, this matrix is modified (and named x' now) based upon the changes in the entity and the room sets. We start with $x' \leftarrow x$ and modify x' as according to the changes in entity and room sets.

- *Entity Set:* If some of the entities are removed from the problem, those entities are removed from x'. Conversely, new entities are added to x'. The non-changing part of the entity set is referred to as E'.

- *Room Set:* If some of the rooms are removed from the problem, then these rooms are removed from the binary matrix x'. If two or more rooms are merged into one room, then x' will contain only one of the rooms and the other rooms will be deleted. The non-changing part of the room set is referred to as R'.

- *Constraint set:* The changes in the constraint set do not affect the binary solution matrix x'. However, the objective function Z should be modified if some of the soft constraints are altered. The updated objective function Z' is formed by searching for the deleted entities and rooms, and then deleting the constraints associated with them. New constraints associated with the new entities and room added to the problem are incorporated into Z' as well.

After these changes, it is easy to compare the two matrices x' and x. Only the part where the entities and rooms that are not changed in this modification is compared in the *re-allocation* problem. Basically, the cells in these parts of the matrices x' and x are compared one by one. If two locations differ, then this shows an entity has been re-allocated to a new room. The reorganization problem deals with the number of reallocated entities. If the number of reallocated entities are high, then this shows a major restructuring on the allocation of the office space (which is usually undesirable in an organization).

Two approaches can be taken when dealing with the number of reallocated entities in the reorganization problem.

- *Hard constraint approach:* The number of re-allocated entities cannot exceed a specific limit.

- *Soft constraint approach:* The number of re-allocated entities can be added to the objective function with a weight as a part of soft constraint violation penalty.

In order to formulate the *re-allocation* problem, another binary decision matrix l of size $|R'||E'|$ is used. Basically, l matrix checks whether or not each cell in x' or x differs

from each other. Each cell in the matrix l is set to 1 if there is a difference between two matrices as in equation 4.3.1:

$$x'_{er} - x_{er} \leq l_{er} \quad x_{er} - x'_{er} \leq l_{er} \quad \forall e \in E' \quad \forall r \in R' \tag{4.3.1}$$

If we take *re-allocation* as a *hard* constraint, then the summation over the l matrix should be lower than a limit value L as in equation 4.3.2:

$$\sum_{r \in |R'|} \sum_{e \in |E'|} l_{er} \leq L \tag{4.3.2}$$

If we take re-organization as a *soft* constraint, then the summation over the l should be added to the objective function with a weight. The modified objective function Z' then becomes as in equation 4.3.3:

$$Z' = Z + w^{reo} \sum_{r \in |R'|} \sum_{e \in |E'|} l_{er} \tag{4.3.3}$$

where w^{reo} is a weight factor for the re-organisation penalty.

If the *soft* constraint set is changed due to addition or removal of constraints, the respective section in the objective function Z' is modified accordingly. If there is an addition or removal of a *hard* constraint, then the respective constraint definition is added or removed from the *hard* constraint set.

4.4 A Model with General Integer Decision Variables

For this thesis, preliminary investigations on different representations besides using binary decision variables were performed. In a general integer programming model, the whole entity-room mapping can be implemented by using an integer array g of size $|E|$ where each location in g_i ($i = \{1, \ldots, |E|\}$) represents the room an entity i is placed. Hence, each location g_i takes a value between 1 and $|R|$.

Although some of *hard* constraints like *allocation, non-allocation, same-room*, and *not same-room* were easy to implement with this representation, modelling of the objective function and the *soft* constraints required plenty of additional binary decision variables. It was quickly noticed that eventually any general integer programming model for OSA turned into a bloated binary integer programming model with additional general integer variables. The model could be thought of a hybrid representation which synchronised the binary integer variable matrix x and the general integer decision variable array g (as described in equation 4.4.1).

$$g_i - \sum_{k=1}^{|R|} kx_{ik} = 0 \quad \forall i \in E \tag{4.4.1}$$

where $g_i = r$ if entity i is placed to room r. The summation $\sum_{k=1}^{|R|} kx_{ik}$ synchronises the binary matrix x (described in Section 4.2) with the general integer representation array g. Since an entity can only be placed in just one room, the summation actually contains only one non-zero component when the summation is taken over R. This non-zero component is rx_{ir} where $k = r$ and $x_{ir} = 1$. Since $g_i = r$ due to the way the general integer programming representation is defined (entity i is placed into room r), the difference between two components in equation 4.4.1 becomes 0.

Preliminary experiments yielded no benefit of using such a hybrid model over a strictly binary variant, so further analysis of this representation is beyond the scope of this thesis.

4.5 Two Binary Integer Programming Models

Based upon the binary decision variables, two variants of mathematical models were developed. Both models were developed by using the same formulations for *allocation*, *non-allocation*, *same room* (Not IF formulation for the soft constraints), *not same room*, *not sharing* (formulation 2), *adjacency* and *capacity* constraints. The second formulation for *not sharing* constraint was used instead of the first formulation. The second formulation utilises a check for column summation being equal to 1 where the entity e is placed. This is due to the fact that first formulation (which makes a row by row comparison between the row of entity e and other rows) consumed too much memory without providing any other computational benefit.

The main difference between two models is the existence or non-existence of floor variables. Floor variables mainly affect the formulations of *nearby* and *away from* constraints since these constraints are defined over floors. Therefore, the model with the floor variables utilises the floor variables in order to define *nearby* and *away from* constraints.

The model with the floor variables is superior to the one without the floor variables given a specific subset of instances. Based upon the instances created by the generator implemented for this thesis, the model with the floor variables provides significantly smaller simplex tables for the integer programming solvers to work on.

However, these two models are not strictly alternatives to each other. The defini-

tions of *nearby* and *away from* constraints are very important whether the floor model can be effective or even be applicable. For the instances considered in this thesis, the *nearby* and *away from* constraints are defined over a floor, that two entities are near or away from each other if they are in the same floor or not respectively. Unfortunately, this definition is not very flexible since the granularity of the relationship is at floor level. The model using the floor variables for the *nearby* and *away from* constraints is not suitable if the *nearby* and *away from* relationship is not defined over the floors but over a segment of a floor, over arbitrarily chosen rooms, or over a distance metric, etc. It might still be possible to modify this formulation to a more limited *section* based formulation. For example, a clustering algorithm might be used to identify the rooms that can be clustered within a section.

Unlike the *floor* or *section* based models, the model without the floor variables will still be able to handle section or floor based relationships regardless of how they are defined because the granularity of the *nearby* or *away from* relationship is at room level. Consequently, the choice between two models is also a trade-off between performance and generalisability.

4.5.1 Effect of Using Floor Variables on the Size of Simplex Tables

In Table 4.1, the information about the simplex tables (the number of rows, columns, non-zero and binary variables) for several test instances using models without and with floor variables is given. This information is acquired immediately after CPLEX solves the linear programming reduction of the mixed integer programming problem and performs other reductions (which is called the pre-solve stage in CPLEX) but not before the start of the branch and bound stage.

The number of rows in the simplex table is directly proportional to the number of constraints/cuts and the number of columns is directly proportional to the number of variables in the model. The non-zero variables refer to the number of non zero locations in the simplex table. A sparse table with a majority of its values being zero is usually easier to solve. The binary column gives the number of binary variables set after the pre-solve stage. It is immediately observed that the model with the floor variables yields a much smaller table after the pre-solve stage: The number of rows (constraints) is reduced to one third of the case where the floor variables are not utilised. The number of columns (variables in the model) is reduced to roughly two thirds of the previous case. The percentage of non-zero variables is also reduced from roughly 10 percent to 6 percent in several *SVe*150 and *PNe*150 instances.

	Model w/o Floor Variables				Model with Floor Variables			
Instance	*Rows*	*Columns*	*Non-zero*	*Binary*	*Rows*	*Columns*	*Non-zero*	*Binary*
$S_{0.00}V_{0.00}$	25128	25814	673588	25722	7966	17332	92898	17240
$S_{0.40}V_{0.80}$	29233	27912	767121	27820	9729	18267	105728	18175
$S_{0.80}V_{0.40}$	26979	26755	706707	26663	9089	17910	100829	17818
$P_{0.00}N_{0.00}$	27125	26818	710879	26726	9095	17903	100751	17811
$P_{0.10}N_{0.20}$	27129	26825	711494	26733	9095	17908	100883	17816
$P_{0.20}N_{0.10}$	27155	26843	712585	26751	9097	17914	100924	17822
*nott*1	32092	34893	1250566	34764	10635	24103	182789	23974
*nott*1b	14492	15084	388788	15007	5314	10563	56802	10486
*nott*1c	10653	12849	318642	12757	3577	9256	71943	9164

Table 4.1: The number of rows, columns, non-zero, and binary values in the simplex table for several data instances

It can be observed that the most significant advantage of using the model with floor variables is the drastic reduction in the number of rows which represent the number of *nearby* and *away from* constraints. The addition of floor variables to the model did not increase the number of columns either. In fact, these variables further helped elimination of other columns in the simplex table during the pre-solve stage.

4.6 Experiments Related to Integer Programming Models

To solve the 0/1 IP formulation, IBM ILOG CPLEX 12.3 [IBM-Ilog, 2013] was used on a PC with a processor Core 2 Duo E8400 3Ghz and 2GB of RAM. The first datasets used were University of Nottingham and Wolverhampton instances (*nott*1 and *wolver*). The instances created by the data generator described in Section 2.5 were also used. All the instances in the *SVe*150 and *PNe*150 dataset were tested.

When dealing with the *soft* constraints, the penalty weights were taken similarly to those used in [Landa-Silva, 2003] apart from *nearby* constraint. For *nott*1 and *wolver* instances, the penalty for the *nearby* constraint was 11.18. We could not trace any reasoning why such a value was chosen for this constraint. Therefore, for simplification, the penalty for *nearby* constraint was taken as 10 in *SVe*150 and *PNe*150 instances. The penalty for each constraint type was previously given in Table 2.1 in Section 2.3.2.

The effect of changing the penalty weights for each *soft* constraint was also tested. The total *soft constraint penalty* was either divided by half (*half soft constraint penalty condition*); taken as it was in Table 2.1 (*normal soft constraint penalty condition*) or multiplied by 2 (*double soft constraint penalty condition*).

In the following experiments, the objective function for total penalty (TP) is taken as the weighted summation of space misuse (SMP) and soft constraint violation penalties (SCP). The formulation for the objective function in binary integer programming models was previously given in equation 4.2.106 in Section 4.2.2. This objective function is going to be minimised subject to *hard* constraints. The numbers of *hard* and *soft* constraints for *nott*1, *SVe*150, and *PNe*150 instances were previously given in Table 2.2 (Section 2.4) and Table 2.4 (Section 2.5.3) in Chapter 2 respectively.

In Section 4.6.1, results on *nott*1 and *wolver* datasets are presented. Effects of *slack space rate* (S), *violation rate* (V), *positive* (P), and *negative slack amount* (N) on percentage and absolute gaps in *SVe*150 and *PNe*150 datasets are presented in Sections 4.6.2 and 4.6.3 respectively. Effects of S, V, P, and N on some key ratios in *SVe*150 and *PNe*150 datasets are given in Sections 4.6.4 and 4.6.5 respectively. The integer programming models with and without floor variables are compared in Section 4.6.6. The complete tabular results related to all experiments in this chapter can be found in Appendix A.

In our experiments, each instance was given 30 minutes of running time in single-thread mode. It was possible to run the solver indefinitely until an optimal solution was found for a specific instance. However, our aim was to test these instances under a fixed amount of running time and observe how the gap values between the incumbent solution and the bound as well as some other key ratios in OSA varied in our parametrised instance sets. It was not trivial to make the analysis in Sections 4.6.2, 4.6.3, 4.6.4, and 4.6.5 objectively by just running the solver indefinitely without a fixed amount of running time. This 30 minutes limit is also adhered in a somewhat different setting tailored for the heuristic based algorithms that are going to be proposed in Chapters 5, 6, and 7.

4.6.1   Results on Nott1 and Wolverhampton Datasets

Table 4.2 summarises the best results obtained after a run of 30 minutes on each problem instance (from *nott*1 to *wolver*). Note that these dataset instances do not contain *non-allocation*, *not in same room*, or *capacity* constraints, the other six constraint types are present in these real-world instances.

Two different experiments were run on the largest problem instance *nott*1. It was observed during experiments that minimising *same room* constraint violations was the most difficult, especially for the *nott*1b instance (value of 80.00). So, an additional experiment was conducted for tackling the *same room* constraint in the *nott*1 instance (largest one). In *nott*1 column, *same room* constraints were all set as *soft*, whereas in

	nott1	nott1*	nott1b	nott1c	nott1d	nott1e	wolver
Allocation	40.00	20.00	0.00	40.00	0.00	0.00	0.00
Same Room	0.00	0.00	80.00	0.00	0.00	0.00	0.00
Not Sharing	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Adjacency	10.00	10.00	0.00	10.00	0.00	0.00	0.00
Group by	11.18	22.36	11.18	11.18	11.18	0.00	0.00
Away From	20.00	30.00	0.00	20.00	0.00	40.00	0.00
Constraint Penalty	81.18	82.36	91.18	81.18	11.18	40.00	0.00
Overuse	130.80	106.40	64.20	182.90	164.70	13.80	486.04
Underuse	134.20	122.00	87.90	41.65	26.85	123.90	148.15
Usage Penalty	265.00	228.40	152.10	224.55	191.55	137.70	634.19
Total Penalty	346.18	310.76	243.28	305.73	202.73	177.70	634.19
Lower Bound	201.86	273.16	131.45	305.73	202.73	177.70	634.19
Percentage Gap	%41.70	%12.10	%46.00	%0.00	%0.00	%0.00	%0.00

Table 4.2: Individual penalties for the best results obtained for each problem instance of the *nott1* and *wolver* datasets by using the model without floor variables

column *nott1∗*, *same room* constraints were all set as *hard*. Notice that this latter setup achieved a lower usage penalty by roughly 35 square meters. In all these instances, the constraint penalty turned out to be significantly lower than the usage penalty. For the instances *nott1c*, *nott1d*, *nott1e*, and *wolver*, optimal results were obtained while instances *nott1* and *nott1b* remained very challenging. Note that in *nott1*, *nott1b*, and *nott1c*, the previous best results reported [Landa-Silva and Burke, 2007] in the literature were 482.2, 417.1, and 315.4 respectively. In these three instances, the best results obtained were significantly improved setting new target for the algorithms in following chapters. These new best results were reported in [Ülker and Landa-Silva, 2011].

4.6.2 Effect of S and V on Percentage and Absolute Gaps

The experiments focused on the four basic generator parameters: *slack space rate* (*S*), *violation rate* (*V*), *positive* (*P*) and *negative slack amount* (*N*). The aim was not only to analyse how these parameters affected the difficulty of the problem but also to see the effect on *space misuse (overuse, underuse)* and (*soft constraint penalty*) because these attributes and the ratios between them could be used for designing algorithms which tracked these values during the search.

The visual representation of the experimental results on percentage and absolute gaps between the results and the best bound obtained in *SVe*150 dataset is depicted in Figures 4.2, 4.3, and 4.4. The two axes *S* and *V* represent different values for *slack* and *violation* rates. The *PG* and *AG* axes represent the *percentage* and *absolute gaps* obtained. Each figure represents the results under *half, normal* or *double soft constraint penalty conditions* respectively (as described in Section 4.6).

As evidenced by the *percentage gaps* between the best objective value obtained and the bound, decreasing the penalty values for each *soft* constraint increased the difficulty of the instance (with respect to the proof of optimality). This was usually due to the fact that the total penalty (which was the denominator in the *percentage gap* equation) became lower due to the lower individual penalties for the *soft* constraints. However, it was noticed that for higher values of *S*, the difference between the gaps over different *soft constraint penalty conditions* became lower. This was related to the increase of importance of *space misuse* over the *soft constraint violation penalty* due to the higher values of *S*.

It was observed that increasing *S* and *V* did not necessarily increase the *percentage gaps* smoothly. In fact, some of the highest *percentage gaps* were obtained when *S* was quite low especially in *half* and *normal soft constraint penalty conditions*. This was probably due to the difficulty of reducing an already low space misuse to strictly zero. However, this was not an issue when the space misuse became less important as in *double soft constraint penalty condition*. The *soft* constraint violation rate *V* had a direct effect on the *percentage gaps*. Increasing *V* increased the *percentage gaps* and unlike the case in *S* the increase was more in *half* and *normal soft constraint penalty condition*. One explanation for this could be that increasing the importance of *soft* constraints by doubling the penalty weights forced the solver to focus on minimising the *soft* constraint violations more.

The *absolute gaps* (difference between the best obtained solution and the bound) were related to the *percentage gaps*. There was an increasing trend in this value with high *S* and the increase was even higher with greater *V* rates. Naturally, the increase became steeper in *double soft constraint penalty condition* due to the fact the *total penalty* was higher in this case.

Finally, the analysis focused on finding if increasing or decreasing the *soft constraint penalty* gave better or worst results. It was observed that on the average case, *double soft constraint penalty condition* gave an improvement of $total\,penalty = 4.875$ per instance over *normal soft constraint penalty condition*. There was a further improvement of $total\,penalty = 3.983$ per instance of using *normal* over *half penalty condition*. Thus, it
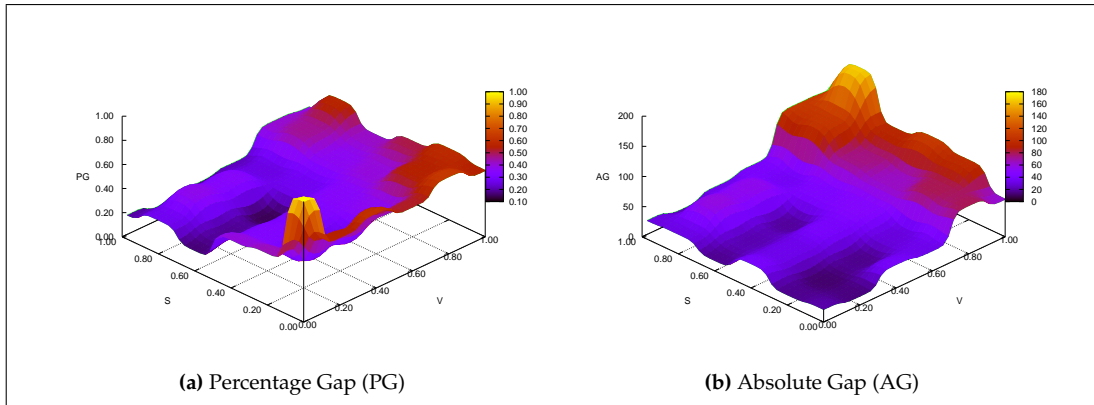
Figure 4.2: Effects of different *S* and *V* on *percentage* and *absolute gaps* under *half soft constraint penalty condition* in *SVe*150 dataset
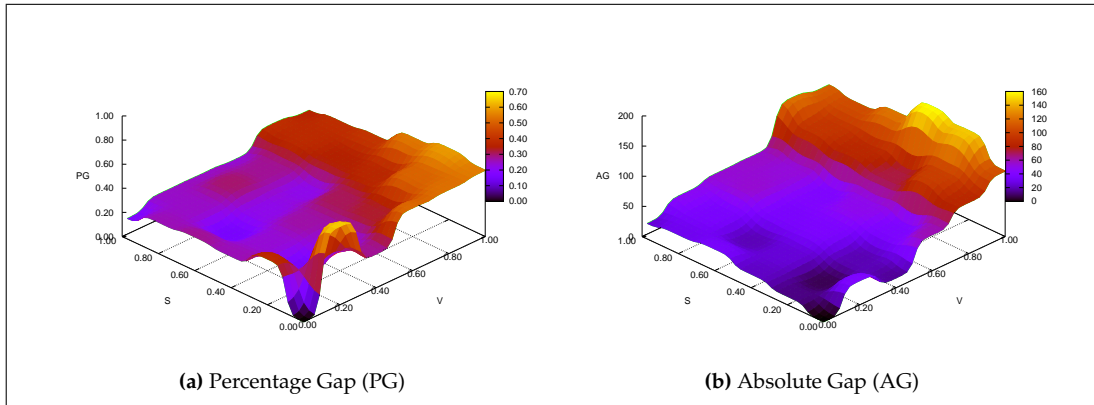
Figure 4.3: Effects of different *S* and *V* on *percentage* and *absolute gaps* under *normal soft constraint penalty condition* in *SVe*150 dataset
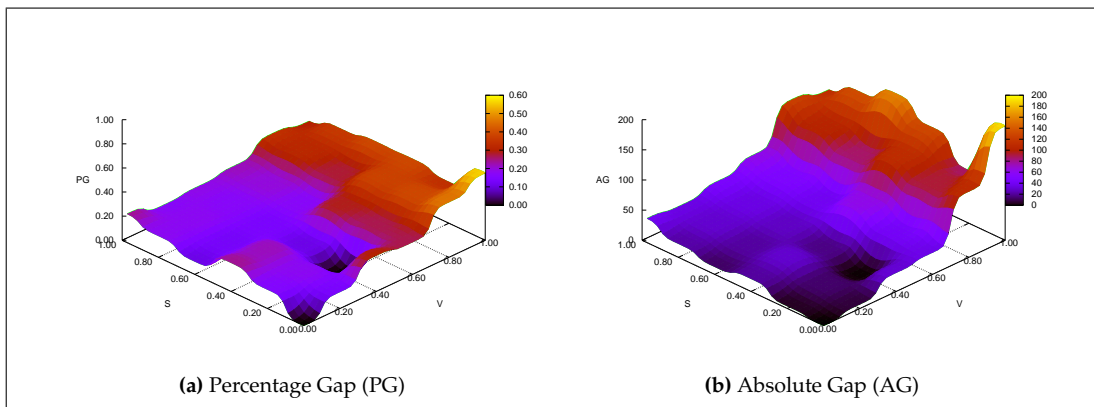
Figure 4.4: Effects of different *S* and *V* on *percentage* and *absolute gaps* under *double soft constraint penalty condition* in *SVe*150 dataset

might be advisable to use higher penalties for *soft* constraints than the ones initially set, solve the problem with these increased *soft* constraint penalties and convert the *total penalty* obtained to a normalised form in the end. However, such an approach may not

be necessary if the space misuse was observed high due to the reasoning that higher S values reduced the importance of *soft constraint penalty* over *space misuse*.

4.6.3   Effect of P and N on Percentage and Absolute Gaps

In Figures 4.5, 4.6, and 4.7, the percentage (PG) and absolute gaps (AG) obtained after experimentation on *PNe*150 dataset under *half, normal*, and *double soft constraint penalty conditions* are presented. The P and N axes stand for *positive* and *negative slack amounts* which adjust the underuse and overuse in an instance. The axes *PG* and *AG* represent the percentage and absolute gaps obtained.

In *PNe*150 tests, a similar scenario as in *SVe*150 was observed. Performing the search with increased *soft constraint penalty* yielded better results: *double soft constraint penalty condition* provided *total penalty* = 4.294 per instance over *normal soft constraint penalty condition*, and a further *total penalty* = 4.269 per instance improvement was observed over *half soft constraint penalty condition*. Again similar to *SVe*150, the *percentage gaps* tended to go higher when the weight of *soft constraint penalty* was reduced.

An important observation was that the difficulty of the instance was not necessarily affected by increasing or decreasing P and N values independently. The percentage gaps were usually highest when P and N were set equal or close to each other, and the percentage gaps usually tended to get lower when the absolute value gap between P and N was increased. The absolute value gap between the objective value, and the bounds exhibited the similar pattern with the percentage gap case. This case was interesting considering the fact that in the current model, *overuse* was penalised more than *underuse* (by a factor of 2) yet that did not affect the P/N ratio (which maximised the percentage gap) in a similar way. The explanation for change in the difficulty of the instance could be attributed to how the solver handled the instance in different *overuse* and *underuse* situations. When P was set low and N was set high, the generator would create an instance with many under-capacitated rooms and not enough rooms with over-capacity to compensate for them. Because of the penalty weight difference between the *overuse* and *underuse*, the solver was expected to allocate whatever space it could allocate (by claiming the very few over capacitated rooms or violating a *soft* constraint immediately) and to reduce the *overuse* of the rooms. On the exact opposite case (when P was high and N was low), the resulting instance would have many rooms with over-capacity and few rooms with under-capacity. In this case, the solver would have more choices to minimise the *overuse* penalty and could also search for a balance between minimising the soft constraint violations and the *underuse* penalty. When N

(a) Percentage Gap (PG) **(b)** Absolute Gap (AG)
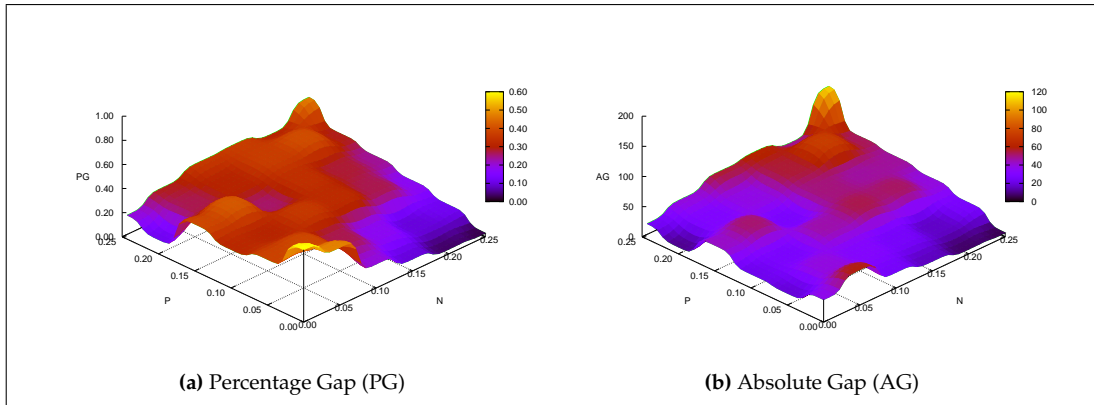
Figure 4.5: Effects of different *P* and *N* on *percentage* and *absolute gaps* under *half soft constraint penalty condition* in *PNe*150 dataset

(a) Percentage Gap (PG) **(b)** Absolute Gap (AG)
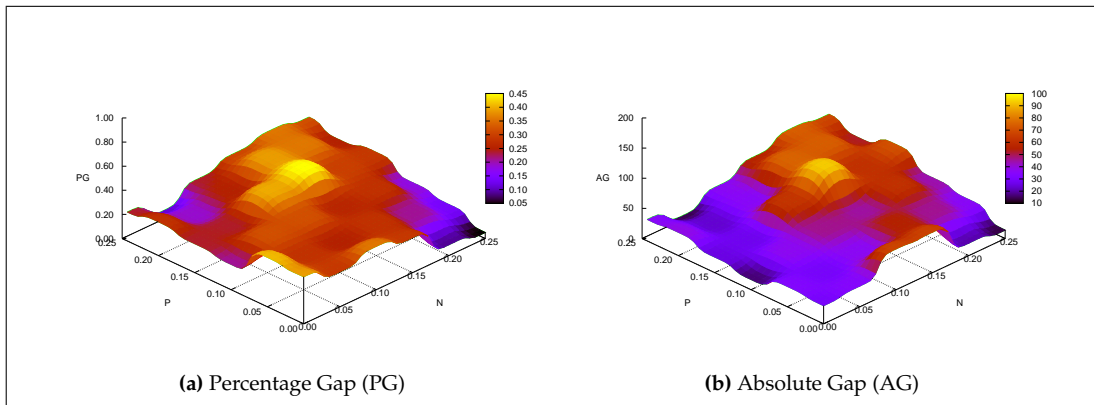
Figure 4.6: Effects of different *P* and *N* on *percentage* and *absolute gaps* under *normal soft constraint penalty condition* in *PNe*150 dataset

(a) Percentage Gap (PG) **(b)** Absolute Gap (AG)
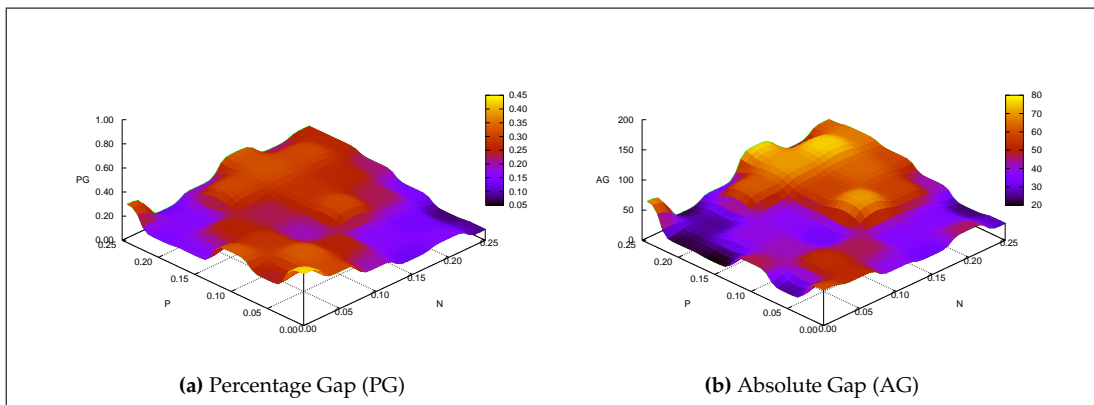
Figure 4.7: Effects of different *P* and *N* on *percentage* and *absolute gaps* under *double soft constraint penalty condition* in *PNe*150 dataset

and *P* were close to each other, the instance was expected to have rooms with over and under capacity, thus the solver would have more choices to minimise *overuse*, *underuse*, or the *soft* constraint violations. This caused instances with higher *percentage gaps*.

97

4.6.4 Effect of S and V on Overuse, Underuse and Soft Constraint Violations

Figures 4.8, 4.9 and 4.10 give a graphical depiction of *O/U* (*overuse / underuse* penalty ratio), and *SCP/SMP* (*soft constraint violation penalty /misuse penalty* ratio) under *half, normal* and *double soft constraint penalty conditions* respectively. We specifically focused on these two ratios to understand how the space misuse (both *overuse* and *underuse*) and constraint penalties were affected under various parameter settings.

It was observed that when *S* (*slack rate*) was set to 0 (corresponding to instances with zero *space misuse*), the IP solver yielded results where the space *overuse* penalties were exactly twice the amount of *underuse* penalties in all *V* (*violation rates*) conditions. This amount coincided with the penalty weight ratio between *overuse* and *underuse* (remember that overusing a room was considered worse than underusing a room, hence the penalty factor of 2). It was observed that increasing *V* decreased the *O/U* ratio. However, the effect of *V* was much less than *S* in this experiment. It was observed that around $S = 0.60, V = 0.00$, the *O/U* ratio was maximised. Increasing the weights for the constraints had little effect on the *O/U* ratio. The *SCP/SMP* ratio was primarily (and naturally) affected by increasing the weights for each constraint. Under *half, normal* and *double constraint penalty conditions*, the gradual increase on the *SCP/SMP* can be observed in Figures 4.8(b), 4.9(b), and 4.10(b). It was observed that decreasing the *S* parameter (decreasing the *space misuse*) and setting it around $S = 0$ (zero *space misuse*) maximised this ratio. Under various settings, the ratio was maximised around medium *V* values (around $V = 0.2$ and $V = 0.6$).

Notice that these instances in the *SVe*150 dataset were designed to have equal amount of space *overuse* and space *misuse*. In ideal conditions during an algorithmic search, we would try to keep the ratio *O/U* close to 2 (because of the weight of the overuse = 2). Since *V* has barely any effect in this scenario, whenever the ratio goes significantly above 2, it might be beneficial to just concentrate on reducing each and every component of *space misuse* (whether it is *overuse* and *overuse*) rather than focusing on reducing the *soft* constraint violation penalty.

4.6.5 Effect of P and N on Overuse, Underuse and Soft Constraint Violations

Figures 4.11, 4.12, and 4.13 give a graphical depiction of *O/U* (*overuse / underuse penalty* ratio), and *SCP/SMP* (*soft constraint penalty / space misuse penalty* ratio) under *half, normal,* and *double soft constraint penalty conditions* respectively.

It was observed that when the $N - P$ difference was increased (which meant that

(a) Overuse / Underuse Penalty (O/U)   **(b)** Constraint / Space Misuse Penalty (SCP/SMP)
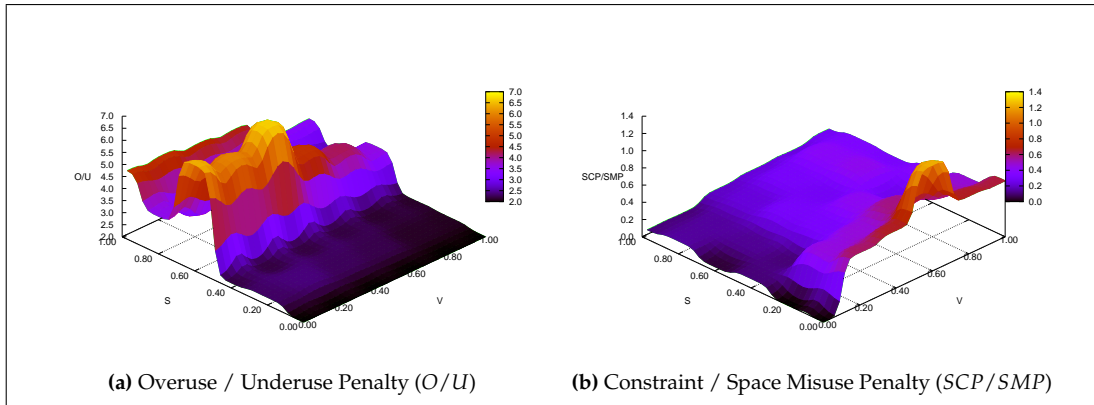
Figure 4.8: Effects of different S and V on *overuse/underuse penalty* and *soft constraint/space misuse penalty* under *half soft constraint penalty condition* in $SVe150$ dataset

(a) Overuse / Underuse Penalty (O/U)   **(b)** Constraint / Space Misuse Penalty (SCP/SMP)
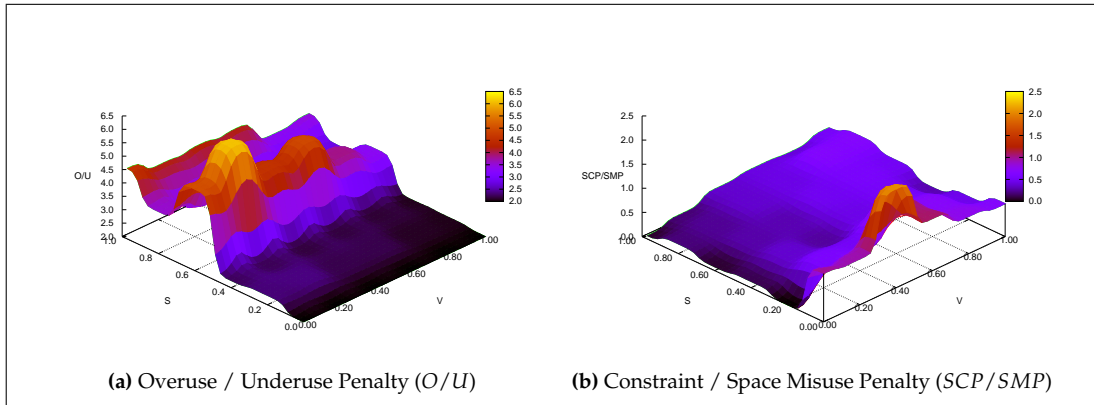
Figure 4.9: Effects of different S and V on *overuse/underuse penalty* and *soft constraint/space misuse penalty* under *normal soft constraint penalty condition* in $SVe150$ dataset

(a) Overuse / Underuse Penalty (O/U)   **(b)** Constraint / Space Misuse Penalty (SCP/SMP)
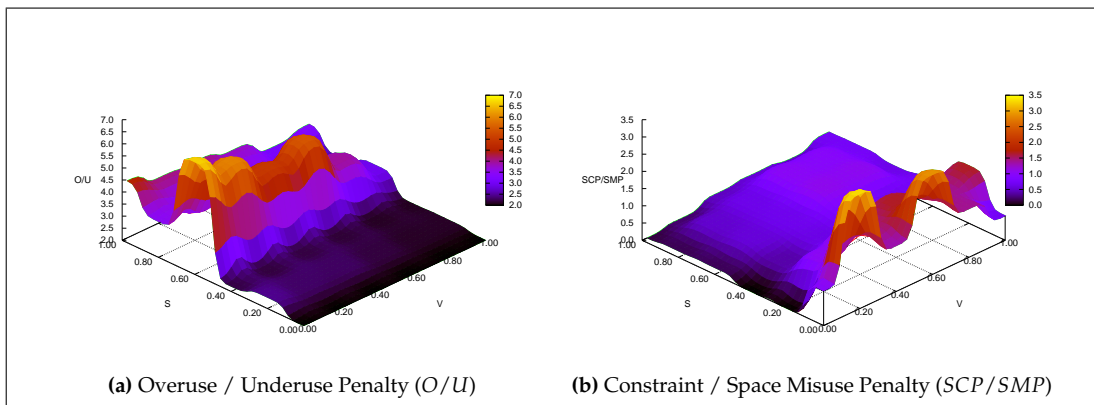
Figure 4.10: Effects of different S and V on *overuse/underuse penalty* and *soft constraint/space misuse penalty* under *double soft constraint penalty condition* in $SVe150$ dataset.

the instance was created with the intent of having very little to no *underuse* and very heavy *overuse*, the O/U ratio showed a very significant increase with a very quick spike. This showed the difficulty of controlling the space *overuse* when there was not enough

(a) Overuse / Underuse Penalty (O/U) **(b)** Constraint / Space Misuse Penalty (SCP/SMP)
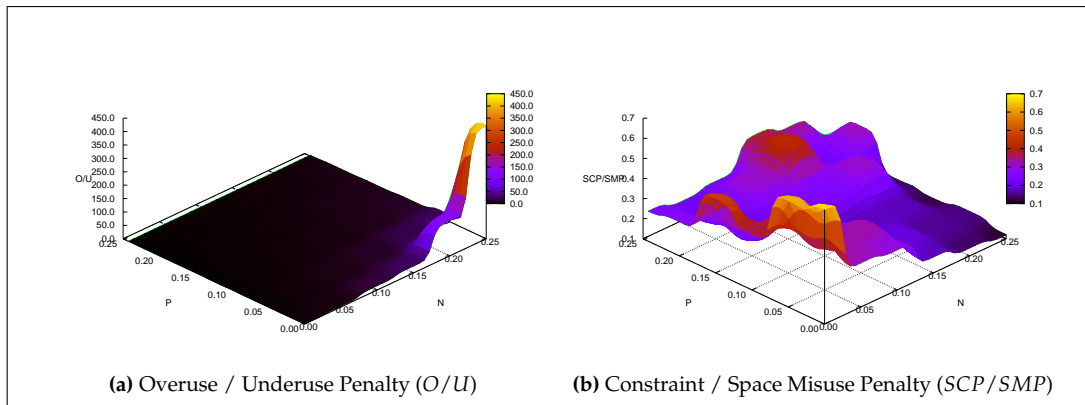
Figure 4.11: Effects of different P and N on *overuse / underuse penalty* and *soft constraint / space misuse penalty* under *half soft constraint penalty condition* in $PNe150$ dataset.

(a) Overuse / Underuse Penalty (O/U) **(b)** Constraint / Space Misuse Penalty (SCP/SMP)
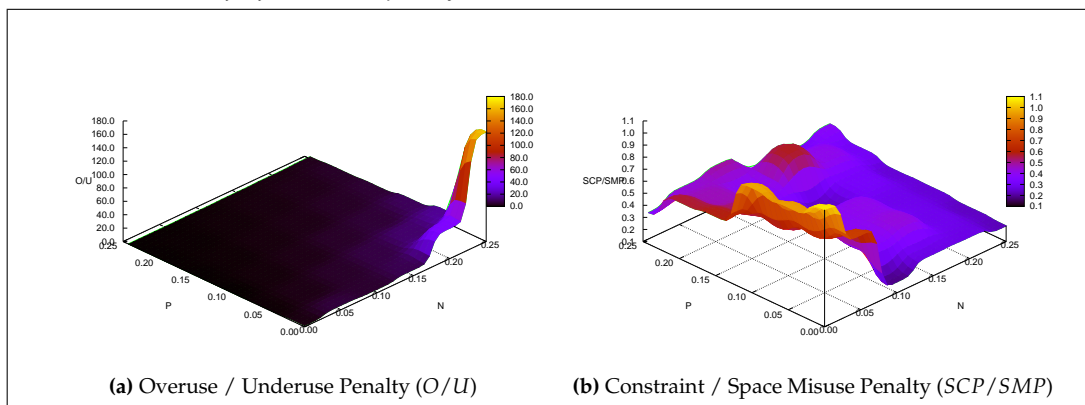
Figure 4.12: Effects of different P and N on *overuse / underuse penalty* and *soft constraint / space misuse penalty* under *normal soft constraint penalty condition* in $PNe150$ dataset.

(a) Overuse / Underuse Penalty (O/U) **(b)** Constraint / Space Misuse Penalty (SCP/SMP)
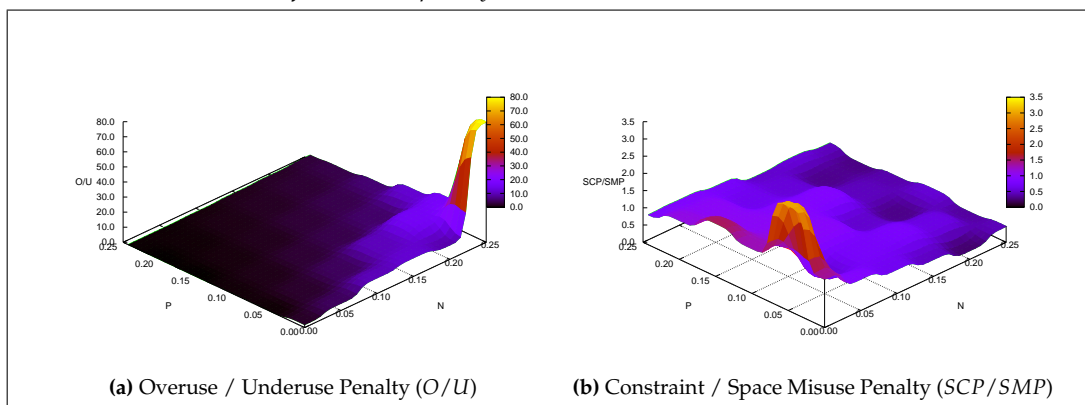
Figure 4.13: Effects of different P and N on *overuse / underuse penalty* and *soft constraint / space misuse penalty* under *double soft constraint penalty condition* in $PNe150$ dataset.

rooms with sufficient space *underuse* to compensate for the rooms with overuse. When the number of rooms with excess space was very low and the number of overused rooms was very high, the IP solver immediately used up the available capacity and

the O/U ratio immediately shot up in this extreme situation. However, these instances were very easy to solve. In design of future algorithms, somewhat counter-intuitively, it may not be desirable to just aggressively reduce the *overuse* when a very high overuse of some room space and very low usage of some room space are observed over a period of time in the search. In this case, the algorithm might have already allocated the very few extra space it has; therefore, it might be beneficial to try to reduce the *soft* constraint violations in this case.

SCP/SMP ratio was maximised when both P and N were set very close to each other (hence very little *space misuse* in this case). In this case, it might be beneficial to design algorithms which try to lower the *soft* constraint violation when individual space *overuse* and *underuse* are observed to be close to each other and SCP/SMP is higher. It was observed that the weight of the *soft* constraint penalty also had an effect on both ratios. O/U tended to get lower and SCP/SMP tended to go higher in *double soft constraint penalty condition* probably due to the IP solver trying to force the *soft* constraints to be satisfied in this case.

4.6.6 Comparison of Models with and without Floor Variables

In this section, the models with and without the floor variables are compared. Both models were given 30 minutes of run-time on a Intel Core 2 Duo E8400 processor and the tests were done on complete $SVe150$ and $PNe150$ datasets under *normal soft constraint penalty condition*.

Figures 4.14 and 4.15 depict the difference in best objective function value obtained and the best bound on it in $SVe150$ and $PNe150$ datasets respectively. The model without and with the floor variables are abbreviated as IP_1 and IP_2 respectively. In the following figures, IP_1 is better in blue regions while IP_2 is better in green regions. As it is discussed next, the performance of the model with the floor variables is significantly superior to the one without these variables.

For all but two instances ($S_{0.20}V_{0.00}$ and $S_{1.00}V_{0.00}$), the model with the floor variables yielded better or at least equal results. As can be seen from Figure 4.14a, the difference between the model increased as the V rate increased. This was expected due to the fact that with higher rates of V, higher amounts of conflicting *nearby* and *away from* constraints would be present in the instances. The differences peaked around mid S and high V rates, the model without the floor variables still provided good results for instances with low *space misuse* and *soft constraint violation* penalties. The difference in best bound obtained in $SVe150$ dataset is depicted in Figure 4.14b. Using floor variables

(a) Δ_{TP} of $IP_1 - IP_2$

(b) Δ_{BND} of $IP_1 - IP_2$

(c) Δ_{SMP} of $IP_1 - IP_2$

(d) Δ_{SCP} of $IP_1 - IP_2$
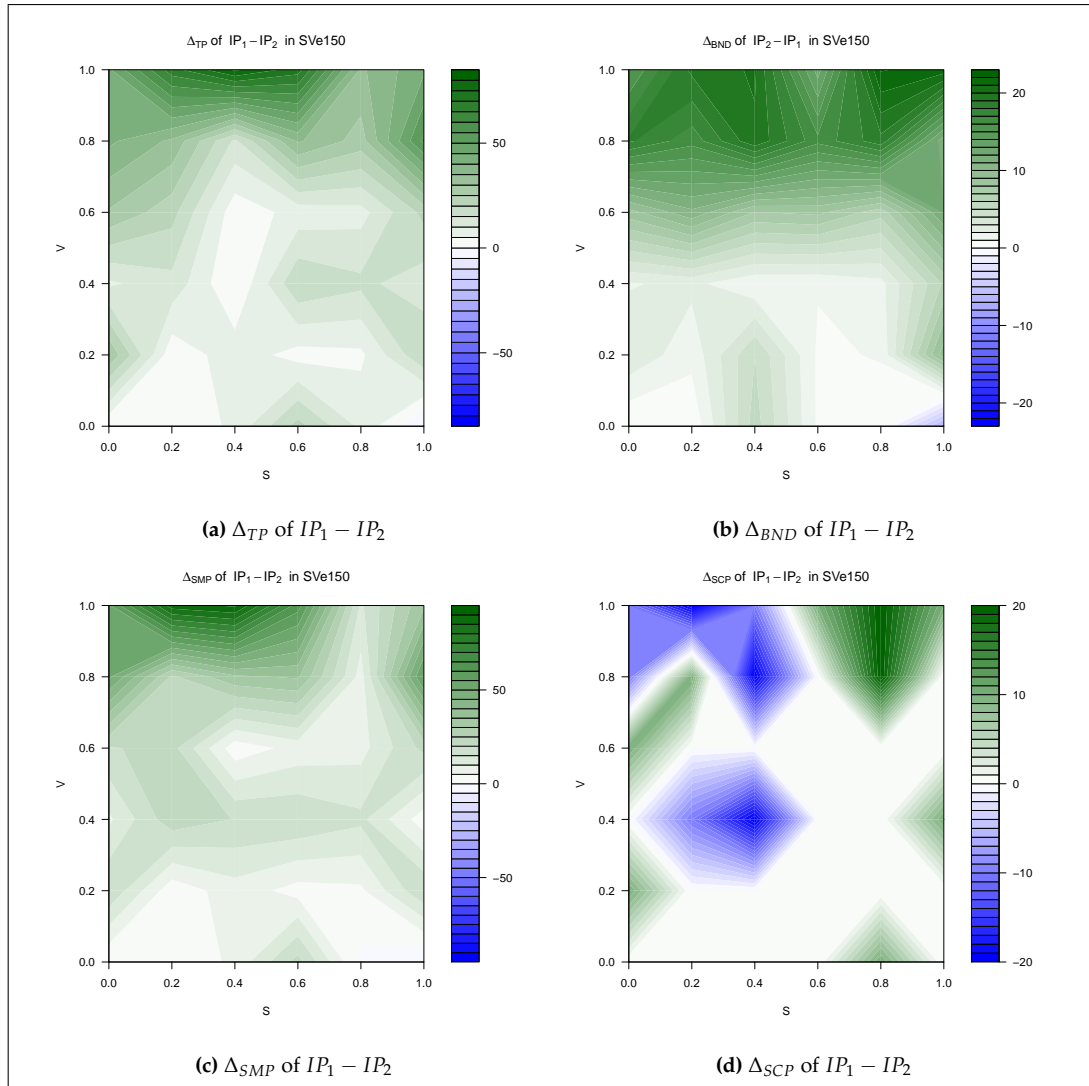
Figure 4.14: Differences in total cost penalty (Δ_{TP}), best bound (Δ_{BND}), space misuse penalty (Δ_{SMP}), and soft constraint penalty (Δ_{SCP}) after applying model without the floor variables (IP_1) and model with the floor variables (IP_2) on $SVe150$ dataset ($IP_1 - IP_2$). IP_1 is better in blue regions while IP_2 is better in green regions.

improved the best bounds obtained as well, and similar to the best objective function value case, most performance increases were observed around high V values.

In Figure 4.15a the differences of best objective function value obtained by two models in $PNe150$ dataset are depicted. Using floor variables provided better results for all instances, the performance gain was most prominent along the region where $|N - P|$ was maximised where the model without the floor variables struggled the most. The best bounds were significantly improved along this diagonal as well (as seen in Figure 4.15b). However, the best bounds improved very slightly alongside both P and N axis over the model without floor variables.

(a) Δ_{TP} of $IP_1 - IP_2$

(b) Δ_{BND} of $IP_1 - IP_2$

(c) Δ_{SMP} of $IP_1 - IP_2$

(d) Δ_{SCP} of $IP_1 - IP_2$
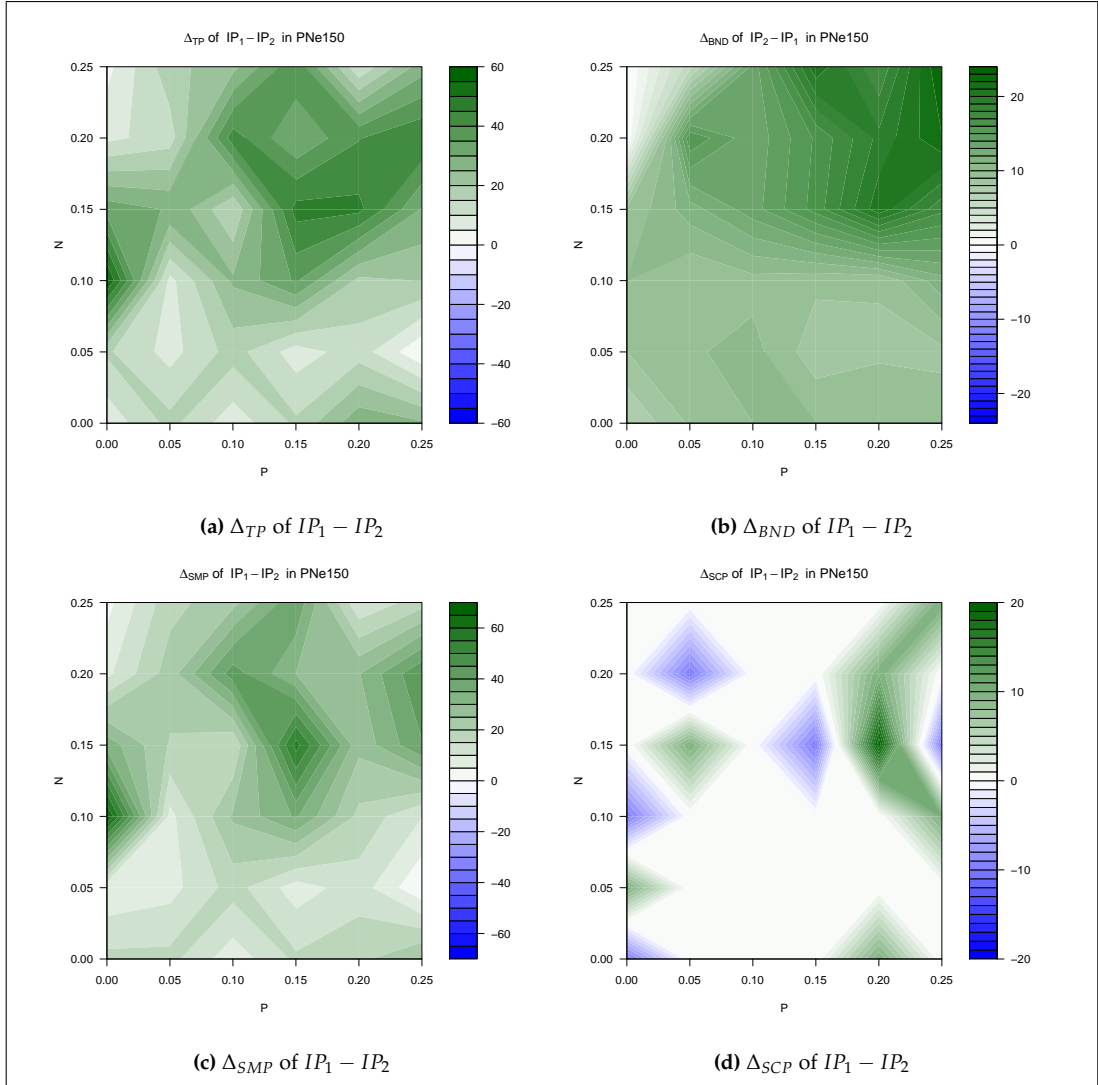
Figure 4.15: Differences in total cost penalty (Δ_{TP}), best bound (Δ_{BND}), space misuse penalty (Δ_{SMP}), and soft constraint penalty (Δ_{SCP}) after applying model without the floor variables (IP_1) and model with the floor variables (IP_2) on $PNe150$ dataset ($IP_1 - IP_2$). IP_1 is better in blue regions while IP_2 is better in green regions.

In Figures 4.14c and 4.14d, the differences of *space misuse* and *soft constraint violation* penalties on $SVE150$ dataset using both models are depicted. It was observed that the performance gain was mostly due to superior handling of the *space misuse*. In fact, for a significant number of instances, the *soft constraint violation* did not improve at all. It should be also noted that in instances with higher expected *space misuse* due to large S rates, the improvements in *space misuse* were not as significant. In fact, the performance of reducing the *space misuse* improved significantly in instances where higher *soft constraint violations* were expected due to large V rates. One possible explanation to this behaviour could be that since the model with the floor variables was already efficient

in handling the large number of *nearby* and *away from* constraints, it could spend more time in minimising the *space misuse components* especially by trading one or two such constraint violations for better space misuse.

In Figures 4.15c and 4.15d, the difference between two models in reducing the *space misuse* and *soft constraint violation* penalties in $PNe150$ dataset is depicted. Similar to the case as in $SVe150$, most performance gains were observed in *space misuse* component, especially around some of the regions along the PN diagonal. The handling of *soft constraint violation* penalties was usually improved around high P values where underuse of rooms was expected. However, in low P rates, it could be more difficult to reduce the *soft constraint violation* over the model without the floor variables.

4.7 Conclusion

In this chapter, two binary integer programming formulations designed for the office space allocation problem were analysed. The analysis was based upon the experimental results on parametrised instances created by the instance generator as described in Section 2.5. Four different parameters for handling *space misuse*, *space overuse*, *space underuse* and *soft constraint violation penalty* were developed during the implementation of this generator.

The experiments focused on the effects of changing the weight for *soft constraint penalty* on two different instances. On the average case, it was observed that increasing the weight of the *soft constraint penalty* improved the results. The improvement was more notable in cases when the *space misuse* was expected to be high. In general, it was observed that reducing an already very low *space misuse* was the greatest hurdle the models struggled with. The instances were actually most difficult to solve when the expected *space overuse* and *underuse* were close to each other and *soft constraint violation penalty* was expected to be high. We also made some preliminary observations on some of the cases where it was desirable to prioritise the *space misuse* components or *soft constraint violation penalty*.

In this chapter, the binary integer programming formulations with and without the floor variables were also compared. It was observed that the model with the floor variables yielded a far simpler simplex table to work on after the pre-solve stage in CPLEX. Significant performance improvements were observed over the model without the floor variables.

In the preliminary research, it was observed that the order the constraints were

handled was instrumental in designing difficult data instances. Although this chapter includes a study of the cumulative effect of *soft constraint violations*, a constraint by constraint study of each nine constraints may be planned for future work.

CHAPTER 5

Local Search Algorithms

5.1 Introduction

In the literature, there are various heuristic approaches already implemented for handling the OSA problem such as [Burke et al., 2001c], [Burke et al., 2001a], [Landa-Silva, 2003], [Landa-Silva and Burke, 2007], [Pereira et al., 2010], and [Lopes and Girimonte, 2010]. This chapter is devoted to the implementation of a general local search heuristic which mostly uses stochastic components. The goal of this chapter is to investigate heuristic algorithms that are easy to implement and that do not depend upon fine-tuned implementations of constraint handling and objective value calculations (which is going to be covered in Chapter 6 in more detail). The algorithms in this chapter are based upon an iterated local search [Lourenco et al., 2002] (ILS) procedure at its core. Solutions are perturbed and improved using a local search method at each step of the algorithm. Several methods for acceptance and rejection of new solutions at the end of each stage of ILS are investigated. These methods include descend methods, simulated annealing [Kirkpatrick et al., 1983], threshold acceptance [Dueck and Scheuer, 1990], and great deluge algorithm [Dueck, 1993]. Initial tests are conducted in order to determine the balance between *relocate* and *swap* moves to explore the neighbourhood of a given solution. Experiments are also conducted if it is beneficial to use the aforementioned acceptance and rejection methods.

In this chapter, the main aim is an initial investigation of local search techniques that are similar in nature to the ones previously proposed in the literature such as [Landa-Silva, 2003], [Pereira et al., 2010], and [Lopes and Girimonte, 2010]. Given the new datasets and new understanding of the problem from Chapter 4, this chapter will serve as a starting point of heuristic development for OSA in this thesis. Some of the ideas presented in this chapter will be revisited and improved in Chapters 6 and 7.

This chapter is organised as follows: Section 5.2 describes the solution representation and data structures used in designing the heuristics not only in this chapter but also in Chapters 6 and 7. Section 5.3 describes the move operators and the neighbourhood structures considered in this chapter. Section 5.4 describes the general framework of the local search algorithm. Section 5.5 describes a procedure for quickly calculating the cost function changes for a random *relocate* or *swap* move. The experiments and discussions on the results are presented in Section 5.6. Finally, the conclusions of this chapter are given in Section 5.7.

5.2 Solution Representation and Data Structures

There are mainly three objects in a typical office space allocation problem: *entity*, *room* and *constraint* objects which hold the information related to each entity, room, and constraint respectively. An OSA problem can be formalised and implemented by using lists of entity, room, and constraint objects, and the relationships between them. An example diagram which depicts such relationships between these three types of objects is given in Figure 5.1.
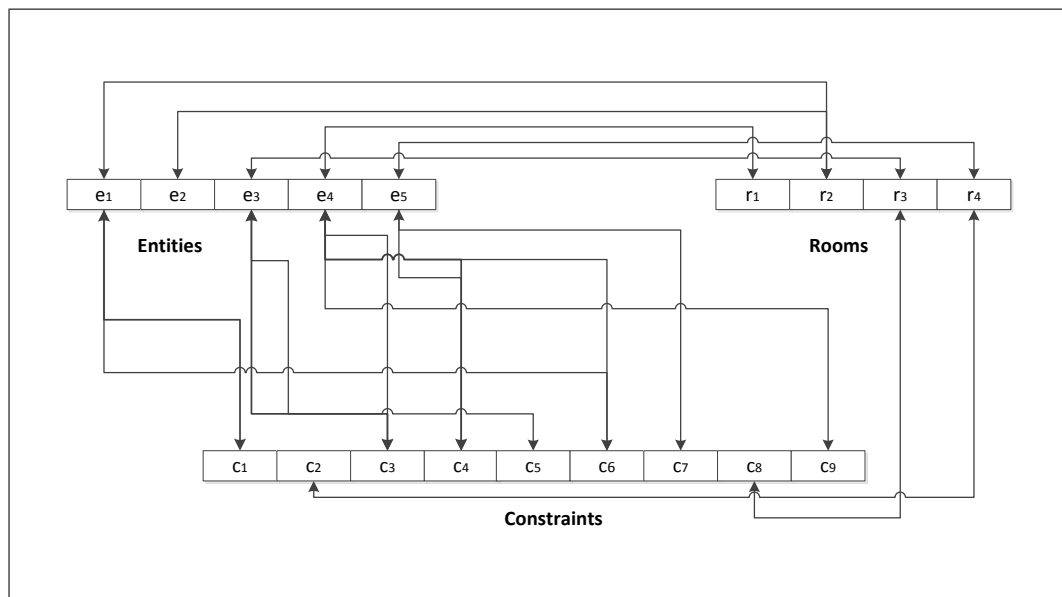
Figure 5.1: Relationships between entity, room and constraint objects

Entity Object: An entity object contains information related to a single entity.

- *Size*: The storage space requirement for the entity.

- *Room*: the room the entity is currently placed.

- *Group*: The organisational group the entity belongs to. Used for *nearby* constraint checking.

- *Links of constraints*: Each entity contains a list of links which point to each individual *hard* and *soft* constraint of each type associated with this entity.

Room Object: A room object contains information related to a single room.

- *Size*: The storage space capacity of the room.

- *Floor*: The floor the room is in. It is used for *nearby* and *away from* cost calculations.

- *Space misuse*: The current overuse, underuse, and total space misuse are kept to facilitate the space misuse calculations.

- *List of entities*: The entities currently placed into the room and specified in a list form.

- *Not sharing entities*: The entities that have *not sharing constraints* and placed into this room are marked to facilitate the *not sharing* constraint checking.

- *List of constraints*: Each *capacity* constraint (the only type of constraint considered that applies exclusively for rooms) associated with the room.

Constraint Object: Each constraint object holds information related to the constraint such as:

- *entity*: For *allocation*, *non-allocation*, and *not sharing* constraints.

- *entity-entity*: For binary constraints (*same room*, *not same-room*, *adjacency*, *nearby*, and *away from*) that involve two entities.

- *room*: For *capacity* constraints.

Entity and *room* type of constraints (first and third types described above) are unary constraints while *entity-entity* types of constraints are binary.

We decided on this representation scheme with these three object structures. The relationships (links) between these objects are represented by pointers in C++. This scheme enables quick access to several features such as the contents of the rooms (the

entities placed into specific rooms), the placement of entities, and the constraints associated with each room and entity. This representation also helps to the efficient implementation of objective function value calculations which are going to be described in Section 5.5 and 6.4.3.

5.3 Neighbourhood Operators

There are three common move operators designed to create the neighbourhoods in OSA research [Burke et al., 2001c], [Burke et al., 2001a], [Landa-Silva, 2003], and [Landa-Silva and Burke, 2007]. These three move operations are as follows:

- *Relocate*: This operator moves one entity from one room to another. This move can also be referred to as *simple move* or *single move* in the literature.

- *Swap*: This operator swaps the allocation of two entities that have been placed in two different rooms.

- *Interchange*: This operator selects two rooms and swaps the entities between these two rooms.

Each move operation has to decide entities and room that will be associated with the operation. The *relocate* move has to decide which entity should be selected initially and which room it should be placed into later. The *swap* move has to decide which two rooms and which two entities in these rooms should be selected for this operation. Alternatively, just two entities can be selected without taking the rooms into consideration. This approach cannot guarantee that two entities are going to be swapped because the selected entities can already be in the same room. Hence, in order to force the swap, the former approach of selecting the rooms first and then selecting two entities later is chosen for the implemented *swap* move.

Similarly, *interchange* move has to decide which two rooms should be selected for the operation. In [Landa-Silva, 2003], different versions of these operators were investigated. These versions differ in terms of the level of greediness or stochasticity of the selection and placement of an entity. The entities can be selected based upon how much constraint violation they are associated with. The decision to select the rooms can be made based upon the space misuse of each room. Conversely, the entity or room selection can be made completely randomly as well which is the approach taken in this chapter.

In this thesis, the main focus will be on the *relocate* operator. The *relocate* operator is the most basic and atomic move operation in OSA. Any complex entity-room re-allocations can be formalised via serialisations of *relocate* moves. The move allows efficient cost calculation procedures both in deciding which entity should be selected for re-allocation and which room the entity should be placed. In most cases during the coding of the algorithm, it was observed that numerous, easy to implement, fast *relocate* moves were preferred over a few complex, hard to implement, slow move operations. A random *swap* operator can easily be implemented using two successive *relocate* operations. Note that this chapter is mainly devoted to the implementations of stochastic *relocate* and *swap* moves.

In this thesis, the *interchange* operator is completely ignored in designing local search algorithms. This decision was made after the design of the mathematical programming models described in Chapter 4 and after analysing the (optimal or near-optimal) solutions for the instances in *nott*1, *SVe*150, and *PNe*150 datasets. It was observed that for the majority of the instances, the final solution was an entity-room mapping where a room was occupied by a single entity. This was mainly due to the sheer number of *not sharing* constraints in the instances. In such cases, *interchange* operator became practically superfluous because two rooms selected for the operation would most likely contain only a single entity. In this case, *interchange* operator would practically be reduced to a *swap* operator instead.

Practically, the main benefit of using an *interchange* operator in most OSA instances will be causing disruptions or corrections in very few rooms; however, such disruptions and corrections can be implemented by a series of straightforward single-point random perturbation (randomly changing the room of an entity) and *relocate* moves.

Neighbourhood: Consequently, the neighbourhood in this algorithm is formed as follows: Given a current solution, first, which type of move is going to be performed is probabilistically decided. The parameter for this selection is *swap rate* which determines the probability that a *swap* move is performed. Naturally, the probability for a *relocate* move is 1 - *swap rate*. Based on the probabilistic selection, a random *relocate* or *swap* move is applied to the current solution to create the new candidate solution. The balance between *relocate* and *swap* move is going to be tested in Section 5.6.1.

5.4 Algorithm Outline

The local search algorithm used in this chapter can be thought of a variation of *iterated local search* [Lourenco et al., 2002] and *ruin and recreate* [Schrimpf et al., 2000]. The algorithm operates on a single solution; however, it keeps track of the best encountered solution during the search. The local search framework depicted in Figure 5.2 will be used throughout this chapter. It also serves a starting point for the evolutionary local search algorithm that is going to be described in Chapter 6.

Input: input file with entities, rooms, and constraints.
Output: solution (an entity-room mapping)
1: x_{best} = Initial solution
2: $x_{start} = x_{best}$
3: **repeat**
4:    $x' \leftarrow x_{start}$
5:    Apply random perturbation to the current best solution x'
6:    **repeat**
7:       Random decision on *relocate* or *swap* move.
8:       Select the entity and rooms for *relocate* or *swap moves*
9:       $x* \leftarrow$ Apply *relocate* or *swap* move on x'
10:       **if** objective function value $obj(x*)$ is better than $obj(x')$ **then**
11:          $x' \leftarrow x*$
12:       **else**
13:          Reject $x*$
14:    **until** Local search step limit in one stage is reached
15:    **if** $obj(x')$ is better than $obj(x_{best})$ **then**
16:       $x_{best} \leftarrow x'$
17:    **if** Acceptance criterion is met according to the local search acceptance method **then**
18:       $x_{start} \leftarrow x'$
19:    **else**
20:       $x_{start} = x_{best}$
21: **until** Time limit is reached

Figure 5.2: Local search (*LS*) algorithm

Initialisation: In order to create the initial solution, first the entities that have *allocation* constraints associated with them are processed. These entities are placed into the respective rooms indicated by *allocation* constraints. Other entities are placed randomly to the available rooms.

Local Search: Each local search stage works on the currently best solution x_{best} en-
countered during the search. At the start of each local search stage, the best solution
x_{best} is copied to the current solution x' where local modifications are performed. Then,
a random perturbation is applied to x'. The size of this perturbation is small, usually
around one to five percent of the total size of the solution (which is taken as the num-
ber of entities). The chosen entities during the perturbation operation are placed into
other rooms randomly. This disruption operator is the first step of introducing new
information and providing diversification to the algorithm.

After the random perturbation, a large number of local search move operations is
performed. The neighbourhood of the current solution is searched by using the *relocate*
and *swap* operators. Any *relocate* or *swap* move that leads to a better solution than the
current one (in terms of objective function value) is accepted. Otherwise, these moves
are rejected and the search continues with the previous best solution.

Acceptance/Rejection of New Solutions: At the end of each local search stage, a
decision is made regarding whether the algorithm should continue operating on the
global best solution x_{best} encountered during the whole search or on the local best so-
lution x' created during that solution. In the most frequent case, the local best solution
x' in that stage will not improve on the global best solution x_{best}, hence the algorithm
has to decide whether to accept or reject this current solution. This step is extremely
crucial and often neglected in previous OSA heuristic algorithm designs like [Landa-
Silva, 2003] and [Landa-Silva and Burke, 2007]. It is observed that the algorithm can
quickly stagnate if it accepts the local best solution encountered during the most recent
local search stage over the global best solution and does not perform frequent returns
back to the global best solution x_{best}.

Several accept-reject mechanisms are tested in order to determine how to proceed
with the usual case when the best solution x' after each local search stage is worse than
the global best solution x_{best}. In this case, the whole sequence of operations within a
local search stage can be taken as a move which leads from the global best solution x_{best}
to x'. Then, x' can be accepted or rejected by using different methods. The methods
implemented are as follows:

- *Only improving moves*: Solution x' is accepted if and only if it leads to a better
  objective function value than solution x_{best}. Any non-improving or equal solution
  is rejected.

- *Better or equal*: Solution x' is accepted if and only if it leads to better than or equal

to solution x_{best} in terms of objective function value. Any non-improving move is rejected.

- *Threshold acceptance* [Dueck and Scheuer, 1990]: Solution x' is accepted if the difference between x' and x_{best} is within a threshold value t. The solution x' is rejected if such difference is beyond this threshold value.

- *Simulated Annealing* [Kirkpatrick et al., 1983]: Solution x' is accepted probabilistically according to the following function in equation 5.4.1:

$$p = e^{-\frac{obj(x') - obj(x_{best})}{T}} \tag{5.4.1}$$

where T is the current *temperature* value. If the randomly chosen decimal value from the range $(0, 1)$ is less than equal to p, x' is accepted, otherwise it is rejected.

- *Great Deluge* [Dueck, 1993]: Solution x' is accepted if the objective function value of the new solution is below a *water-level* value. Solution x' is rejected if the objective function value of the new solution is above the current *water-level*.

In *threshold acceptance*, *simulated annealing*, and *great deluge* acceptance-reject methods, there is an associated parameter value (threshold t, temperature T, and water level w respectively). These values are not static, they change over time. They are initialised to higher values first to enable a large number of non-improving moves being accepted during the starting stages of the algorithm. These values are lowered gradually during the algorithm. This limits the amount of uphill (non-improving) moves. The decreases Δ_t, Δ_T, and Δ_w on threshold, temperature, and the water-level are made whenever a new best solution is encountered in the solution. As long as the algorithm can find better solutions, the threshold, temperature, or water-levels will eventually reach zero; in this case, the algorithm is going to accept only improving moves.

5.5 Fast Cost Calculation for *Relocate* and *Swap* Moves

In this section, a cost calculation procedure for *relocate* and *swap* moves is proposed. In many optimisation problems, the cost function calculation process (the measurement of the objective function value of a given solution) is the most time consuming stage. In the literature, there are various *delta evaluation* methods for other problems to calculate the Δ (difference) between the objective function values of two successive solutions such as in [Ross et al., 1994]. Although cost change procedures for *relocate* and *swap* moves were proposed in [Landa-Silva, 2003] and [Landa-Silva and Burke, 2007], these

give *approximate* change in the cost calculation. In this chapter, an *exact* cost change procedure for these moves is described. This procedure is going to be referred to as Delta (Δ) stage from now on.

This Δ procedure is similar to the one implemented in [Landa-Silva, 2003] but is adjusted to new constraints and representation of objects and move implementation of local search in this thesis. There is going to be a more complex and elaborate cost update procedure in Section 6.4.3 of Chapter 6 that is built upon this implementation. The objective function for total penalty (TP) is taken as the weighted summation of space misuse (SMP) and soft constraint violation penalties (SCP). The respective formulations for SMP, SCP, and TP were previously given in equations 2.3.1, 2.3.2 and 2.3.3 in Section 2.3.2. The numbers of *hard* and *soft* constraints for *nott1*, *SVe*150, and *PNe*150 instances were previously given in Table 2.2 (Section 2.4) and Table 2.4 (Section 2.5.3) in Chapter 2 respectively. This definition of the objective function is also going to be used for the rest of the meta-heuristics developed in Chapters 6 and 7.

The Delta procedure for the *relocate* move that sends entity e from r_1 to r_2 goes through all the constraints associated with e, r_1, and r_2 and calculates the cost change δ in total objective function.

The *swap* move select two entities e_1 and e_2 in two different rooms r_1 and r_2 and reverses the allocations. A random *swap* move can be implemented using two successive *relocate* moves: (e_1, r_1, r_2) followed by (e_2, r_2, r_1). In this case, first the delta procedure for the *relocate* move (e_1, r_1, r_2) is performed to calculate the cost change δ_{e_1, r_1, r_2}. The entity e_1 is placed into room r_2. Only then, the delta procedure for the *relocate* move (e_2, r_2, r_1) is performed to calculate δ_{e_2, r_2, r_1}. The total cost change then becomes $\delta \leftarrow \delta_{e_1, r_1, r_2} + \delta_{e_2, r_2, r_1}$.

Given a *relocate* move (e, r_1, r_2), the value δ (the change in objective function value) is calculated according to the constraints associated with e. If entity e has one of the unary constraints or it has a binary constraint with another entity f, the rooms associated with e and f have to checked before and after the *relocate* move. The δ calculation also has to check the entities in r_1 and r_2 for *not sharing* and *capacity* constraints.

Allocation and Non-Allocation Constraints: Room r_a is the room (specified in the respective constraint) that entity e has to be allocated to.

- If room $r_1 = r_a$ and $r_2 \neq r_a$, increase δ with *penalty$_{al}$*.

- If room $r_2 = r_a$ and $r_1 \neq r_a$, decrease δ with *penalty$_{al}$*.

Non-allocation is the opposite case of *allocation*, so reverse the conditions.

Same Room and Not Same Room Constraints: Room r_f is the room the second entity f (which is specified in the respective constraint definition) is allocated to.

- If room $r_1 = r_f$ and $r_2 \neq r_f$, increase δ with *penalty$_{sr}$*.

- If room $r_2 = r_f$ and $r_1 \neq r_f$, decrease δ with *penalty$_{sr}$*.

Not same room is the opposite case of *same room*, so reverse the conditions.

Input: Move (e, r_1, r_2)
Output: δ_{nsh}
1: $\delta_{nsh} \leftarrow 0$
2: **if** entity e does not have a *not sharing* constraint **then**
3:    **if** room r_1 has 1 entity with *not sharing* constraint AND has 2 entities **then**
4:        $\delta_{nsh} \leftarrow \delta_{nsh} - penalty_{nsh}$
5:    **if** room r_2 has 1 entity with *not sharing* constraint AND has 1 entity **then**
6:        $\delta_{nsh} \leftarrow \delta_{nsh} + penalty_{nsh}$
7: **else**
8:    **if** room r_1 has more than 2 entities **then**
9:        $\delta_{nsh} \leftarrow \delta_{nsh} - penalty_{nsh}$
10:       **if** room r_1 has 2 entities each with *not sharing* constraint **then**
11:           $\delta_{nsh} \leftarrow \delta_{nsh} - penalty_{nsh}$
12:    **if** room r_2 has more than 1 entity **then**
13:        $\delta_{nsh} \leftarrow \delta_{nsh} + penalty_{nsh}$
14:       **if** room r_2 contains just single entity with *not sharing* constraint **then**
15:           $\delta_{nsh} \leftarrow \delta_{nsh} + penalty_{nsh}$

Figure 5.3: Delta algorithm for *not-sharing* constraint

Not Sharing: The rooms r_1 and r_2 are the rooms the entity e is placed before and after the *relocate* move. There are two conditions based on whether the entity e has a *not sharing* constraint defined on it or not. This constraint requires several case checks related to the number of entities with *not-sharing* constraints in rooms r_1 and r_2. The algorithm is described in the pseudo-code in Figure 5.3. Based on specific conditions related to the number of entities, the change in cost value due to *not sharing* constraint (δ_{nsh}) can be increased once or twice the penalty value of the *not sharing* constraint. This is due to elimination or introduction of one or two *not sharing* constraint violations due to transfer of entity e from room r_1 to r_2.

Proximity Constraints: Room r_f is the room the other entity f associated with the constraint is allocated to.

- If r_1 is adjacent to r_f and r_2 is not adjacent to r_f, increase δ with *penalty*$_{ad}$

- If r_2 is adjacent to r_f and r_1 is not adjacent to r_f, decrease δ with *penalty*$_{ad}$

For *nearby* constraint, use nearby relationships (sets) instead of adjacency. *Away from* is the opposite case of *nearby*, so reverse the conditions.

Space Misuse: For both rooms r_1 and r_2, the space misuses before and after the move are calculated. Since entity e is leaving r_1, r_1 may go from an overuse situation to an underuse situation. Conversely, room r_2 may go from an underuse situation to overuse. Cost changes δ_{space} on both rooms r_1 and r_2 are calculated based on the difference between current space misuse and the previous one.

Capacity Constraint: This constraint is handled similar to the space misuse cost change calculation. There are two conditions checked:

- Room r_1 has a *capacity* constraint. In this case, the difference in space misuse in room r_1 before and after the move is calculated. If there is currently no overuse situation while there was one before, decrease δ_{cap}.

- Room r_2 has a *capacity* constraint. Again, the difference in space misuse in room r_1 and r_2 before and after the move is calculated. If there is currently an overuse situation while there was not one before, increase δ_{cap}.

5.6 Experiments Related to Local Search Algorithm

Specific experiments were performed on eight instances: three instances from $SVe150$ dataset ($S_{0.00}V_{0.00}$, $S_{0.60}V_{0.60}$, $S_{1.00}V_{1.00}$); three instances from $PNe150$ dataset ($P_{0.00}N_{0.00}$, $P_{0.15}N_{0.15}$, $P_{0.25}N_{0.25}$); and two *nott*1 instances (*nott*1, *nott*1b). Also, complete experiments on the whole $SVe150$ and $PNe150$ datasets were performed as well by using only improving criterion with 10 runs each 180 seconds. Tests were performed on a Core 2 Duo E8400 3GHz Intel machine.

In the following experiments, the objective function for total penalty (TP) is taken as the weighted summation of space misuse (SMP) and soft constraint violation penalties (SCP). The respective formulations for SMP, SCP and TP were previously given

in equations 2.3.1, 2.3.2 and 2.3.3 in Section 2.3.2. This objective function is going to be minimised subject to *hard* constraints. The numbers of *hard* and *soft* constraints for *nott1*, *SVe*150 and *PNe*150 instances were previously given in Table 2.2 (Section 2.4) and Table 2.4 (Section 2.5.3) in Chapter 2 respectively.

Experiments related to the balance between the *relocate* and *swap* moves described in 5.3 are presented in Section 5.6.1. The acceptance/rejection methods described in Section 5.4 are compared in Section 5.6.2. Complete results on *SVe*150 and *PNe*150 datasets are given in Section 5.6.3. The local search algorithm is compared to the integer programming models proposed in Chapter 4 in Section 5.6.4.

5.6.1 Balance Between Relocate and Swap Moves

In this section, whether using two different stochastic neighbourhoods with *relocate* or *swap* moves is beneficial over using just one move is explored. At each step in a stage of iterated local search, it is probabilistically decided which move operator is going to be chosen. At each step, only one move type can be used, the *swap rate* determines the probability the *swap* move is going to be chosen. Naturally, the *relocate* rate then becomes 1− *swap rate*. The results on four different instances are presented in Figure 5.4 as box-whisker plots. These instances were given 20 runs (each 75 seconds) on a Core 2 Duo E8400 3GHz Intel machine.

As can be clearly seen from the box-whisker plot, combining random *relocate* and *swap* moves in the algorithm yields better results that are statistically significant. At both extreme spectra where only *relocate* or *swap* move is used in the algorithm, the performance is significantly worse in all four instances. In the middle range of *swap rates* (0.25 to 0.75), the performance is significantly better which signifies the synergy between two move types. For these four instances, taking the *swap rate* as around 0.75 gives more tightly packed results in terms of lower and upper quartiles. The difference between the minimum and the maximum results is usually low as well. However, *swap rate* at 0.50 can give better minimum results and low median and mean values as well. However, the spread between the lower and upper quartiles can be large. In any case, all the values between 0.25 to 0.75 can be chosen without significantly affecting the performance of the algorithm unlike the borderline cases of 0.00 and 1.00 as *swap rate*.

5.6.2 Comparison of Acceptance/Rejection Mechanisms

These experiments study the acceptance/rejection methods described in Section 5.4 which decide whether a local non-improving solution should be allowed or discarded

(a) $S_{0.60}V_{0.60}$

(b) $S_{1.00}V_{1.00}$

(c) $P_{0.15}N_{0.15}$
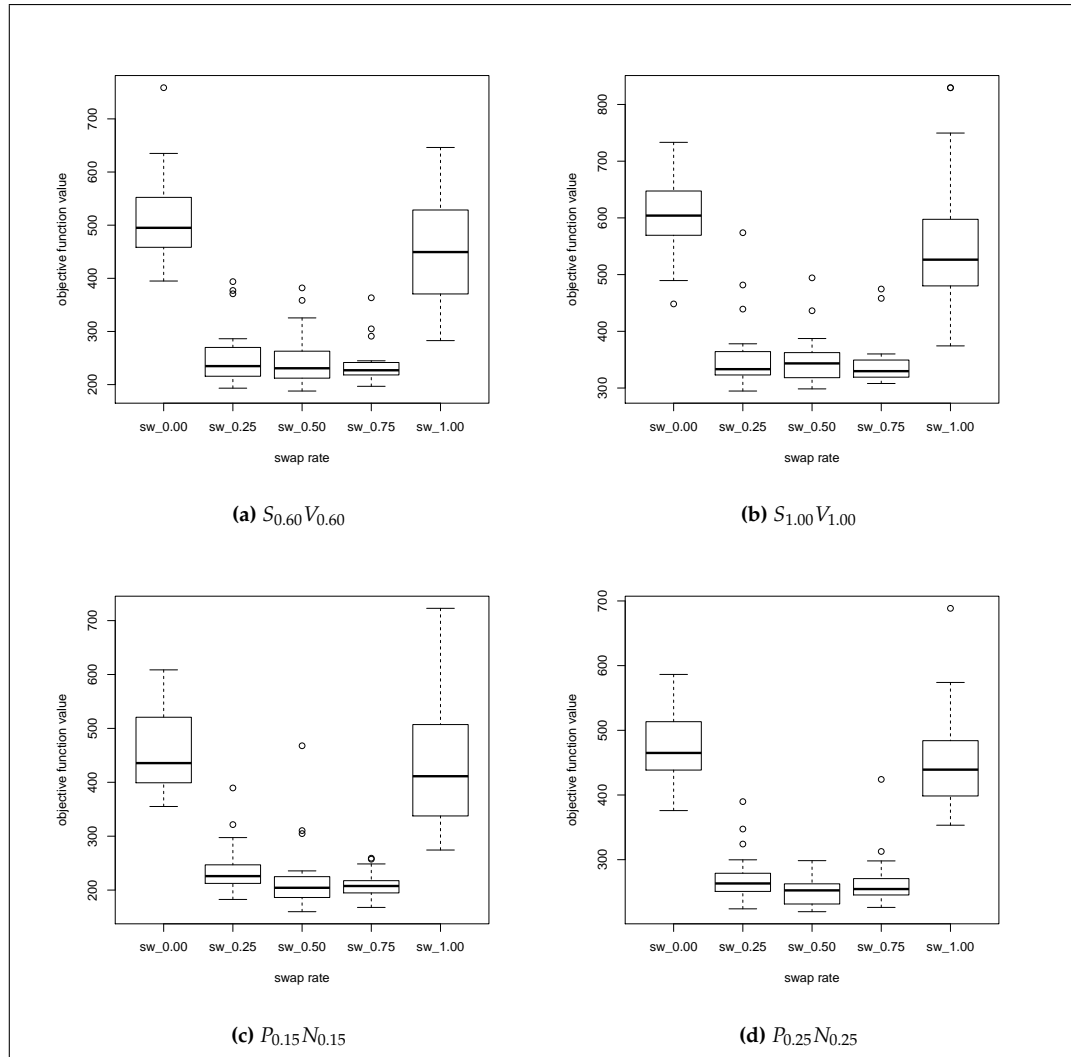
(d) $P_{0.25}N_{0.25}$

Figure 5.4: Objective function value plots of different *swap move* rates for instances $S_{0.60}V_{0.60}$, $S_{1.00}V_{1.00}$, $P_{0.15}N_{0.15}$, and $P_{0.25}N_{0.25}$

if it does not improve the global best solution (x_{best}) are tested. Five methods are tested: Only improving (OI), better or equal (BE), simulated annealing (SA), great deluge (GD), and threshold acceptance (TA).

SA, GD, and TA each requires the management of two parameters. The first parameter is the temperature, water level, threshold for each algorithm respectively. The second parameter is the reduction step applied to the first parameter. The number of combinations and fine-tuning will require great amount of experimental time, so it is decided to simplify this process as follows: The reduction of the temperature, water level or threshold will be made only when there is an improvement to the global best solution. After preliminary experimentation with OI and BE, it is observed that around 500 improvements occur to the global best solution during the runtime of the search,

(a) *nott*1

(b) [*nott*1*b*

(c) $S_{0.00}V_{0.00}$

(d) $S_{0.60}V_{0.60}$

(e) $S_{1.00}V_{1.00}$

(f) $P_{0.00}N_{0.00}$

(g) $P_{0.15}N_{0.15}$

(h) $P_{0.25}N_{0.25}$
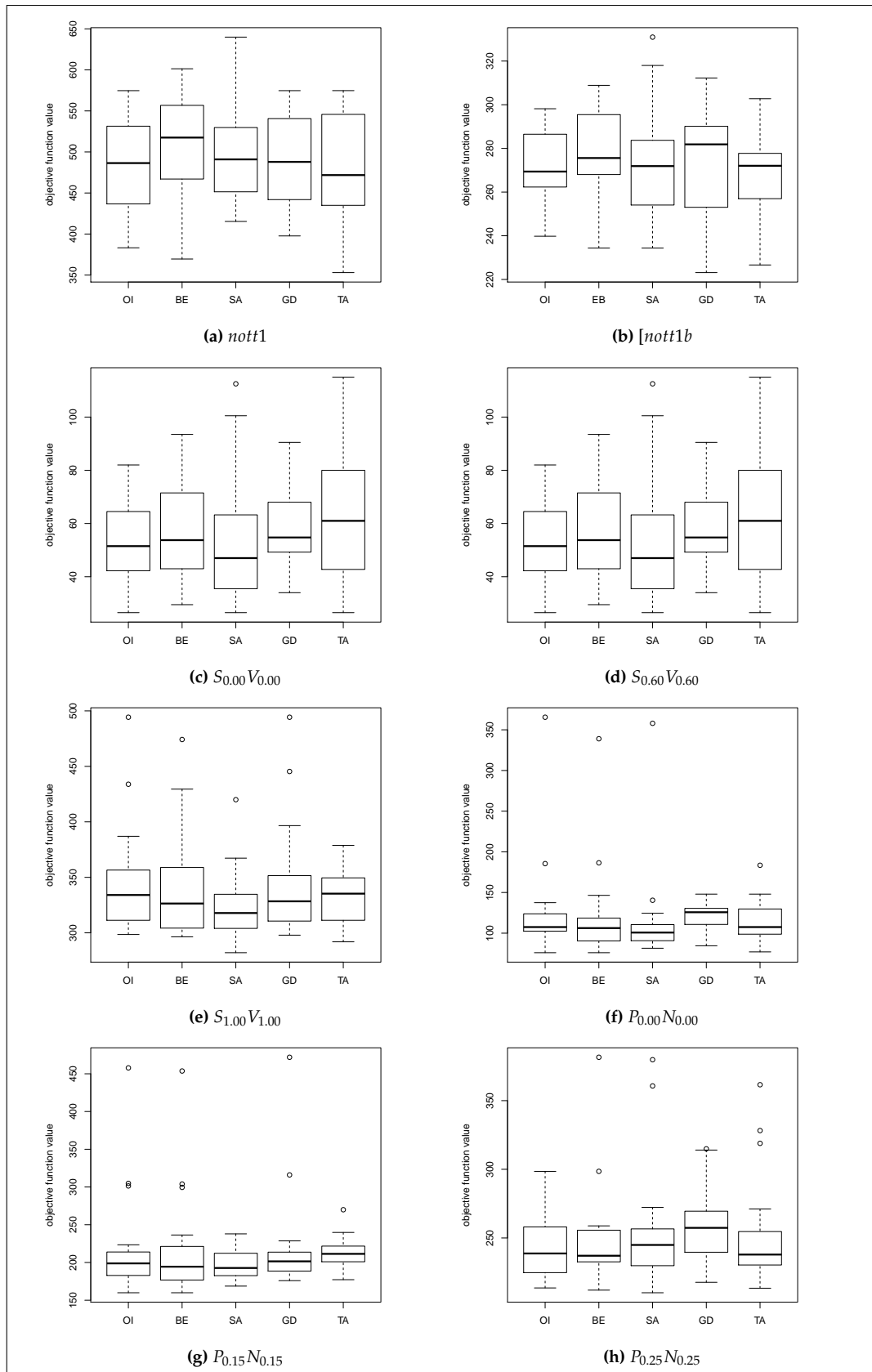
Figure 5.5: Objective function value plots for several instances in various acceptance/rejection methods

so the reductions in SA, GD, and TA are fine-tuned to make these algorithms operate in OI mode after 500 improvements were made to the global best solution. In order to achieve this balance, the following parameters are chosen:

- *Simulated Annealing:* initial temperature $T = 50$, decrement step $d = 0.1$.

- *Great Deluge:* initial water level $W = 500$, decrement step $d = 1$.

- *Threshold Acceptance:* initial threshold $T = 50$, decrement step $d = 0.1$.

In this test setup, each instance was given 20 runs (75 seconds each). The results are given for eight different instances as box-whisker plots in Figure 5.5.

As it can be seen from the figures, it is not easy to decide which method performs better than another. The large standard deviations of the results make each method close to each other especially in terms of spread of the lower and upper quartiles. Also, there are plenty of bad solutions encountered in the outliers of the box plots which is mostly due to the algorithm getting stuck in a local optima pretty quickly.

However, it is observed that the performance of the accepting only improving solutions over the global best solution is no worse than any other method with two parameters that needed to be fine-tuned. This gives only improving (OI) and to some degree better or equal (BE) methods an advantage over other mechanisms because they are not susceptible to poor selection of parameters associated with other acceptance/rejection methods.

5.6.3 Complete Results on SVe150 and PNe150 Datasets

The complete results for *SVe*150 and *PNe*150 datasets are given in Table 5.1a and 5.1b respectively. The results are obtained after 10 runs (180 seconds each). Columns μ σ, and *min* represent the average best value, standard deviation, and minimum result obtained after 10 runs. Columns SMP, SCP represent the average space misuse and constraint penalties over 10 runs.

The most important drawback of the local search methods described in this chapter is the high standard deviations as can be seen from Table 5.1a and 5.1b. Several instances have standard deviations in the range of 60-80 while the average can be 2 or 3 times of this standard deviation. This indicates the instability of the algorithm. In certain runs, this local search framework with mostly stochastic moves has a tendency to yield very poor results due to inability to escape from local optima.

S	V	μ	σ	SMP	SCP	min
0.00	0.00	47.90	18.42	27.90	20.00	23.50
0.00	0.20	85.05	13.51	37.05	48.00	63.00
0.00	0.40	133.75	68.50	42.75	91.00	72.00
0.00	0.60	143.60	52.94	45.60	98.00	102.00
0.00	0.80	160.05	17.15	49.05	111.00	138.50
0.00	1.00	180.95	18.13	55.95	125.00	160.00
0.20	0.00	63.72	16.12	44.72	19.00	46.80
0.20	0.20	105.78	15.02	59.78	46.00	74.30
0.20	0.40	118.36	8.67	62.36	56.00	104.60
0.20	0.60	135.15	16.37	62.15	73.00	113.50
0.20	0.80	191.01	23.58	79.01	112.00	153.30
0.20	1.00	202.76	23.53	88.76	114.00	176.80
0.40	0.00	124.63	16.84	83.63	41.00	97.90
0.40	0.20	157.58	23.34	97.58	60.00	134.20
0.40	0.40	185.00	27.95	104.00	81.00	154.70
0.40	0.60	203.61	53.89	97.61	106.00	171.90
0.40	0.80	237.19	41.29	112.19	125.00	186.20
0.40	1.00	246.06	19.18	119.06	127.00	214.30
0.60	0.00	165.15	52.14	115.15	50.00	128.10
0.60	0.20	190.61	66.16	117.61	73.00	156.50
0.60	0.40	210.83	54.97	125.83	85.00	167.00
0.60	0.60	226.92	56.49	128.92	98.00	186.10
0.60	0.80	250.79	20.55	138.79	112.00	225.80
0.60	1.00	264.74	26.06	137.74	127.00	241.80
0.80	0.00	144.76	11.39	108.76	36.00	122.40
0.80	0.20	200.70	68.07	114.70	86.00	164.20
0.80	0.40	201.25	54.94	123.25	78.00	170.90
0.80	0.60	207.44	17.32	125.44	82.00	182.60
0.80	0.80	250.38	11.97	134.38	116.00	230.50
0.80	1.00	279.64	29.00	150.64	129.00	244.60
1.00	0.00	193.55	15.17	162.55	31.00	171.20
1.00	0.20	232.89	18.19	161.89	71.00	209.20
1.00	0.40	243.00	16.05	166.00	77.00	219.20
1.00	0.60	253.50	13.01	167.50	86.00	234.90
1.00	0.80	293.33	23.06	178.33	115.00	254.80
1.00	1.00	334.42	61.51	181.42	153.00	286.70

(a) *SVe*150 Instances

P	N	μ	σ	SMP	SCP	min
0.00	0.00	124.60	81.42	36.60	88.00	84.50
0.00	0.05	144.82	30.11	78.82	66.00	109.20
0.00	0.10	188.05	31.46	115.05	73.00	162.50
0.00	0.15	208.13	12.47	142.13	66.00	191.20
0.00	0.20	240.01	8.24	180.01	60.00	225.60
0.00	0.25	270.60	11.24	215.60	55.00	262.30
0.05	0.00	98.88	15.88	34.88	64.00	73.20
0.05	0.05	132.47	15.79	76.47	56.00	110.20
0.05	0.10	185.23	70.63	102.23	83.00	151.10
0.05	0.15	204.24	36.64	127.24	77.00	183.70
0.05	0.20	227.16	14.89	158.16	69.00	208.10
0.05	0.25	266.66	18.52	196.66	70.00	242.10
0.10	0.00	108.16	10.83	52.16	56.00	90.40
0.10	0.05	141.42	20.77	75.42	66.00	112.60
0.10	0.10	197.39	80.02	95.39	102.00	144.20
0.10	0.15	201.49	40.49	120.49	81.00	163.20
0.10	0.20	232.60	20.74	150.60	82.00	208.90
0.10	0.25	244.82	5.76	183.82	61.00	235.60
0.15	0.00	122.06	10.21	68.06	54.00	107.10
0.15	0.05	150.15	51.90	81.15	69.00	116.20
0.15	0.10	182.23	46.48	96.23	86.00	137.20
0.15	0.15	208.96	84.57	118.96	90.00	153.00
0.15	0.20	225.85	19.42	146.85	79.00	193.60
0.15	0.25	248.49	22.12	171.49	77.00	219.60
0.20	0.00	125.79	8.82	79.79	46.00	113.10
0.20	0.05	181.75	70.93	93.75	88.00	139.10
0.20	0.10	171.86	19.94	102.86	69.00	144.70
0.20	0.15	205.51	29.21	118.51	87.00	161.00
0.20	0.20	214.18	16.38	138.18	76.00	190.40
0.20	0.25	231.85	15.94	156.85	75.00	209.50
0.25	0.00	160.84	58.20	95.84	65.00	137.80
0.25	0.05	192.69	77.31	105.69	87.00	148.10
0.25	0.10	175.61	16.90	106.61	69.00	160.70
0.25	0.15	196.61	57.89	114.61	82.00	163.90
0.25	0.20	212.17	17.50	133.17	79.00	190.50
0.25	0.25	226.08	19.15	146.08	80.00	204.90

(b) *PNe*150 Instances

Table 5.1: Experimental results on the *SVe*150 and *PNe*150 dataset instances using the local search algorithm

5.6.4 Comparison of Local Search and Integer Programming Models

In this section, we are going to analyse the difference between the local search algorithm (*LS*) described in this chapter and the integer programming models presented in Chapter 4. The mathematical models without and with the floor variables are abbreviated as IP_1 and IP_2 respectively.

(a) Δ_{min} of $IP_1 - LS$

(b) Δ_{min} of $IP_2 - LS$
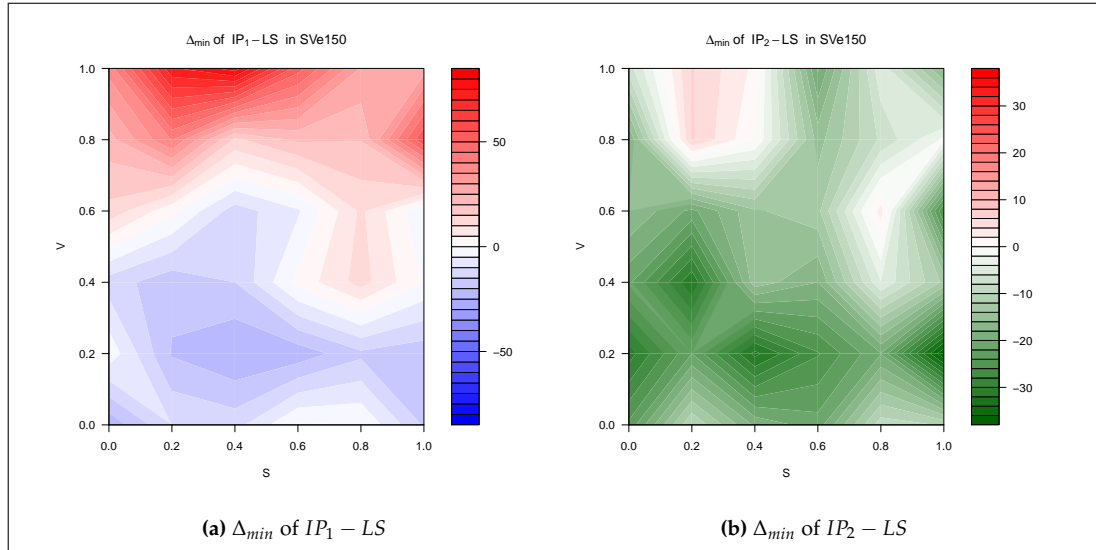
Figure 5.6: Differences in minimum penalty (Δ_{min}) after applying IP models without/with floor variables (IP_1 and IP_2) and local search (LS) on $SVe150$ dataset ($IP_1 - LS$ and $IP_2 - LS$). IP_1, IP_2, and LS are represented by blue, green, and red regions respectively.

(a) Δ_{min} of $IP_1 - LS$

(b) Δ_{min} of $IP_2 - LS$
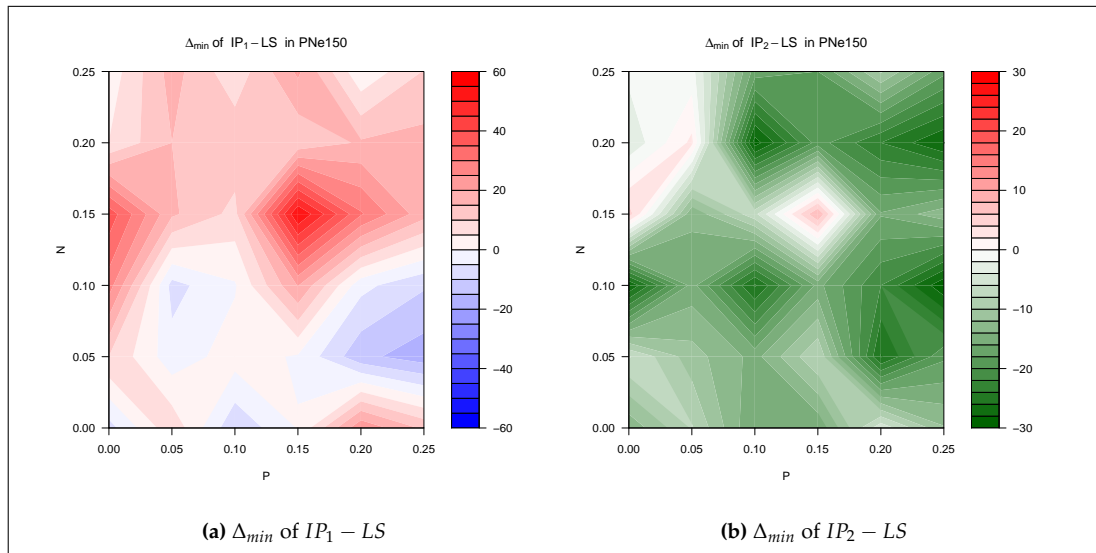
Figure 5.7: Differences in minimum penalty (Δ_{min}) after applying IP models without/with floor variables (IP_1 and IP_2) and local search (LS) on $PNe150$ dataset ($IP_1 - ELS$ and $IP_2 - LS$). IP_1, IP_2, and LS are represented by blue, green, and red regions respectively.

123

In Figures 5.6 and 5.7, the minimum total penalty obtained for each instance with LS is compared to the ones obtained by IP_1 and IP_2 using $SVe150$ and $PNe150$ datasets respectively. IP_1 or IP_2 and LS were given equal amount of running time: LS was given 10 runs (3 minutes each), while IP_1 and IP_2 were given a single run of 30 minutes.

The x and y axes represent *slack space rate* (S) and *violation rate* (V) in Figure 5.6 while they represent *positive* (P) and *negative* (N) *slack space rate* in Figure 5.7. In these contour plots, IP_1 or IP_2 is better than LS in obtaining the minimum total penalty when a region becomes more blue or green respectively. LS is better in redder regions. IP models and LS algorithms are about equal in obtaining the minimum penalty in white or pale coloured regions.

From a general analysis of the results, while the results in LS are of acceptable quality, there are certain drawbacks of the current algorithm. The results are close to the first IP model without the floor variables (IP_1). However, they are significantly behind the second IP model that utilise the floor variables (IP_2).

If the instances in $SVe150$ are analysed in Figure 5.6a, it is observed that IP_1 is better than LS in obtaining the minimum total penalty when the soft constraint violation penalty is expected to be low (in blue regions). However, when the parameter V increases, it is observed that LS begins to perform better in obtaining the minimum total penalty (in redder regions). It is observed that regardless of S rate, the variation of expected space misuse does not differentiate either IP_1 or LS in obtaining the minimum total penalty.

The performance is better for LS when the instances in $PNe150$ are analysed in Figure 5.7a. LS is generally better than IP_1 in this instance set (pale to darker red regions). The performance of LS is usually better than IP_1 in high N region where the instance is expected to have high overuse penalty due to lack of capacity. LS is significantly better than IP_1 in $P = N = 0.15$ region where IP_1 tends to struggle.

However, LS is not competitive with IP_2 in obtaining the minimum total penalty. In $SVe150$ instances (Figure 5.6b), the dominance of IP_2 becomes stronger in low V rates (darker green regions). In $PNe150$ instances (Figure 5.7b), IP_2 is consistently better than LS apart from $P = N = 0.15$ (similar to IP_1 in this case).

5.7 Conclusion

In this chapter, a general iterated local search framework that used stochastic *relocate* and *swap* moves was proposed. This iterated local search framework periodically retracted back to the best obtained solution encountered during the search and continuously improved this best solution. This repeated retracting back to the best solution was not investigated in previous heuristic designs in OSA literature and it might be one of the most important reasons for poor performance of previous heuristic algorithms for this problem (when such results were compared to the results obtained by integer programming models proposed in Chapter 4). Therefore, this framework will be kept in upcoming local search based algorithms that will be proposed in Chapters 6 and 7.

It was observed that combining *relocate* and *swap* move operators to define and traverse the neighbourhood was beneficial to the performance of this algorithm. For borderline cases where the algorithm operated with just a single random move operator, the results were less than satisfactory. This signified the importance of using multiple neighbourhood structures when the majority of the components of a solution method was stochastic.

Another issue considered in this chapter was a mechanism which decided how to proceed when a solution after a stage of local search was worse than the current best global solution. Descent methods, simulated annealing, threshold acceptance, and great deluge algorithm were tested. The methods apart from descent methods required additional parameters that needed to be fine-tuned. Although a general purpose parameter handling procedure was implemented, it was not observed that results were significantly better than only accepting the solutions that were better than the global best solution. For the rest of the thesis, the acceptance criterion will be taken as *only improving* in order to reduce the number of parameters in the algorithms in Chapters 6 and 7.

One of the most obvious drawbacks of the algorithm developed in this chapter was the large standard deviation of the results. Several runs were observed where the algorithm got stuck in local optima points quickly and was unable to improve the results. Compared to the integer programming models described in Sections 4.2 and 4.5, the local search algorithm was only competitive with the first integer programming model without the floor variables (IP_1) while it was completely dominated by the model with the floor variables (IP_2) when the minimum penalty obtained was minimised. In the next chapter, more greedy move operators will be proposed in order to tackle the large deviation in best obtained total penalties due non-avoidance from local optima.

Evolutionary Local Search Algorithm

6.1 Introduction

In this chapter, an evolutionary local search algorithm is proposed for solving the OSA problem. At its core, this algorithm retains the iterated local search framework utilised in Chapter 5 to intensify the search around the global best solution; however, an almost opposite approach is taken here in regards to defining and traversing the neighbourhood of a given solution. The algorithm mostly relies upon the *relocate* move that sends an entity from one room to another. The selection of the move and acceptance/rejection of such a move are performed in a very greedy manner. In order for this method to work efficiently, a very fast objective function that can handle nine different constraints and space misuse components is developed. A local search algorithm operating with a *relocate* move (sending an entity to another room) works on a search space that changes very slowly over successive move operations. By focusing on a very efficient implementation of a *relocate* move in local search, finding a very good move over a large search space becomes easier. But more importantly, this greedy implementation allows the update of specific locations that are affected by the move. This in turn speeds up finding the next best move in the successive operations.

This chapter is organised as follows: Section 6.2 presents the outline for the proposed evolutionary local search algorithm (*ELS*). The details of the evolutionary components of the algorithm are explained in Section 6.3. The implementation of the local search stage of the algorithm with delta and update stages is explained in Section 6.4. Experimental results on effects of the parameters of the algorithm and comparative analysis are presented in Section 6.5. Conclusions are given in Section 6.6.

6.2 The Algorithm Outline

An *evolutionary local search* algorithm (*ELS*), also referred to as a *memetic algorithm* is developed for solving the OSA problem. A memetic algorithm [Moscato, 1989] is traditionally the hybridisation of a genetic algorithm [Goldberg, 1989] with local search mechanisms. The goal of the evolutionary components of the algorithm is typically jumping to a different region of the search space (*exploration* or *diversification*). The local search is then applied to quickly explore this specific region of the search space (exploitation or intensification).

A general outline of the *ELS* algorithm is given in Figure 6.1. *ELS* contains a series of iterations of evolutionary and local search operators until a termination criterion is met (which is usually the time limit being reached). The evolutionary components include selection of parents for crossover, crossover, mutation, and replacement operation. The children solutions generated by the evolutionary components are improved using a local search method which is similar to *iterated local search* [Lourenco et al., 2002] and *ruin and recreate* [Schrimpf et al., 2000]. The local search includes methods for searching the whole *relocate* neighbourhood as quickly as possible and allows rapid updates to the cost change of each move in the neighbourhood. The local search method is described in great detail in Section 6.4.

Input: input file with entities, rooms, and constraints.
Output: best solution (an entity-room mapping)
 1: Randomly initialise a population of solutions
 2: **repeat**
 3:     Randomly select two parent solutions for crossover
 4:     Apply *Crossover* to produce two children
 5:     Apply *Random mutation* on two children
 6:     Apply *Allocation mutation* on two children
 7:     Apply *Same room* and *proximity constraint based mutation* (optional)
 8:     Apply *Local Search* on two children produced
 9:     Replace the parents with the best solution encountered so far in *ELS*
10: **until** Time limit is reached

Figure 6.1: Evolutionary local search (*ELS*) algorithm

6.3 Evolutionary Components

Solution Representation and Population: The algorithm proposed here uses a standard encoding of fixed-size arrays. The length of a single solution is equal to the number of entities and the content of each location represents the room the entity is placed into. Small populations of 10 to 20 solutions are utilised. The solutions in the population are initialised with a mixture of randomness and enforcement of *allocation* constraints. A majority of entities associated with *allocation* constraints (about ninety percent) is placed first, then the remaining entities are allocated to random rooms.

At each generation, two solutions are selected randomly from the population. Different selection mechanisms such as roulette-wheel, ranking, or tournament based selection are tested. Since the number of solutions in the population is rather small, and the search is mostly governed by the local search heuristic, the population usually contains a good number of high quality solutions with low diversity. The diversity measure between two candidate solutions in the population can be taken as the number of different locations in two solutions (the number of entities that are placed in different rooms). The population diversity can be measured as the sum of pairwise diversities of each two solutions in the population. Coupled with a rather aggressive replacement strategy, non-random selection methods usually reduce the diversity of the population too drastically. Due to these issues, it is decided on the random selection of solutions.

In the current implementation, a steady state population which creates two children from two parents at each generation was implemented. A trans-generational population that creates and replaces a whole size population at each generation was also tested. However, preliminary experiments yielded no discernible performance differences.

Crossover: Two solutions (parents) are selected for crossover. Different crossover operators are examined including one and two point or uniform crossovers. Also, several problem specific crossover operators based on the preservation of allocations within rooms and floors are proposed. These crossover operators will be explained in more detail in Section 6.3.1.

Mutation: After the crossover, different mutation operations are applied to the produced children. The first one is the traditional random mutation operator in which a randomly selected location is changed (which effectively sends the entity from one room to another). The mutation rate m (which is the percentage of the entities that are

randomly subjected to mutation) is an important parameter of the algorithm. The second mutation operator is based upon the *allocation* constraint. The operator selects a randomly high number of violated allocation constraints and forces them to be satisfied by setting the corresponding location in the solution accordingly. Other mutation operators based on the enforcement of different constraints are also proposed. These mutations will be described in 6.3.2.

Replacement: After the application of mutation, each child is improved using the local search procedure which is described in Section 6.4. If the local search procedure produces a best solution during this stage starting with the child, this new best obtained solution encountered during the search replaces the parent. Otherwise, the previous best solution encountered throughout the search replaces the parent. This can be considered as a very aggressive replacement strategy which can reduce the diversity of the population quite fast. However, due to improved performance, this is the chosen strategy.

Figure 6.2: Traditional crossover operators for office space allocation

130

6.3.1 Crossover Operators

Several traditional crossovers are considered for the evolutionary stages of the algorithm. These crossovers are *one-point-crossover* (1-PTX), *two-point-crossover* (2-PTX), and *uniform crossover* (UX) as described in Section 3.6.1. These crossovers operate on general integer variable fixed size array formulations where each location in the array corresponds to an entity and the content of each location signifies the room the entity is going to be allocated to. A sample application of three traditional crossover operators on two parent solutions is depicted in Figure 6.2.

Several OSA specific crossover operators that utilise the room and floor information in OSA are also developed and tested. These crossover operators are inspired from the *greedy partition crossover* proposed for solving the graph colouring problem [Galinier and Hao, 1999]. These problem specific crossovers are:
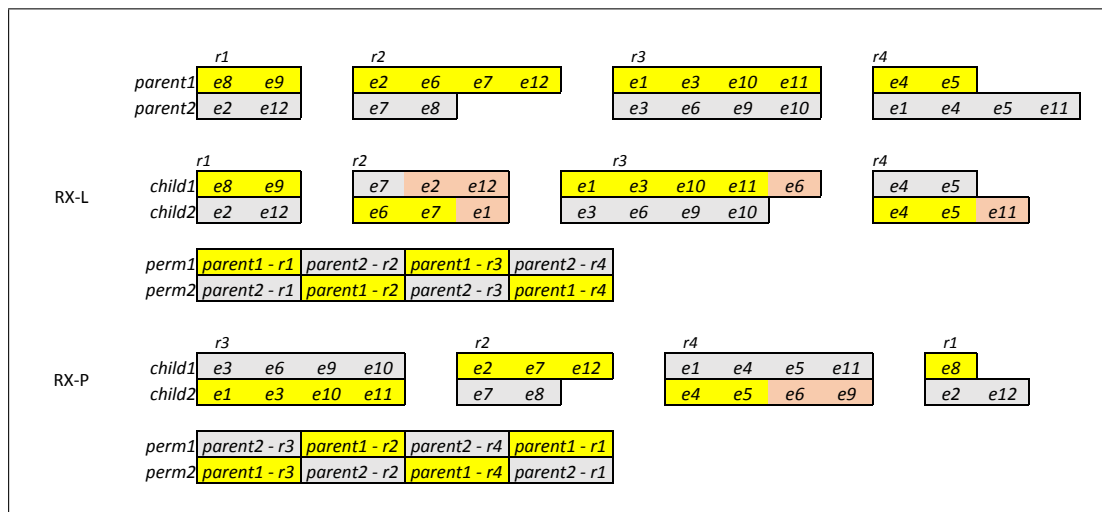
Figure 6.3: Room based crossovers (RX-L and RX-P)

Room Based Crossover: In this crossover, the goal is to transmit the allocations in each room from the parent to the children as close as possible. This operator first creates a room content list for each parent: it goes through each entity, finds the room the entity is allocated to in that parent and places that entity into the corresponding location in the room content list. In *linear room based crossover* (RX-L), the algorithm starts with the room having the first index (which is r_1). It transmits the entities in this room from a randomly selected parent and transmits them to one of the children. The entities in the room content list of the other parent are transferred to the other children. The entities that are transmitted from one parent are deleted from the room content list of the other parent after each transfer.

The transmission-deletion continues from the first room to the last one in *linear room based crossover* (RX-L). However, this crossover has an inherent bias to the rooms indexed with lower ids because it prioritises the transmission of these rooms due to the simpler implementation. In order to reduce this bias, a random permutation of rooms is created in *permutation room based crossover* (RX-P) at each crossover attempt. This allows different orderings for transmissions of the room contents at each crossover. Entities that cannot be allocated in this way are assigned randomly to the children at the end.

Example applications of room based crossovers (RX-L and RX-P) are presented in Figure 6.3. Two children solutions (*child*1 and *child*2) are produced from two parent solutions (*parent*1 and *parent*2) using the depicted permutation arrays (*perm*1 and *perm*2).
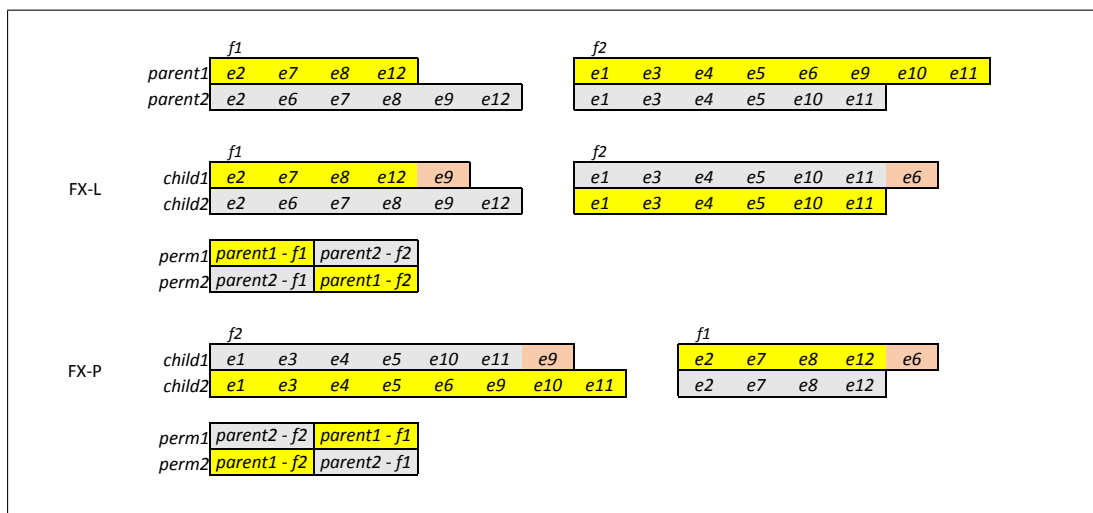
Figure 6.4: Floor based crossovers (FX-L and FX-P)

Floor Based Crossover: In this crossover, the goal is to transmit the allocations within a whole floor in the parent to the children. This crossover is intended for OSA cases where there is a clear structuring of rooms in floors (or sections of the buildings) and preserving promising floor structures to future generations in *ELS* may be beneficial. For each parent, a floor content list is generated. Each location in a floor content list is filled with the entities allocated to that floor. There are again two variants similar to the *room based crossover*. In *linear floor based crossover* (FX-L), the algorithm starts with the first floor content of a randomly selected parent and transmits it to the first child. The first floor content of the non-chosen parent is transmitted to the second child. Transmitted entities are deleted from the opposite parents in order to prevent duplications. This operator continues like a uniform crossover, selecting the next floor content in order and transmits the entity-room mapping from a randomly chosen parent to the

children.

This transmission and deletion process goes from the first floor to the last one in *linear floor based crossover* (FX-L). In the end, some of the children may lack some of the entity-room mappings, in this case the missing mappings are randomly assigned. Since FX-L biases the floors with low indices, *permutation floor based crossover* (FX-P) is also developed. Similar to RX-P, in this crossover, a random permutation of floors is created, and the entity-room mappings in the floors are transmitted according to the order in the permutation.

Example applications of floor based crossovers (FX-L and FX-P) are presented in Figure 6.4. Two children solutions (*child*1 and *child*2) are produced from two parent solutions (*parent*1 and *parent*2) using the depicted permutation arrays (*perm*1 and *perm*2).

6.3.2 Mutation Operators

There are various mutations considered over the development of the algorithm. Most commonly used one is the traditional one point mutation. This mutation procedure goes through all entities and randomly chooses a low percentage of them for mutation. The disruption is performed by randomly assigning a new room different from the one the entity is currently allocated to. This random mutation operator is one of the most important steps in the *ELS* algorithm because it is the main method for introducing new information to the search by creating new entity-room allocations that are not present in the population.

Apart from the traditional one point mutation, several constraint based mutations that enforce the satisfaction of such constraints are also considered. Some of these mutations are as follows:

Allocation Mutation: In this mutation, the algorithm creates a very large subset of the *hard* and *soft allocation* constraints (usually %80-%90) and relocates the entity to the room specified in the constraint. It is observed that this mutation is very effective when coupled with the traditional one point mutation; therefore, these two form an important basis for the *ELS* algorithm.

Same Room Mutation: The algorithm takes a subset of *hard* and *soft same room* constraints and moves and checks the entities associated with such constraints in the current allocation. If the two entities are not in the same room, then one of the entities is

moved to the room the other one is currently in. If the two entities are already in the same room, one of them might be moved to another room with a very low probability, otherwise they stay in the room they are currently allocated to.

Greedy Mutations based on Proximity Constraints: Based on the three proximity constraints (*adjacency*, *nearby* and *away from*), three greedy mutations that try to correct the respective constraint violations were developed. These operators can also be regarded as *repair* operators tailored to specific constraints due to the greedy corrective nature of the operator. Note that these mutation operators are optional in this evolutionary local search algorithm. It was observed that addition of these mutations did not significantly improve or worsen the performance of the algorithm. However, for potential future instances with large number of proximity constraints, it may be beneficial to use these greedy mutation operators.

- *Adjacency Mutation:* The entities in a subset of *adjacency* constraints are considered in this mutation. If two entities are allocated to the rooms that are not adjacent to each other, one of the entities is chosen and randomly sent to one of the rooms that are adjacent to the room the other entity associated to the constraint is currently placed. If two entities are already in adjacent rooms, an entity might be sent to a non-adjacent room with a very low probability.

- *Nearby Mutation:* This mutation aims to fix the *nearby* constraints. Each binary *nearby* constraint is associated with a group head and several binary *nearby* constraints form a group structure. Thus, the mutation checks the entities associated with a group, and moves the entities to the rooms that are near (in the same floor) to the room the groups head is allocated to. If the entity is already in the same floor with the group head, it might be sent to another floor with a very low probability.

- *Away from Mutation:* In this mutation, the entities in a subset of *hard* and *soft away from* constraints are considered. If two entities are near to each other (in the same floor), then one of the entities is moved to rooms that are away from the other one (to another floor). If these entities are already in different floors, one of them might be moved to the floor the other entity is placed to with a very low probability.

134

Input: input solution
Output: output solution
 1: Initial calculation of the Delta matrix Δ (Delta Stage)
 2: **for** $= 1 \rightarrow h$ iterations **do**
 3:     Select $E' = |E|/d$ entities randomly from the entity set
 4:     Find minimum (e, r_2) in $\forall e \in E' \ \forall r \in R \ (e, r)$
 5:     Make move (e, r_1, r_2)
 6:     Update the locations in Δ affected by the move (e, r_1, r_2)

Figure 6.5: Local search stage in *ELS*

6.4 Local Search

The main design goal in the local search operation is to apply a short but aggressive search to cover the *relocate* neighbourhood of a candidate solution as quickly as possible. A local search method is implemented based upon searching a large portion of the *relocate* (one entity moved from one room to another one) neighbourhood of the current solution as fast as possible.

There are efficient and fast update algorithms on the objective cost function in the literature such as *delta evaluation* in [Ross et al., 1994]. For an efficient implementation of greedy local search in this thesis, fast cost calculation methods that involve the space misuse and the nine different types of constraint violations are required. In order to identify the cost changes associated with each move, a cost change matrix called Δ is used. The Δ matrix is of size $|E||R|$ where each location (e, r) corresponds to a move of sending the entity e from its current location to another room r and holds the cost change of making such a move. There are two steps to maintain the Δ matrix: delta and update stages. These two will be described in Sections 6.4.2 and 6.4.3 respectively. The outline of the local search stage is given in Figure 6.5.

6.4.1 Neighbourhood in Evolutionary Local Search Algorithm

ELS algorithm described in this section only uses greedy *relocate* moves to define and traverse the neighbourhood unlike the *LS* algorithm described in Chapter 5 which uses both *relocate* and *swap* moves. The *swap* operator was dropped in *ELS* due to following reasons:

Delta Table Updates for Swap Moves: One of the design goals for developing *ELS* was to test the greedy implementation of move operator using Δ table as described in

Section 6.4.3. An *exact* implementation of *swap* operator that gives correct changes in the objective cost function can be too costly in design and coding time.

Regardless of the current solution during the search, all *relocate* moves apart from the ones that send the entity to the room it is already allocated to will be valid. Therefore, all *relocate* moves that sends any entity to any room can be defined beforehand and whole Δ table can be statically kept in the memory. This is not the case in *swap* move because not all *swap* moves are allowed given a current solution (two entities in the same room cannot be swapped for example); therefore, Δ table for *swap* move will require some dynamic checks or it might be recreated dynamically in each iteration.

A bigger problem is the cost update calculation for the *swap* moves. It is easy to update the cost change values in Δ table for *relocate* moves after a *swap* move. The *swap* move can be divided into two successive *relocate* moves and two update operations for *relocate* move can be performed. However, the converse case of updating the cost change values in Δ table for *swap* moves after a *relocate* or *swap* move is not easy. This is a complex operation and not necessarily an efficient one especially after a *swap* move. It has to consider the potential changes of *swap* moves in Δ table after two successive *relocate* moves from which a single *swap* move is built.

An approximate cost update procedure was tested that created the Δ table for *swap* operation by adding two different Δ tables that were generated by applying two *relocate* moves individually. However, the results with this approximate method were never better than just using the *relocate* move alone in the search.

Hybrids of LS and ELS: Since, it is not easy to develop an update procedure for greedy *swap* move similar to greedy *relocate* move as in Section 6.4.3, several hybrids of *LS* and *ELS* were tested in the local search stage. The *swap* move was taken as stochastic as in Section 5.3. The *relocate* move was taken as greedy as in this chapter.

Several performance issues were encountered during the tests of this hybrid. The length of the local search stage was one of the major issues. The random *swap* move in *LS* required a higher number of local search move operations at each generation of *ELS*. On the other hand, greedy *relocate* move required lower amount of local search move operations. This was due to the opposite nature of random and greedy move operators. A single random *swap* move would be quick but more of them will be required to find a quality solution. A single greedy *relocate* move that searched a large section of the complete neighbourhood would be expensive; however, a smaller amount of such moves would be required to find high quality solutions.

Hybridisation of these two moves within *ELS* was not very successful. If random *swap* moves were embedded within the greedy *relocate* moves in a short local search stage, poor results were observed. The algorithm was also implemented by not embedding these two moves as well. In this case, it was observed that usually one move (either *relocate* or *swap*) took over the search process and poor synergy between two moves was observed as a result unlike the case in Section 5.6.1.

Consequently; due to the implementation, coding, embedding and poor synergy problems associated with *swap* move, *ELS* only used *relocate* moves that were greedily implemented using fast cost update procedure with Δ table.

6.4.2 Delta Stage

This stage is the initial calculation of the Δ matrix given a current solution. This stage incrementally calculates the whole single-move neighbourhood of a candidate solution. It is a very costly operation and must be used once after a large change is made on the candidate solution. Typical examples of a large change on the candidate solution are crossover operators, large mutation/perturbation operators, partial or full reinitialisation of a candidate solution. In these scenarios, it is usually necessary to perform a recalculation of the Δ matrix through the delta stage. In the current algorithm, this is the first step in the local search phase after children solutions are generated by the crossover and the mutation operators. This algorithm is derived from the fast cost calculation procedure for the *relocate* move described in Section 5.5. The delta stage in the *ELS* algorithm is a cumulative calculation of each possible *relocate* move.

6.4.3 Update Stage

After a single move (e, r_1, r_2) has been applied to the solution (entity e is moved from room r_1 to r_2), certain regions of the Δ matrix have to be updated. The algorithm goes through all constraints associated with entity e, and rooms r_1 and r_2 and only updates the locations that are affected by this single move (e, r_1, r_2). The affected locations (illustrated in Figure 6.6) are as follows:

When an entity e is moved from room r_1 to r_2, we must check whether there should be a change in space misuse of moving the rest of the entities to these two rooms in the next iteration of the local search. This corresponds to the locations $\forall e' \in (E - e)(e', r_1)$ and (e', r_2) in Δ matrix (blue columns). Since the space usage in r_1 is going to be reduced and the space usage in r_2 is going to be increased, the algorithm checks the difference of the space misuse of sending the entities $\forall e' \in (E - e)$ to r_1 and r_2 before and after
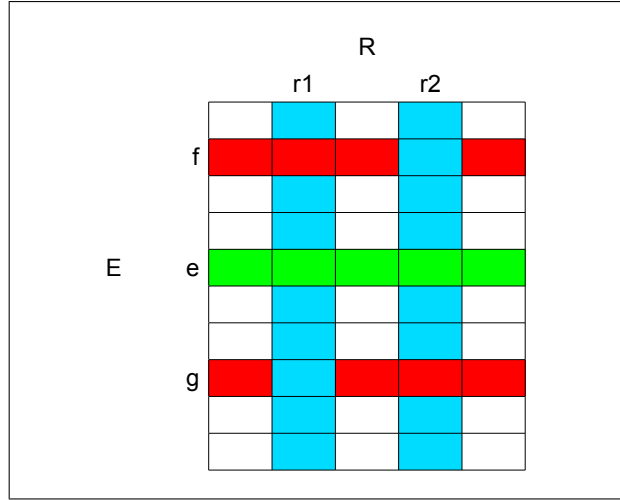
Figure 6.6: Locations affected by the move (e, r_1, r_2). The e row, and r_1 and r_2 columns are always affected at each *relocate* move. Update on f and g rows depend upon the constraint and the allocations in r_1 and r_2

the move is made and updates the corresponding location accordingly. Additionally, the locations $\forall r \in (R - r_2)$ $f \in E_{r_1}(f, r)$ and $\forall r' \in (R - r_1)$ $g \in E_{r_2}(g, r')$ also need to be updated (red regions). This corresponds to sending an entity that is already present in either r_1 or r_2 to all the other rooms and updating the space misuse accordingly.

The *capacity* constraint (that a specific room should not be overused) needs a similar update on the same locations as in space misuse calculations described above. The algorithm checks whether affected rooms have space overuse before and after the move, whether moving the other entities to rooms r_1 or r_2 prevents or causes an overuse before and after the move, and updates the corresponding locations accordingly.

For the *not sharing* constraint (that an entity should not share a room with others), the locations $\forall e' \in (E - e)(e', r_1)$ and (e', r_2) need to be updated because the numbers of entities in rooms r_1 and r_2 are changed. Sending any other entity to r_1 or r_2 might change the *not sharing* penalty if the rooms r_1 and r_2 contain entities that should not share the rooms with others. Additionally, the locations $\forall r \in (R - r_2)$ $f \in E_{r_1}(f, r)$ and $\forall r' \in (R - r_1)$ $g \in E_{r_2}(g, r')$ might or might not require an update operation.

If entity e has any of *same room*, *not same room*, *adjacency*, *nearby*, and *away from* constraints with another entity f, then the locations $\forall r \in R$ (f, r) need updating. These locations correspond to the moves that send the entity f to other rooms. If the constraint is satisfied before the move and not satisfied later or in the opposite case (that the constraint is not satisfied before the move and now it is satisfied), depending upon the constraint, the respective constraint penalty should be either subtracted or added to $\forall r \in R$ (f, r).

There are four sub-conditions in *same room*, *not same room*, *adjacency*, *nearby* and *away from* constraints while evaluating how the updates should be performed. In *same room* and *not same room* constraints, the first check is about whether the entities were in the same room or not *before* the *relocate* move. The second check is performed based upon whether the entities are in the same room or not *after* the *relocate* move is performed. There are four possible outcomes based on these two checks (each check has a true or false outcome). Update algorithms work by checking these four possible outcomes and update the corresponding locations appropriately. In *adjacency*, *nearby*, and *away from* constraints, the checks are about whether the rooms the entities are placed into are adjacent or near to each other before and after the *relocate* move. Again, the update algorithms check four different outcomes and update the corresponding locations.

During the experiments, it is observed that roughly three to five percent of the Δ matrix needs to be updated whenever a move is made. This percentage is expected to go down if the number of rooms is increased. This is due to the fact that regardless of the number of rooms in the instance, there will always be two rooms (the rooms r_1 and r_2 where the entity e is placed before and after the *relocate* move) that have to checked after each *relocate* move for Δ update. As a result, the update stage will yield in very high speed-ups for algorithms that search a large portion of the search space at each step (such as tabu search or steepest descent). However, it is not desirable to use the update stage in algorithms that operate on the acceptance or rejection of random moves (such as the local search methods tested in Chapter 5).

The update algorithm must also consider the hard constraints. In this work, it is decided not to use a very strict hard constraint checking mechanism that disables each move that violates a hard constraint. Instead, a very large penalty value is assigned for each hard constraint violation. This eliminates the need for strict hard constraint check coding because the moves that lead to invalid solutions (due to the violation of hard constraints) have already high penalty values therefore these moves are least likely to be selected. This approach has a theoretical drawback of traversing solutions in the infeasible region. In practice, it is observed that the algorithm quickly eliminates all hard constraints and begins working on the minimisation of the soft constraint violations and space misuse.

The following subsections describe the respective update algorithms for space misuse and each constraint in more detail. In order to preserve the format and flow of this thesis, the pseudo-codes associated with each update algorithm are moved to Appendix B.

Update Algorithm for Space Misuse

The update algorithm for space misuse is the most important part of the whole update stage. This update algorithm has to run after each *relocate* neighbourhood move and affects each entity in the problem. In order to speed up this process, matrix Δ_{sp} of size $|E||R|$ is used. This matrix Δ_{sp} is similar to the total cost change matrix Δ but it only holds the cost changes in *space misuse*.

The update algorithm iterates through each entity i and checks the room each i is allocated to. There are three conditions considered. The pseudo-code of the update algorithm for space misuse calculations is given in Figure B.1. For the sake of simplification, the symmetric case (similar to the condition as in room r_1) associated with room r_2 is omitted.

- This condition is given at lines 3-23 of Figure B.1. Entity i is in room r_1 (the room entity e is moved from). In this step, first the previous (before the move) and current (after the move) misuse penalties with and without entity i being moved to room r_1 are calculated. By using these four values, the first component of change in space misuse (δ^{sm1}) is calculated. In the second step, the algorithm iterates through each room. For the case of room r_2, the previous and current misuse penalties, and the necessary cost change value (δ^{sm2}) are calculated. For the location Δ_{ir_2} the incremental change required is $\delta^{sm1} + \delta^{sm2}$. For other locations Δ_{ij}, the incremental change is simply δ^{sm1}.

- This condition is given at lines 24-26 of Figure B.1. Entity i is in room r_2 (the room entity e is moved to). This case is a symmetrical version of the first one. Hence, the cost change in this condition is calculated similar to the first condition by swapping room r_1 with r_2.

- This condition is given at lines 28-36 of Figure B.1. Entity i is in neither room r_1 nor room r_2. This is the most encountered and hence the most important case because the majority of the entities will not be either in r_1 or r_2. The algorithm calculates the space misuse in rooms r_1, r_2 and room entity i is in and by using the Δ_{sp} and Δ matrices, updates the locations Δ_{ir_1} and Δ_{ir_2} accordingly.

Update Algorithm for Same Room and Not Same Room Constraints

The pseudo-codes for the update algorithms for *same room* and *not same room* are given in Figures B.2 and B.3 respectively.

The update algorithm for *same room* and *not same room* contains four conditions. For the *same room* constraint, the application of the update algorithm in these conditions is as follows:

- Entities e_1 and e_2 are in the same room before and after the *relocate* move. This condition requires no update. This condition is given at lines 1-2 of Figures B.2 and B.3 for *same room* and *not same room* constraints respectively.

- Entities e_1 and e_2 were previously in the same room but now e_1 is moved to another room. In this case, all cost changes associated with the rooms that e_2 can be sent to, go through a penalty reduction. This condition is given at lines 3-8 of Figures B.2 and B.3 for *same room* and *not same room* constraints respectively.

- Entities e_1 and e_2 were previously not in the same room but now they are in the same room. In this case, all cost changes associated with the rooms that e_2 can be sent go through a penalty increase. This condition is given at lines 9-14 of Figures B.2 and B.3 for *same room* and *not same room* constraints respectively.

- Entities e_1 and e_2 were previously not in the same room and they are still not in the same room. In this case, the cost change of moving e_2 to the previous room of e_1 is increased while the cost change of moving e_2 to the current room of e_1 is decreased. This condition is given at lines 15-18 of Figures B.2 and B.3 for *same room* and *not same room* constraints respectively.

Not same room constraint is the exact opposite of *same room* constraint. Therefore, at each step, the negative operation is performed. The penalty increases at specific location are replaced by penalty reductions and vice versa.

Update Algorithm for Not Sharing Constraint

The update algorithm for the *not sharing* constraint is performed every time a *relocate* move is performed. The update operation involves every entity because the cost changes of moving each entity to the two rooms affected by the *relocate* move have to be modified. This is because of the possibility of a change in number of entities with the *not-sharing* constraint in either of the two rooms affected.

The pseudo-codes for the update algorithm for not sharing constraint is given in Figure B.4 and B.5. For the sake of simplicity, the penalty values for hard and soft constraints are taken equal in the pseudo-code.

There are two stages in this update algorithm. The first stage updates the move costs of sending every entity to room r_1 (the room the entity e was previously located in). The second stage updates the moves costs of sending every entity to room r_2 (the room entity e is moved to after the *relocate* move operation).

Stage 1 - Updating the cost changes of moving entities to room r_1:   Entity e is moved out of room r_1, The algorithm iterates through each other entity i in the entity set E and checks the room it is located in. There are two sub-conditions in this stage:

- This condition is given at lines 3-21 of Figure B.4. Entity i is already in room r_1. The algorithm checks the number of entities in the room. If room r_1 has currently only one entity and neither entity e nor i has a *not sharing* constraint defined over them, then no update is necessary. However, if both of them have the constraint, then an increment in cost change is necessary, because moving i out of room r_1 no longer eliminates the constraint violation since entity e is already removed. If there are two entities residing in room r_1, the algorithm checks the number of entities which have *not sharing* constraint. If there is at least one such entity, then a penalty reduction is required because moving entity i out of room r_1 now eliminates a constraint violation. After the necessary change is calculated, then the algorithm iterates through each room j and adds the change to each Δ_{ij} location.

- This condition is given at lines 23-32 of Figure B.4. Entity i is in some other room other than room r_1. Again, the number of entities in room r_1 is checked. If room r_1 is currently empty, then whether entities e and i have *not sharing* or not is checked. If none of them has the constraint, then no update is required. If only one of them has the constraint, then a single penalty reduction is applied because one violation no longer occurs when entity i is moved to room r_1. If both entities i and e have the constraint, then double penalty reduction is required, because two constraint violations no longer can happen.

Stage 2 - Updating the cost changes of moving entities to room r_2:   Entity e is moved to room r_2. The algorithm iterates through each other entity i and checks the room it is located in. There are again two sub-conditions in this stage:

- This condition is given at lines 3-17 of Figure B.5. Entity i is already in room r_2. In this case, if the number of entities in room r_2 is currently 2, and either entity i or e have a *not sharing* constraint, then a penalty reduction is applied. If both

142

entities in room r_2 have *not sharing* constraint, then an additional penalty reduction is applied. This is due to the elimination of one or two previous violation of moving i to r_2. If there are three entities in room r_2 instead and the third entity has a *not sharing* constraint on itself, then a penalty increase is necessary due to a potential violation caused by moving i to room r_2. After the necessary change is calculated, the algorithm iterates through each room j and adds the change to each Δ_{ij} location.

- This condition is given at lines 19-30 of Figure B.5. Entity i is not in room r_2. Again, the number of entities in room r_2 is checked. If there is currently only a single entity (which is e) with *not sharing* constraint and entity i does not have the constraint, then penalty increase is necessary because moving i to room r_2 will now cause a penalty. If entity i has a *not sharing* constraint on it, one or two steps of penalty increase is necessary depending on whether e has the constraint as well. If room r_2 has another entity besides e, then depending on the possible constraint on that entity, a penalty reduction is possible.

Update Algorithm for Adjacency, Nearby and Away from Constraints

The update algorithms for *adjacency* and *nearby* constraint are the same except from the lists that are used during the algorithm. Adjacency lists A are used in the update algorithm for *adjacency* whereas near lists N are used in *nearby* update algorithm.

The pseudo-codes for the update algorithms for *adjacency*, *nearby* and *away from* are given in Figures B.6, B.7, and B.8 respectively.

Regardless of the constraint type, four conditions are considered:

- Entities e_1 and e_2 were previously in adjacent (or near) rooms. After the *relocate* move, they are still in adjacent (or near) rooms. In this case, all the rooms that are adjacent (or near) to the previous room of e_1 are checked, and the cost changes associated with e_2 and these rooms go through a penalty reduction. The Δ locations associated with e_2 and the rooms that are adjacent (or near) to the current room of e_1 need a penalty increase. This condition is given at lines 1-7 of Figures B.6, B.7, and B.8 for *adjacency*, *nearby* and *away from* constraints respectively.

- Entities e_1 and e_2 were previously in adjacent (or near) rooms. After the *relocate* move, they are now not in adjacent (or near) rooms. In this case, the cost change locations associated with e_2 and all the rooms need a penalty reduction. If any of

the rooms are adjacent (or near) to the current room of e_1 another penalty reduction is required in that specific location in Δ matrix. If any of the rooms is adjacent (or near) to the previous room of e_1, then a penalty increase is applied to that individual location in Δ matrix. This condition is given at lines 8-16 of Figures B.6, B.7, and B.8 for *adjacency*, *nearby* and *away from* constraints respectively.

- Entities e_1 and e_2 were previously not in adjacent (or near) rooms. After the *relocate* move, they are now in adjacent (or near) rooms. In this case, the locations associated with e_2 and all rooms need a penalty increase (opposite case of the second condition). However, the increase and reduction of penalty in specific locations related to e_2 and certain rooms are adjusted in exactly the same way as in second condition. This condition is given at lines 17-25 of Figures B.6, B.7, and B.8 for *adjacency*, *nearby* and *away from* constraints respectively.

- Entities e_1 and e_2 were previously not in adjacent (or near) rooms. After the *relocate* move, they are still not in adjacent (or near) rooms. This condition is treated the same way as the first condition. This condition is given at lines 26-32 of Figures B.6, B.7 and B.8 for *adjacency*, *nearby*, and *away from* constraints respectively.

The *away from* constraint is the opposite of *nearby* constraint. The same conditions are checked as in *nearby* constraint update function; however, the penalty reductions and increases in modified locations are reversed.

Update Algorithm for Capacity Constraint

The update algorithm for the *capacity constraint* has to consider the changes associated with each entity. Each entity is affected by the *relocate* move (Entity e being moved from room r_1 to r_2).

The pseudo-code for the update algorithm for *capacity* constraint is given in Figures B.11, B.9, and B.10. In order to simplify this pseudo-code, hard and soft constraint penalties are taken as equal and simply referred to as $penalty_{cp}$.

The update algorithm iterates through each entity i and checks the room i is allocated to. There are four cases the algorithm has to consider:

- This condition is given at lines 1-12 of Figure B.9. Entity i is allocated to room r_1 and r_1 has a *capacity* constraint on it. First, the space misuses in room r_1 before and after the *relocate* move are checked. An overuse situation in room r_1 may still continue after the *relocate* move operation. The space required for entity i may

be between previous overuse and current overuse values in room r_1. In this case, the algorithm iterates through each room r_j except room r_1 and decreases the cost change in each location Δ_{ij}. If a previous overuse situation is rectified due to entity e being moved out of r_1, then the algorithm checks whether the previous overuse is less than the space required for entity i. If this is the case, then the algorithm increases cost changes in each location Δ_{ij} for each room j except room r_1.

- This condition is given at lines 13-24 of Figure B.9. Entity i is allocated to room r_2 which has a *capacity* constraint on it. This case is similar to the first one. There may be an ongoing overuse situation before and after the move. However, since entity e is moved to room r_2, the overuse of room r_2 is now greater. If the space required for entity i is less than the current overuse amount but greater than or equal to the previous overuse, then the locations Δ_{ij} for each room j except room r_2 need a cost increment. Conversely, there might not be an overuse situation before the move but there might be one due to entity e is being moved to room r_2. In this case, the locations Δ_{ij} for each room j except r_2 are decreased if the current overuse is less than or equal to the space required for the entity i.

- This condition is given at lines 1-12 of Figure B.10. Entity i is not allocated to room r_1 or r_2. The first sub-case involves sending entity i to room r_1. The space misuses of the room r_1 before and after the *relocate* move are calculated. If there is currently no misuse of space, then sending i to room r_1 will cause an overuse, therefore, the cost change in location Δ_{ir_1} should be increased. If there is an underuse situation occurring after the overuse of the room is rectified by the removal of entity e, then the cost change in location Δ_{ir_1} should be increased if the space required for entity i is greater than the current underuse. If room r_1 is underused before and after the move and the space required for entity i is between the current and previous underuse, then the cost change in location Δ_{ir_1} should be decreased.

- This condition is given at lines 13-24 of Figure B.10. Entity i is not allocated to room r_1 or r_2. This sub-case involves sending entity i to room r_2. The space misuses of the room r_2 before and after the *relocate* move are calculated. If there was no space misuse in room r_1 before the *relocate* move, then the cost change in location Δ_{ir_2} should be decreased. There might be a case where room r_2 was previously underused but now overused due to entity e being moved into this room. In this case, the cost change in location Δ_{ir_2} is decreased if the previous underuse of r_2 is less than the space required for entity i. It is also possible for the underuse situation in room r_2 still persisting even if entity e is moved into this

room. The cost change in location Δ_{ir_2} should be increased if the space required for entity i lies between the current and previous underuse in room r_2.

6.4.4 Partial Local Search

During the local search stage, the Δ matrix is calculated first. At each one of the h iterations, a random subset E' of entities is selected, and the minimum location (e, r) is searched in that subsection of the Δ matrix. The divisor parameter d paramater adjusts the greediness of this search. The size of the explored neighbourhood is $|E|/d$. When $d = 1$, the whole neighbourhood is checked. When $d = 2$ or $d = 3$, one half or one third of the search space is explored respectively. In practice, poor results are observed with a very greedy local search when the whole neighbourhood is checked, that is why additional experiments with different values of d are performed.

Remember that in the Δ matrix, the columns represent the entities, while the rows represent the rooms. In the default algorithm, search over the Δ matrix is based on columns i.e some of the entities are going to be chosen first and the cost changes of sending these entities to other rooms are considered later. This corresponds to the approach of finding the best room to allocate the chosen entity. Alternatively, the rows in the Δ matrix can be chosen instead. In this case, the cost changes of sending each entity to that specific rooms are considered. This corresponds to the approach of finding the best entity that can be sent to that specific chosen room. A third alternative is randomly switching between columns (entity selection) and rows (room selection) at each *relocate* move operation of the local search. This enables the search switch between finding the best entity for a specific room, or finding the best room for a specific entity at different iterations.

6.4.5 Application of Tabu Search

As an alternative to the local search stage in *ELS*, a tabu search (*TS*) [Glover, 1989] algorithm was also implemented to replace this stage. *TS* algorithm utilises a data structure called a *tabu list* to prevent certain moves from being carried out to prevent the search getting stuck in local optima. In the current implementation, whenever an entity e is moved from room r_1 to r_2, the moves that send the entity back to r_1 are considered *tabu* and hence forbidden until the end of that specific local search stage. However, if a tabu move produces the best obtained solution encountered during the entire search, the move is still made (*aspiration criteria*). Another consideration in designing a TS algorithm is the management of the tabu tenure (that how long a tabu condition is

going to be kept in tabu lists). In this work, the tabu tenure is restricted to the number of tabu search iterations after each crossover. Another method to populate tabu lists within an integer-programming and tabu search combination is going to be explained in Chapter 7.

6.5 Experiments Related to Evolutionary Local Search

The algorithm is implemented using the Microsoft Visual Studio 2010 C++ compiler. All experiments are carried out on a Windows PC with Intel Core 2 Duo E8400 (3 Ghz) processor.

In the experiments, the $SVe150$ and $PNe150$ datasets designed in Chapter 2 are used. For several tests, a subset of instances are used. These instances are $S_{0.00}V_{0.00}$, $S_{0.40}V_{0.80}$, $S_{0.80}V_{0.40}$, $P_{0.00}N_{0.00}$, $P_{0.10}N_{0.20}$, and $P_{0.20}N_{0.10}$.

Experiments were carried out in order to determine the best parameter values and operators. There are itemised below. The sections that these tested methods are described before are given between the parenthesis.

- Ordering of local search described in Section 6.4.4 and tested in Section 6.5.1.

- Different crossover methods described in Section 6.3.1 and tested in Section 6.5.2.

- Random mutation rate (m) described in Section 6.3.2 and tested in Section 6.5.3.

- Number of local search iterations (h) after each mutation described in Section 6.4.4 and tested in Section 6.5.3.

- Divisor value (d) which adjusts the size of the neighbourhood explored by the local search in each iteration described in Section 6.4.4 and tested in Section 6.5.4.

- Population size (ps) described in Section 6.3 and tested in Section 6.5.5.

- Comparison of using hill climbing local search and tabu search described in Section 6.4.5 and tested in Section 6.5.6.

In addition to the tests of these individual components of ELS, complete results on $SVe150$ and $PNe150$ datasets are given in Section 6.5.7. ELS algorithm is compared to the integer programming models described in Chapter 4 in Section 6.5.8. ELS is also compared to the local search heuristic described in Chapter 5 in Section 6.5.9.

In these experiments, each instance is given 20 runs of 90 seconds each for a total execution time of 30 minutes. Our base parameter values are $m = 0.03$, $h = 100$, $d = 3$,

and $ps = 20$ unless one of them is changed in its respective test. In the proposed *ELS* algorithm, the default operators are entity based ordering (ENT_{HC}) during the search of Δ matrix, one-point crossover (*1-PTX*), random and *allocation* mutations.

In the following experiments, the objective function for total penalty (*TP*) is taken as the weighted summation of space misuse (*SMP*) and soft constraint violation penalties (*SCP*). The respective formulations for *SMP*, *SCP*, and *TP* were previously given in equations 2.3.1, 2.3.2, and 2.3.3 in Section 2.3.2. This objective function is going to be minimised subject to *hard* constraints. The numbers of *hard* and *soft* constraints for *nott1*, *SVe*150, and *PNe*150 instances were previously given in Table 2.2 (Section 2.4) and Table 2.4 (Section 2.5.3) in Chapter 2 respectively.

6.5.1 Effect of Orderings of Local Search

In this test, the effect of using different orderings of local search while searching for a best move within a sub-section of the Δ matrix is investigated. For local search operation, the hill climbing and tabu search algorithms are used. There are four different orderings of searching through the Δ matrix considered in this chapter. In $BEST_{HC}$ and $BEST_{TS}$, the whole Δ table is searched using hill climbing and local search algorithms respectively. This ordering looks for the best local *entity-room* pairing for that *relocate* move. In ENT_{HC} and ENT_{TS}, a subset of entities of size $|E|/d$ is chosen first, and the best room the entity can be sent to is searched. The room based ordering is performed in $ROOM_{HC}$ and $ROOM_{TS}$, a subset of rooms of size $|R|/d$ is randomly chosen, and the best entity that can be sent to one of these rooms is searched. The methods $MIXED_{HC}$ and $MIXED_{TS}$ alternate between entity and room based orderings at each *relocate* move.

The results after 20 runs of 90 seconds (average best objective function value μ, standard deviation σ, minimum and maximum values) on six different *SVe*150 and *PNe*150 instances are given in Table 6.1a and 6.1b, respectively. The performances of different ordering methods were similar due to the close average objective function values and rather large standard deviations. However, certain trends could still be observed. The tabu search version of the orderings were usually worse than the hill climbing versions with the exception of *BEST* orderings that searched the whole Δ table. In this special case, $BEST_{TS}$ was clearly superior to $BEST_{HC}$. Most successful orderings were $MIXED_{HC}$ and ENT_{HC} while $ROOM_{HC}$ trailed slightly behind. It was observed that the approach of "searching the best room for specific entities" was slightly better than "searching the best entity for specific rooms". Hybridisation of both approaches

instance	order	μ	σ	min	max
$S_{0.00}V_{0.00}$	$BEST_{HC}$	77.68	24.07	47.00	139.50
	$BEST_{TS}$	69.53	18.75	48.00	106.00
	ENT_{HC}	53.63	17.99	19.50	97.00
	ENT_{TS}	64.63	23.35	34.50	122.00
	$ROOM_{HC}$	50.78	15.31	22.00	84.00
	$ROOM_{HC}$	60.23	21.73	22.50	118.00
	$MIXED_{HC}$	**49.33**	18.11	**12.00**	90.50
	$MIXED_{TS}$	65.85	17.80	36.50	102.00
$S_{0.40}V_{0.80}$	$BEST_{HC}$	261.06	23.23	223.40	314.00
	$BEST_{TS}$	254.49	18.30	220.80	284.70
	ENT_{HC}	233.62	23.27	**190.70**	293.00
	ENT_{TS}	248.50	15.01	209.00	268.30
	$ROOM_{HC}$	242.92	22.97	205.10	297.60
	$ROOM_{HC}$	235.81	19.06	196.90	264.00
	$MIXED_{HC}$	**228.61**	11.38	210.90	247.80
	$MIXED_{TS}$	254.46	22.35	215.00	295.50
$S_{0.80}V_{0.40}$	$BEST_{HC}$	220.43	25.17	183.70	262.40
	$BEST_{TS}$	211.29	15.30	187.20	235.40
	ENT_{HC}	200.34	15.03	168.90	222.80
	ENT_{TS}	211.45	19.84	188.20	260.00
	$ROOM_{HC}$	203.37	15.43	178.90	231.70
	$ROOM_{HC}$	200.64	15.12	177.00	233.40
	$MIXED_{HC}$	**191.36**	16.70	**163.60**	236.90
	$MIXED_{TS}$	206.19	12.72	183.50	233.00

(a) $SVe150$ instances

instance	order	μ	σ	min	max
$P_{0.00}N_{0.00}$	$BEST_{HC}$	135.78	27.09	89.00	206.00
	$BEST_{TS}$	124.83	21.95	80.00	162.00
	ENT_{HC}	**108.35**	18.89	74.00	145.50
	ENT_{TS}	123.03	17.21	90.00	165.50
	$ROOM_{HC}$	119.70	16.12	88.00	157.00
	$ROOM_{HC}$	125.10	27.40	77.50	197.50
	$MIXED_{HC}$	114.98	18.62	**71.50**	153.50
	$MIXED_{TS}$	120.30	20.23	75.50	158.50
$P_{0.10}N_{0.20}$	$BEST_{HC}$	257.82	21.37	215.10	292.00
	$BEST_{TS}$	244.06	21.67	212.40	291.50
	ENT_{HC}	227.19	19.08	201.30	273.80
	ENT_{TS}	253.17	21.93	215.10	287.60
	$ROOM_{HC}$	228.81	13.52	209.90	266.50
	$ROOM_{HC}$	231.82	14.26	206.60	259.10
	$MIXED_{HC}$	**225.03**	12.78	207.60	249.60
	$MIXED_{TS}$	234.50	21.09	**198.20**	279.40
$P_{0.20}N_{0.10}$	$BEST_{HC}$	202.45	22.06	169.20	269.40
	$BEST_{TS}$	188.68	25.63	151.60	246.60
	ENT_{HC}	166.33	14.56	140.50	199.40
	ENT_{TS}	190.39	14.96	165.80	221.80
	$ROOM_{HC}$	176.84	18.44	152.50	220.30
	$ROOM_{HC}$	172.47	23.15	142.30	218.90
	$MIXED_{HC}$	**162.92**	16.17	**137.80**	202.10
	$MIXED_{TS}$	179.08	15.37	152.10	211.60

(b) $PNe150$ instances

Table 6.1: Impacts of different order of local search on instances $S_{0.00}V_{0.00}$, $S_{0.40}V_{0.80}$, $S_{0.80}V_{0.40}$, $P_{0.00}N_{0.00}$, $P_{0.10}N_{0.20}$, and $P_{0.20}N_{0.10}$

still worked well due to the synergy of using both approaches.

6.5.2 Effect of Using Different Crossover Operators

In this section, the effect of using different crossover operators described in Section 6.3.1 on the solution quality is investigated. For this, three traditional crossover operators and four problem specific crossovers described in Section 6.3.1 were tested. These crossover operators are as follows:

- *1-PTX* One point crossover.

- *2-PTX* Two point crossover.

- *UX* Uniform crossover.

- *RX-L* Linear room based crossover.

- *RX-P* Permutated room based crossover.

- *FX-L* Linear floor based crossover.

- *FX-P* Permutated floor based crossover.

Each crossover was tested on six different $SVe150$ and $PNe150$ instances. Each instance was given 20 runs of 90 seconds each. The results are given in Table 6.2a and 6.2b for $SVe150$ and $PNe150$ instances respectively. Columns μ, σ, min and max represent the best average result, standard deviation, minimum and maximum results obtained over 20 runs.

instance	crossover	μ	σ	min	max
	1-PTX	53.63	17.99	19.50	97.00
	2-PTX	56.30	15.93	35.50	89.00
	UX	54.63	14.30	30.00	84.00
$S_{0.00}V_{0.00}$	RX-L	55.70	18.27	31.50	105.50
	RX-P	56.88	22.34	32.50	124.50
	FX-L	**48.78**	**17.78**	19.50	105.50
	FX-P	55.75	14.74	24.00	80.50
	1-PTX	233.62	23.27	190.70	293.00
	2-PTX	233.77	15.15	208.60	263.40
	UX	237.67	15.01	216.20	267.90
$S_{0.40}V_{0.80}$	RX-L	**231.75**	26.66	199.20	287.20
	RX-P	235.68	22.36	200.80	276.70
	FX-L	246.65	19.23	216.50	289.70
	FX-P	236.21	21.31	**197.50**	269.80
	1-PTX	200.34	15.03	168.90	222.80
	2-PTX	208.66	24.04	175.70	256.60
	UX	195.51	18.12	**162.80**	225.40
$S_{0.80}V_{0.40}$	RX-L	201.15	16.11	174.90	232.50
	RX-P	**193.52**	14.59	160.20	227.30
	FX-L	199.23	15.94	166.80	232.40
	FX-P	196.10	11.27	169.00	218.10

(a) $SVe150$ instances

instance	crossover	μ	σ	min	max
	1-PTX	**108.35**	18.89	74.00	145.50
	2-PTX	117.33	22.90	87.50	179.50
	UX	111.60	14.09	89.00	136.00
$P_{0.00}N_{0.00}$	RX-L	112.25	17.98	**74.50**	143.50
	RX-P	119.78	21.73	77.00	156.00
	FX-L	116.48	15.97	92.50	153.00
	FX-P	113.73	17.50	86.00	147.00
	1-PTX	228.09	18.92	201.30	273.80
	2-PTX	227.86	16.24	204.60	261.10
	UX	225.41	15.13	**196.40**	253.70
$P_{0.10}N_{0.20}$	RX-L	229.14	13.93	203.90	253.30
	RX-P	231.21	18.55	203.60	267.30
	FX-L	225.46	22.36	200.30	271.70
	FX-P	**224.30**	21.24	200.00	272.50
	1-PTX	165.60	15.06	140.50	199.40
	2-PTX	172.10	23.58	142.00	246.90
	UX	**163.04**	14.63	**137.50**	188.00
$P_{0.20}N_{0.10}$	RX-L	174.66	17.79	149.20	214.40
	RX-P	165.69	13.23	143.10	191.40
	FX-L	165.71	17.38	141.90	220.00
	FX-P	168.09	21.35	143.80	231.30

(b) $PNe150$ instances

Table 6.2: Impacts of different crossover types on instances $S_{0.00}V_{0.00}$, $S_{0.40}V_{0.80}$, $S_{0.80}V_{0.40}$, $P_{0.00}N_{0.00}$, $P_{0.10}N_{0.20}$, and $P_{0.20}N_{0.10}$

As can be observed from the close averages and rather large standard deviations, no crossover operator could be deemed superiour to others. The best average and the minimum results were distributed evenly among the six instances without any observable pattern. Three claims can be made why problem specific crossovers performed not significantly better than traditional crossovers:

- Since the *ELS* algorithm contains a rather greedy local search procedure, the crossover operator lacks importance, does not really need to preserve the information in the parents and works more like a mutation operator.

- Preservation of room allocations as in *RX-L* and *RX-P* is not very important due to the nature of the OSA instances tested. In most OSA instances, the final solution is an entity-room mapping where the majority of the rooms contains just one entity (especially due to *not sharing* constraint). Since a room does not contain too many entities in it, the allocations in rooms are not likely to be disrupted even using traditional crossover operators.

- Due to the low genetic diversity in the population, the children will be similar to the parents regardless of crossover operator used.

6.5.3 Effect of Mutation Rate and Local Search Iterations

Four different mutation rates ($m = 0.00$, $m = 0.01$, $m = 0.03$, and $m = 0.05$) and four different values for number of local search move operations after each mutation ($h = 50$, $h = 100$ $h = 250$, and $h = 500$) were tested. The mutation rates are given as percentages. For example if $m = 0.03$, three percent of the entities will be probabilistically relocated on the average. Since a large mutation rate might cause a large disruption on the solution, there might be a need for a longer local search afterwards to obtain better results. Conversely, a small perturbation on the solution can be corrected with a shorter local search. Therefore, a large perturbation and longer local search afterwards will work as a *diversification* of the search. On the other hand, small perturbation and shorter local search afterwards can be seen as an *intensification* strategy.

Results for mutation rate and local search iterations are given in Table 6.3a and 6.4a for $SVe150$ instances and in Table 6.3b and 6.4b for $PNe150$ instances respectively. Columns m and h represent the mutation rate and the number of local search iterations respectively. Columns μ, σ, min and max represent the average best objective value, the standard deviation on the best objective value, the minimum and maximum objective value obtained after 20 runs respectively.

It was observed that the mutation was essential to the success of the algorithm; without random mutation (the case $m = 0.00$), the algorithm quickly got stuck in a local optimum and produced uncompetitive results. We observed that the algorithm worked best when the mutation rate was around ($m = 0.01 - 0.03$), although $m = 0.05$ still yielded acceptable results. It was observed that the local search iterations should be kept quite low (usually $h = 50$, $h = 100$) for the best results to be obtained. Increasing h beyond 500 affected the performance of the algorithm adversely, therefore, using a long extensive local search after each crossover and mutation operator is not recommended. Another benefit of using a lower h is that it reduced the effect of us-

instance	m	μ	σ	min	max
$S_{0.00}V_{0.00}$	0.00	677.35	200.09	241.50	1107.50
	0.01	64.35	19.42	38.50	111.00
	0.03	**51.85**	20.91	24.00	92.00
	0.05	52.33	18.84	**22.50**	84.50
$S_{0.40}V_{0.80}$	0.00	755.83	188.12	412.20	1158.10
	0.01	239.47	20.13	211.50	278.30
	0.03	**230.49**	14.90	**205.50**	253.20
	0.05	243.77	22.11	208.90	286.30
$S_{0.80}V_{0.40}$	0.00	841.56	209.57	505.50	1242.10
	0.01	199.24	16.64	**171.30**	249.00
	0.03	**198.99**	17.38	171.60	239.60
	0.05	204.43	15.51	186.40	243.90

(a) *SVe*150 instances

instance	m	μ	σ	min	max
$P_{0.00}N_{0.00}$	0.00	760.75	229.60	297.50	1046.50
	0.01	111.25	19.22	82.00	146.50
	0.03	116.88	23.52	**78.00**	161.50
	0.05	**110.93**	17.74	79.00	141.50
$P_{0.10}N_{0.20}$	0.00	787.31	154.35	574.30	1059.60
	0.01	228.22	12.25	209.30	256.50
	0.03	**226.18**	16.94	**196.10**	261.70
	0.05	231.74	16.07	207.00	275.20
$P_{0.20}N_{0.10}$	0.00	688.20	215.08	353.70	1032.90
	0.01	173.17	19.82	**143.80**	207.50
	0.03	**167.27**	17.14	145.30	205.70
	0.05	178.24	17.43	144.50	215.70

(b) *PNe*150 instances

Table 6.3: Impacts of different mutation rates (m) on instances $S_{0.00}V_{0.00}$, $S_{0.40}V_{0.80}$, $S_{0.80}V_{0.40}$, $P_{0.00}N_{0.00}$, $P_{0.10}N_{0.20}$, and $P_{0.20}N_{0.10}$

ing different mutation rates between $m = 0.01$ and $m = 0.05$, therefore, setting the mutation rate precisely became less important.

instance	h	μ	σ	min	max
$S_{0.00}V_{0.00}$	50	56.43	15.42	28.00	87.50
	100	**51.85**	20.91	24.00	92.00
	250	56.33	22.01	**13.50**	110.50
	500	68.80	15.10	46.00	105.00
$S_{0.40}V_{0.80}$	50	236.48	21.74	**193.60**	291.40
	100	**230.49**	14.90	205.50	253.20
	250	245.78	18.88	204.00	279.80
	500	240.98	25.35	206.20	292.20
$S_{0.80}V_{0.40}$	50	**197.94**	19.93	**171.00**	232.10
	100	198.99	17.38	171.60	239.60
	250	207.67	18.86	183.40	258.30
	500	212.96	15.25	183.60	235.10

(a) *SVe*150 instances

instance	h	μ	σ	min	max
$P_{0.00}N_{0.00}$	50	117.78	18.23	92.50	168.50
	100	**116.88**	23.52	**78.00**	161.50
	250	127.98	23.61	87.50	175.00
	500	126.43	18.42	97.50	166.50
$P_{0.10}N_{0.20}$	50	**225.84**	12.42	207.20	246.70
	100	226.18	16.94	**196.10**	261.70
	250	240.57	24.55	216.40	298.70
	500	245.79	23.06	204.00	278.10
$P_{0.20}N_{0.10}$	50	174.15	17.96	**133.50**	208.50
	100	**167.27**	17.14	145.30	205.70
	250	171.14	15.78	146.50	207.00
	500	176.58	17.13	145.50	201.10

(b) *PNe*150 instances

Table 6.4: Impacts of different local search iterations (h) on instances $S_{0.00}V_{0.00}$, $S_{0.40}V_{0.80}$, $S_{0.80}V_{0.40}$, $P_{0.00}N_{0.00}$, $P_{0.10}N_{0.20}$, and $P_{0.20}N_{0.10}$

6.5.4 Effect of Local Search Size

The idea of dividing the whole search space in local search was previously explained in Section 6.4.4. Table 6.5a and Table 6.5b depict the effect of using different d (divisor) values on the *SVe*150 and *PNe*150 instances. Five different values were tested: $d = 1$ (100%), $d = 2$ (50%), $d = 3$ (33%), $d = 4$ (25%), and $d = 5$ (20%). One immediate obser-

vation was that searching for the whole single-move neighbourhood ($d = 1$) greedily produced the worst results in all of the instances. The algorithm had a tendency to get stuck in a local optima when $d = 1$ (although not severely as with the no mutation case); however, by simply increasing d (and hence decreasing the greediness of the search), the algorithm gave the best results when d was around 3 (33.3% of the Δ matrix searched at each iteration). However, this could be attributed to the size of a specific instance and our implementation of the algorithm. During the profiling tests of our code using the Visual Studio 2010 Profiler tool, it was observed that the majority of the execution time was spent in two sections of the algorithm: searching the Δ matrix for the best move and the update stage after making this best move. When $d = 3$, these two steps took roughly equal amount of time practically maximising the amount of iterations per unit time.

Consequently, our conjecture after these experiments is that d should be further increased when the size of the instance (the number of entities times the number of rooms) is increased because the time required for searching the Δ matrix is inversely proportional to d while the time required for the update stage is just directly proportional to the number of constraints (and hence mostly constant).

instance	d	μ	σ	min	max
	1	83.50	26.26	49.00	147.00
	2	56.30	20.39	28.00	110.00
$S_{0.00}V_{0.00}$	3	**51.40**	20.58	24.00	92.00
	4	57.18	14.89	**23.50**	85.00
	5	65.65	19.99	35.50	127.50
	1	273.77	18.38	233.10	302.90
	2	231.15	18.67	**202.20**	265.30
$S_{0.40}V_{0.80}$	3	**230.35**	14.93	205.50	253.20
	4	244.49	21.66	213.50	295.70
	5	259.57	29.65	205.90	321.90
	1	228.49	24.44	194.90	276.40
	2	203.68	16.51	184.20	238.30
$S_{0.80}V_{0.40}$	3	**199.44**	17.60	171.60	239.60
	4	208.61	18.79	184.80	254.30
	5	205.72	23.95	**171.40**	253.80

(a) $SVe150$ instances

instance	d	μ	σ	min	max
	1	134.93	14.00	108.00	159.00
	2	118.03	13.06	91.00	138.50
$P_{0.00}N_{0.00}$	3	**117.45**	23.10	**78.00**	161.50
	4	132.10	20.96	98.00	173.50
	5	120.33	19.34	82.00	146.50
	1	256.54	16.68	217.20	288.50
	2	229.00	19.64	201.80	275.80
$P_{0.10}N_{0.20}$	3	**225.96**	17.05	**196.10**	261.70
	4	236.48	16.07	206.90	266.00
	5	246.22	21.03	200.30	291.80
	1	215.03	23.89	172.00	255.60
	2	178.51	26.73	144.40	228.80
$P_{0.20}N_{0.10}$	3	**167.12**	16.80	145.30	202.70
	4	174.12	16.64	149.90	212.90
	5	169.94	20.07	**140.60**	212.30

(b) $PNe150$ instances

Table 6.5: Impacts of different divisor values (d) on instances $S_{0.00}V_{0.00}$, $S_{0.40}V_{0.80}$, $S_{0.80}V_{0.40}$, $P_{0.00}N_{0.00}$, $P_{0.10}N_{0.20}$, and $P_{0.20}N_{0.10}$

6.5.5 Effect of Population Size

The final parameter tests were performed on ps, the number of solutions in a population. The population sizes were chosen as $ps = 5$, $ps = 10$, $ps = 20$, $ps = 50$, and $ps = 100$. The results for some $SVe150$ and $PNe150$ instances are given in Table 6.6a and Table 6.6b respectively. It was observed that for the average best objective function value μ, there was not any statistically significant difference between any of the ps values. However, for $ps = 20$ solutions, it was observed that the chance of finding a best minimum value after 20 runs was slightly higher. The main benefit of using a population-based algorithm was to keep a number of different solutions so that the algorithm has a chance to backtrack from a local optimum to another section of the search space by means of crossover and mutation operators.

instance	ps	μ	σ	min	max
	5	52.63	16.11	33.00	98.00
	10	58.30	16.68	31.00	82.50
$S_{0.00}V_{0.00}$	20	51.40	20.58	24.00	92.00
	50	63.28	24.99	28.50	109.50
	100	**49.80**	20.34	**16.50**	99.50
	5	237.45	14.21	202.80	268.60
	10	241.33	16.90	206.40	278.10
$S_{0.40}V_{0.80}$	20	**230.94**	14.98	206.40	253.20
	50	246.35	23.00	**200.40**	290.20
	100	245.74	17.79	214.80	272.30
	5	203.72	16.59	182.70	239.10
	10	201.01	13.82	177.90	226.90
$S_{0.80}V_{0.40}$	20	**199.34**	17.49	**171.60**	239.60
	50	208.02	17.95	173.80	247.00
	100	204.41	16.69	185.70	239.30

(a) $SVe150$ instances

instance	ps	μ	σ	min	max
	5	117.23	13.52	97.50	146.50
	10	119.40	13.85	95.00	146.00
$P_{0.00}N_{0.00}$	20	116.88	23.52	**78.00**	161.50
	50	118.43	25.58	79.00	173.00
	100	**116.10**	15.39	87.50	140.50
	5	226.54	16.28	201.30	261.80
	10	223.87	12.78	198.50	248.70
$P_{0.10}N_{0.20}$	20	**226.18**	16.94	**196.10**	261.70
	50	226.88	14.31	205.80	261.40
	100	232.62	17.66	202.80	273.40
	5	167.77	16.53	140.20	195.30
	10	**162.21**	12.14	**139.50**	190.60
$P_{0.20}N_{0.10}$	20	167.27	17.14	145.30	205.70
	50	167.85	12.94	141.90	189.90
	100	165.65	16.57	144.00	207.40

(b) $PNe150$ instances

Table 6.6: Impacts of different population (ps) sizes on instances $S_{0.00}V_{0.00}$, $S_{0.40}V_{0.80}$, $S_{0.80}V_{0.40}$, $P_{0.00}N_{0.00}$, $P_{0.10}N_{0.20}$, and $P_{0.20}N_{0.10}$

6.5.6 Tabu Search vs Basic Local Search

In this section, hill climbing based local search method described in Section 6.4 is replaced with tabu search (described in Section 6.4.5) within the evolutionary local search algorithm.

Table 6.7 represents the results obtained using tabu search. Columns μ, σ, and min give the average best objective value, the standard deviation on the best objective value and the minimum best objective value obtained after 20 runs of 90 seconds each.

It was observed that replacing the local search with the tabu search offered no benefits for obtaining better average or minimum results. After using a small population, crossover, and mutation operators and a divisor parameter to adjust the greediness of the search, adding a *tabu list* structure did not offer any additional benefit over the local search. A likely explanation for this might be the restriction of traversable search space due to tabu moves. The search might have to go through some of the solutions restricted by tabu moves in order to reach higher quality solutions.

	Local Search			Tabu Search		
instance	μ	σ	**min**	μ	σ	**min**
$S_{0.00}V_{0.00}$	**53.65**	18.16	**19.50**	64.40	23.38	34.50
$S_{0.40}V_{0.80}$	**233.62**	23.27	**190.70**	248.48	14.99	209.00
$S_{0.80}V_{0.40}$	**200.34**	15.03	**168.90**	211.59	20.05	188.20
$P_{0.00}N_{0.00}$	**108.65**	19.55	**74.00**	123.03	17.21	90.00
$P_{0.10}N_{0.20}$	**227.01**	18.83	**201.30**	253.17	21.93	215.10
$P_{0.20}N_{0.10}$	**165.25**	14.78	**140.50**	190.22	14.97	165.80

Table 6.7: Results on several *SVe*150 and *PNe*150 instances using local and tabu search within the *ELS*

Tabu search algorithm will be re-evaluated in a different setting (within an integer programming - tabu search heuristic combination) in Chapter 7.

6.5.7 Complete Results on SVe150 and PNe150 Datasets

By using the parameters obtained after experimentation on a subset of *SVe*150 and *PNe*150, additional experiments were carried out on the full set of instances. The following parameters were used: mutation rate $m = 0.03$, population size $ps = 20$, the number of local search iterations $h = 100$ and the divisor value of the local search $d = 3$. Entity based ordering with hill climbing (ENT_{HC}) was used in these experiments. Traditional one-point crossover and one-point random mutation were used as crossover and mutation operators respectively.

Tables 6.8a and 6.8b represent the results obtained. Columns *S*, *V*, *P*, and *N* represent the four different parameters: *slack space rate*, *soft constraint violation rate*, *positive*, and *negative slack rates* respectively. Columns μ and σ represent the average and standard deviation of the total penalty obtained after 10 runs respectively. Columns *SMP* and *SCP* give the average space misuse and soft constraint violation penalty after 10 runs respectively. Column *min* gives the minimum total penalty obtained after 10 runs.

S	V	μ	σ	SMP	SCP	min	P	N	μ	σ	SMP	SCP	min
0.00	0.00	49.30	15.11	39.30	10.00	31.50	0.00	0.00	103.75	14.60	45.75	58.00	79.00
0.00	0.20	90.80	27.30	52.80	38.00	61.50	0.00	0.05	138.90	9.06	79.90	59.00	120.80
0.00	0.40	102.40	17.43	56.40	46.00	89.50	0.00	0.10	174.45	9.22	117.45	57.00	161.70
0.00	0.60	118.40	12.31	50.40	68.00	99.00	0.00	0.15	216.14	12.34	153.14	63.00	198.40
0.00	0.80	169.85	18.04	65.85	104.00	138.50	0.00	0.20	244.64	17.54	183.64	61.00	222.90
0.00	1.00	192.25	18.89	65.25	127.00	165.00	0.00	0.25	283.79	10.31	220.79	63.00	269.20
0.20	0.00	64.04	14.29	55.04	9.00	38.90	0.05	0.00	111.01	21.50	55.01	56.00	87.90
0.20	0.20	105.24	22.72	62.24	43.00	77.90	0.05	0.05	133.29	13.50	76.29	57.00	113.90
0.20	0.40	133.23	22.45	75.23	58.00	115.40	0.05	0.10	173.08	16.11	108.08	65.00	146.90
0.20	0.60	139.92	14.46	69.92	70.00	117.10	0.05	0.15	201.85	17.62	135.85	66.00	180.40
0.20	0.80	182.19	17.11	85.19	97.00	157.90	0.05	0.20	227.66	14.74	168.66	59.00	206.60
0.20	1.00	213.71	28.83	90.71	123.00	181.60	0.05	0.25	268.28	11.81	204.28	64.00	250.60
0.40	0.00	125.27	16.59	95.27	30.00	108.40	0.10	0.00	128.74	19.64	60.74	68.00	100.50
0.40	0.20	157.45	25.90	101.45	56.00	118.80	0.10	0.05	142.71	23.96	82.71	60.00	106.00
0.40	0.40	173.27	18.85	110.27	63.00	144.60	0.10	0.10	169.92	17.08	105.92	64.00	149.40
0.40	0.60	197.05	18.05	123.05	74.00	163.90	0.10	0.15	203.62	13.83	134.62	69.00	182.40
0.40	0.80	218.23	17.24	117.23	101.00	185.60	0.10	0.20	217.43	17.65	155.43	62.00	192.20
0.40	1.00	246.95	19.30	126.95	120.00	219.70	0.10	0.25	254.31	15.28	192.31	62.00	237.00
0.60	0.00	137.63	11.85	115.63	22.00	116.60	0.15	0.00	116.55	13.32	64.55	52.00	91.50
0.60	0.20	170.23	10.72	122.23	48.00	154.70	0.15	0.05	136.17	15.01	82.17	54.00	120.20
0.60	0.40	198.65	10.45	134.65	64.00	189.00	0.15	0.10	157.63	11.56	104.63	53.00	143.50
0.60	0.60	209.00	14.25	136.00	73.00	189.20	0.15	0.15	185.43	13.92	123.43	62.00	159.30
0.60	0.80	254.23	12.97	143.23	111.00	233.60	0.15	0.20	219.86	17.90	151.86	68.00	203.10
0.60	1.00	280.13	17.34	153.13	127.00	261.90	0.15	0.25	236.59	12.55	176.59	60.00	213.70
0.80	0.00	151.31	13.52	120.31	31.00	131.40	0.20	0.00	119.61	11.29	73.61	46.00	108.60
0.80	0.20	170.63	14.38	124.63	46.00	152.90	0.20	0.05	148.65	10.12	94.65	54.00	138.90
0.80	0.40	188.42	13.09	127.42	61.00	165.00	0.20	0.10	155.50	12.12	105.50	50.00	141.70
0.80	0.60	202.94	18.59	132.94	70.00	175.90	0.20	0.15	186.63	18.79	127.63	59.00	161.50
0.80	0.80	255.37	23.72	141.37	114.00	229.20	0.20	0.20	204.08	11.13	142.08	62.00	184.10
0.80	1.00	284.52	12.18	159.52	125.00	259.20	0.20	0.25	236.96	23.25	166.96	70.00	204.40
1.00	0.00	186.75	15.25	160.75	26.00	167.70	0.25	0.00	139.03	12.99	89.03	50.00	126.00
1.00	0.20	226.99	17.17	172.99	54.00	201.00	0.25	0.05	150.55	9.51	101.55	49.00	133.90
1.00	0.40	243.02	12.53	176.02	67.00	225.30	0.25	0.10	164.91	12.96	115.91	49.00	151.20
1.00	0.60	260.61	18.44	174.61	86.00	238.30	0.25	0.15	185.92	9.77	125.92	60.00	170.70
1.00	0.80	298.76	11.16	195.76	103.00	284.90	0.25	0.20	199.60	7.71	132.60	67.00	189.80
1.00	1.00	309.96	19.46	187.96	122.00	280.80	0.25	0.25	233.41	13.21	158.41	75.00	212.40

(a) *SVe*150 instances · · · · · · · · · · · · · · · · · **(b)** *PNe*150 instances

Table 6.8: Experimental results on the *SVe*150 and *PNe*150 dataset instances using the evolutionary local search algorithm

6.5.8 Comparison of Integer Programming Models and Evolutionary Local Search

In this section, we are going to analyse the difference in performance between the evolutionary local search algorithm (*ELS*) described in this chapter and the integer programming models presented in Chapter 4. The mathematical models without and with the floor variables are abbreviated as IP_1 and IP_2 respectively.

In Figures 6.7 and 6.8, minimum total penalty obtained for each instance with *ELS* is compared to the ones IP_1 and IP_2 using *SVe*150 and *PNe*150 datasets respectively. IP_1 or IP_2 and *ELS* were given equal amount of running time: *ELS* was given 10 runs (3 minutes each), while IP_1 and IP_2 were given a single run of 30 minutes.

The *x* and *y* axes represent *slack space rate* (*S*) and *violation rate* (*V*) in Figure 6.7 while they represent *positive* (*P*) and *negative* (*N*) *slack space rate* in Figure 6.8. In these contour plots, IP_1 or IP_2 is better than *ELS* in obtaining the minimum total penalty when a region becomes more blue or green respectively. *ELS* is better in grey regions. IP models and *ELS* algorithms are about equal in obtaining the minimum total penalty in white or pale coloured regions.

If IP_1 and *ELS* are compared in *SVe*150 instances in Figure 6.7a, it is observed that for lower *V* values where the soft constraint violation penalty is expected to be low, IP_1 perform better than *ELS* in obtaining minimum total penalty (blue regions). However, when the expected soft constraint violation penalty increases with larger *V*, *ELS* begins to perform better (grey regions). It is also observed that variation of the space misuse penalty over *S* parameter does not differentiate IP_1 or *ELS*.

For the *SVe*150 dataset, out of the 36 instances, *ELS* algorithm yields better results for 18, and the IP_1 formulation is more successful for the remaining 18 of them. However, *ELS* still achieves 5.38 penalty improvement per instance over IP_1.

ELS generally performs better than IP_1 in *PNe*150 instances (Figure 6.8a). For larger *N* values (where the instance is expected to have high overuse due to lack of capacity), *ELS* performs better, especially around $P = N = 0.15$ region. For some low *N* values, IP_1 provides slightly better results in a few instances (pale blue regions). For 24 instances, *ELS* performs better, for the other 12 instance IP_1 performs better than *ELS* in obtaining minimum total penalty. Average improvement per instance is around 9.

ELS is not competitive with IP_2 in obtaining the minimum total penalty for either *SVe*150 (Figure 6.7b) or *PNe*150 (Figure 6.8b) datasets. In both figures, the dominance

(a) Δ_{min} of $IP_1 - ELS$ **(b)** Δ_{min} of $IP_2 - ELS$
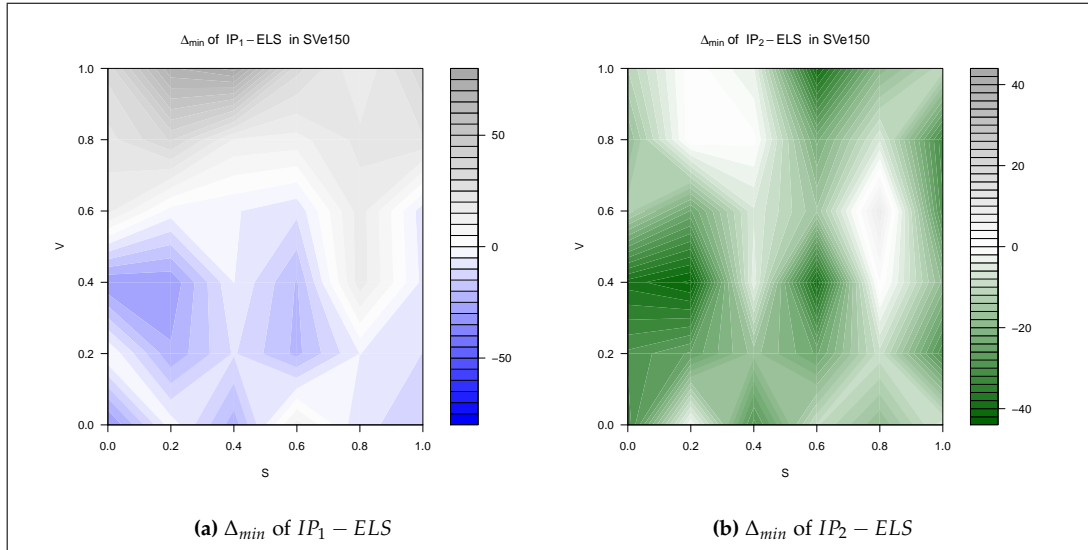
Figure 6.7: Differences in minimum penalty (Δ_{min}) after applying IP models without/with floor variables (IP_1 and IP_2) and evolutionary local search (ELS) on $SVe150$ dataset ($IP_1 - ELS$ and $IP_2 - ELS$). IP_1, IP_2 and ELS are represented by blue, green, and grey regions respectively.

(a) Δ_{min} of $IP_1 - ELS$ **(b)** Δ_{min} of $IP_2 - ELS$
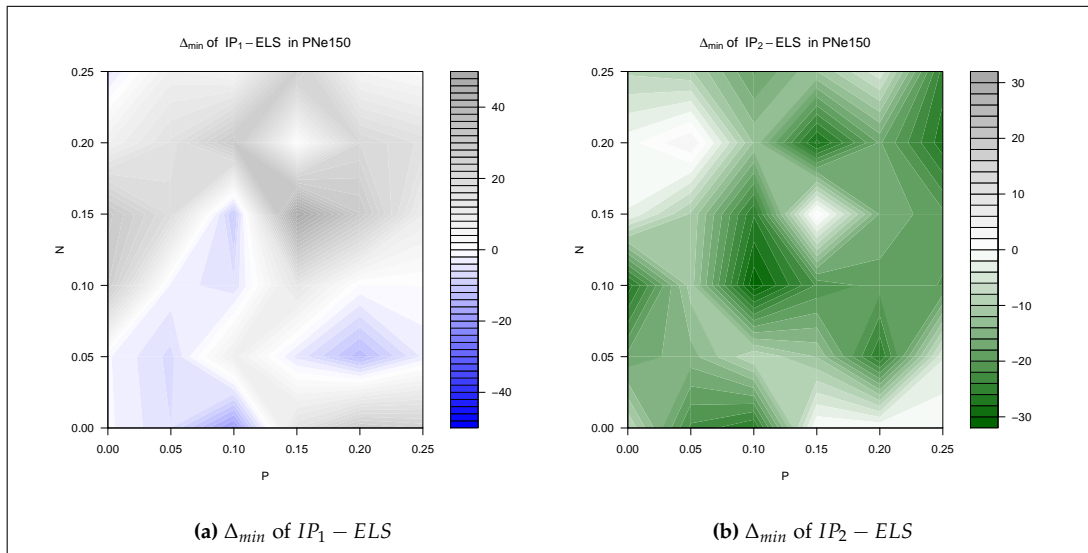
Figure 6.8: Differences in minimum penalty (Δ_{min}) after applying IP models without/with floor variables (IP_1 and IP_2) and evolutionary local search (ELS) on $PNe150$ dataset ($IP_1 - ELS$ and $IP_2 - ELS$). IP_1, IP_2 and ELS are represented by blue, green, and grey regions respectively.

of dark green regions signifies the superiority of IP_2 over ELS for providing better minimum total penalties in both datasets.

The situation here is similar to the local search and integer programming comparison case (described in Section 5.6.4). The performance of ELS is between IP_1 and IP_2. It is more competitive with IP_1 than LS was. However, IP_2 is still better than both ELS and LS although the gap between IP_2 and ELS is smaller than the gap between IP_2 and LS.

6.5.9 Comparison of Local Search vs Evolutionary Local Search Algorithms

In this section, we are going to analyse the difference in performance between the evolutionary local search algorithm (ELS) described in this chapter and the local search algorithm (LS) presented in Chapter 5.

Both LS and ELS were given 10 runs (3 minutes each). In Figures 6.9 and 6.10, four performance measures are compared for instances in $SVe150$ and $PNe150$ datasets respectively. These measures are differences in average total penalty (Δ_{TP}), minimum total penalty (Δ_{min}), average space misuse penalty (Δ_{SMP}) and average soft constraint violation penalty (Δ_{SCP}) after 10 runs.

The x and y axes represent *slack space rate* (S) and *violation rate* (V) in Figure 6.9 while they represent *positive* (P) and *negative* (N) *slack space rate* in Figure 6.10. In these contour plots, LS is better than ELS when a region becomes more red, while ELS is better in grey regions. LS and ELS algorithms give comparable performance in white or pale red/grey regions.

It is observed that ELS is generally better in obtaining the average total cost penalty (Δ_{TP}) in $SVe150$ dataset (Figure 6.9a). The average performance of ELS is especially better in medium S range (where the expected space misuse penalty is about medium) although LS can yield better results in higher V range where the expected soft constraint violation penalty is large. LS and ELS are comparable in obtaining the minimum total penalty (min) after 10 runs as evidenced by the abundance of white and pale red/grey regions in Figure 6.9b. Similar to the average total cost penalty case, ELS tends to perform better in medium S range (except $S_{0.60}V_{0.40}$ instance) and performs worse than LS in high V range.

When the average space misuse and soft constraint violation penalties are evaluated in Figures 6.9c and 6.9d, it is observed that LS is clearly better in minimising the space misuse penalty SMP (red regions in Figure 6.9c). However, ELS is superior in minimising the soft constraint violation penalty SCP (grey regions in 6.9d). Since ELS

(a) Δ_{TP} of $LS - ELS$

(b) Δ_{min} of $LS - ELS$

(c) Δ_{SMP} of $LS - ELS$

(d) Δ_{SCP} of $LS - ELS$
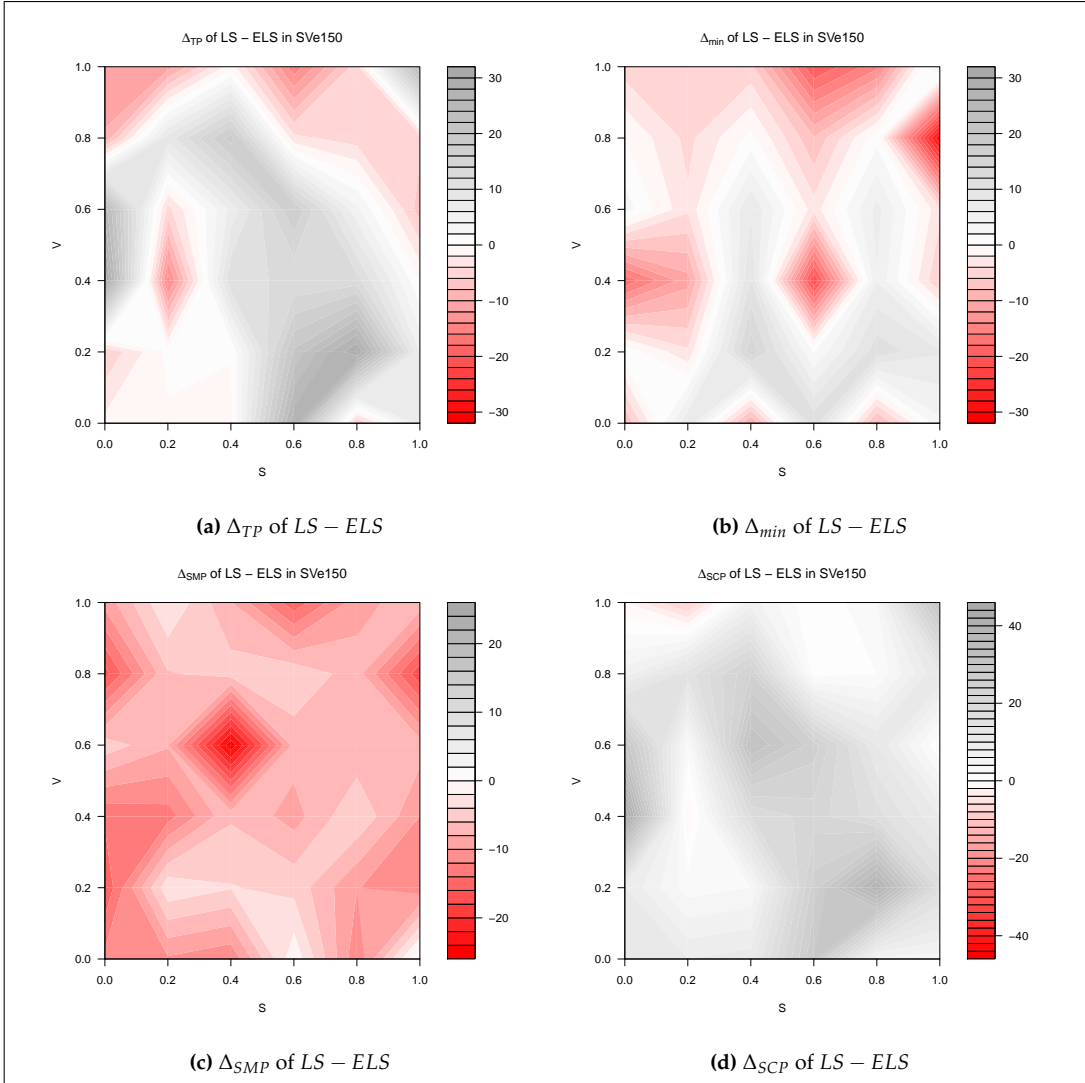
Figure 6.9: Differences in average total cost penalty (Δ_{TP}), minimum total penalty (Δ_{min}), average space misuse penalty (Δ_{SMP}), and average soft constraint penalty (Δ_{SCP}) after applying local search (LS) and evolutionary local search (ELS) on $SVe150$ dataset ($LS - ELS$). LS and ELS are represented by red and grey regions respectively.

minimises SCP better than LS minimises SMP, on the average case, ELS yields better results in total penalty TP on $SVe150$ dataset.

The results in $PNe150$ give a similar picture as in $SVe150$ instances. ELS is superior in minimising the average total penalty (TP) (the grey regions in Figure 6.10a). The difference is greater in favour of ELS when the expected space underuse is high (large P) and space overuse is low (small N). The case of difference in minimum total penalty (Δ_{min}) is interesting as in Figure 6.10b: LS usually gives better results when N and P values are close (where the instances are usually difficult to solve as analysed in Section 4.6.3). ELS is usually better than LS around the edges of Figure 6.10b in grey regions.

(a) Δ_{TP} of $LS - ELS$

(b) Δ_{min} of $LS - ELS$

(c) Δ_{SMP} of $LS - ELS$

(d) Δ_{SCP} of $LS - ELS$
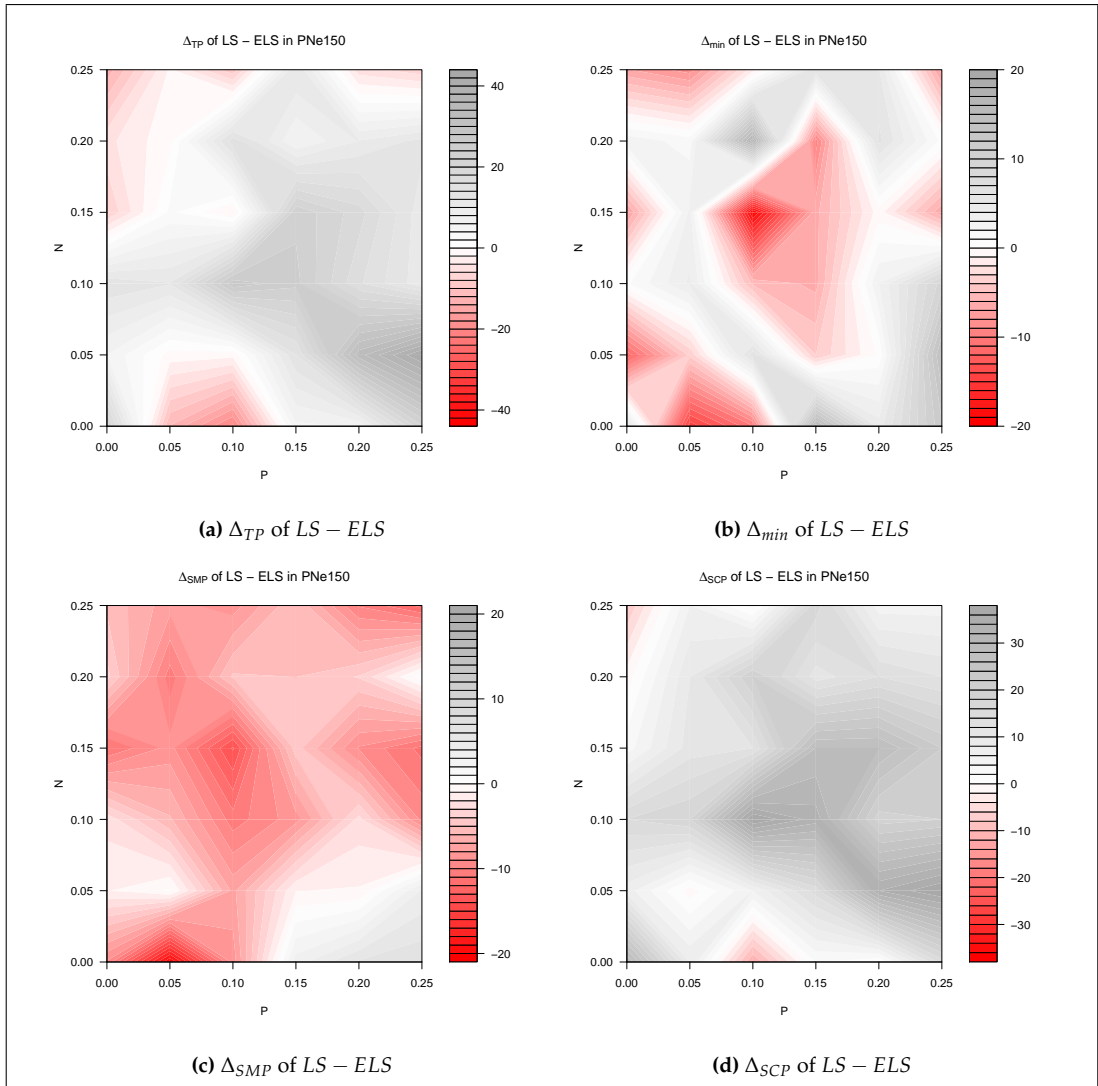
Figure 6.10: Differences in average total cost penalty (Δ_{TP}), minimum total penalty (Δ_{min}), average space misuse penalty (Δ_{SMP}), and average soft constraint penalty (Δ_{SCP}) after applying local search (LS) and evolutionary local search (ELS) on $PNe150$ dataset ($LS - ELS$). LS and ELS are represented by red and grey regions respectively.

A similar situation is observed in minimisation of space misuse (SMP) and soft constraint violation penalties (SCP) as in $SVe150$ instances. Figure 6.10c is predominantly red which signifies the superiority of LS in minimisation of SMP. Conversely, grey regions in Figure 6.10d shows that ELS is better in minimisation of SCP. Since ELS minimises SCP better than LS minimises SMP, on the average case, ELS yields better results in total penalty TP in $PNe150$ dataset.

6.6 Conclusion

In this chapter, an evolutionary local search algorithm (*ELS*) was presented. The evolutionary components of the algorithm included a population of solutions selected for traditional crossover and mutation operators. Additionally, problem specific crossover and mutation operators were also developed.

However, the most important component of this algorithm was the local search which utilised a very fast cost calculation procedure. This mechanism enabled searching for a promising *relocate* move (sending an entity from one room to another) within a large neighbourhood. More importantly, after such a move was made, this procedure enabled quick and complete update to the cost changes of any move in the *relocate* move neighbourhood. Significant speed-ups were observed. However, *swap* move as in Chapter 5 had to be dropped due to implementation efficiency problems. *Relocate* and *swap* had also poor synergy with each other in a preliminary hybrid method with approximate update algorithms.

The experiments focused on mutation rates, length of local search after each crossover operation, size of the *relocate* move neighbourhood that was going to be searched at each local search step, different orderings (entity or room based) to sample a fixed size region of the neighbourhood for the search, and population size.

It was observed that the mutation was very integral to the success to the algorithm. The traditional random mutation which changed the room of an entity stochastically performed quite well when coupled with another mutation operator which set the entities that were associated with the *allocation* constraints. The duration of the local search that was needed was observed to be proportional to the extent of the mutation. Larger mutation rates required a longer local search in order to correct the disruptions to the solution.

Another important issue in designing the local search was to decide on the size of the neighbourhood that was going to be searched. Searching the whole *relocate* move neighbourhood and taking the best move within it yielded poor results. Therefore, at each step, the neighbourhood was stochastically divided into regions and only the locations within this sub-region was searched. It was experimentally observed that most efficient approach in deciding the size of this region was the balance between the sub-region search time and the update time after a *relocate* move was made.

Overall, while *ELS* improved the results over *LS* algorithm Chapter 5, it was still not competitive with integer programming model with floor variables from Chapter 4. There will be further modifications to *ELS* in next chapter to address this issue.

Combining Mathematical Programming and Heuristics

7.1 Introduction

This chapter is devoted to final implementations of mathematical programming and local search heuristics and the approaches to combine these two together in a framework for solving the office space allocation problem. This final chapter proposes refinements, modifications and hybridisations to the algorithms designed in Chapters 4, 5, and 6. For the mathematical programming component, the binary integer programming formulation with the floor variables that is described in Chapter 4 is used. For the heuristic component, the evolutionary local search algorithm described in Chapter 6 is reduced to a single-solution local search algorithm. This reduction is done to simplify the embedding process of heuristic within a mathematical programming framework. The necessary modifications to this reduction are explained in this chapter. This new local search algorithm is also improved by using a large mutation based backtracking method which allows the search to return back to a different mutated solution. Also, different methods to combine the mathematical programming and local search components are described. Several implementation and coding issues are encountered due to the application programming interfaces of integer programming solvers (CPLEX [IBM-Ilog, 2013] and Gurobi [Gurobi-Optimization, 2010]). These issues related to the application programming interfaces (APIs) are described in this chapter.

This chapter is organised as follows: Section 7.2 describes methods to combine mathematical programming and heuristics in a single framework. The modifications to the local search heuristic that is used together or embedded into the mathematical programming are explained in Section 7.3. Section 7.4 presents the experimental re-

sults and discussions of the efficacy of the combination techniques. Finally, Section 7.5 presents the conclusions of the hybrid mathematical programming and heuristic research.

7.2 Combination Methodologies

In this section, some of the approaches that can be considered in utilising the heuristic and the IP solver together are investigated. These methods are depicted in Figures 7.2.

- *Initialising a solution for the IP solver using a meta-heuristic.* In this simple approach, the meta-heuristic is allowed to run for a small period of time until a sufficiently high quality solution is obtained. This solution is then fed into the IP solver as the starting solution and the IP continues without any additional input from the heuristic apart from this starting solution generated by the heuristic. This method is depicted in Figure 7.2a(a).

- *Running heuristic and IP in tandem.* In this method, first, one of the approaches is chosen and that approach is allowed to run for a fixed period of time. The heuristic is the preferred choice in this case due to the fact it has a higher chance of producing a sufficiently high quality valid solution sooner than an IP solver. Assuming heuristic is chosen first, the solution generated by the heuristic after a fixed period of time is then fed to the IP solver. The IP solver is allowed to operate and improve this solution. The best solution generated by the IP solver is then sent back to the heuristic for improvement. The algorithm ends after a number of successive heuristic-IP iterations. This method is depicted in Figure 7.2a(b).

- *Passing additional information from the IP to the solver.* The IP solvers like CPLEX can generate some important information like lower and upper bound values on the decision variables. If the lower and upper bound of a variables are equal to each other, the decision variable is fixed to a specific value. In the case of an OSA problem, this information can be used to determine if an entity must or must not be allocated to a specific room or floor. This is depicted in Figure 7.1. This entity-room-floor information can be used within a heuristic for preventing some entities to be sent to specific rooms or floors. In a typical tabu search algorithm [Glover and Laguna, 1997], this information (e, r) can be used to fill tabu lists. For example, if a decision variable associated with an entity-room allocation is fixed to 1, this means any single *relocate* move that moves the entity e to any other

room than *r* can be set as tabu. Another improvement can be made in cost calculation procedures. Since it is known that certain entity-room-floor relationships can never happen due to certain fixed variables, cost calculation or neighbourhood search operators can simply ignore the locations associated with such relationships. The availability of such locations related to these fixed variables can help more efficient implementation of the updates on the Δ matrix which were described in Section 6.4 previously. This method is depicted in Figure 7.2a(c).
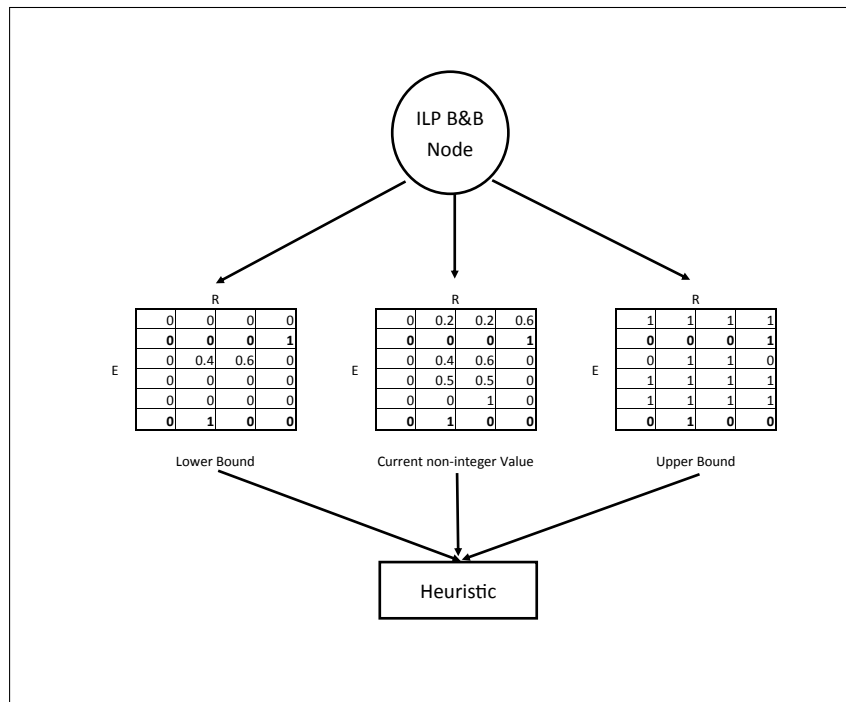
Figure 7.1: Transferring bounds and fixed variables from IP solver to heuristics

- *Incorporating a heuristic within the IP using heuristic callback.* The CPLEX API allows access to the nodes of the branch and bound tree during the search. An *incumbent callback* allows access to a new best-known solution (*incumbent solution*) whenever such a solution is found. This solution can be fed to the heuristic using an *incumbent callback* and the heuristic is allowed to improve this new best-known solution. If a new best solution is obtained using the heuristic, this new *incumbent solution* is passed back to the IP solver. This method is depicted in Figure 7.2b(d). Alternatively, the heuristic can be called at different nodes in the branch and bound process using a *heuristic callback*. The solutions at these nodes will probably be incomplete. Some of the decision variables will be non-integer. Hence, the heuristic can decide on the integer value to which these non-integer variables can be set. The heuristic might also try to improve the solution in these sub-nodes with the hope of generating a new-incumbent that can be passed back

to the IP solver. This method is depicted in Figure 7.2b(e).

(a)

a. Initialisation only

b. Running heuristic and IP solver in tandem

c. Passing information between IP and heuristic

(b)

d. Incorporating heuristic within IP nodes

e. New incumbent solution being passed to the heuristic
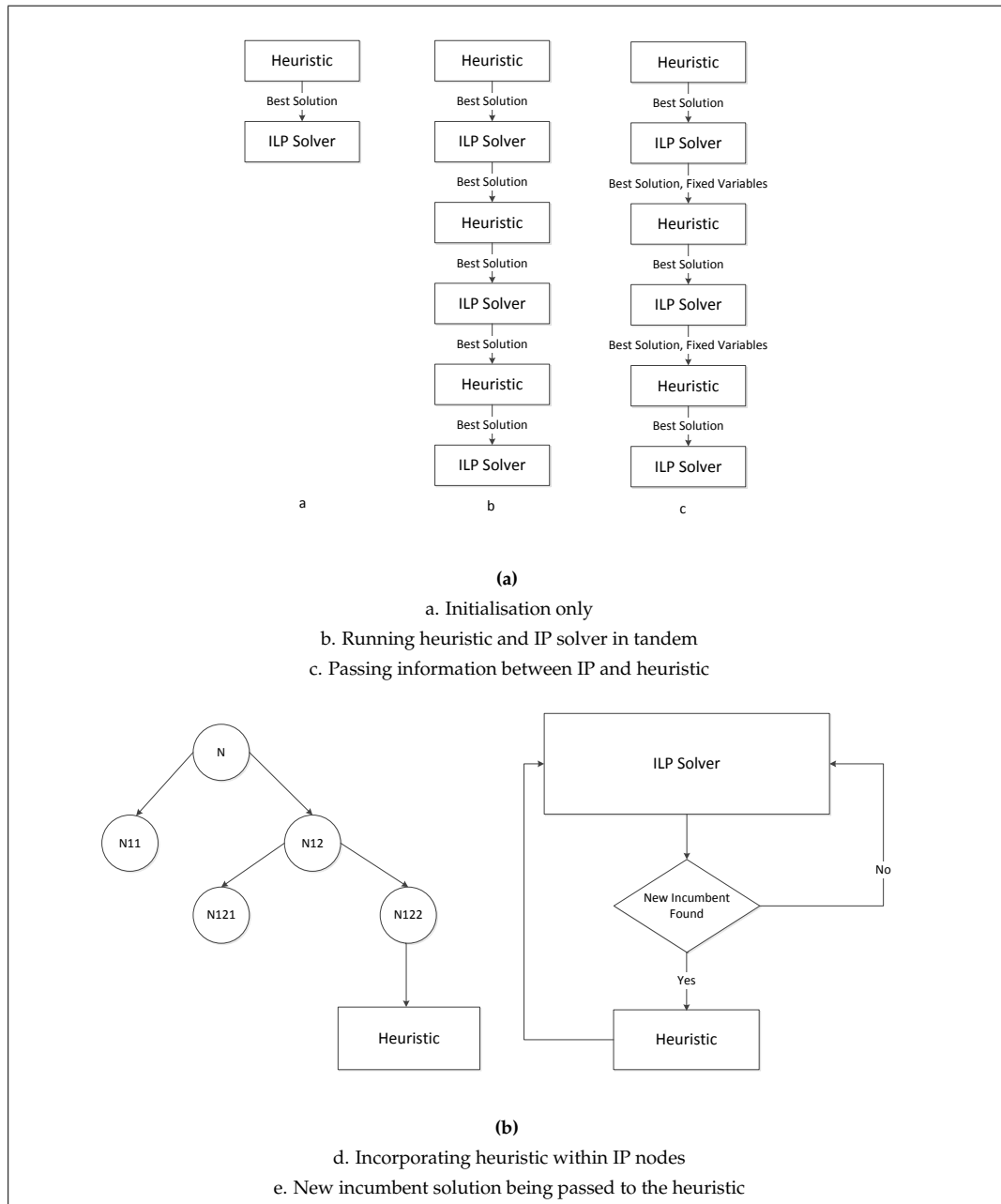
Figure 7.2: Combination methods of mathematical programming and heuristics

7.2.1 Implementation Issues with CPLEX API

During the implementation of mathematical models, the C++ Concert API of CPLEX was used. Unfortunately, some of the limitations of the application programming interface (API) of CPLEX prevented some features from being implemented.

Any heuristic that is going to be embedded within CPLEX has to be implemented using a callback function via the CPLEX callback interface. The most important problem occurs due to the implementation of the callbacks using CPLEX and how addition of such a heuristic callback within the branch and bound procedure changes the way CPLEX operates. By default, CPLEX operates in a mode called *dynamic search* which is a combination of *branch and bound*, *cutting planes*, and *heuristics* designed for IP problems. However, implementing a heuristic callback to incorporate the desired features eliminates the *dynamic search* and CPLEX begins to work in a straight-forward *traditional branch and bound* mode.

The second important problem is the limited access to the lower and upper bound values of the decision variables during the IP search. Unfortunately, due to C++ API issues, it is only possible to access the fixed decision variables (where the lower and upper bound values on the variable are equal to each other) that are determined in the pre-solve stage. Although CPLEX tightens these bounds and fixes additional decision variables during the search, it is not possible to determine and access these fixed decision variables in a heuristic callback in C++ API.

7.3 Modifications to the Local Search Heuristic

Here, the evolutionary local search algorithm (*ELS*) designed in Chapter 6 is chosen as a basis for heuristic to be used together with the mathematical model. There are several problems using such a population based algorithm with an off-the-shelf integer programming solver like CPLEX. First problem is to determine the solution in the population that should be fed into the solver when transferring from the heuristic stage back to branch-and-bound procedure in the IP solver. The most obvious answer to this problem is selecting the best solution in the population which is the technique used in our implementation. Alternatively, non-best solutions in the population can also be fed to the solver; however, after preliminary experimentation it is decided against such an approach. In an embedded framework where a heuristic works within the IP solver, passing a non-best solution obtained in a heuristic can be worse than the best obtained solution by the IP solver. This is more evident especially in the later stages of the algorithm when improvement of the overall best solution becomes excessively more difficult. In this case, the best obtained solution transferred from the IP solver is simply overridden by this non-best solution from the heuristic. However, depending upon the combination method of IP and the heuristic, the IP solver can quickly regenerate the best previously obtained solution. In most cases, IP solver effectively eliminates

any benefit of passing a non-best solution generated by the heuristic in the first place.

A greater issue arises when transferring the solution from the IP stage to the heuristic stage. Although the IP solver can provide a list of best and non-best solutions, due to the nature of branch-and-bound procedure and the IP Solver, there can be huge gaps between the best and the non-best solutions. These gaps between the best and the non-best solutions can be a lot higher than the successive best solutions in a typical meta-heuristic algorithm. Therefore, passing these non-best solutions back to a population based algorithm might be ineffectual due to the fact that these will be overridden immediately by the meta-heuristic.

Another problem is that the IP solver may not be able generate enough solutions to initialise the population of the ELS. There might be cases where the IP solver can transfer only a few quality solution back to the heuristic. In this case, solutions in the population have to be initialised again using these few solutions. The current population in the ELS algorithm has already low diversity of the solutions due to the aggressively greedy nature of the local search component. Hence, the situation is exacerbated due to few diverse solutions passed from the IP.

Given the above issues, it was decided to morph the population based algorithm (*ELS*) from Chapter 6 into a local-search based variant operating on a single solution. This simplifies the solution passing mechanism between the two stages (IP and heuristic searches). This reduction into a local-search algorithm allowed further implementation optimisations as well. The outline of the modified algorithm is depicted in Figure 7.3. Two modifications made to the *ELS* algorithm (described in Chapter 6) are described in Sections 7.3.1 and 7.3.2. The final algorithm is described in Section 7.3.3.

7.3.1 Elimination of Repeated Delta Stage in the Local Search Heuristic

The elimination of the population based algorithm and the genetic operators allows further optimisation in the coding stage. As described in Section 6.4.2, the delta stage is responsible for the initial calculation of the Δ table which holds the changes in objective function value for each relocate move (sending an entity e from room r_1 to r_2). This delta stage is a costly operation and has to be repeated after each crossover operation in the ELS algorithm. This is due to the large potential change on the entity-room mapping from parents to offspring. Elimination of the crossover operator and the population of solutions allows easier tracking of the changes in the Δ table because in this form, rest of the operations in the local search all involves simple moves or a series of simple moves and hence the changes on the Δ table can be easily done using the update

stage algorithms described in Chapter 6.

As described in Section 6.4, the local search stage operates by the application of several mutation operators and then a short number of *relocate* move operations (which is reduced to 25 operations in the new version) with fast cost change calculations on the Δ table. Each local search stage begins on the global best solution x_{best} (the best solution encountered in the whole search). If a new best solution is obtained during this single step, this new best solution is set as the new global best solution. If such a new best solution is not obtained, then the current global best solution replaces the current solution and the new local search step continues again to improve the global best solution.

7.3.2 Backtracking

Additional problems can arise when reducing a population based meta-heuristic into a single-solution one. The most important problem is the elimination of population diversity. It is observed that the single solution heuristic might be more prone to getting stuck in local optima than the population based one due to the nature of the algorithm which tries to improve a best-known solution greedily. In order to alleviate this issue, methods which take the search to a previous position (*backtracking*) can be considered. The backtracking mechanisms considered in this thesis are as follows:

- Applying a large random mutation to the solution and setting this solution as the new best one. This new best solution is allowed to violate the *hard* constraints. In this case, the penalty for the *hard* constraint can be set to an arbitrarily large number (to represent infinite penalty). There are two algorithm parameters in this approach. The first one is the *large mutation rate* and this represents how disruptive the random mutation is going to be. The second one is the *mutation iteration rate* which sets up when a mutation is going to be performed. This parameter is tied to an iteration counter which checks how many iterations have passed since any improvement on the global best known solution is observed.

- Using a list of previous solutions encountered during the search and returning back to one of these solutions during the search. The algorithm will return back to a previous solution if no improvement can be achieved to the current best solution after a number of iterations. The list can be formed either by only adding a best known solution to it whenever such a best-known solution is encountered the first time during the search. Alternatively, a sorted list of solutions ordered

according to the objective value can also be used although with additional keep-up costs. In this type of backtracking, one of the most important issue is the decision of backtracking depth: that how many previous solutions we should backtrack to if there is still no improvement to the best-known solution during the search.

After initial experimentation, it was decided to use the large mutation method instead of returning back to a previous best solution. This was because the previous solution approach did not necessarily allow escape from local optima and led to cycling runs between the backtracked solution and the best solution encountered in the search. Therefore, this thesis is going to focus on the first approach (large mutation backtracking).

In this thesis, backtracking approach was applied to the single solution local search algorithm (explained in Section 7.3) that was stripped down from the evolutionary local search (*ELS*) from Chapter 6. Our preliminary backtracking experiments with local search (*LS*) algorithm from Chapter 5 did not give satisfactory results. *LS* algorithm and backtracking had poor synergy together and as a result, no improvement on the solution quality was observed when backtracking was applied on *LS*.

7.3.3 Single Solution Local Search Algorithm with Backtracking

In this section, the final modified single solution local search algorithm with backtracking (*BCK*) is explained. Figure 7.3 depicts the pseudo-code for *BCK*.

This algorithm works similarly to the local search stage of *ELS* which was previously described in Section 6.4. At each iteration, the global best solution is passed as the solution the current iteration of the algorithm will work on. The global Δ table is passed via C++ pointers for efficiency. Random mutation and allocation mutation described in Section 6.3.2 are applied to the current solution. A shorter local search compared to *ELS* is then applied to improve the solution. The update algorithms described in Section 6.4.3 are again applied in this step. Finally, after a number of local search iterations, the algorithm moves back to the global best solution encountered during the search. If a new global best solution is obtained in the local search stage, this global best solution and the Δ table associated with this new solution are stored.

In this algorithm, backtracking is applied as follows: If there is no improvement to the global best solution after a fixed number of iterations, then the global best solution is perturbed with the random one-point mutation operator. This perturbed solution is then set as the new global best solution. The respective Δ table is calculated using Delta

algorithm described in Sections 5.5 and 6.4.2. *BCK* continues from this new perturbed solution.

Input: input file with entities, rooms, and constraints.
Output: global best solution (an entity-room mapping)
1: Calculate the Δ table
2: $\Delta_{best} \leftarrow \Delta$
3: **repeat**
4:    current solution $\leftarrow$ global best solution
5:    $\Delta \leftarrow \Delta_{best}$
6:    Apply *Random Mutation* on the current solution
7:    Apply *Allocation Mutation* on the current solution
8:    Apply *Local Search* on the current solution
9:    **if** new global best solution is found **then**
10:      global best solution $\leftarrow$ current solution
11:      Backup Δ_{best} table
12:    **if** Backtracking criterion is met (no improvement in solution) **then**
13:      Apply backtracking to the global best solution
14: **until** Time limit is reached

Figure 7.3: Single solution local search meta-heuristic with backtracking (*BCK*)

7.4 Experiments Related to Single Solution Local Search with Backtracking, Integer Programming and Heuristic Combination Techniques

In this section, experiments related to the single solution local search with backtracking algorithm (*BCK*) and the combinations of integer programming with this heuristic are presented. First, experiments showing the effect of backtrack iterations and mutation rate (as described in Section 7.3) are presented in Section 7.4.1. Complete results on *SVe*150 and *PNe*150 dataset instances using the single solution local search with backtracking are given in Section 7.4.2. Comparisons of *BCK* algorithm with integer programming models described in Chapter 4 are given in Section 7.4.3. Comparisons of *BCK* to previous heuristics (local search in Chapter 5 and evolutionary local search in Chapter 6) are given in Sections 7.4.4 and 7.4.5 respectively. The performance of local search meta-heuristics on some *nott*1 instances were compared in Section 7.4.6. Results on combination of integer programming and meta-heuristics are presented in Section 7.4.7.

In the following experiments, the objective function for total penalty (TP) is taken as the weighted summation of space misuse (SMP) and soft constraint violation penalties (SCP). The respective formulations for SMP, SCP and TP were previously given in equations 2.3.1, 2.3.2, and 2.3.3 in Section 2.3.2. This objective function is going to be minimised subject to *hard* constraints. The numbers of *hard* and *soft* constraints for *nott1*, $SVe150$ and $PNe150$ instances were previously given in Table 2.2 (Section 2.4) and Table 2.4 (Section 2.5.3) in Chapter 2 respectively.

7.4.1 Effect of Backtrack Iterations and Backtrack Mutation Rate

In this experiment, the effect of *backtrack iterations* and *backtrack mutation rate* is explored. Six data instances are chosen for experiments: $S_{0.00}V_{0.00}$, $S_{0.40}V_{0.80}$, $S_{0.80}V_{0.40}$, $P_{0.00}N_{0.00}$, $P_{0.10}N_{0.20}$, and $P_{0.20}N_{0.10}$. Each instance is given 20 runs (90 seconds each) on a Core 2 Duo E8400 Processor. The number of *backtrack iterations* (that the best solution cannot be improved for the past such number of iterations) are 2500, 5000, and 10000. The *backtrack mutation rates* are chosen as 1.5, 3.0, and 6.0 as number of entities chosen for random mutation on average.

The results for several $SVe150$ and $PNe150$ instances are given on Tables 7.1a and 7.1b respectively. Columns b and bm represent the *backtrack iterations* and *backtrack mutation rate* respectively. Columns μ, σ, min, and max give the average, the standard deviation, minimum and maximum results obtained over 20 runs. As evidenced from the close average and standard deviations, the performance of modified local search algorithm does not overly depend on the variations of these two parameters in this range. However, it is still observed that the performance tends to get slightly better as the number of *backtrack iterations* is reduced to 2500 range. However, for *backtrack mutation* rate, no conclusive result is obtained in this range.

However, it should be noted that in other preliminary experiments, it was not possible to obtain satisfactory results under different parameter ranges (large or really low). For larger *backtrack iterations*, the algorithm spent too much time on a concentrated small search space. It was not possible to take the *backtrack iterations* really lower than 1000 either. In this opposite case, the algorithm was not allowed to extensively explore a specific location in the search space and it was observed that the algorithm backtracked very frequently.

Finally, the preliminary implementations and experiments on a dynamic *backtrack iterations* were not successful. These implementations included either increasing or decreasing the number of *backtrack iterations* as the search went on. However, it was

instance	b	bm	μ	σ	min
$S_{0.00}V_{0.00}$	2500	1.50	25.10	8.74	9.00
	2500	3.00	24.35	7.51	10.50
	2500	6.00	23.20	6.63	7.50
	5000	1.50	25.10	9.73	7.50
	5000	3.00	26.05	7.95	12.00
	5000	6.00	25.55	3.55	18.00
	10000	1.50	27.70	7.64	12.00
	10000	3.00	25.35	8.52	14.50
	10000	6.00	22.75	8.04	13.50
$S_{0.40}V_{0.80}$	2500	1.50	199.87	11.24	182.00
	2500	3.00	197.78	6.79	184.40
	2500	6.00	204.60	10.43	190.00
	5000	1.50	202.75	10.32	183.20
	5000	3.00	204.38	10.27	187.30
	5000	6.00	202.90	6.61	188.50
	10000	1.50	205.47	11.28	188.60
	10000	3.00	199.97	13.39	183.20
	10000	6.00	200.31	6.46	188.60
$S_{0.80}V_{0.40}$	2500	1.50	168.33	8.00	160.20
	2500	3.00	171.68	9.87	156.80
	2500	6.00	172.73	11.25	162.00
	5000	1.50	172.40	7.30	162.20
	5000	3.00	173.52	9.66	162.00
	5000	6.00	171.84	5.99	160.10
	10000	1.50	175.78	13.66	155.10
	10000	3.00	174.54	12.34	160.30
	10000	6.00	171.39	11.03	159.00

(a) $SVe150$ instances

instance	b	bm	μ	σ	min
$P_{0.00}N_{0.00}$	2500	1.50	80.50	9.00	67.00
	2500	3.00	81.80	7.59	69.50
	2500	6.00	81.90	5.79	76.00
	5000	1.50	81.95	9.64	74.00
	5000	3.00	83.30	5.91	74.50
	5000	6.00	81.05	5.39	73.00
	10000	1.50	80.85	4.87	69.50
	10000	3.00	81.95	10.08	66.50
	10000	6.00	82.75	9.58	74.00
$P_{0.10}N_{0.20}$	2500	1.50	197.06	3.58	191.90
	2500	3.00	198.28	4.32	192.20
	2500	6.00	203.44	4.20	196.10
	5000	1.50	200.39	6.60	186.20
	5000	3.00	199.21	5.71	190.10
	5000	6.00	195.38	4.09	188.60
	10000	1.50	199.20	4.64	191.30
	10000	3.00	201.66	7.35	193.10
	10000	6.00	199.44	7.11	190.70
$P_{0.20}N_{0.10}$	2500	1.50	140.00	5.69	131.20
	2500	3.00	137.32	6.35	130.50
	2500	6.00	139.01	7.54	127.50
	5000	1.50	142.55	8.70	131.40
	5000	3.00	135.32	6.70	121.20
	5000	6.00	136.60	4.63	129.40
	10000	1.50	143.72	7.12	129.90
	10000	3.00	143.78	6.38	135.00
	10000	6.00	140.81	5.70	132.90

(b) $PNe150$ instances

Table 7.1: Impacts of different backtrack iterations and backtrack mutation rate on instances $S_{0.00}V_{0.00}$, $S_{0.40}V_{0.80}$, $S_{0.80}V_{0.40}$, $P_{0.00}N_{0.00}$, $P_{0.10}N_{0.20}$, and $P_{0.20}N_{0.10}$

observed that performance could sometimes be significantly worse than using a simple static *backtrack iteration* limit.

7.4.2 Complete Results of Single Solution Local Search with Backtracking on SVe150 and PNe150 Datasets

Tables 7.2a and 7.2b represent the results obtained. Columns *S*, *V*, *P*, and *N* represent the four different parameters: *slack space rate*, *soft constraint violation rate*, *positive*, and *negative slack rates* respectively. Columns μ and σ represent the average and standard deviation of the total penalty obtained after ten runs (three minutes each) respectively. Columns *SMP* and *SCP* give the average misuse and soft constraint violation penalty after ten runs respectively. Column *min* gives the minimum total penalty obtained after

ten runs.

S	V	μ	σ	SMP	SCP	min		P	N	μ	σ	SMP	SCP	min
0.00	0.00	24.20	7.32	16.20	8.00	10.50		0.00	0.00	81.80	7.59	31.80	50.00	69.50
0.00	0.20	48.70	8.06	23.70	25.00	35.00		0.00	0.00	114.52	6.32	69.52	45.00	98.90
0.00	0.40	77.95	6.55	28.95	49.00	65.50		0.00	0.00	156.17	4.26	106.17	50.00	147.40
0.00	0.60	91.15	7.66	33.15	58.00	79.50		0.00	0.00	194.23	6.79	141.23	53.00	186.10
0.00	0.80	138.65	9.36	43.65	95.00	126.50		0.00	0.00	227.07	6.29	177.07	50.00	220.10
0.00	1.00	158.80	12.35	49.80	109.00	140.50		0.00	0.00	262.89	3.10	213.89	49.00	257.70
0.20	0.00	44.60	6.41	41.60	3.00	33.80		0.05	0.00	80.40	4.49	34.40	46.00	73.00
0.20	0.20	76.29	4.24	45.29	31.00	70.40		0.05	0.05	113.79	7.27	62.79	51.00	104.20
0.20	0.40	93.61	6.32	52.61	41.00	83.10		0.05	0.10	142.80	8.06	96.80	46.00	128.40
0.20	0.60	111.99	4.70	50.99	61.00	105.90		0.05	0.15	178.44	5.56	125.44	53.00	171.00
0.20	0.80	153.64	9.15	67.64	86.00	138.10		0.05	0.20	212.00	7.82	159.00	53.00	200.60
0.20	1.00	173.51	4.35	71.51	102.00	168.10		0.05	0.25	248.25	5.09	195.25	53.00	242.10
0.40	0.00	99.36	10.03	80.36	19.00	90.30		0.10	0.00	85.34	4.91	40.34	45.00	77.40
0.40	0.20	128.76	6.76	85.76	43.00	117.90		0.10	0.05	114.31	5.58	71.31	43.00	108.10
0.40	0.40	148.61	6.86	94.61	54.00	138.10		0.10	0.10	132.07	5.73	88.07	44.00	122.70
0.40	0.60	156.73	5.39	91.73	65.00	147.90		0.10	0.15	169.27	3.67	112.27	57.00	165.30
0.40	0.80	197.73	6.83	103.73	94.00	184.40		0.10	0.20	198.34	4.28	143.34	55.00	192.20
0.40	1.00	223.31	7.90	112.31	111.00	212.00		0.10	0.25	227.19	4.35	177.19	50.00	221.40
0.60	0.00	120.18	6.02	106.18	14.00	110.00		0.15	0.00	95.32	4.49	54.32	41.00	91.50
0.60	0.20	152.93	3.70	112.93	40.00	146.90		0.15	0.05	117.15	5.42	72.15	45.00	109.60
0.60	0.40	172.86	8.71	119.86	53.00	156.50		0.15	0.10	135.83	7.44	87.83	48.00	125.70
0.60	0.60	186.76	6.23	123.76	63.00	177.10		0.15	0.15	161.29	5.91	107.29	54.00	152.70
0.60	0.80	230.34	6.39	132.34	98.00	223.10		0.15	0.20	188.20	7.10	130.20	58.00	175.10
0.60	1.00	248.22	8.54	138.22	110.00	238.40		0.15	0.25	219.11	4.83	167.11	52.00	211.80
0.80	0.00	124.95	4.74	104.95	20.00	118.10		0.20	0.00	113.26	5.12	69.26	44.00	108.60
0.80	0.20	151.53	6.95	107.53	44.00	141.50		0.20	0.05	129.40	6.07	86.40	43.00	119.20
0.80	0.40	171.52	9.87	117.52	54.00	156.80		0.20	0.10	137.32	6.35	90.32	47.00	130.50
0.80	0.60	176.47	6.53	113.47	63.00	163.30		0.20	0.15	161.22	5.36	105.22	56.00	156.30
0.80	0.80	225.27	8.63	133.27	92.00	211.90		0.20	0.20	180.26	8.56	124.26	56.00	169.20
0.80	1.00	253.56	12.79	140.56	113.00	238.50		0.20	0.25	211.18	10.11	157.18	54.00	189.60
1.00	0.00	172.53	7.58	152.53	20.00	159.50		0.25	0.00	126.00	0.00	86.00	40.00	126.00
1.00	0.20	209.60	5.26	163.60	46.00	201.60		0.25	0.05	139.08	4.23	97.08	42.00	133.30
1.00	0.40	222.32	8.67	161.32	61.00	211.80		0.25	0.10	148.83	6.60	105.83	43.00	135.60
1.00	0.60	239.59	9.70	164.59	75.00	219.50		0.25	0.15	165.26	5.32	116.26	49.00	157.80
1.00	0.80	265.80	8.64	173.80	92.00	252.60		0.25	0.20	181.71	7.82	127.71	54.00	172.50
1.00	1.00	288.64	10.17	171.64	117.00	273.00		0.25	0.25	205.60	9.22	148.60	57.00	191.70

(a) *SVe*150 instances   **(b)** *PNe*150 instances

Table 7.2: Experimental results on the *SVe*150 and *PNe*150 dataset instances using single solution local search heuristic with backtracking

7.4.3 Comparison of Single Solution Local Search with Backtracking and Integer Programming Models

In this section, we are going to analyse the difference in performance between single solution local search algorithm with backtracking (BCK) described in this chapter and the integer programming models presented in Chapter 4. The mathematical models without and with the floor variables are abbreviated as IP_1 and IP_2 respectively.

In Figures 7.4 and 7.5, the minimum total penalty obtained for each instance with BCK is compared to ones obtained by IP_1 and IP_2 using $SVe150$ and $PNe150$ datasets respectively. IP_1 or IP_2 and BCK were given equal amount of running time: BCK was given 10 runs (3 minutes each), while IP_1 and IP_2 were given a single run of 30 minutes.

The x and y axes represent *slack space rate* (S) and *violation rate* (V) in Figure 7.4 while they represent *positive* (P) and *negative* (N) *slack space rate* in Figure 7.5. In these contour plots, IP_1 or IP_2 is better than BCK in obtaining the minimum total penalty when a region becomes more blue or green respectively. BCK is better in yellow regions. IP models and BCK algorithms are about equal in obtaining the minimum total penalty in white or pale coloured regions.

As evidenced from the dominance of yellow regions in Figure 7.4a for $SVe150$ instances, BCK is clearly superior to IP_1 in obtaining the minimum total penalty. IP_1 can only provide equal or slightly better results when expected soft constraint violation penalty is low with small V values. There is not a discernible influence of different levels of space misuse through S. These are similar to the cases evidenced in Section 5.6.4 and 6.5.8.

BCK is also better than IP_1 in $PNe150$ dataset in obtaining the best minimum total penalties. The yellow dominance in Figure 7.5a signifies that BCK is better than IP_1 especially in medium to high N values (where the overuse penalty is expected to be higher). IP_1 provides a few competitive results in low N values.

BCK is a lot more competitive with IP_2 than LS or ELS was. For low V values, IP_2 yields better minimum total penalties in green regions and for higher V values, BCK performs better in yellow regions. It is observed that around $S = 0.60$, IP_2 tends to perform better than BCK while other S values do not differentiate either method.

IP_2 usually performs better than BCK in $PNe150$ instances. Figure 7.5b is mostly dark green because of this. However, for low P values where the expected space underuse is small, BCK performs better than IP_2. BCK is also better in $P_{0.15}N_{0.15}$ instance where the heuristic methods like LS and ELS tend to give high quality results over IP

(a) Δ_{min} of $IP_1 - BCK$

(b) Δ_{min} of $IP_2 - BCK$
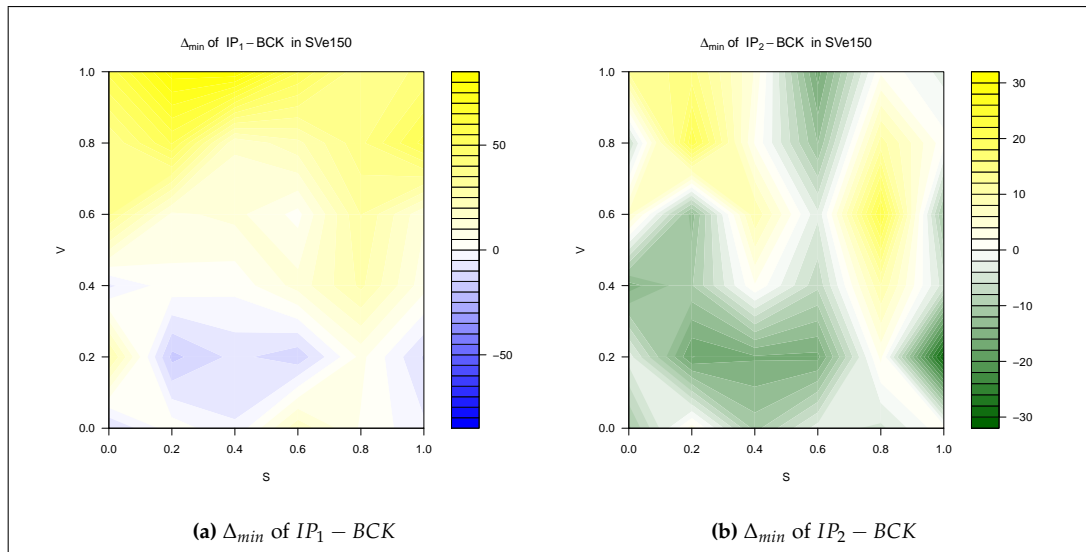
Figure 7.4: Differences in minimum penalty (Δ_{min}) after applying IP models without/with floor variables (IP_1 and IP_2) and single solution local search algorithm with backtracking (BCK) on $SVe150$ dataset ($IP_1 - BCK$ and $IP_2 - BCK$). IP_1, IP_2 and BCK are represented by blue, green, and yellow regions respectively.

(a) Δ_{min} of $IP_1 - BCK$

(b) Δ_{min} of $IP_2 - BCK$
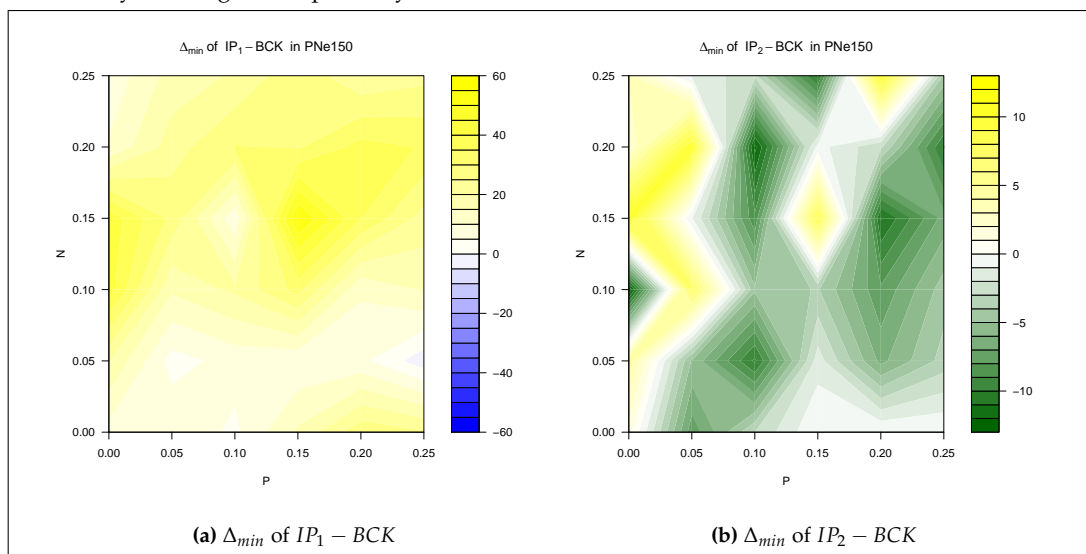
Figure 7.5: Differences in minimum penalty (Δ_{min}) after applying IP models without/with floor variables (IP_1 and IP_2) and single solution local search algorithm with backtracking (BCK) on $PNe150$ dataset ($IP_1 - BCK$ and $IP_2 - BCK$). IP_1, IP_2 and BCK are represented by blue, green, and yellow regions respectively.

models.

BCK is clearly superior to IP_1 in both datasets and gives competitive results with IP_2 (which LS and ELS usually fail). This is the major reason BCK was chosen to be used in development of hybrid methods with integer programming model IP_2.

7.4.4 Comparison of Single Solution Local Search with Backtracking and Local Search Algorithm

In this section, we are going to analyse the difference in performance between the local search algorithm (*LS*) described in Chapter 5 and the single solution local search algorithm with backtracking (*BCK*) presented in this chapter.

Both *LS* and *BCK* were given 10 runs (3 minutes each). In Figures 7.6 and 7.7, four performance measures are compared for instances in *SVe*150 and *PNe*150 datasets respectively. These measures are differences in average total penalty (Δ_{TP}), minimum total penalty (Δ_{min}), average space misuse penalty (Δ_{SMP}) and average soft constraint violation penalty (Δ_{SCP}) after 10 runs.

The *x* and *y* axes represent *slack space rate* (*S*) and *violation rate* (*V*) in Figure 7.6 while they represent *positive* (*P*) and *negative* (*N*) *slack space rate* in Figure 7.7. In these contour plots, *LS* is better than *BCK* when a region becomes more red, while *BCK* is better in yellow regions. *LS* and *BCK* algorithms give comparable performance in white or pale red/yellow regions.

As it can be seen from the yellow regions in Figure 7.6, *BCK* is clearly superiour to *LS* in all four difference metrics (Δ_{TP}, Δ_{min}, Δ_{SMP}, and Δ_{SCP}) in *SVe*150 dataset. The difference in space misuse penalty (Δ_{SCP}) is larger for small *S* values and gets smaller as *S* grows larger as shown in Figure 7.6c. *LS* usually has good performance in minimising the space misuse penalty compared to *ELS* (which was analysed in Section 6.5.9); therefore, the difference between *LS* and *BCK* which was derived from *ELS* is not drastic in this metric. *BCK* also minimises the soft constraint violation penalty *SCP* better. *LS* was already weak compared to *ELS* in this metric as analysed in Section 6.5.9. Average *SCP* and *SMP* combined together, the average *TP* of *BCK* is clearly better than *LS* in Figure 7.6a; however, the difference in minimum total penalty Δ_{min} obtained after 10 runs is not as drastic in Figure 7.6b.

Similar to the case in *SVe*150, *BCK* is clearly superior to *LS* in every difference metric for *PNe*150 instances in Figure 7.7. The differences Δ_{min} in Figure 7.7b and Δ_{SMP} in Figure 7.7c are not drastically in favour of *BCK*; however, *BCK* fixes the weak performance of *ELS* in space misuse penalty. Combined with a stronger performance in soft constraint violation penalty reduction as shown in Figure 7.7d, *BCK* clearly pulls ahead in average total penalty minimisation especially in medium *P* and *N* values in Figure 7.7a.
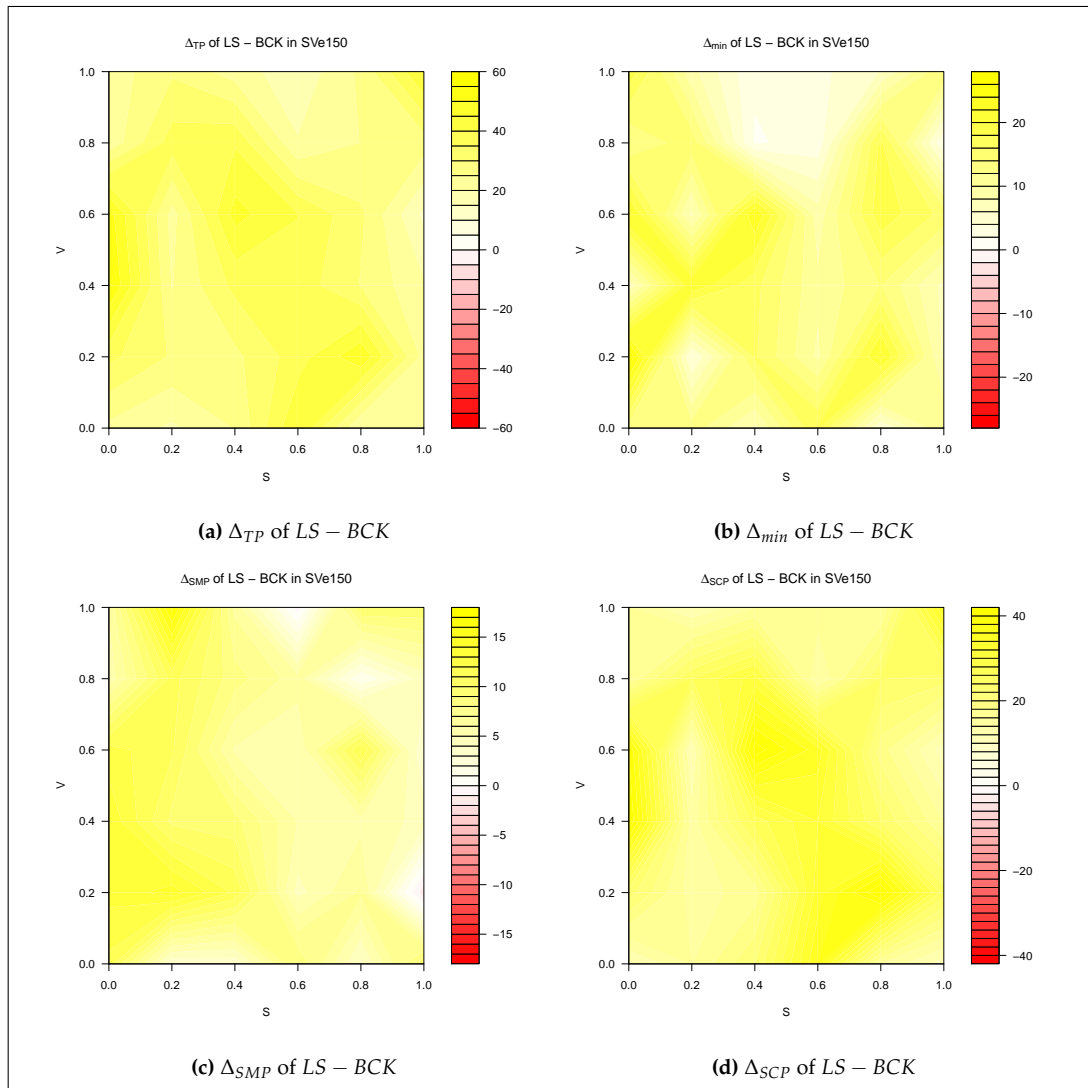
Figure 7.6: Differences in average total cost penalty (Δ_{TP}), minimum penalty (Δ_{min}), average space misuse penalty (Δ_{SMP}) and average soft constraint penalty (Δ_{SCP}) after applying local search (*LS*) and single solution local search with backtracking (*BCK*) on *SVe*150 dataset (*LS* − *BCK*). *LS* and *BCK* are represented by red and yellow regions respectively.

(a) Δ_{TP} of $LS - BCK$

(b) Δ_{min} of $LS - BCK$

(c) Δ_{SMP} of $LS - BCK$

(d) Δ_{SCP} of $LS - BCK$
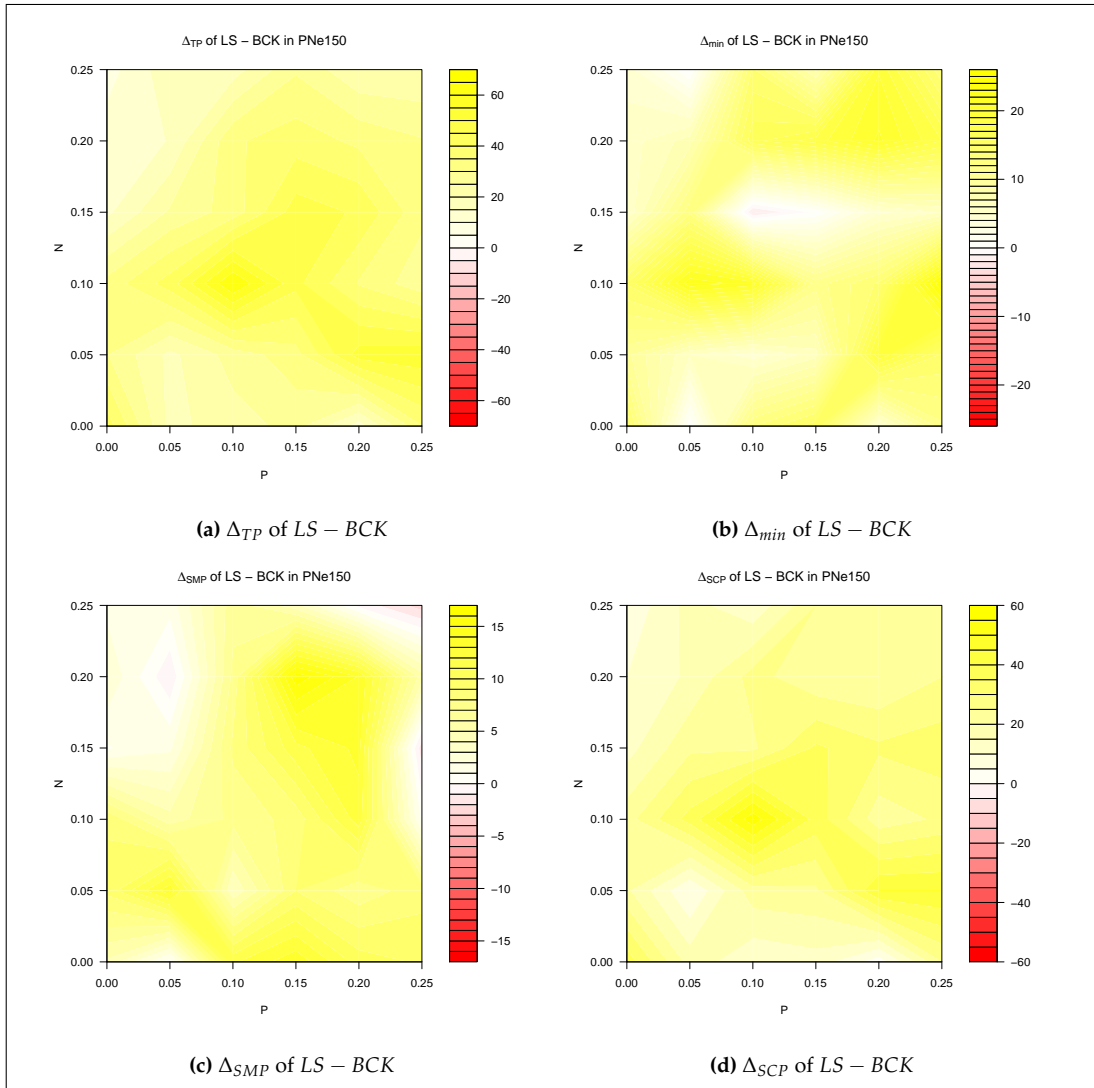
Figure 7.7: Differences in average total cost penalty (Δ_{TP}), minimum penalty (Δ_{min}), average space misuse penalty (Δ_{SMP}), and average soft constraint penalty (Δ_{SCP}) after applying local search (*LS*) and single solution local search with backtracking (*BCK*) on *PNe*150 dataset (*LS* − *BCK*). *LS* and *BCK* are represented by red and yellow regions respectively.

7.4.5 Comparison of Single Solution Local Search with Backtracking and Evolutionary Local Search Algorithm

In this section, we are going to analyse the difference in performance between the evolutionary local search algorithm (*ELS*) described in Chapter 6 and the single solution local search algorithm with backtracking (*BCK*) presented in this chapter.

Both *ELS* and *BCK* were given 10 runs (3 minutes each). In Figures 7.8 and 7.9, four performance measures are compared for instances in *SVe*150 and *PNe*150 datasets respectively. These measures are differences in average total penalty (Δ_{TP}), minimum total penalty (Δ_{min}), average space misuse penalty (Δ_{SMP}), and average soft constraint violation penalty (Δ_{SCP}) after 10 runs.

The *x* and *y* axes represent *slack space rate* (*S*) and *violation rate* (*V*) in Figure 7.8 while they represent *positive* (*P*) and *negative* (*N*) *slack space rate* in Figure 7.9. In these contour plots, *ELS* is better than *BCK* when a region becomes more grey, while *BCK* is better in yellow regions. *ELS* and *BCK* algorithms give comparable performance in white or pale grey/yellow regions.

As evidenced by Figure 7.8, *BCK* is superior to *ELS* in every difference metric in *SVe*150 instances. *BCK* fixes the weak performance of *ELS* in space misuse penalty (*SMP*) as shown in Figure 7.8c. The improvement in this metric is greater for smaller *S*-medium *V* valued instances. *BCK* also improves the comparatively strong performance of *ELS* in soft constraint violation penalty (*SCP*) especially for larger *V* values as shown in Figure 7.8d. With the combined improvement in *SCP* and *SMP*, *BCK* shows a smooth improvement in average total penalty in Figure 7.8a over *ELS* which it was derived from. There is also an improvement in obtaining better minimum total penalties especially for larger *V* valued instances as shown in Figure 7.8b.

BCK performs better in all difference metrics in *PNe*150 instances as shown in Figure 7.9. The improvement in *SMP* is notable in region where *P* and *N* values are close (yellow area along the diagonal in Figure 7.9c). There is an improvement in *SCP* towards small to medium *P* and *N* valued instances as shown by the slightly yellow region in the lower left corner of Figure 7.9d. The end result is a smooth improvement in obtaining average total penalties (*TP*) especially along *P* = 0.10 region as shown in Figure 7.9a. Minimum total penalties are also improved especially along the diagonals of Figure 7.9b.

(a) Δ_{TP} of $ELS - BCK$

(b) Δ_{min} of $ELS - BCK$

(c) Δ_{SMP} of $ELS - BCK$

(d) Δ_{SCP} of $ELS - BCK$
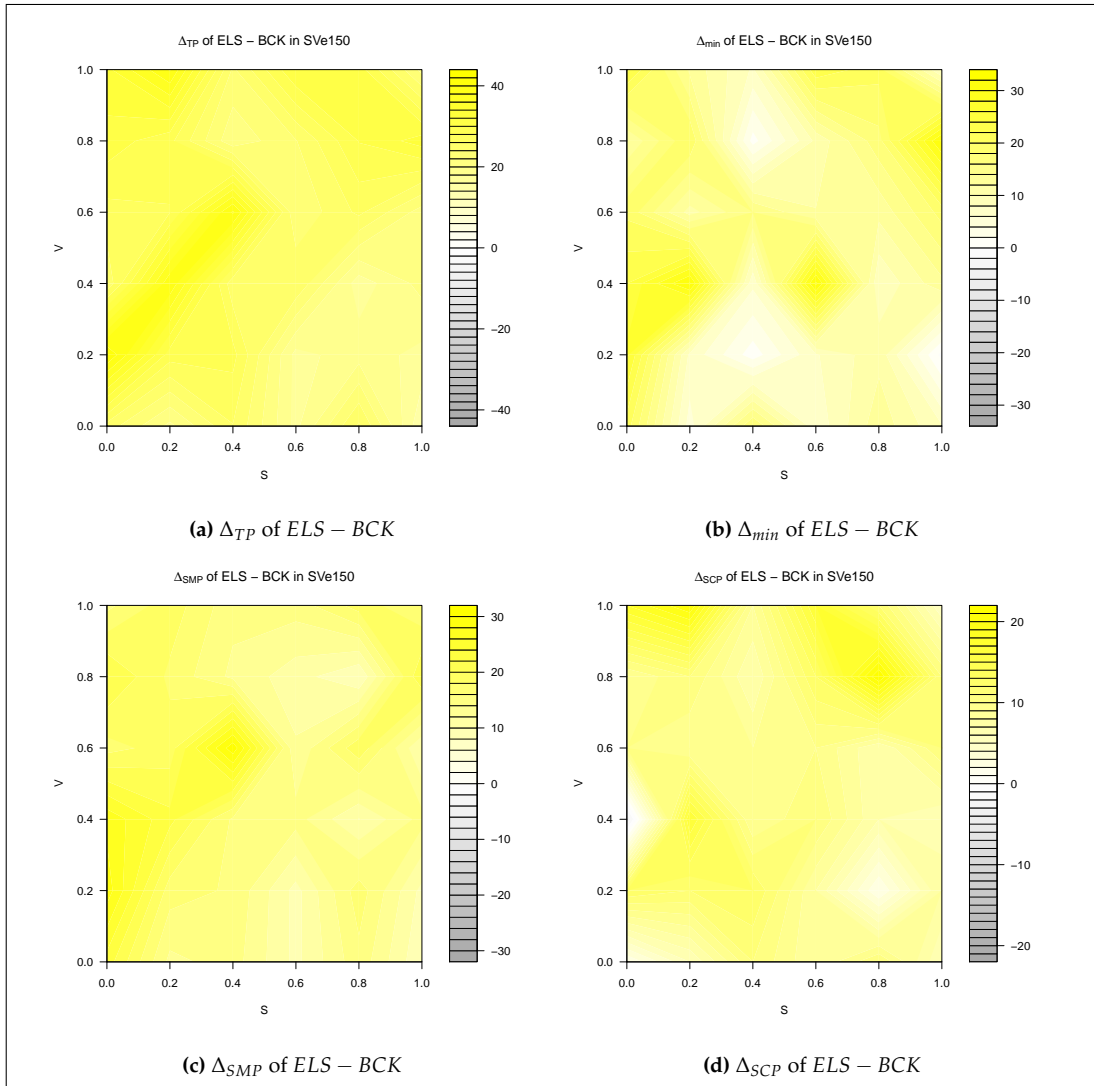
Figure 7.8: Differences in average total cost penalty (Δ_{TP}), minimum penalty (Δ_{min}), average space mis-use penalty (Δ_{SMP}), and average soft constraint penalty (Δ_{SCP}) after applying evolutionary local search (*ELS*) and single solution local search with backtracking (*BCK*) on *SVe*150 dataset (*ELS* − *BCK*). *ELS* and *BCK* are represented by grey and yellow regions respectively.

(a) Δ_{TP} of $ELS - BCK$

(b) Δ_{min} of $ELS - BCK$

(c) Δ_{SMP} of $ELS - BCK$

(d) Δ_{SCP} of $ELS - BCK$
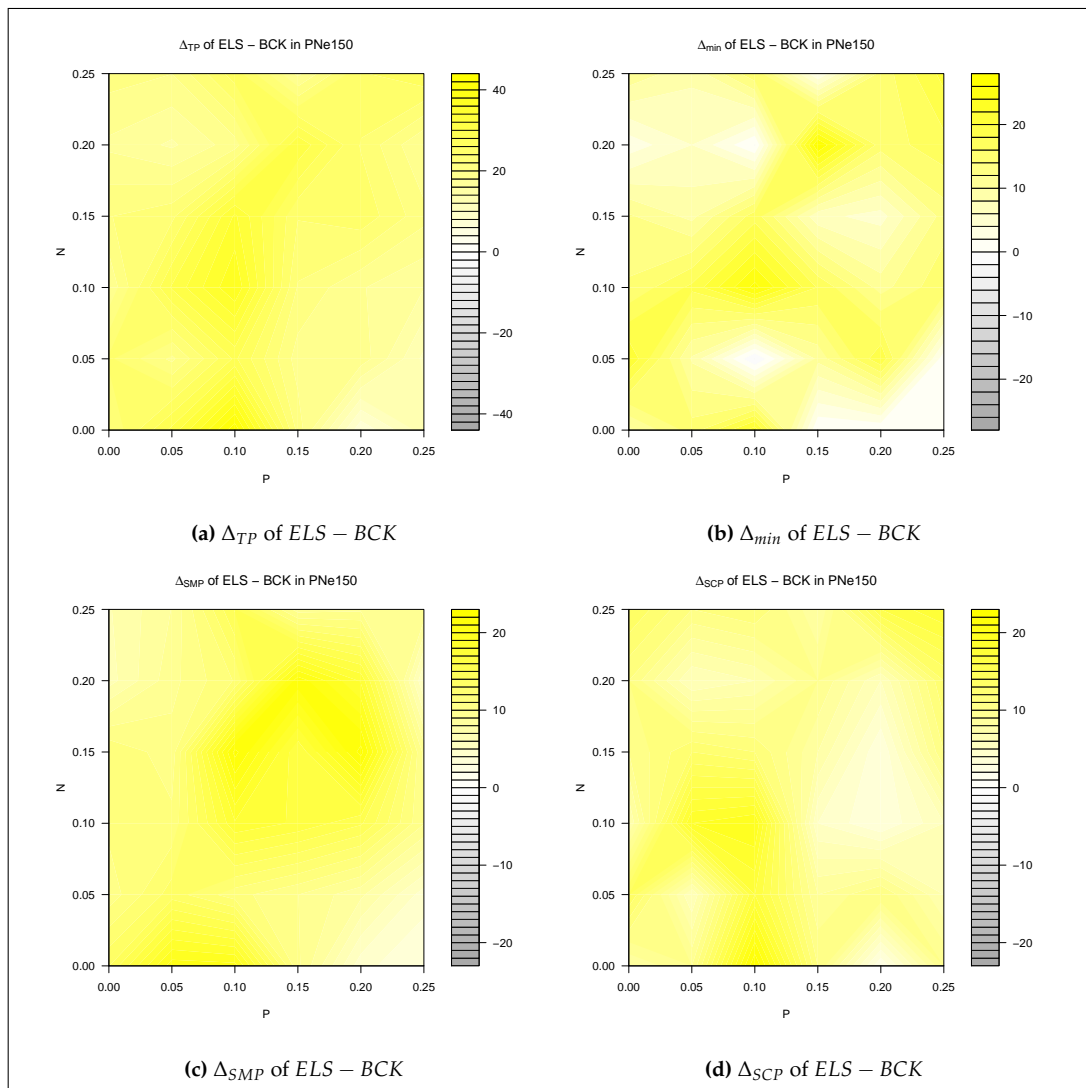
Figure 7.9: Differences in average total cost penalty (Δ_{TP}), minimum penalty (Δ_{min}), average space mis-use penalty (Δ_{SMP}), and average soft constraint penalty (Δ_{SCP}) after applying evolutionary local search (ELS) and single solution local search with backtracking (BCK) on $PNe150$ dataset ($ELS - BCK$). ELS and BCK are represented by grey and yellow regions respectively.

Method	nott1					nott1b				
	TP	σ	SMP	SCP	min	TP	σ	SMP	SCP	min
LS	479.46	72.58	275.98	203.48	**349.06**	258.98	24.36	127.80	131.18	233.13
ELS	417.58	26.77	313.93	103.65	379.26	251.97	15.63	130.91	121.06	228.98
BCK	**391.66**	23.53	298.30	93.36	353.62	**235.20**	7.20	134.37	100.83	**223.13**

Table 7.3: Results on *nott*1 and *nott*1*b* instances after applying *LS*, *ELS*, and *BCK*

7.4.6 Comparison of Heuristic Methods in nott1 Instances

In this section, results of applying local search (*LS* - described in Chapter 5), evolutionary local search (*ELS* - described in Chapter 6) and single solution local search with backtracking (*BCK* - described in Section 7.3) on two *nott*1 instances are presented. Each instance was given 10 runs (3 minutes each). Results are presented in Table 7.3. Columns *TP*, σ, *SMP*, *SCP*, and *min* represent the average total penalty, standard deviation of total penalty, average space misuse penalty, soft constraint violation penalty and minimum total penalty after 10 runs.

Results show the progression of algorithms through the thesis when applied to *nott*1 and *nott*1*b* instances. *TP* and *SCP* decrease by going from *LS* to *ELS* and then from *ELS* to *BCK*. On the other hand, it is observed that *LS* algorithm seems to be prioritising *SMP* over other components, that is why it is better than *ELS* and *BCK* in this component.

A big drawback of *LS* over *ELS* and *BCK* is its inconsistency as evidenced by the large standard deviation (σ) value. Actually, the minimum result for *nott*1 instance is obtained by *LS* in this test despite its poorest average performance. In any case, all local search heuristic methods described in this thesis yield better results for *nott*1 and *nott*1*b* instances than any meta-heuristic approach reported previously in the literature.

BCK algorithm is the highest performing local search heuristic in most parameters. Its average performance is best and it gives consistent results with very small standard deviations. This is another reason *BCK* is the desired local search heuristic that will be hybridised with integer programming methods.

7.4.7 Comparison of Combination Methodologies

For these experiments, the IP model with the floor variables was chosen and implemented using CPLEX. For the heuristic to be used with IP, the single solution local search and tabu search heuristics were used with backtracking with large mutation. In

Tables 7.4a and 7.4b, results obtained after 10 runs (6 minutes each) using various combination methodologies are given. Columns μ and σ represent the mean and standard deviation of 10 runs with each combination. The combination methodologies tested are as follows:

- **PureIP:** Running the IP solver alone for 360 seconds.

- **HeurHC6:** Running just the single solution local search heuristic for 360 seconds.

- **HeurInitPureIP:** Running the local search heuristic for 90 seconds and feeding the best solution obtained to the IP solver as a solution initialisation method. IP runs for 270 seconds.

- **HCIPTandem:** Running the Local Search heuristic + IP solver in tandem three times (3 x (60 seconds HC + 60 seconds IP)) while passing only the best solution obtained between two stages.

- **TSIPTandem:** Running the Tabu Search heuristic + IP in tandem three times (3 x (60 seconds HC + 60 seconds IP)). The best solution obtained is passed between two stages. Additionally, fixed variables are passed from the IP to the Tabu Search heuristic in order to initialise the tabu list.

- **HCIPMixed:** Whenever a best solution is obtained with the IP solver, the single solution local search heuristic with backtracking is called to improve this best solution for 10 seconds.

- **TSIPMixed:** Whenever a best solution is obtained with the IP solver, the single solution tabu search heuristic is called to improve this best solution for 10 seconds. Fixed variables are also transferred to the tabu Search heuristic.

Strictly comparing the average best results over 15 instances produced by the above combinations, it was observed that the most successful method was *HeurInitPureIP* which initialised the MIP solver with the hill climbing local search heuristic with backtracking. *HeurInitPureIP* provided best average results for six of the instances. Its performance in other instances was competitive as well. *HeurInitPureIP* was followed by strictly IP or heuristic methods (*PureIP* and *HeurHC6* respectively) with three best results each. However, the performance of *PureIP* and *HeurHC6* could sometimes be inconsistent providing either best or worst results for certain instances. Other combination methods were not particularly successful in providing the best results for the instances. However, their average performances were consistent across the board and were not significantly affected by certain instances.

Instance	PureIP μ	PureIP σ	HeurHC6 μ	HeurHC6 σ	HeurInitPureIP μ	HeurInitPureIP σ
$S_{0.00}V_{0.00}$	**10.50**	0.00	18.90	6.69	13.80	12.39
$S_{0.20}V_{0.40}$	93.50	0.00	94.47	7.24	**86.53**	9.33
$S_{0.40}V_{0.80}$	237.20	0.00	192.49	6.32	**191.97**	7.73
$S_{0.60}V_{0.60}$	188.50	0.00	**181.15**	6.19	183.57	8.09
$S_{0.80}V_{0.40}$	184.10	0.00	163.00	4.43	**161.76**	7.77
$S_{1.00}V_{1.00}$	300.50	0.00	276.98	7.66	279.74	11.86
$P_{0.00}N_{0.00}$	92.00	0.00	76.70	4.43	**74.65**	4.70
$P_{0.05}N_{0.10}$	144.00	0.00	140.35	4.87	139.42	10.51
$P_{0.10}N_{0.20}$	203.30	0.00	195.85	5.94	**191.66**	4.40
$P_{0.15}N_{0.15}$	184.90	0.00	**156.44**	5.48	158.50	6.41
$P_{0.20}N_{0.10}$	133.50	0.00	133.81	4.30	**131.46**	6.24
$P_{0.25}N_{0.25}$	**196.60**	0.00	201.22	6.07	202.65	7.75
*nott*1	**327.16**	0.00	379.18	15.71	339.42	14.05
*nott*1b	286.71	0.00	**231.15**	8.17	239.48	8.48
*nott*1c	305.73	0.00	310.10	8.68	305.73	0.00

(a) Combination methods - Part 1

Instance	HCIPTandem μ	HCIPTandem σ	TSIPTandem μ	TSIPTandem σ	HCIPMixed μ	HCIPMixed σ	TSIPMixed μ	TSIPMixed σ
$S_{0.00}V_{0.00}$	11.65	11.11	12.00	13.17	23.05	8.07	24.60	8.54
$S_{0.20}V_{0.40}$	95.94	5.17	101.26	8.66	96.94	9.93	94.85	13.17
$S_{0.40}V_{0.80}$	195.25	7.35	199.86	10.49	194.32	10.79	202.74	18.76
$S_{0.60}V_{0.60}$	184.11	6.68	184.41	6.87	185.83	10.92	189.34	11.22
$S_{0.80}V_{0.40}$	166.17	9.24	174.51	6.69	174.47	8.75	176.69	9.28
$S_{1.00}V_{1.00}$	278.02	9.84	281.11	12.85	**273.65**	10.50	288.81	16.06
$P_{0.00}N_{0.00}$	78.10	6.95	82.15	5.14	78.20	5.77	81.70	8.84
$P_{0.05}N_{0.10}$	**138.48**	8.21	145.74	11.19	142.08	11.18	139.93	10.62
$P_{0.10}N_{0.20}$	194.04	4.75	197.80	7.13	194.14	7.80	197.57	8.70
$P_{0.15}N_{0.15}$	162.62	8.11	164.93	6.32	158.92	5.28	161.51	6.99
$P_{0.20}N_{0.10}$	134.04	5.58	139.56	6.67	136.48	7.00	145.08	11.12
$P_{0.25}N_{0.25}$	204.30	6.71	205.47	9.41	206.48	11.85	201.59	9.51
*nott*1	340.62	15.71	346.20	22.28	337.22	14.84	342.12	16.03
*nott*1b	235.84	9.45	237.44	8.81	242.59	9.21	244.10	15.61
*nott*1c	305.73	0.00	305.73	0.00	305.73	0.00	305.73	0.00

(b) Combination methods - Part 2

Table 7.4: Results obtained in several instances using different hybridisations
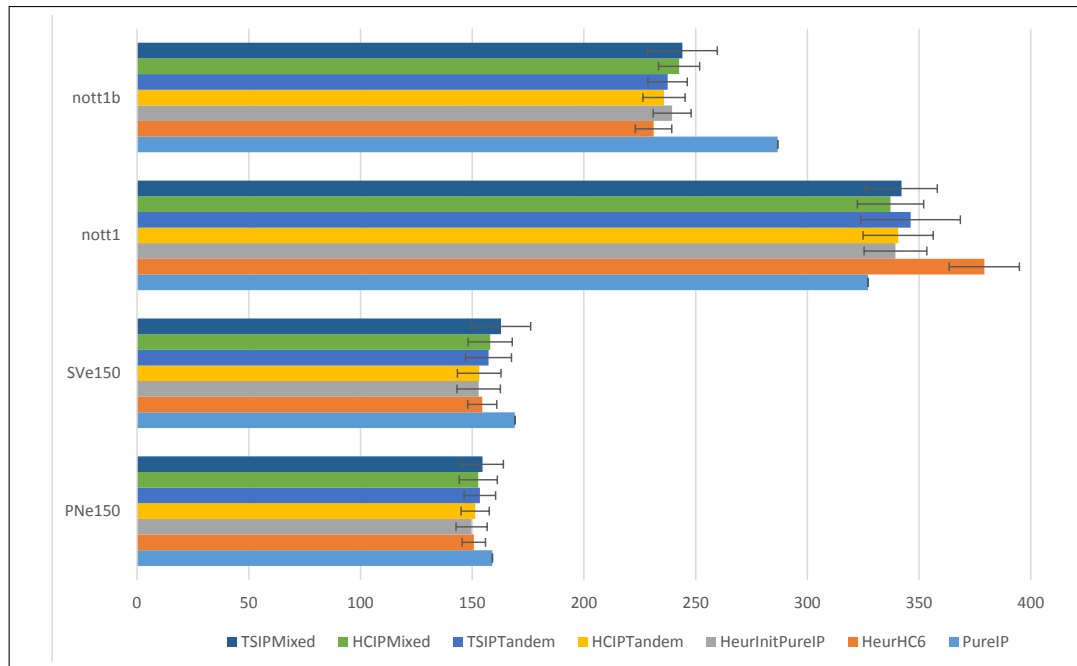
Figure 7.10: Average total penalties in instances

Figure 7.10 depicts the average total penalty and standard deviation on the total penalty in PNe150 and SVe150 (over 6 different instances) in addition to *nott*1 and *nott*1*b* instances respectively.

As it can be seen from Figure 7.10, for the average total penalty and standard deviation over PNe150 and SVe150 instances, all seven test methods yielded close results. However, a slight improvement was observed when the heuristic was used as an initialisation method to the IP method (*HeurInitPureIP*). In any case, just running the IP model alone yielded poorer results in PNe150 and SVe150 instances, combining the heuristic with/within the CPLEX always provided better results in five different combination methods. However, running the hill climbing heuristic (*HeurHC6*) alone always gave higher solutions over the heuristic embedded IP methods (*HCIPInTandem*, *TSIPInTandem*, *HCIPMixed*, *TSIPMixed*). Therefore, improvements over IP were due to efficacy of the heuristic rather than the combination technique.

For *nott*1 instance, a completely different outcome was observed. This instance was always solved more efficiently by IP solvers and the outcome was not different in this test case either. Running just the pure integer programming method offered by far the best results, while running just the hill climbing heuristic provided the worst ones. Any combination technique reduced the deficiency of the heuristic considerably and provided sufficient average performance in this instance.

However, the case with *nott*1*b* instance was similar to *PNe*150 and *SVe*150 instances but completely opposite to *nott*1 instance. The IP solver results were considerably behind any pure heuristic or heuristic-IP combination technique. The hill climber heuristic was the fastest in this case, while any combination method provided close competitive results and they were still far ahead of the IP Solver.

Figures 7.11, 7.12, 7.13, and 7.14 are graphical representations of each space penalty component (*overuse*, *underuse*, *total space misuse*, and *soft constraint violation penalties*) in a subset of *PNe*150, *SVe*150 instances in addition the *nott*1 and *nott*1*b* instances respectively. Since six different *PNe*150 and *SVe*150 instances each were tested in experiments, Figures 7.11 and 7.12 depict the results averaged over the six *PNe*150 and *SVe*150 instances respectively. The small vertical lines in bars in each graph stand for the standard deviation over 10 runs.

The individual analysis of each component of the total penalty was performed to see whether a technique was more efficient in one aspect of the performance criteria. For the overuse and underuse penalties in *PNe*150 and *SVe*150 instances, all methods provided very close average results and standard deviations apart from the *PureIP*; therefore, no definitive conclusion could be drawn about the overuse/underuse bias of an algorithm. Consequentially, a bias in total space misuse penalty was not observed especially with methods that utilised heuristics. However, for constraint penalty, it was observed that the IP solver had a bias for the soft constraint violations in SV instances. Although the IP solver provided worst results for the total space misuse penalty, it yielded the best soft constraint violation penalty values.

While analysing the results in *nott*1 instance, the main shortcoming of the hill climbing heuristic was noticed: Although *HeurHC6* was most successful in reducing the soft constraint violation penalties, it really struggled in reducing the space misuse which explained its overall poor performance in this instance. Other methods had a slight bias in reducing the overuse penalty over underuse in this instance. In *nott*1*b* instance, the heuristic and heuristic/IP combinations provided close results in individual penalties, the performance in each sub-penalty component was consistent with their overall strong performance in total penalty while the *PureIP* was really poor in each sub-component which explained its poor performance in this instance.

7.4.8   Discussion of Results

After the experiments on *PNe*150, *SVe*150, and *nott*1 instances, it was observed that there was not necessarily a clear winner across the board. For *PNe*150 and *SVe*150 in-

(a) Overuse (O) and Underuse (U) Penalties

(b) Constraint (SCP) and Misuse (SMP) Penalties
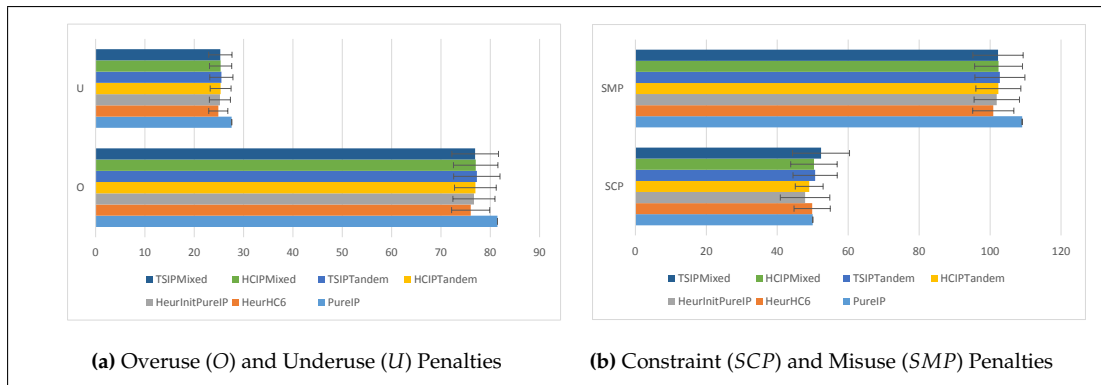
Figure 7.11: Average individual penalty values in a subset of $PNe150$ instances

(a) Overuse (O) and Underuse (U) Penalties

(b) Constraint (SCP) and Misuse (SMP) Penalties
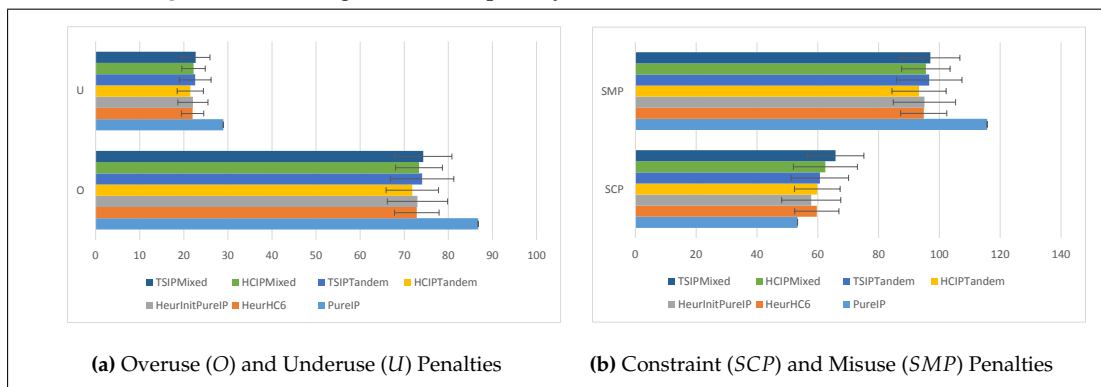
Figure 7.12: Average individual penalty values in a subset of $SVe150$ instances

(a) Overuse (O) and Underuse (U) Penalties

(b) Constraint (SCP) and Misuse (SMP) Penalties
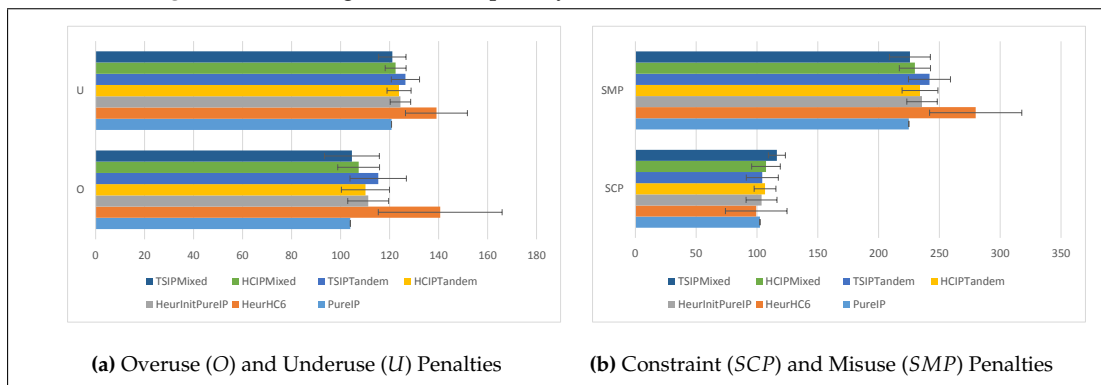
Figure 7.13: Average individual penalty values in $nott1$ instance

(a) Overuse (O) and Underuse (U) Penalties

(b) Constraint (SCP) and Misuse (SMP) Penalties
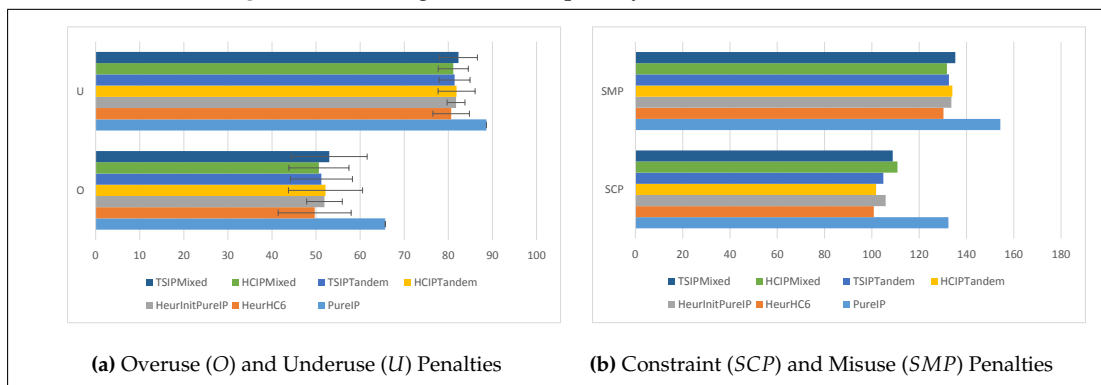
Figure 7.14: Average penalty values in $nott1b$ instance

188

stances, the *HeurInitPureIP* (using the hill climbing heuristic as an initialisation function to the heuristic) performed slightly better than other methods. For *nott*1 instance, pure IP solver yielded significantly better results over heuristic while it ended far behind in *nott*1*b* instance after the heuristic and hybrid methods.

The methods other than *HeurInitPureIP* were not particularly successful in providing best results for instances and were usually beaten by heuristic or MIP only methods in this criterion. However, these heuristic-IP combination methods were not susceptible to poor performance in certain instances unlike *PureIP* and *HeurHC6*, their average performances were still quite competitive. Therefore, they might be still preferred if more stable but not necessarily best results after a number of runs are sufficient if these methods are applied to another set of OSA problem.

Initially desired results for access and usage of fixed variables to guide the local search were not obtained. Passing fixed variables from the IP solver to the heuristic to populate tabu lists in the tabu search heuristic provided no benefit. This was similar to the case as in Chapter 6 where using tabu search algorithm was worse than using the hill climber algorithm. One explanation to this phenomenon could be the greediness of the local search algorithms. At each iteration in the local search, the best move was searched and selected from roughly one-third of the available *relocate* neighbourhood. Restricting not only potentially cyclic moves but also the fixed variables passed from the IP solver by tabu lists in this greedy framework, might prevent some of the moves that had to be made in order to find better moves later. Therefore, in the future, algorithms that are less restrictive in penalising some section of the search space can be explored with IP models. There should probably be more research that has to be performed on different tabu search implementations. One alternative would be not supplying all fixed variables to the tabu search algorithm to fill the tabu lists. Instead, a dynamically or probabilistically created subset of these variables can be used to populate the tabu lists.

Another big hurdle with the combination methods (especially with the ones that embed heuristic within the IP) was the API restrictions. Whenever a heuristic was used within CPLEX via an implementation of a heuristic callback function (as in *HCIPMixed* and *TSIPMixed*), CPLEX turned down its default dynamic search behaviour and operated in the traditional branch and cut mode for solving the MIP problem. Unfortunately, the performance of CPLEX in traditional branch and cut mode was really poor compared to dynamic search mode. As a result, the performance of embedded combinations (*HCIPMixed* and *TSIPMixed*) suffered as well as the method that transferred fixed variables from CPLEX to heuristic *TSIPTandem*.

7.5 Conclusion

In this chapter, an initial investigation of several combination techniques for hybridising mathematical programming and heuristics for solving the office space allocation problem was presented. The evolutionary local search algorithm proposed in Chapter 6 was stripped off its evolutionary components and reduced to a single solution local search heuristic. The goal of this approach was to simplify the embedding of the heuristic within an IP solver.

The single solution local search heuristic was augmented with a large mutation based backtracking method in order to return back to a previous location in the search space to avoid local optima. Further optimisations of the delta and update stages made this heuristic the best performing one to solve the office space allocation problem as evidenced in Sections 7.4.4 and 7.4.5. This meta-heuristic by itself was also competitive against the integer programming model with the floor variables in Chapter 4 unlike local search in Chapter 5 and evolutionary local search algorithm in 6.

Various combinations methodologies to hybridise the mathematical programming and heuristic were proposed. These included using heuristic as an initialisation function to the IP solver, using heuristic and IP solver in tandem, passing fixed variables from the IP solver to the heuristic to initialise tabu lists, and calling the heuristic whenever an incumbent (best) solution was found with the IP solver.

It was observed that the best performance was observed when the heuristic was fed into the IP solver as an initialisation routine. The performance of the CPLEX solver significantly improved when a high quality solution was set as a starting point for its dynamic search. Unfortunately, more advanced ideas, such as passing the fixed variables to populate the tabu list in a tabu search algorithm were not successful. Similar to the case as in Chapter 6.5, restricting some of the available moves with the tabu lists reduced the performance of the local search.

Embedding the heuristic within CPLEX as a heuristic callback and running the heuristic whenever a new best solution found did not offer any benefits over non-embedded techniques. Unfortunately, CPLEX shut down its dynamic search whenever a heuristic callback was implemented and this was the main reason the performance was not as good as the non-embedded techniques.

Conclusion

In this chapter, the summary of the research performed from the perspectives of the office space allocation problem, mathematical programming, and meta-heuristic is presented in Section 8.1. Section 8.2 describes some of the proposed work that can be done in the future. Final remarks are given in Section 8.3.

8.1 Summary of Work

8.1.1 From the Office Space Allocation Problem Perspective

In this thesis, the optimisation problem of office space allocation was considered. The definition of the optimisation problem was derived from the works of Burke et al. [2001b], Burke et al. [2001c], Burke et al. [2001a], Burke et al. [2005], Landa-Silva [2003], and Landa-Silva and Burke [2007]. The problem definition was extended with three more hard and soft constraints. The number of data instances during the beginning of this work was quite limited. In order to overcome this hurdle, a data instance generator was implemented with the goal of creating test cases where an approximate bound on the optimal value could be determined. Two parametrised data instance sets ($SVe150$ and $PNe150$) were generated with properties similar to the $nott1$ dataset which was previously investigated in the literature.

8.1.2 Mathematical Models for Solving Office Space Allocation

In this thesis, various mathematical models were considered and proposed for the first time for this variant OSA problem. The focus was on binary integer formulations instead of general integer models. Two binary models were developed by using and not using the floor variables (which held the information about the floor an entity was al-

located to). The model without the floor variables offered more generalisability but less performance while the model with the floor variables yielded superior results although it was only applicable to instances where the *nearby* and *away from* relationships were defined using the floors. We believe that identifying and exploiting structures like floor and sections and formulating the constraints around them would be paramount for obtaining high quality results with integer programming.

Using the model without the floor variables, further analysis of the problem was conducted. The space misuse (overuse, underuse) and the soft constraint violations were analysed over two datasets encompassing seventy-two instances specially created for this analysis. It was observed that the most important aspect about determining the difficulty of an instance was the balance between the overuse and underuse of space in rooms. In a more balanced instance where there was a roughly comparable amount of rooms with either overuse or underuse, the instance became more difficult to solve. It was also observed that increasing the weight of the soft constraint violations improved the solution quality.

8.1.3 Heuristic Approaches for Solving Office Space Allocation

Various meta-heuristic approaches were implemented to solve the office space allocation problem. Initial implementations (Chapter 5) started with general stochastic local search algorithms like threshold acceptance, great deluge, and simulated annealing. These algorithms were similar in nature to the methods devised previously in the literature (Landa-Silva [2003], Landa-Silva and Burke [2007], Lopes and Girimonte [2010], etc.). The goal was to observe the performance of local search with a stochastic neighbourhood which was comprised of *relocate* and *swap* moves randomly selected and applied to the current incumbent solution). It was observed using the combination of these two moves to generate the neighbourhood gave superior performance rather than just using a single move.

It was observed that the performance of using such meta-heuristic approaches could be uneven across the problem instances used. The algorithm might sometimes get stuck in very poor local optima and as a result, quite high standard deviations were observed in results. The relative performance compared to integer programming models described in Chapter 4 were not satisfactory. Therefore, a more advanced evolutionary local search heuristic algorithm was proposed in Chapter 6. It was observed that exploring a significantly large section of a given neighbourhood was necessary to find good *relocate* moves. A very fast cost calculation method allowed quick modifi-

cations of the changes in objective function value of performing each *relocate* move in a neighbourhood given a certain solution. Significant speed-ups were observed using cost update algorithms for nine different constraints and the space misuse calculations. However, *swap* move had to be dropped in this algorithm due to poor synergy with *relocate* move and difficulty of implementing an exact cost update method that could handle *swap* moves efficiently.

The final addition to the local search heuristic was implemented due to the tendency of non-improvement of the solution quality in the evolutionary local search algorithm. Since the algorithm was applied greedily (by selecting locally high quality moves within a subset of the neighbourhood), it was more likely to get stuck in local optima. A large mutation based backtracking method which disrupted the current best solution considerably was applied if no improvement to the solution quality was observed for a period of past iterations. This final local search heuristic with large mutation backtracking (proposed in Chapter 7) was the most successful heuristic design in this thesis.

At the core of local search heuristic designs in this thesis, there was an *iterated local search* [Lourenco et al., 2002] inspired framework. The algorithm periodically retracted back to the global best solution encountered during the search and tried to improve this best solution continuously after a small perturbation was applied on it. Even the replacement strategy in *ELS* had to be modified according to this framework. Results were not particularly satisfactory and they were prone to getting stuck in local optima if this ILS inspired framework was not used and the algorithm continuously operated on the current solution. This was the first time an ILS based approach was used in the literature for this problem. We believe that the *intensification* of search around the global best solution was one of the major reasons for success of the local search methods proposed in this thesis.

8.1.4 Hybridisation of Mathematical Modelling and Heuristics

The ultimate goal of this project was to develop OSA solution frameworks which combined mathematical programming with meta-heuristics. The benefit of such an approach was to take advantage of exact methods (in terms of the guarantee of optimality or degree of closeness to it) and meta-heuristics (the ability of generating quick solutions and handling larger problems that the mathematical programming models could not handle). Modifications to the meta-heuristics and mathematical programming models were done in order to create various combination methodologies.

However, mixed results were obtained after this research. It was observed that the most successful approach was simply to use the meta-heuristic as a short initialisation function for the integer linear programming solver. However, the results obtained were not significantly better than using the meta-heuristic alone. The other combination techniques that tried to embed the meta-heuristic within the IP solver as a solution improver, or that used the IP as a means of passing fixed variables to a tabu search meta-heuristic were unfortunately not very successful. Some API restrictions in CPLEX limited the ability to access the bounds of the variables as the search deepened. Additionally, CPLEX disabled its advanced functionality (dynamic search) when meta-heuristic was embedded within it via heuristic callbacks.

8.1.5 Overall Summary

In this thesis, various mathematical programming models and meta-heuristics were implemented to solve the office space allocation problem. The definition of the problem was extended from Landa-Silva [2003]. Best results in the literature were obtained for the instances in *nott1* dataset. Some of these results were proved optimal by using CPLEX as well. For small to medium instances, the order of algorithm quality was observed as follows:

- Model with floor variables initialised with single solution local search heuristic with backtracking.

- Single solution local search heuristic with large mutation backtracking.

- Model with floor variables.

- Evolutionary local search algorithm without backtracking.

- Model without floor variables.

- Stochastic local search algorithms based on acceptance/rejection of moves and simple cost calculation procedures.

However, any of these algorithms can still be used as a basis for future office space allocation problems if a specific variant of the problem disallows other algorithms. During the process of this research, results obtained by integer programming models increased considerably due to improvements on the commercial IP solvers like CPLEX. However, due to the licensing costs of these commercial tools, meta-heuristic approaches proposed in this thesis can be preferred if quick but not necessarily provably optimal results is sufficient.

8.2 Future Work

8.2.1 Modification of Office Space Allocation Problem

The space allocation problem considered in this thesis was already an extension of the variant described in Landa-Silva [2003]. In this thesis, the proximity constraints (*adjacency, nearby, away from*) were defined using lists of rooms which held the information about these relationships. However, in certain cases, the proximity relationship between the entities can be described using metric distance relationships instead of such lists. For example, one nearby relationship can be defined over several entities being allocated within 20 square meters of the group head, while another nearby relationship is defined over 10 or 30 square meters. Additionally, the *nearby* constraint can be embedded within the objective function as the minimisation of the cumulative distance to the group head within the group similar to as in a typical clustering problem [Everitt et al., 2009].

8.2.2 Re-allocation Problem in Office Space Allocation

The re-allocation problem is the task of re-optimisation of a previous allocation due to changes in entity, room structures and the constraints associated with them. Although a simple model was given for the re-allocation problem in Chapter 4, more adjustments and implementations are required. Since the complexity of the re-allocation problem is directly related to the amount of changes applied to the structures in the problem, test cases should be designed considering these aspects.

Some of the case studies that can be considered in the future are as follows:

- The number of new entities that has to be allocated.

- The number of old entities that no longer should be allocated.

- The amount of reorganisation of the room and floor structure. The number of rooms that have to be split, merged, destroyed, newly built or went through space adjustment has to be considered.

- The number and the type of the constraints that are related to the new entity and room structures have to be considered.

- The importance of restructuring, or any other aspect in addition to the number of reallocated entries should be considered while modelling the new formulation of the objective function.

8.2.3 Modification of Mathematical Programming Models

Although major performance improvements were observed using a model with the floor variables (at the expense of generalisability), current models were not sufficient to handle larger instances. In fact, some preliminary experiments with a larger dataset (roughly twice the number of rooms and entities than the current instances) showed no improvement after the pre-solve stage of CPLEX. There was zero progression in the objective function value of the incumbent solution and very little improvement in the best bound obtained during the branch and bound stage even after a few hours of running time of CPLEX.

It was observed that the number of non-zero locations in simplex tables in the current models were really low (six percent to 10 percent of the whole table). Also, due to the constraints and the nature of the current problem (that an entity can only be allocated to one room), it may be possible to exploit several structures in the simplex table. A method that is considered to address these issues is *column generation* [Lübbecke and Desrosiers, 2005].

8.2.4 Modification of Heuristic Approaches

Throughout the design of the heuristic algorithms, the move operator that was frequently used was the *relocate* move (that an entity is moved from one room to another). Although this move allows the implementation of very fast and efficient cost calculation procedures, it may not be enough in making jumps in the search space to avoid local optima. In addition to the *relocate* move, the *swap* move was also incorporated in the search. In the future, more complex neighbourhood structures which involve reallocation of more than one entity can be considered. A simple approach is to divide a series of moves into a chain of *relocate* moves and use the update procedures described in Chapter 6. A more complex approach is to re-design the update procedures to handle the re-organisation of more than one entity. Since most of the current constraints are binary (that most involve two entities), two entity move update procedures should be the first approach. These procedures can also help designing constraint based move operators as well. However, it should be noted that initial attempts of coding update algorithms that could handle *swap* moves efficiently were quite challenging.

Another successful component of the heuristics was backtracking to return to a previous location in the search space. Although the preliminary research on making the backtracking dynamic was not successful, more attention should be given to this area. This is due the fact that the current static backtracking algorithm, while successful

early on during the search, can really struggle later. Planned research on backtracking can be on multiple backtrack points, the adjustment of mutation rate, and the number of (dynamic or static) iterations for a decision to backtrack.

In this thesis, some of the major heuristic building blocks (such as random and greedy components) were tested. One further extension is to use hyper-heuristics [Burke et al., 2003], [Burke et al., 2010] to govern some of the successful building blocks designed. Hyper-heuristics are high level heuristic methods that choose low level heuristics. The current focus on hyper-heuristic research is to build automated systems which allow quick prototyping of easy to implement building blocks with emphasis on generalisability.

8.2.5 Improving the Hybrid Mat-heuristic Methods

Current implementations of the mat-heuristics struggle due to some of the API restrictions of the IP solvers. Gurobi solver [Gurobi-Optimization, 2010] offers a simple callback interface which makes some of the hybrid implementations difficult. CPLEX solver [IBM-Ilog, 2013] on the other hand has a very extensive callback library, yet it still has limitations on accessing the upper-bounds on the variables in C++ Concert interface. Unfortunately, CPLEX disables its dynamic search whenever a callback function is implemented. As a result, performances of embedded and bound information based heuristics suffered compared to the non-embedded and no-bound information combinations. Revisit of these previous ideas is planned after the release of future versions of these commercial IP solvers. Application of non-commercial open source IP solvers can also be considered in order to avoid the licence costs of the commercial solvers.

8.3 Final Remarks

Office space allocation is an important problem in many organisations especially in situations space is a premium. In this thesis, a specific variant of this problem was investigated and analysed. State of the art integer programming models and meta-heuristic approaches for this problem were proposed. This thesis should serve as a seminal work in this area, and the analysis and algorithms presented in this thesis should provide a solid foundation for future research in this important problem.

Tables Associated with Chapter 4

In this section, some of the tables associated with Chapter 4 are given here. The tables are moved from Chapter 4 in order to improve the flow in the chapter.

A.1   Tables Associated with the Analysis of S, P, N, V by using Model without the Floor Variables

Tables A.1, A.2, and A.3 represent the results obtained under different S (*slack space*) and V (*soft constraint violation* rates) under *half, normal*, and *double soft constraint penalty* conditions respectively. Columns C, B, %, and AG represent the best objective function value, best lower bound, the percentage, and absolute gaps between the best objective value and bound respectively. Columns O, U, and O/U represent the overuse, underuse penalties, and the ratio between them respectively. Columns SCP, SMP, and SCP/SMP represent the soft constraint violation penalty, total space misuse penalty, and the ratio between them respectively. The respective analysis of these tables can be found in Sections 4.6.2 and 4.6.3.

Tables A.4, A.5, and A.6 represent the results obtained under different P (*positive slack space rate*) and N (*negative slack space rate*) under *half, normal*, and *double soft constraint penalty* conditions respectively. Other column descriptions are the same as in A.1, A.2, and A.3 as described in the paragraph above. The respective analysis of these tables can be found in Sections 4.6.4 and 4.6.5.

S	V	C	B	%	AG	O	U	$\frac{O}{U}$	SMP	SCP	$\frac{SCP}{SMP}$
0.00	0.00	21.00	0.00	1.00	21.00	14.00	7.00	2.00	21.00	0.00	0.00
0.00	0.20	26.50	11.65	0.56	14.85	11.00	5.50	2.00	16.50	10.00	0.61
0.00	0.40	62.50	23.46	0.62	39.04	25.00	12.50	2.00	37.50	25.00	0.67
0.00	0.60	63.50	29.89	0.53	33.61	19.00	9.50	2.00	28.50	35.00	1.23
0.00	0.80	125.00	44.86	0.64	80.14	50.00	25.00	2.00	75.00	50.00	0.67
0.00	1.00	114.00	51.68	0.55	62.32	46.00	23.00	2.00	69.00	45.00	0.65
0.20	0.00	39.80	22.72	0.43	17.08	27.80	12.00	2.32	39.80	0.00	0.00
0.20	0.20	47.60	34.76	0.27	12.84	23.00	9.60	2.40	32.60	15.00	0.46
0.20	0.40	62.80	45.87	0.27	16.93	29.80	13.00	2.29	42.80	20.00	0.47
0.20	0.60	81.80	51.62	0.37	30.18	35.80	16.00	2.24	51.80	30.00	0.58
0.20	0.80	145.90	65.54	0.55	80.36	65.20	30.70	2.12	95.90	50.00	0.52
0.20	1.00	168.60	69.86	0.59	98.74	77.00	36.60	2.10	113.60	55.00	0.48
0.40	0.00	101.40	61.06	0.40	40.34	63.00	28.40	2.22	91.40	10.00	0.11
0.40	0.20	104.40	70.40	0.33	34.00	65.00	29.40	2.21	94.40	10.00	0.11
0.40	0.40	123.80	83.12	0.33	40.68	64.60	29.20	2.21	93.80	30.00	0.32
0.40	0.60	134.40	86.80	0.35	47.60	65.00	29.40	2.21	94.40	40.00	0.42
0.40	0.80	162.70	97.99	0.40	64.71	87.20	40.50	2.15	127.70	35.00	0.27
0.40	1.00	214.40	105.74	0.51	108.66	105.00	49.40	2.13	154.40	60.00	0.39
0.60	0.00	105.60	88.14	0.17	17.46	87.00	13.60	6.40	100.60	5.00	0.05
0.60	0.20	119.80	97.51	0.19	22.29	89.80	15.00	5.99	104.80	15.00	0.14
0.60	0.40	128.20	108.93	0.15	19.27	85.40	12.80	6.67	98.20	30.00	0.31
0.60	0.60	157.20	113.30	0.28	43.90	101.40	20.80	4.88	122.20	35.00	0.29
0.60	0.80	200.50	126.12	0.37	74.38	113.60	26.90	4.22	140.50	60.00	0.43
0.60	1.00	226.10	131.65	0.42	94.45	134.00	37.10	3.61	171.10	55.00	0.32
0.80	0.00	117.70	85.06	0.28	32.64	80.20	22.50	3.56	102.70	15.00	0.15
0.80	0.20	153.90	95.26	0.38	58.64	101.00	32.90	3.07	133.90	20.00	0.15
0.80	0.40	160.30	104.73	0.35	55.57	98.60	31.70	3.11	130.30	30.00	0.23
0.80	0.60	147.60	109.89	0.26	37.71	86.80	25.80	3.36	112.60	35.00	0.31
0.80	0.80	228.90	122.48	0.46	106.42	121.00	42.90	2.82	163.90	65.00	0.40
0.80	1.00	293.20	126.45	0.57	166.75	157.20	61.00	2.58	218.20	75.00	0.34
1.00	0.00	154.90	123.85	0.20	31.05	116.00	23.90	4.85	139.90	15.00	0.11
1.00	0.20	156.10	133.71	0.14	22.39	116.80	24.30	4.81	141.10	15.00	0.11
1.00	0.40	177.10	143.43	0.19	33.67	120.80	26.30	4.59	147.10	30.00	0.20
1.00	0.60	186.00	145.09	0.22	40.91	123.40	27.60	4.47	151.00	35.00	0.23
1.00	0.80	261.90	154.40	0.41	107.50	164.00	47.90	3.42	211.90	50.00	0.24
1.00	1.00	268.60	162.68	0.39	105.92	161.80	46.80	3.46	208.60	60.00	0.29

Table A.1: Effects of *slack* and *violation* rates on results under *half soft constraint penalty condition* in *SVe*150 dataset instances

S	V	C	B	%	AG	O	U	$\frac{O}{U}$	SMP	SCP	$\frac{SCP}{SMP}$
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	NA	0.00	0.00	NA
0.00	0.20	61.50	21.47	0.65	40.03	21.00	10.50	2.00	31.50	30.00	0.95
0.00	0.40	59.50	43.08	0.28	16.42	13.00	6.50	2.00	19.50	40.00	2.05
0.00	0.60	115.50	55.76	0.52	59.74	37.00	18.50	2.00	55.50	60.00	1.08
0.00	0.80	161.50	76.58	0.53	84.92	61.00	30.50	2.00	91.50	70.00	0.77
0.00	1.00	197.00	88.31	0.55	108.69	78.00	39.00	2.00	117.00	80.00	0.68
0.20	0.00	36.80	22.76	0.38	14.04	25.80	11.00	2.35	36.80	0.00	0.00
0.20	0.20	53.80	45.15	0.16	8.65	23.80	10.00	2.38	33.80	20.00	0.59
0.20	0.40	86.30	66.21	0.23	20.09	38.80	17.50	2.22	56.30	30.00	0.53
0.20	0.60	114.80	76.42	0.33	38.38	37.80	17.00	2.22	54.80	60.00	1.09
0.20	0.80	193.60	94.30	0.51	99.30	77.00	36.60	2.10	113.60	80.00	0.70
0.20	1.00	250.90	103.45	0.59	147.45	115.20	55.70	2.07	170.90	80.00	0.47
0.40	0.00	85.80	63.10	0.26	22.70	52.60	23.20	2.27	75.80	10.00	0.13
0.40	0.20	109.40	83.67	0.24	25.73	55.00	24.40	2.25	79.40	30.00	0.38
0.40	0.40	139.70	101.42	0.27	38.28	75.20	34.50	2.18	109.70	30.00	0.27
0.40	0.60	159.40	110.11	0.31	49.29	75.00	34.40	2.18	109.40	50.00	0.46
0.40	0.80	200.10	126.72	0.37	73.38	88.80	41.30	2.15	130.10	70.00	0.54
0.40	1.00	296.70	137.80	0.54	158.90	153.20	73.50	2.08	226.70	70.00	0.31
0.60	0.00	128.90	92.97	0.28	35.93	99.20	19.70	5.04	118.90	10.00	0.08
0.60	0.20	133.30	112.59	0.16	20.71	88.80	14.50	6.12	103.30	30.00	0.29
0.60	0.40	168.50	131.12	0.22	37.38	105.60	22.90	4.61	128.50	40.00	0.31
0.60	0.60	180.40	140.55	0.22	39.85	100.20	20.20	4.96	120.40	60.00	0.50
0.60	0.80	246.70	161.17	0.35	85.53	124.40	32.30	3.85	156.70	90.00	0.57
0.60	1.00	293.90	171.97	0.41	121.93	149.20	44.70	3.34	193.90	100.00	0.52
0.80	0.00	122.40	91.07	0.26	31.33	80.00	22.40	3.57	102.40	20.00	0.20
0.80	0.20	148.00	111.18	0.25	36.82	90.40	27.60	3.28	118.00	30.00	0.25
0.80	0.40	183.00	127.75	0.30	55.25	100.40	32.60	3.08	133.00	50.00	0.38
0.80	0.60	193.60	139.86	0.28	53.74	100.80	32.80	3.07	133.60	60.00	0.45
0.80	0.80	250.70	156.48	0.38	94.22	112.20	38.50	2.91	150.70	100.00	0.66
0.80	1.00	274.20	167.70	0.39	106.50	121.20	43.00	2.82	164.20	110.00	0.67
1.00	0.00	155.60	129.85	0.17	25.75	119.80	25.80	4.64	145.60	10.00	0.07
1.00	0.20	190.90	143.81	0.25	47.09	130.00	30.90	4.21	160.90	30.00	0.19
1.00	0.40	218.80	163.61	0.25	55.19	128.60	30.20	4.26	158.80	60.00	0.38
1.00	0.60	230.30	174.42	0.24	55.88	129.60	30.70	4.22	160.30	70.00	0.44
1.00	0.80	309.40	187.68	0.39	121.72	169.00	50.40	3.35	219.40	90.00	0.41
1.00	1.00	314.40	200.16	0.36	114.24	159.00	45.40	3.50	204.40	110.00	0.54

Table A.2: Effects of *slack* and *violation* rates on results under *normal soft constraint penalty condition* in *SVe*150 dataset instances

S	V	C	B	%	AG	O	U	$\frac{O}{U}$	SMP	SCP	$\frac{SCP}{SMP}$
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	NA	0.00	0.00	NA
0.00	0.20	52.00	44.51	0.14	7.49	8.00	4.00	2.00	12.00	40.00	3.33
0.00	0.40	105.00	73.50	0.30	31.50	30.00	15.00	2.00	45.00	60.00	1.33
0.00	0.60	134.50	97.77	0.27	36.73	23.00	11.50	2.00	34.50	100.00	2.90
0.00	0.80	239.00	129.32	0.46	109.68	66.00	33.00	2.00	99.00	140.00	1.41
0.00	1.00	338.00	148.06	0.56	189.94	132.00	66.00	2.00	198.00	140.00	0.71
0.20	0.00	28.10	23.71	0.16	4.39	20.00	8.10	2.47	28.10	0.00	0.00
0.20	0.20	78.60	64.82	0.18	13.78	27.00	11.60	2.33	38.60	40.00	1.04
0.20	0.40	100.10	99.65	0.00	0.45	28.00	12.10	2.31	40.10	60.00	1.50
0.20	0.60	159.90	117.45	0.27	42.45	41.20	18.70	2.20	59.90	100.00	1.67
0.20	0.80	238.90	145.76	0.39	93.14	67.20	31.70	2.12	98.90	140.00	1.42
0.20	1.00	255.60	168.36	0.34	87.24	65.00	30.60	2.12	95.60	160.00	1.67
0.40	0.00	89.30	67.77	0.24	21.53	61.60	27.70	2.22	89.30	0.00	0.00
0.40	0.20	137.40	107.24	0.22	30.16	67.00	30.40	2.20	97.40	40.00	0.41
0.40	0.40	155.00	137.47	0.11	17.53	65.40	29.60	2.21	95.00	60.00	0.63
0.40	0.60	216.30	153.24	0.29	63.06	79.60	36.70	2.17	116.30	100.00	0.86
0.40	0.80	294.50	183.57	0.38	110.93	118.40	56.10	2.11	174.50	120.00	0.69
0.40	1.00	333.40	203.38	0.39	130.02	131.00	62.40	2.10	193.40	140.00	0.72
0.60	0.00	118.20	105.13	0.11	13.07	85.40	12.80	6.67	98.20	20.00	0.20
0.60	0.20	164.80	143.12	0.13	21.68	89.80	15.00	5.99	104.80	60.00	0.57
0.60	0.40	199.20	171.05	0.14	28.15	99.40	19.80	5.02	119.20	80.00	0.67
0.60	0.60	231.40	191.61	0.17	39.79	94.20	17.20	5.48	111.40	120.00	1.08
0.60	0.80	310.70	224.17	0.28	86.53	120.40	30.30	3.97	150.70	160.00	1.06
0.60	1.00	404.30	246.30	0.39	158.00	182.80	61.50	2.97	244.30	160.00	0.65
0.80	0.00	126.00	102.98	0.18	23.02	82.40	23.60	3.49	106.00	20.00	0.19
0.80	0.20	172.60	142.25	0.18	30.35	86.80	25.80	3.36	112.60	60.00	0.53
0.80	0.40	203.40	167.93	0.17	35.47	94.00	29.40	3.20	123.40	80.00	0.65
0.80	0.60	238.00	189.29	0.20	48.71	90.40	27.60	3.28	118.00	120.00	1.02
0.80	0.80	323.30	219.69	0.32	103.61	120.60	42.70	2.82	163.30	160.00	0.98
0.80	1.00	372.20	239.63	0.36	132.57	153.20	59.00	2.60	212.20	160.00	0.75
1.00	0.00	167.70	127.30	0.24	40.40	121.20	26.50	4.57	147.70	20.00	0.14
1.00	0.20	206.60	170.99	0.17	35.61	133.80	32.80	4.08	166.60	40.00	0.24
1.00	0.40	248.10	201.06	0.19	47.04	134.80	33.30	4.05	168.10	80.00	0.48
1.00	0.60	282.80	219.86	0.22	62.94	144.60	38.20	3.79	182.80	100.00	0.55
1.00	0.80	368.50	243.59	0.34	124.91	188.40	60.10	3.13	248.50	120.00	0.48
1.00	1.00	376.40	264.24	0.30	112.16	167.00	49.40	3.38	216.40	160.00	0.74

Table A.3: Effects of *slack* and *violation* rates on results under *double soft constraint penalty condition* in *SVe*150 dataset instances

P	N	C	B	%	AG	O	U	$\frac{O}{U}$	SMP	SCP	$\frac{SCP}{SMP}$
0.00	0.00	62.50	25.09	0.60	37.41	25.00	12.50	2.00	37.50	25.00	0.67
0.00	0.05	123.10	64.33	0.48	58.77	76.20	16.90	4.51	93.10	30.00	0.32
0.00	0.10	131.40	105.10	0.20	26.30	95.60	5.80	16.48	101.40	30.00	0.30
0.00	0.15	166.80	150.40	0.10	16.40	136.60	5.20	26.27	141.80	25.00	0.18
0.00	0.20	197.90	189.90	0.04	8.00	171.60	1.30	132.00	172.90	25.00	0.14
0.00	0.25	236.70	230.20	0.03	6.50	211.20	0.50	422.40	211.70	25.00	0.12
0.05	0.00	52.00	32.30	0.38	19.70	9.80	22.20	0.44	32.00	20.00	0.63
0.05	0.05	101.20	61.85	0.39	39.35	53.40	22.80	2.34	76.20	25.00	0.33
0.05	0.10	120.80	88.60	0.27	32.20	77.00	13.80	5.58	90.80	30.00	0.33
0.05	0.15	150.80	124.27	0.18	26.53	114.40	11.40	10.04	125.80	25.00	0.20
0.05	0.20	181.30	164.04	0.10	17.26	149.00	7.30	20.41	156.30	25.00	0.16
0.05	0.25	229.30	201.64	0.12	27.66	191.40	7.90	24.23	199.30	30.00	0.15
0.10	0.00	72.70	49.40	0.32	23.30	12.20	40.50	0.30	52.70	20.00	0.38
0.10	0.05	90.70	62.78	0.31	27.92	35.00	30.70	1.14	65.70	25.00	0.38
0.10	0.10	126.60	80.82	0.36	45.78	72.80	28.80	2.53	101.60	25.00	0.25
0.10	0.15	162.20	108.14	0.33	54.06	110.60	26.60	4.16	137.20	25.00	0.18
0.10	0.20	192.30	139.99	0.27	52.31	141.60	20.70	6.84	162.30	30.00	0.18
0.10	0.25	215.30	175.53	0.18	39.77	174.00	16.30	10.67	190.30	25.00	0.13
0.15	0.00	109.30	66.50	0.39	42.80	15.20	59.10	0.26	74.30	35.00	0.47
0.15	0.05	122.00	70.70	0.42	51.30	47.80	54.20	0.88	102.00	20.00	0.20
0.15	0.10	106.30	78.45	0.26	27.85	51.20	35.10	1.46	86.30	20.00	0.23
0.15	0.15	148.00	97.71	0.34	50.29	86.40	31.60	2.73	118.00	30.00	0.25
0.15	0.20	167.80	120.82	0.28	46.98	107.20	20.60	5.20	127.80	40.00	0.31
0.15	0.25	197.60	150.23	0.24	47.37	150.80	21.80	6.92	172.60	25.00	0.14
0.20	0.00	95.20	83.60	0.12	11.60	4.40	70.80	0.06	75.20	20.00	0.27
0.20	0.05	115.10	84.39	0.27	30.71	31.80	63.30	0.50	95.10	20.00	0.21
0.20	0.10	126.20	87.61	0.31	38.59	46.40	49.80	0.93	96.20	30.00	0.31
0.20	0.15	151.00	95.08	0.37	55.92	67.00	39.00	1.72	106.00	45.00	0.42
0.20	0.20	185.60	111.65	0.40	73.95	111.00	39.60	2.80	150.60	35.00	0.23
0.20	0.25	185.80	132.82	0.29	52.98	118.20	22.60	5.23	140.80	45.00	0.32
0.25	0.00	126.90	101.00	0.20	25.90	10.60	91.30	0.12	101.90	25.00	0.25
0.25	0.05	137.80	97.85	0.29	39.95	32.00	80.80	0.40	112.80	25.00	0.22
0.25	0.10	155.00	98.68	0.36	56.32	54.00	71.00	0.76	125.00	30.00	0.24
0.25	0.15	163.60	102.33	0.37	61.27	63.80	54.80	1.16	118.60	45.00	0.38
0.25	0.20	157.90	109.59	0.31	48.31	77.60	40.30	1.93	117.90	40.00	0.34
0.25	0.25	238.40	126.10	0.47	112.30	145.00	53.40	2.72	198.40	40.00	0.20

Table A.4: Effects of *negative* and *positive slack amounts* on results under *half soft constraint penalty condition* in *PNe*150 dataset.

P	N	C	B	%	AG	O	U	$\frac{O}{U}$	SMP	SCP	$\frac{SCP}{SMP}$
0.00	0.00	77.50	47.41	0.39	30.09	25.00	12.50	2.00	37.50	40.00	1.07
0.00	0.05	119.10	84.61	0.29	34.49	60.20	8.90	6.76	69.10	50.00	0.72
0.00	0.10	189.90	122.10	0.36	67.80	134.60	25.30	5.32	159.90	30.00	0.19
0.00	0.15	229.30	162.01	0.29	67.29	161.60	17.70	9.13	179.30	50.00	0.28
0.00	0.20	230.70	209.90	0.09	20.80	176.80	3.90	45.33	180.70	50.00	0.28
0.00	0.25	264.10	250.20	0.05	13.90	212.80	1.30	163.69	214.10	50.00	0.23
0.05	0.00	81.90	48.15	0.41	33.75	16.40	25.50	0.64	41.90	40.00	0.95
0.05	0.05	107.20	77.41	0.28	29.79	47.40	19.80	2.39	67.20	40.00	0.60
0.05	0.10	144.30	107.65	0.25	36.65	86.00	18.30	4.70	104.30	40.00	0.38
0.05	0.15	199.60	144.57	0.28	55.03	123.60	16.00	7.73	139.60	60.00	0.43
0.05	0.20	223.10	176.84	0.21	46.26	160.20	12.90	12.42	173.10	50.00	0.29
0.05	0.25	259.20	223.08	0.14	36.12	198.00	11.20	17.68	209.20	50.00	0.24
0.10	0.00	81.30	64.40	0.21	16.90	4.60	36.70	0.13	41.30	40.00	0.97
0.10	0.05	115.00	77.92	0.32	37.08	41.20	33.80	1.22	75.00	40.00	0.53
0.10	0.10	143.70	97.18	0.32	46.52	74.20	29.50	2.52	103.70	40.00	0.39
0.10	0.15	171.60	126.47	0.26	45.13	100.20	21.40	4.68	121.60	50.00	0.41
0.10	0.20	222.50	161.06	0.28	61.44	148.40	24.10	6.16	172.50	50.00	0.29
0.10	0.25	242.10	197.98	0.18	44.12	175.20	16.90	10.37	192.10	50.00	0.26
0.15	0.00	107.40	81.50	0.24	25.90	10.60	56.80	0.19	67.40	40.00	0.59
0.15	0.05	115.60	88.24	0.24	27.36	30.20	45.40	0.67	75.60	40.00	0.53
0.15	0.10	157.50	94.62	0.40	62.88	72.00	45.50	1.58	117.50	40.00	0.34
0.15	0.15	208.10	114.67	0.45	93.43	119.80	48.30	2.48	168.10	40.00	0.24
0.15	0.20	204.50	143.00	0.30	61.50	125.00	29.50	4.24	154.50	50.00	0.32
0.15	0.25	240.60	169.78	0.29	70.82	162.80	27.80	5.86	190.60	50.00	0.26
0.20	0.00	137.20	98.60	0.28	38.60	12.40	74.80	0.17	87.20	50.00	0.57
0.20	0.05	125.50	101.65	0.19	23.85	25.40	60.10	0.42	85.50	40.00	0.47
0.20	0.10	141.30	105.54	0.25	35.76	49.80	51.50	0.97	101.30	40.00	0.39
0.20	0.15	191.60	116.17	0.39	75.43	77.40	44.20	1.75	121.60	70.00	0.58
0.20	0.20	206.10	131.90	0.36	74.20	108.00	38.10	2.83	146.10	60.00	0.41
0.20	0.25	210.30	157.30	0.25	53.00	131.20	29.10	4.51	160.30	50.00	0.31
0.25	0.00	151.80	116.00	0.24	35.80	17.20	94.60	0.18	111.80	40.00	0.36
0.25	0.05	130.30	115.98	0.11	14.32	17.00	73.30	0.23	90.30	40.00	0.44
0.25	0.10	151.30	114.97	0.24	36.33	38.20	63.10	0.61	101.30	50.00	0.49
0.25	0.15	180.80	124.20	0.31	56.60	78.60	62.20	1.26	140.80	40.00	0.28
0.25	0.20	207.20	132.70	0.36	74.50	103.80	53.40	1.94	157.20	50.00	0.32
0.25	0.25	215.20	146.37	0.32	68.83	116.20	39.00	2.98	155.20	60.00	0.39

Table A.5: Effects of *negative* and *positive slack amounts* on results under *normal soft constraint penalty condition* in *PNe150* dataset

P	N	C	B	%	AG	O	U	$\frac{O}{U}$	SMP	SCP	$\frac{SCP}{SMP}$
0.00	0.00	137.00	77.82	0.43	59.18	38.00	19.00	2.00	57.00	80.00	1.40
0.00	0.05	168.30	115.76	0.31	52.54	73.00	15.30	4.77	88.30	80.00	0.91
0.00	0.10	194.00	155.73	0.20	38.27	104.00	10.00	10.40	114.00	80.00	0.70
0.00	0.15	242.80	210.40	0.13	32.40	150.60	12.20	12.34	162.80	80.00	0.49
0.00	0.20	296.20	249.90	0.16	46.30	213.80	22.40	9.54	236.20	60.00	0.25
0.00	0.25	318.30	290.20	0.09	28.10	215.60	2.70	79.85	218.30	100.00	0.46
0.05	0.00	106.30	80.29	0.24	26.01	6.00	20.30	0.30	26.30	80.00	3.04
0.05	0.05	163.20	109.92	0.33	53.28	71.40	31.80	2.25	103.20	60.00	0.58
0.05	0.10	184.00	137.27	0.25	46.73	85.80	18.20	4.71	104.00	80.00	0.77
0.05	0.15	223.70	187.67	0.16	36.03	113.00	10.70	10.56	123.70	100.00	0.81
0.05	0.20	262.00	226.08	0.14	35.92	152.80	9.20	16.61	162.00	100.00	0.62
0.05	0.25	289.50	265.16	0.08	24.34	198.20	11.30	17.54	209.50	80.00	0.38
0.10	0.00	141.70	94.40	0.33	47.30	18.20	43.50	0.42	61.70	80.00	1.30
0.10	0.05	152.00	109.88	0.28	42.12	39.20	32.80	1.20	72.00	80.00	1.11
0.10	0.10	162.40	129.46	0.20	32.94	60.00	22.40	2.68	82.40	80.00	0.97
0.10	0.15	229.80	159.43	0.31	70.37	119.00	30.80	3.86	149.80	80.00	0.53
0.10	0.20	248.40	190.78	0.23	57.62	159.00	29.40	5.41	188.40	60.00	0.32
0.10	0.25	282.50	239.76	0.15	42.74	168.80	13.70	12.32	182.50	100.00	0.55
0.15	0.00	132.10	111.50	0.16	20.60	0.40	51.70	0.01	52.10	80.00	1.54
0.15	0.05	158.30	118.64	0.25	39.66	32.00	46.30	0.69	78.30	80.00	1.02
0.15	0.10	164.50	127.22	0.23	37.28	50.00	34.50	1.45	84.50	80.00	0.95
0.15	0.15	209.90	151.95	0.28	57.95	81.00	28.90	2.80	109.90	100.00	0.91
0.15	0.20	249.20	185.57	0.26	63.63	134.80	34.40	3.92	169.20	80.00	0.47
0.15	0.25	268.80	209.98	0.22	58.82	161.60	27.20	5.94	188.80	80.00	0.42
0.20	0.00	150.70	128.60	0.15	22.10	1.40	69.30	0.02	70.70	80.00	1.13
0.20	0.05	158.90	132.84	0.16	26.06	21.00	57.90	0.36	78.90	80.00	1.01
0.20	0.10	198.10	136.00	0.31	62.10	61.00	57.10	1.07	118.10	80.00	0.68
0.20	0.15	221.40	150.71	0.32	70.69	90.60	50.80	1.78	141.40	80.00	0.57
0.20	0.20	243.30	167.39	0.31	75.91	132.80	50.50	2.63	183.30	60.00	0.33
0.20	0.25	256.60	190.03	0.26	66.57	155.40	41.20	3.77	196.60	60.00	0.31
0.25	0.00	214.80	146.00	0.32	68.80	19.20	95.60	0.20	114.80	100.00	0.87
0.25	0.05	171.20	148.33	0.13	22.87	17.60	73.60	0.24	91.20	80.00	0.88
0.25	0.10	182.20	147.53	0.19	34.67	38.80	63.40	0.61	102.20	80.00	0.78
0.25	0.15	234.60	157.87	0.33	76.73	87.80	66.80	1.31	154.60	80.00	0.52
0.25	0.20	216.80	169.99	0.22	46.81	90.20	46.60	1.94	136.80	80.00	0.58
0.25	0.25	246.30	183.30	0.26	63.00	123.60	42.70	2.89	166.30	80.00	0.48

Table A.6: Effects of *negative* and *positive slack amounts* on results under *double soft constraint penalty condition* in *PNe150* dataset

A.2 Tables Associated with the Comparison of Models with and without Floor Variables

In Tables A.7, A.8, A.9, and A.10, the results obtained from $SVe150$ and $PNe150$ datasets using a 30 minute CPLEX run by using models without and with floor variables are represented respectively. Columns S, V, P, and N represent the *slack space rate*, *soft constraint violation rate*, *positive*, and *negative slack space rate* respectively. Columns C, B, % represent the best cost obtained, the best bound, and the percentage gap between the best solution obtained, and the best bound respectively. Columns SMP, O, U, $and SCP$ represent the total space misuse, overuse, underuse, and soft constraint violation penalties. The respective analysis related to these tables can be found in Section 4.6.6.

S	V	C	B	%	SMP	O	U	SCP
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.20	61.50	21.47	0.65	31.50	21.00	10.50	30.00
0.00	0.40	59.50	43.08	0.28	19.50	13.00	6.50	40.00
0.00	0.60	115.50	55.76	0.52	55.50	37.00	18.50	60.00
0.00	0.80	161.50	76.58	0.53	91.50	61.00	30.50	70.00
0.00	1.00	197.00	88.31	0.55	117.00	78.00	39.00	80.00
0.20	0.00	36.80	22.76	0.38	36.80	25.80	11.00	0.00
0.20	0.20	53.80	45.15	0.16	33.80	23.80	10.00	20.00
0.20	0.40	86.30	66.21	0.23	56.30	38.80	17.50	30.00
0.20	0.60	114.80	76.42	0.33	54.80	37.80	17.00	60.00
0.20	0.80	193.60	94.30	0.51	113.60	77.00	36.60	80.00
0.20	1.00	250.90	103.45	0.59	170.90	115.20	55.70	80.00
0.40	0.00	85.80	63.10	0.26	75.80	52.60	23.20	10.00
0.40	0.20	109.40	83.67	0.24	79.40	55.00	24.40	30.00
0.40	0.40	139.70	101.42	0.27	109.70	75.20	34.50	30.00
0.40	0.60	159.40	110.11	0.31	109.40	75.00	34.40	50.00
0.40	0.80	200.10	126.72	0.37	130.10	88.80	41.30	70.00
0.40	1.00	296.70	137.80	0.54	226.70	153.20	73.50	70.00
0.60	0.00	128.90	92.97	0.28	118.90	99.20	19.70	10.00
0.60	0.20	133.30	112.59	0.16	103.30	88.80	14.50	30.00
0.60	0.40	168.50	131.12	0.22	128.50	105.60	22.90	40.00
0.60	0.60	180.40	140.55	0.22	120.40	100.20	20.20	60.00
0.60	0.80	246.70	161.17	0.35	156.70	124.40	32.30	90.00
0.60	1.00	293.90	171.97	0.41	193.90	149.20	44.70	100.00
0.80	0.00	122.40	91.07	0.26	102.40	80.00	22.40	20.00
0.80	0.20	148.00	111.18	0.25	118.00	90.40	27.60	30.00
0.80	0.40	183.00	127.75	0.30	133.00	100.40	32.60	50.00
0.80	0.60	193.60	139.86	0.28	133.60	100.80	32.80	60.00
0.80	0.80	250.70	156.48	0.38	150.70	112.20	38.50	100.00
0.80	1.00	274.20	167.70	0.39	164.20	121.20	43.00	110.00
1.00	0.00	155.60	129.85	0.17	145.60	119.80	25.80	10.00
1.00	0.20	190.90	143.81	0.25	160.90	130.00	30.90	30.00
1.00	0.40	218.80	163.61	0.25	158.80	128.60	30.20	60.00
1.00	0.60	230.30	174.42	0.24	160.30	129.60	30.70	70.00
1.00	0.80	309.40	187.68	0.39	219.40	169.00	50.40	90.00
1.00	1.00	314.40	200.16	0.36	204.40	159.00	45.40	110.00

Table A.7: Results obtained in *SVe*150 dataset under *normal soft constraint penalty* condition using model without floor variables (IP_1)

S	V	C	B	%	SMP	O	U	SCP
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.20	32.00	24.29	0.24	12.00	8.00	4.00	20.00
0.00	0.40	50.50	44.99	0.11	10.50	7.00	3.50	40.00
0.00	0.60	86.00	64.36	0.25	36.00	24.00	12.00	50.00
0.00	0.80	120.50	94.74	0.21	40.50	27.00	13.50	80.00
0.00	1.00	153.00	103.25	0.33	63.00	42.00	21.00	90.00
0.20	0.00	36.80	22.93	0.38	36.80	25.80	11.00	0.00
0.20	0.20	52.30	46.46	0.11	32.30	22.80	9.50	20.00
0.20	0.40	72.90	68.46	0.06	32.90	23.20	9.70	40.00
0.20	0.60	92.90	87.18	0.06	32.90	23.20	9.70	60.00
0.20	0.80	159.60	110.68	0.31	89.60	61.00	28.60	70.00
0.20	1.00	183.00	122.71	0.33	83.00	56.60	26.40	100.00
0.40	0.00	79.50	68.58	0.14	69.50	48.40	21.10	10.00
0.40	0.20	101.60	88.66	0.13	71.60	49.80	21.80	30.00
0.40	0.40	139.90	102.57	0.27	89.90	62.00	27.90	50.00
0.40	0.60	157.60	118.05	0.25	107.60	73.80	33.80	50.00
0.40	0.80	186.20	146.52	0.21	96.20	66.20	30.00	90.00
0.40	1.00	215.80	157.96	0.27	135.80	92.60	43.20	80.00
0.60	0.00	107.00	93.80	0.12	97.00	84.60	12.40	10.00
0.60	0.20	130.30	113.41	0.13	100.30	86.80	13.50	30.00
0.60	0.40	149.00	132.20	0.11	109.00	92.60	16.40	40.00
0.60	0.60	173.50	148.81	0.14	113.50	95.60	17.90	60.00
0.60	0.80	211.00	177.47	0.16	121.00	100.60	20.40	90.00
0.60	1.00	221.50	184.52	0.17	131.50	107.60	23.90	90.00
0.80	0.00	113.60	91.42	0.20	103.60	80.80	22.80	10.00
0.80	0.20	144.10	112.26	0.22	114.10	87.80	26.30	30.00
0.80	0.40	166.80	129.43	0.22	116.80	89.60	27.20	50.00
0.80	0.60	185.50	146.24	0.21	125.50	95.40	30.10	60.00
0.80	0.80	222.90	175.14	0.21	142.90	107.00	35.90	80.00
0.80	1.00	240.10	189.78	0.21	150.10	111.80	38.30	90.00
1.00	0.00	161.00	123.43	0.23	151.00	123.40	27.60	10.00
1.00	0.20	171.40	153.41	0.11	141.40	117.00	24.40	30.00
1.00	0.40	206.70	169.79	0.18	156.70	127.20	29.50	50.00
1.00	0.60	208.70	186.50	0.11	138.70	115.20	23.50	70.00
1.00	0.80	253.30	199.03	0.21	163.30	131.60	31.70	90.00
1.00	1.00	270.20	221.71	0.18	170.20	136.20	34.00	100.00

Table A.8: Results obtained in *SVe*150 dataset under *normal soft constraint penalty* condition using model with floor variables (IP_2)

P	N	C	B	%	SMP	O	U	SCP
0.00	0.00	77.50	47.41	0.39	37.50	25.00	12.50	40.00
0.00	0.05	119.10	84.61	0.29	69.10	60.20	8.90	50.00
0.00	0.10	189.90	122.10	0.36	159.90	134.60	25.30	30.00
0.00	0.15	229.30	162.01	0.29	179.30	161.60	17.70	50.00
0.00	0.20	230.70	209.90	0.09	180.70	176.80	3.90	50.00
0.00	0.25	264.10	250.20	0.05	214.10	212.80	1.30	50.00
0.05	0.00	81.90	48.15	0.41	41.90	16.40	25.50	40.00
0.05	0.05	107.20	77.41	0.28	67.20	47.40	19.80	40.00
0.05	0.10	144.30	107.65	0.25	104.30	86.00	18.30	40.00
0.05	0.15	199.60	144.57	0.28	139.60	123.60	16.00	60.00
0.05	0.20	223.10	176.84	0.21	173.10	160.20	12.90	50.00
0.05	0.25	259.20	223.08	0.14	209.20	198.00	11.20	50.00
0.10	0.00	81.30	64.40	0.21	41.30	4.60	36.70	40.00
0.10	0.05	115.00	77.92	0.32	75.00	41.20	33.80	40.00
0.10	0.10	143.70	97.18	0.32	103.70	74.20	29.50	40.00
0.10	0.15	171.60	126.47	0.26	121.60	100.20	21.40	50.00
0.10	0.20	222.50	161.06	0.28	172.50	148.40	24.10	50.00
0.10	0.25	242.10	197.98	0.18	192.10	175.20	16.90	50.00
0.15	0.00	107.40	81.50	0.24	67.40	10.60	56.80	40.00
0.15	0.05	115.60	88.24	0.24	75.60	30.20	45.40	40.00
0.15	0.10	157.50	94.62	0.40	117.50	72.00	45.50	40.00
0.15	0.15	208.10	114.67	0.45	168.10	119.80	48.30	40.00
0.15	0.20	204.50	143.00	0.30	154.50	125.00	29.50	50.00
0.15	0.25	240.60	169.78	0.29	190.60	162.80	27.80	50.00
0.20	0.00	137.20	98.60	0.28	87.20	12.40	74.80	50.00
0.20	0.05	125.50	101.65	0.19	85.50	25.40	60.10	40.00
0.20	0.10	141.30	105.54	0.25	101.30	49.80	51.50	40.00
0.20	0.15	191.60	116.17	0.39	121.60	77.40	44.20	70.00
0.20	0.20	206.10	131.90	0.36	146.10	108.00	38.10	60.00
0.20	0.25	210.30	157.30	0.25	160.30	131.20	29.10	50.00
0.25	0.00	151.80	116.00	0.24	111.80	17.20	94.60	40.00
0.25	0.05	130.30	115.98	0.11	90.30	17.00	73.30	40.00
0.25	0.10	151.30	114.97	0.24	101.30	38.20	63.10	50.00
0.25	0.15	180.80	124.20	0.31	140.80	78.60	62.20	40.00
0.25	0.20	207.20	132.70	0.36	157.20	103.80	53.40	50.00
0.25	0.25	215.20	146.37	0.32	155.20	116.20	39.00	60.00

Table A.9: Results obtained in *PNe*150 dataset under *normal soft constraint penalty* condition using models without floor variables (IP_1)

P	N	C	B	%	SMP	O	U	SCP
0.00	0.00	71.00	54.37	0.23	21.00	14.00	7.00	50.00
0.00	0.05	103.40	93.60	0.09	63.40	56.40	7.00	40.00
0.00	0.10	134.50	132.10	0.02	94.50	91.00	3.50	40.00
0.00	0.15	196.30	171.12	0.13	146.30	139.60	6.70	50.00
0.00	0.20	222.00	209.90	0.05	172.00	171.00	1.00	50.00
0.00	0.25	260.80	250.20	0.04	210.80	210.60	0.20	50.00
0.05	0.00	65.40	57.30	0.12	25.40	5.40	20.00	40.00
0.05	0.05	99.10	87.27	0.12	59.10	42.00	17.10	40.00
0.05	0.10	135.90	116.68	0.14	95.90	80.40	15.50	40.00
0.05	0.15	170.10	156.11	0.08	120.10	110.60	9.50	50.00
0.05	0.20	210.60	192.67	0.09	150.60	145.20	5.40	60.00
0.05	0.25	240.90	228.71	0.05	190.90	185.80	5.10	50.00
0.10	0.00	74.40	74.40	0.00	34.40	0.00	34.40	40.00
0.10	0.05	97.90	88.29	0.10	57.90	29.80	28.10	40.00
0.10	0.10	117.90	106.82	0.09	77.90	57.00	20.90	40.00
0.10	0.15	156.60	140.03	0.11	106.60	90.20	16.40	50.00
0.10	0.20	179.90	174.33	0.03	129.90	120.00	9.90	50.00
0.10	0.25	218.70	211.32	0.03	168.70	159.60	9.10	50.00
0.15	0.00	91.50	91.50	0.00	51.50	0.00	51.50	40.00
0.15	0.05	108.10	96.64	0.11	68.10	25.20	42.90	40.00
0.15	0.10	121.80	103.84	0.15	81.80	48.20	33.60	40.00
0.15	0.15	161.10	131.09	0.19	111.10	81.80	29.30	50.00
0.15	0.20	174.50	159.16	0.09	124.50	105.00	19.50	50.00
0.15	0.25	201.60	190.55	0.05	151.60	136.80	14.80	50.00
0.20	0.00	108.60	108.60	0.00	68.60	0.00	68.60	40.00
0.20	0.05	112.90	110.46	0.02	72.90	17.00	55.90	40.00
0.20	0.10	122.70	114.63	0.07	82.70	37.40	45.30	40.00
0.20	0.15	145.20	136.80	0.06	95.20	59.80	35.40	50.00
0.20	0.20	166.40	151.18	0.09	116.40	88.20	28.20	50.00
0.20	0.25	199.80	174.34	0.13	149.80	124.20	25.60	50.00
0.25	0.00	126.00	125.59	0.00	86.00	0.00	86.00	40.00
0.25	0.05	129.70	124.73	0.04	89.70	16.60	73.10	40.00
0.25	0.10	131.10	126.58	0.03	91.10	31.40	59.70	40.00
0.25	0.15	151.20	141.47	0.06	101.20	52.20	49.00	50.00
0.25	0.20	162.50	154.66	0.05	112.50	74.00	38.50	50.00
0.25	0.25	187.50	169.46	0.10	137.50	104.40	33.10	50.00

Table A.10: Results obtained in $PNe150$ dataset under *normal soft constraint penalty* condition using models with floor variables (IP_2)

Pseudo-codes Associated with Chapter 6

In this section, the pseudo-codes associated with the update algorithms for each constraint (except *allocation* and *non-allocation*) as described in Chapter 6 are presented. The pseudo-codes were moved to the appendix to remove the clutter in the corresponding chapter. The respective explanations for these pseudo-codes can be found in Section 6.4.3.

The update algorithm for the space misuse calculations is given in Figure B.1. Figures B.2 and B.3 depict the pseudo-code for the update algorithm for *same room* and *not same room* constraints respectively. The pseudo-code for the update algorithm for *not sharing* constraint is divided into two (Figures B.4 and B.5). The update algorithm for three proximity constraints (*adjacency*, *nearby*, and *away from*) are given in Figures B.6, B.7 and B.8 respectively. The update algorithm for the *capacity* constraint is divided into three figures for clarity in Figure B.11, B.9, and B.10 respectively.

Input: cost change matrix Δ, space cost change matrix Δ^{sp}, Entity set E, Room set R

Output: cost change matrix Δ, space cost change matrix Δ^{sp}

1: **for** $i \leftarrow 1$ to $|E|$ except e **do**

2:     $room \leftarrow e_i.room$

3:     **if** $room = r_1$ **then**

4:         $prevSM1^{+i} \leftarrow$ Calculate previous space misuse in r_1 with entity e_i

5:         $prevSM1^{-i} \leftarrow$ Calculate previous space misuse in r_1 without entity e_i

6:         $currentSM1^{+i} \leftarrow$ Calculate current space misuse in r_1 with entity e_i

7:         $currentSM1^{-i} \leftarrow$ Calculate current space misuse in r_1 without entity e_i

8:         $\delta^{sm1} \leftarrow (currentSM1^{-i} - currentSM1^{+i}) - (prevSM1^{-i} - prevSM1^{+i})$

9:         **for** $j \leftarrow 1$ to $|R|$ **do**

10:             **if** $j = r_2$ **then**

11:                 $prevSM2^{+i} \leftarrow$ Calculate previous space misuse in r_2 with entity e_i

12:                 $prevSM2^{-i} \leftarrow$ Calculate previous space misuse in r_2 without entity e_i

13:                 $currentSM2^{+i} \leftarrow$ Calculate current space misuse in r_3 with entity e_i

14:                 $currentSM2^{-i} \leftarrow$ Calculate current space misuse in r_2 without entity e_i

15:                 $\delta^{sm2} \leftarrow (currentSM2^{-i} - currentSM2^{+i}) - (prevSM2^{-i} - prevSM2^{+i})$

16:                 $\Delta_{ij} \leftarrow \Delta_{ij} + \delta^{sm1} + \delta^{sm2}$

17:                 $\Delta_{ij}^{sp} \leftarrow \Delta_{ij}^{sp} + \delta^{sm1} + \delta^{sm2}$

18:             **else if** $j \neq r_1$ **then**

19:                 $\Delta_{ij} \leftarrow \Delta_{ij} + \delta^{sm1}$

20:                 $\Delta_{ij}^{sp} \leftarrow \Delta_{ij}^{sp} + \delta^{sm1}$

21:             **else**

22:                 $\Delta_{ir_1} \leftarrow 0$

23:                 $\Delta_{ir_1}^{sp} \leftarrow 0$

24:     **else if** $room = r_2$ **then**

25:         Symmetric condition for r_2 similar to case with r_1

26:         Adjust and swap r_1 with r_2 (shorthand notation)

27:     **else**

28:         $currentMisuse_1 \leftarrow$ Calculate the current misuse in r_1

29:         $currentMisuse_2 \leftarrow$ Calculate the current misuse in r_2

30:         $currentMisuse_i \leftarrow$ Calculate the current misuse in room entity e_i is in

31:         $\delta^{sm1} \leftarrow currentMisuse_i + currentMisuse_1 - r_1.misuse$

32:         $\delta^{sm2} \leftarrow currentMisuse_i + currentMisuse_2 - r_2.misuse$

33:         $\Delta_{ir_1} \leftarrow \Delta_{ir_1} + \delta^{sm1} - \Delta_{ir_1}^{sp}$

34:         $\Delta_{ir_2} \leftarrow \Delta_{ir_2} + \delta^{sm2} - \Delta_{ir_2}^{sp}$

35:         $\Delta_{ir_1}^{sp} \leftarrow \delta^{sm1}$

36:         $\Delta_{ir_2}^{sp} \leftarrow \delta^{sm2}$

Figure B.1: Update algorithm for Space Misuse

Input: cost change matrix Δ, Entity set E, Room set R

Output: cost change matrix Δ

1: **if** $e_1.prevRoom = e_2.prevRoom$ AND $e_1.currentRoom = e_2.prevRoom$ **then**

2:      return

3: **if** $e_1.prevRoom = e_2.prevRoom$ AND $e_1.currentRoom \neq e_2.prevRoom$ **then**

4:      **for** $i \leftarrow 1$ to $|R|$ **do**

5:          $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} - penalty_{sr}$

6:      $\Delta_{e_2 e_1.currentRoom} \leftarrow \Delta_{e_2 e_1.currentRoom} - penalty_{sr}$

7:      $\Delta_{e_2 e_1.prevRoom} \leftarrow 0$

8:      return

9: **if** $e_1.prevRoom \neq e_2.prevRoom$ AND $e_1.currentRoom = e_2.prevRoom$ **then**

10:      **for** $i \leftarrow 1$ to $|R|$ **do**

11:          $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} + penalty_{sr}$

12:      $\Delta_{e_2 e_1.prevRoom} \leftarrow \Delta_{e_2 e_1.prevRoom} + penalty_{sr}$

13:      $\Delta_{e_2 e_1.currentRoom} \leftarrow 0$

14:      return

15: **if** $e_1.prevRoom \neq e_2.prevRoom$ AND $e_1.currentRoom \neq e_2.prevRoom$ **then**

16:      $\Delta_{e_2 e_1.prevRoom} \leftarrow \Delta_{e_2 e_1.prevRoom} + penalty_{sr}$

17:      $\Delta_{e_2 e_1.currentRoom} \leftarrow \Delta_{e_2 e_1.currentRoom} - penalty_{sr}$

18:      return

Figure B.2: Update algorithm for *same room* constraint

Input: cost change matrix Δ, Entity set E, Room set R

Output: cost change matrix Δ

1: **if** $e_1.prevRoom = e_2.prevRoom$ AND $e_1.currentRoom = e_2.prevRoom$ **then**

2:     return

3: **if** $e_1.prevRoom = e_2.prevRoom$ AND $e_1.currentRoom \neq e_2.prevRoom$ **then**

4:     **for** $i \leftarrow 1$ to $|R|$ **do**

5:         $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} + penalty_{nsr}$

6:     $\Delta_{e_2 e_2.currentRoom} \leftarrow \Delta_{e_2 e_2.currentRoom} + penalty_{nsr}$

7:     $\Delta_{e_2 e_1.prevRoom} \leftarrow 0$

8:     return

9: **if** $e_1.prevRoom \neq e_2.prevRoom$ AND $e_1.currentRoom = e_2.prevRoom$ **then**

10:     **for** $i \leftarrow 1$ to $|R|$ **do**

11:         $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} - penalty_{nsr}$

12:     $\Delta_{e_2 e_1.prevRoom} \leftarrow \Delta_{e_2 e_1.prevRoom} - penalty_{nsr}$

13:     $\Delta_{e_2 e_1.currentRoom} \leftarrow 0$

14:     return

15: **if** $e_1.prevRoom \neq e_2.prevRoom$ AND $e_1.currentRoom \neq e_2.prevRoom$ **then**

16:     $\Delta_{e_2 e_1.prevRoom} \leftarrow \Delta_{e_2 e_1.prevRoom} - penalty_{nsr}$

17:     $\Delta_{e_2 e_1.currentRoom} \leftarrow \Delta_{e_2 e_1.currentRoom} + penalty_{nsr}$

18:     return

Figure B.3: Update algorithm for *not same room* constraint

Input: cost change matrix Δ, Entity set E, Room set R

Output: cost change matrix Δ

1: **for** $i \leftarrow 1$ to $|E|$ except e **do**

2:      $\delta_{nsh} \leftarrow 0$

3:      **if** $r_1 = e_i.room$ **then**

4:          **if** number of entities in r_1 is 1 **then**

5:              **if** entities e_e and e_i both have no *not sharing* constraints **then**

6:                  $\delta_{nsh} \leftarrow 0$

7:              **else**

8:                  $\delta_{nsh} \leftarrow penalty_{nsh}$

9:                  **if** entities e_e and e_i have *not sharing* constraints **then**

10:                      $\delta_{nsh} \leftarrow penalty_{nsh}$

11:          **else if** number of entities in r_1 is 2 **then**

12:              **if** number of entities with *not sharing* is 0 **then**

13:                  $\delta_{nsh} \leftarrow 0$

14:              **else if** entity e_i does not have a *not sharing* constraint **then**

15:                  $\delta_{nsh} \leftarrow -penalty_{nsh}$

16:              **else if** r_1 contains just two entities each with *not sharing* constraint **then**

17:                  $\delta_{nsh} \leftarrow -penalty_{nsh}$

18:              **else**

19:                  $\delta_{nsh} \leftarrow 0$

20:              **for** $j \leftarrow 1$ to $|R|$ except r_1 **do**

21:                  $\Delta_{ij} \leftarrow \Delta_{ij} + \delta_{nsh}$

22:      **else**

23:          **if** r_1 has no entities **then**

24:              **if** entity e_i and e_e both do not have *not sharing* constraints **then**

25:                  $\delta_{nsh} \leftarrow 0$

26:              **else if** either entity e_e or e_i has *not sharing* constraint but not both **then**

27:                  $\delta_{nsh} \leftarrow -penalty_{nsh}$

28:              **else**

29:                  $\delta_{nsh} \leftarrow -2 x penalty_{nsh}$

30:          **else if** r_1 contains only one entity and that entity has *not sharing* constraint **then**

31:              $\delta_{nsh} \leftarrow penalty_{nsh}$

32:      $\Delta_{ir_1} \leftarrow \Delta_{ir_1} \delta_{nsh}$

Figure B.4: Part 1 for the update algorithm for *not sharing* constraint

Input: cost change matrix Δ, Entity set E, Room set R

Output: cost change matrix Δ

1: **for** $i \leftarrow 1$ to $|E|$ except e **do**
2:      $\delta_{nsh} \leftarrow 0$
3:      **if** $r_2 = e_i.room$ **then**
4:          **if** r_2 has two entities **then**
5:              **if** entities e_i and e_e both do not have *not sharing* constraint **then**
6:                  $\delta_{nsh} \leftarrow 0$
7:              **else**
8:                  $\delta_{nsh} \leftarrow -penalty_{nsh}$
9:                  **if** r_2 has two entities with *not sharing* constraint **then**
10:                      $\delta_{nsh} \leftarrow \delta_{nsh} - penalty_{nsh}$
11:          **else if** r_2 has three entities **then**
12:              **if** the third entity in r_2 has *not sharing* constraint **then**
13:                  $\delta_{nsh} \leftarrow \delta_{nsh} + penalty_{nsh}$
14:              **else**
15:                  $\delta_{nsh} \leftarrow 0$
16:          **for** $j \leftarrow 1$ to $|R|$ except r_2 **do**
17:              $\Delta_{ij} \leftarrow \Delta_{ij} + \delta_{nsh}$
18:      **else**
19:          **if** r_2 contains just one entity **then**
20:              **if** entity e_i does not have *not sharing* constraint **then**
21:                  **if** r_2 has an entity with *not sharing* constraint **then**
22:                      $\delta_{nsh} \leftarrow penalty_{nsh}$
23:              **else**
24:                  $\delta_{nsh} \leftarrow penalty_{nsh}$
25:                  **if** r_2 has an entity with *not sharing* constraint **then**
26:                      $\delta_{nsh} \leftarrow \delta_{nsh} + penalty_{nsh}$
27:          **else if** r_2 contains two entities **then**
28:              **if** if the other entity in r_2 besides e_e has not sharing constraint **then**
29:                  $\delta_{nsh} \leftarrow -penalty_{nsh}$
30:          $\Delta_{ir_2} \leftarrow \Delta_{ir_2} + \delta_{nsh}$

Figure B.5: Part 2 for the update algorithm for *not sharing* constraint

Input: cost change matrix Δ, Entity set E, Room set R

Output: cost change matrix Δ

1: **if** $IsAdj(e_1.prevRoom, e_2.prevRoom)$ AND $IsAdj(e_1.currentRoom, e_2.prevRoom)$ **then**

2:     **for** $i \leftarrow 1$ to $|A_{e_1.prevRoom}|$ **do**

3:         $\Delta_{e_2 A_{e_1.prevRoom}(i)} \leftarrow \Delta_{e_2 A_{e_1.prevRoom}(i)} + penalty_{ad}$

4:     **for** $i \leftarrow 1$ to $|A_{e_1.currentRoom}|$ **do**

5:         $\Delta_{e_2 A_{e_1.currentRoom}(i)} \leftarrow \Delta_{e_2 A_{e_1.currentRoom}(i)} - penalty_{ad}$

6:     $\Delta_{e_2 e_2.prevRoom} \leftarrow 0$

7:     return

8: **if** $IsAdj(e_1.prevRoom, e_2.prevRoom)$ AND $\neg IsAdj(e_1.currentRoom, e_2.prevRoom)$ **then**

9:     **for** $i \leftarrow 1$ to $|R|$ **do**

10:         $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} - penalty_{ad}$

11:         **if** $IsAdj(i, e_1.currentRoom) = true$ **then**

12:             $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} - penalty_{ad}$

13:         **if** $IsAdj(i, e_1.prevRoom) = true$ **then**

14:             $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} + penalty_{ad}$

15:     $\Delta_{e_2 e_2.prevRoom} \leftarrow 0$

16:     return

17: **if** $\neg IsAdj(e_1.prevRoom, e_2.prevRoom)$ AND $IsAdj(e_1.currentRoom, e_2.prevRoom)$ **then**

18:     **for** $i \leftarrow 1$ to $|R|$ **do**

19:         $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} + penalty_{ad}$

20:         **if** $IsAdj(i, e_1.currentRoom) = true$ **then**

21:             $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} - penalty_{ad}$

22:         **if** $IsAdj(i, e_1.prevRoom) = true$ **then**

23:             $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} + penalty_{ad}$

24:     $\Delta_{e_2 e_2.prevRoom} \leftarrow 0$

25:     return

26: **if** $\neg IsAdj(e_1.prevRoom, e_2.prevRoom)$ AND $\neg IsAdj(e_1.currentRoom, e_2.prevRoom)$ **then**

27:     **for** $i \leftarrow 1$ to $|A_{e_1.prevRoom}|$ **do**

28:         $\Delta_{e_2 A_{e_1.prevRoom}(i)} \leftarrow \Delta_{e_2 A_{e_1.prevRoom}(i)} + penalty_{ad}$

29:     **for** $i \leftarrow 1$ to $|A_{e_1.currentRoom}|$ **do**

30:         $\Delta_{e_2 A_{e_1.currentRoom}(i)} \leftarrow \Delta_{e_2 A_{e_1.currentRoom}(i)} - penalty_{ad}$

31:     $\Delta_{e_2 e_2.prevRoom} \leftarrow 0$

32:     return

Figure B.6: Update algorithm for *adjacency* constraint

Input: cost change matrix Δ, Entity set E, Room set R

Output: cost change matrix Δ

1: **if** $IsNear(e_1.prevRoom, e_2.prevRoom)$ AND $IsNear(e_1.currentRoom, e_2.prevRoom)$ **then**

2:     **for** $i \leftarrow 1$ to $|N_{e_1.prevRoom}|$ **do**

3:         $\Delta_{e_2 N_{e_1.prevRoom}(i)} \leftarrow \Delta_{e_2 N_{e_1.prevRoom}(i)} + penalty_{nr}$

4:     **for** $i \leftarrow 1$ to $|N_{e_1.currentRoom}|$ **do**

5:         $\Delta_{e_2 N_{e_1.currentRoom}(i)} \leftarrow \Delta_{e_2 N_{e_1.currentRoom}(i)} - penalty_{nr}$

6:     $\Delta_{e_2 e_2.prevRoom} \leftarrow 0$

7:     return

8: **if** $IsNear(e_1.prevRoom, e_2.prevRoom)$ AND $\neg IsNear(e_1.currentRoom, e_2.prevRoom)$ **then**

9:     **for** $i \leftarrow 1$ to $|R|$ **do**

10:         $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} - penalty_{nr}$

11:         **if** $IsNear(i, e_1.currentRoom) = true$ **then**

12:             $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} - penalty_{nr}$

13:         **if** $IsNear(i, e_1.prevRoom) = true$ **then**

14:             $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} + penalty_{nr}$

15:     $\Delta_{e_2 e_2.prevRoom} \leftarrow 0$

16:     return

17: **if** $\neg IsNear(e_1.prevRoom, e_2.prevRoom)$ AND $IsNear(e_1.currentRoom, e_2.prevRoom)$ **then**

18:     **for** $i \leftarrow 1$ to $|R|$ **do**

19:         $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} + penalty_{nr}$

20:         **if** $IsNear(i, e_1.currentRoom) = true$ **then**

21:             $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} - penalty_{nr}$

22:         **if** $IsNear(i, e_1.prevRoom) = true$ **then**

23:             $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} + penalty_{nr}$

24:     $\Delta_{e_2 e_2.prevRoom} \leftarrow 0$

25:     return

26: **if** $\neg IsNear(e_1.prevRoom, e_2.prevRoom)$ AND $\neg IsNear(e_1.currentRoom, e_2.prevRoom)$ **then**

27:     **for** $i \leftarrow 1$ to $|N_{e_1.prevRoom}|$ **do**

28:         $\Delta_{e_2 N_{e_1.prevRoom}(i)} \leftarrow \Delta_{e_2 N_{e_1.prevRoom}(i)} + penalty_{nr}$

29:     **for** $i \leftarrow 1$ to $|N_{e_1.currentRoom}|$ **do**

30:         $\Delta_{e_2 N_{e_1.currentRoom}(i)} \leftarrow \Delta_{e_2 N_{e_1.currentRoom}(i)} - penalty_{nr}$

31:     $\Delta_{e_2 e_2.prevRoom} \leftarrow 0$

32:     return

Figure B.7: Update algorithm for *nearby* constraint

Input: cost change matrix Δ, Entity set E, Room set R

Output: cost change matrix Δ

1: **if** $IsNear(e_1.prevRoom, e_2.prevRoom)$ AND $IsNear(e_1.currentRoom, e_2.prevRoom)$ **then**

2:     **for** $i \leftarrow 1$ to $|N_{e_1.prevRoom}|$ **do**

3:         $\Delta_{e_2 N_{e_1.prevRoom}(i)} \leftarrow \Delta_{e_2 N_{e_1.prevRoom}(i)} - penalty_{aw}$

4:     **for** $i \leftarrow 1$ to $|N_{e_1.currentRoom}|$ **do**

5:         $\Delta_{e_2 N_{e_1.currentRoom}(i)} \leftarrow \Delta_{e_2 N_{e_1.currentRoom}(i)} + penalty_{aw}$

6:     $\Delta_{e_2 e_2.prevRoom} \leftarrow 0$

7:     **return**

8: **if** $IsNear(e_1.prevRoom, e_2.prevRoom)$ AND $\neg IsNear(e_1.currentRoom, e_2.prevRoom)$ **then**

9:     **for** $i \leftarrow 1$ to $|R|$ **do**

10:         $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} + penalty_{aw}$

11:         **if** $IsNear(i, e_1.prevRoom) = true$ **then**

12:             $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} - penalty_{aw}$

13:         **if** $IsNear(i, e_1.currentRoom) = true$ **then**

14:             $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} + penalty_{aw}$

15:     $\Delta_{e_2 e_2.prevRoom} \leftarrow 0$

16:     **return**

17: **if** $\neg IsNear(e_1.prevRoom, e_2.prevRoom)$ AND $IsNear(e_1.currentRoom, e_2.prevRoom)$ **then**

18:     **for** $i \leftarrow 1$ to $|R|$ **do**

19:         $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} - penalty_{aw}$

20:         **if** $IsNear(i, e_1.prevRoom) = true$ **then**

21:             $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} - penalty_{aw}$

22:         **if** $IsNear(i, e_1.currentRoom) = true$ **then**

23:             $\Delta_{e_2 i} \leftarrow \Delta_{e_2 i} + penalty_{aw}$

24:     $\Delta_{e_2 e_2.prevRoom} \leftarrow 0$

25:     **return**

26: **if** $\neg IsNear(e_1.prevRoom, e_2.prevRoom)$ AND $\neg IsNear(e_1.currentRoom, e_2.prevRoom)$ **then**

27:     **for** $i \leftarrow 1$ to $|N_{e_1.prevRoom}|$ **do**

28:         $\Delta_{e_2 N_{e_1.prevRoom}(i)} \leftarrow \Delta_{e_2 N_{e_1.prevRoom}(i)} - penalty_{aw}$

29:     **for** $i \leftarrow 1$ to $|N_{e_1.currentRoom}|$ **do**

30:         $\Delta_{e_2 N_{e_1.currentRoom}(i)} \leftarrow \Delta_{e_2 N_{e_1.currentRoom}(i)} + penalty_{aw}$

31:     $\Delta_{e_2 e_2.prevRoom} \leftarrow 0$

32:     **return**

Figure B.8: Update algorithm for *away from* constraint

Input: cost change matrix Δ, Entity set E, Room set R

Output: cost change matrix Δ

1: **if** r_1 has a *capacity* constraint AND *room* $= r_1$ **then**

2:     *currentOveruse* $\leftarrow r_1.usage - r_1.space$

3:     *previousOveruse* $\leftarrow currentDifference + e_{ent}.space$

4:     **if** *previousOveruse* > 0 **then**

5:         **if** *currentOveruse* > 0 **then**

6:             **if** *previousOveruse* $> e_i.space$ AND *currentOveruse* $\leq e_i.space$ **then**

7:                 **for** $j \leftarrow 1$ to $|R|$ except r_1 **do**

8:                     $\Delta_{ij} \leftarrow \Delta_{ij} - penalty_{cp}$

9:         **else**

10:             **if** *previousOveruse* $\leq e_i.space$ **then**

11:                 **for** $j \leftarrow 1$ to $|R|$ except r_1 **do**

12:                     $\Delta_{ij} \leftarrow \Delta_{ij} + penalty_{cp}$

13: **else if** r_2 has a *capacity* constraint AND *room* $= r_2$ **then**

14:     *currentOveruse* $\leftarrow r_2.usage - r_2.space$

15:     *previousOveruse* $\leftarrow currentOveruse - e_e.space$

16:     **if** *currentOveruse* > 0 **then**

17:         **if** *previousOveruse* > 0 **then**

18:             **if** *previousOveruse* $\leq e_i.space$ AND *currentOveruse* $> e_i.space$ **then**

19:                 **for** $j \leftarrow 1$ to $|R|$ except r_2 **do**

20:                     $\Delta_{ij} \leftarrow \Delta_{ij} + penalty_{cp}$

21:         **else**

22:             **if** *currentOveruse* $\leq e_i.space$ **then**

23:                 **for** $j \leftarrow 1$ to $|R|$ except r_2 **do**

24:                     $\Delta_{ij} \leftarrow \Delta_{ij} - penalty_{cp}$

Figure B.9: Update algorithm for *capacity* constraint (conditions 1 and 2)

Input: cost change matrix Δ, Entity set E, Room set R

Output: cost change matrix Δ

1: **if** r_1 has a *capacity* constraint AND *room* $\neq r_1$ **then**

2:    *currentOveruse* $\leftarrow r_1.usage - r_1.space$

3:    *previousOveruse* $\leftarrow currentOveruse + e_e.space$

4:    **if** *currentOveruse* $= 0$ **then**

5:        $\Delta_{ir_1} \leftarrow \Delta_{ir_1} + penalty_{cp}$

6:    **else if** *currentOveruse* < 0 **then**

7:        **if** *previousOveruse* > 0 **then**

8:            **if** *currentUnderuse* $< e_i.space$ **then**

9:                $\Delta_{ir_1} \leftarrow \Delta_{ir_1} + penalty_{cp}$

10:        **else**

11:            **if** *previousUnderuse* $< e_i.space$ AND *currentUnderuse* $\geq e_i.space$ **then**

12:                $\Delta_{ir_1} \leftarrow \Delta_{ir_1} - penalty_{cp}$

13: **else if** r_2 has a *capacity* constraint AND *room* $\neq r_2$ **then**

14:    *currentOveruse* $\leftarrow r_2.usage - r_2.space$

15:    *previousOveruse* $\leftarrow currentDifference - e_e.space$

16:    **if** *previousOveruse* $= 0$ **then**

17:        $\Delta_{ir_2} \leftarrow \Delta_{ir_2} - penalty_{cp}$

18:    **else if** *previousOveruse* < 0 **then**

19:        **if** *currentOveruse* > 0 **then**

20:            **if** *previousUnderuse* $< e_i.space$ **then**

21:                $\Delta_{ir_2} \leftarrow \Delta_{ir_2} - penalty_{cp}$

22:        **else**

23:            **if** *currentUnderuse* $< e_i.space$ AND *previousUnderuse* $\geq e_i.space$ **then**

24:                $\Delta_{ir_2} \leftarrow \Delta_{ir_2} + penalty_{cp}$

Figure B.10: Update algorithm for *capacity* constraint (conditions 3 and 4)

Input: cost change matrix Δ, Entity set E, Room set R

Output: cost change matrix Δ

1: **for** $i \leftarrow 1$ to $|E|$ except entity e **do**

2:    *room* $\leftarrow e_i.room$

3:    Condition 1

4:    Condition 2

5:    Condition 3

6:    Condition 4

Figure B.11: Update algorithm for *capacity* constraint

References

E. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization*. Wiley, first edition, 2003.

A. O. Adewumi and M. M. Ali. A multi-level genetic algorithm for a multi-stage space allocation problem. *Mathematical and Computer Modelling*, 51(1-2):109– 126, 2010.

D. T. Anh and N. T. Trung. Three improved variants of simulated annealing for optimising dorm room assignments. *International J. Intell. Inf. Database Syst.*, 5(3):296–312, May 2011. ISSN 1751-5858.

M. A. Awadallah, A. T. Khader, M. A. Al-Betar, and P. C. Woon. Office-space-allocation problem using harmony search algorithm. In Tingwen Huang, Zhigang Zeng, Chuandong Li, and Chi-Sing Leung, editors, *ICONIP (2)*, volume 7664 of *Lecture Notes in Computer Science*, pages 365–374. Springer, 2012.

R. Bai. *An Investigation of Novel Approaches for Optimising Retail Shelf Space Allocation*. PhD thesis, School of Computer Science and Information Technology, University of Nottingham, 2005.

J. E. Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms*, pages 101–111, Mahwah, NJ, USA, 1985. Lawrence Erlbaum Associates, Inc.

J. M. Baldwin. *Adaptive Individuals in Evolving Populations: Models and Algorithms - A New Factor in Evolution*, pages 59–80. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

W. B. Ball. Space utilization optimization. White Paper, 2009. URL `http://gis.larc.nasa.gov/spaceopt/`.

M. Beckman and T. C. Koopmans. Assignment problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.

C. Benjamin, I. Ehie, and Y. Omurtag. Planning facilities at the university of missouri-rolla. *Interfaces*, 22(4):94–105, 1992.

C. Beyrouthy. *Models, Solution Methods and Threshold Behaviour for the Teaching Space Allocation Problem*. PhD thesis, School of Computer Science and Information technology, University of Nottingham, 2008.

C. Beyrouthy, E. K. Burke, B. McCollum, P. McMullan, J. D. Landa-Silva, and A. J. Parkes. Towards improving the utilisation of university teaching space. *The Journal of Operational Research Society*, 60:130–143, 2009.

C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, September 2003.

A. Brindle. *Genetic Algorithms for Function Optimization*. PhD thesis, University of Alberta, Edmonton, Canada, 1981.

E. K. Burke and D. B. Varley. Space allocation: An analysis of higher education requirements. In *The Practice and Theory of Automated Timetabling II (PATAT 1997), LNCS, Vol. 1408*, pages 20–33. Springer-Verlag, 1998.

E. K. Burke, P. Cowling, and J. D. Landa-Silva. Hybrid population-based metaheuristic approaches for the space allocation problem. In *Proceedings of the 2001 Congress on Evolutionary Computation (CEC 2001)*, pages 232–239, 2001a.

E. K. Burke, P. Cowling, J. D. Landa-Silva, and B. McCollum. Three methods to automate the space allocation process in UK universities. In *The Practice and Theory of Automated Timetabling III (PATAT 2004), LNCS, Vol. 2079*, pages 254–273. Springer, 2001b.

E. K. Burke, P. Cowling, J. D. Landa-Silva, and S. Petrovic. 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)*, pages 1252–1259, 2001c.

E. K. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-Heuristics: An Emerging Direction in Modern Search Technology. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, International Series in Operations Research and Management Science, chapter 16, pages 457–474. Kluwer Academic Publishers, 2003.

E. K. Burke, J. D. Landa-Silva, and E. Soubeiga. *Multi-Objective Hyper-Heuristic Approaches for Space Allocation and Timetabling*, pages 129–158. Springer, 2005.

E. K. Burke, M. Hyde, G., G. Ochoa, E. Ozcan, and J. R. Woodward. A classification of hyper-heuristics approaches. In M. Gendreau and J-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research and Management Science*, chapter 15, pages 449–468. Springer, second edition, 2010.

D. G. Cattrysse and L. N. Van Wassenhove. A survey of algorithms for the generalized assignment problem. *European Journal of Operational Research*, 60(3):260–272, 1992.

J. W. Chinneck. *Practical Optimization: A Gentle Introduction*. Carleton University, 2012.

C. A. Coello Coello, G. B. Lamont, and D. A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer-Verlag, Secaucus, NJ, USA, second edition, 2006.

D. Corne and P. Ross. Peckish initialisation strategies for evolutionary timetabling. In Edmund Burke and Peter Ross, editors, *Practice and Theory of Automated Timetabling*, volume 1153 of *Lecture Notes in Computer Science*, pages 227–240. Springer Berlin Heidelberg, 1996.

C. Cotta-Porras. A study of hybridisation techniques and their application to the design of evolutionary algorithms. *AI Communications*, 11(3):223–224, 1998.

G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

G. Dueck. New optimization heuristics the great deluge algorithm and the record-to-record travel. *Journal of Computational Physics*, 104:86–92, 1993.

G. Dueck and T. Scheuer. Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, 90(1):161–175, 1990.

I. Dumitrescu and T. Stützle. Combinations of local search and exact algorithms. In S. Cagnoni, C. G. Johnson, J. J. R. Cardalda, R. Marchiori, D. W. Corne, J. Meyer, J. Gottlieb, M. Middendorf, A. Guillot, G. Raidl, and E. Hart, editors, *Applications of Evolutionary Computing*, volume 2611 of *Lecture Notes in Computer Science*, pages 211–223. Springer Berlin Heidelberg, 2003.

B. S. Everitt, S. Landau, and M. Leese. *Cluster Analysis*. Hodder Arnold Publication. John Wiley & Sons, 2009. ISBN 9780340761199.

P. Galinier and J. K. Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379–397, 1999.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

Z. W. Geem. Novel derivative of harmony search algorithm for discrete design variables. *Applied Mathematics and Computation*, 199(1):223–230, 2008.

Z. W. Geem, J. H. Kim, and G. V. Loganathan. A New Heuristic Optimization Algorithm: Harmony Search. *SIMULATION*, 76(2):60–68, feb 2001.

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

F. W. Glover. Tabu search, part 1. *ORSA Journal on Computing*, 1:190–206, 1989.

F. W. Glover. Tabu search, part 2. *ORSA Journal on Computing*, 2(1):4–32, 1990.

F. W. Glover and G. A. Kochenberger. Handbook of metaheuristics, January 2003.

F. W. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, first edition, 1989.

R. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.

Gurobi-Optimization. Gurobi, 2010. URL `http://www.gurobi.com`.

B. Hajek. Cooling schedules for optimal annealing. *Mathematics of Operations Research*, 13(2):311–329, 1988.

Y. C. Ho and D. L. Pepyne. Simple Explanation of the No-Free-Lunch Theorem and Its Implications. *Journal of Optimization Theory and Applications*, 115(3):549–570, dec 2002.

J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, third edition, 2006.

IBM-Ilog. Cplex, 2013. URL `http://www-01.ibm.com/software/integration/optimization/cplex/`.

M. Juenger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, and L. A. Wolsey G . Rinaldi. *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Springer, 2010.

N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984. ISSN 0209-9683.

H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer, 2004. ISBN 978-3-540-40286-2.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

J. D. Landa-Silva. *Metaheuristics and Multiobjective Approaches for Space Allocation*. PhD thesis, School of Computer Science and Information Technology, University of Nottingham, 2003.

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

R. Lopes and D. Girimonte. The office-space-allocation problem in strongly hierarchized organizations. In *Evolutionary Computation in Combinatorial Optimization*, volume 6022 of *LNCS*, pages 143–153. Springer, 2010.

H. R. Lourenco, O. C. Martin, and T. Stutzle. Iterated local search. In *Handbook of Metaheuristics, volume 57 of International Series in Operations Research and Management Science*, pages 321–353. Kluwer Academic Publishers, 2002.

M. E. Lübbecke and J. Desrosiers. Selected topics in column generation. *Operations Research*, 53 (6):1007–1023, 2005. ISSN 0030-364X.

V. Maniezzo, T. Stützle, and S. Voss. Matheuristics: Hybridizing metaheuristics and mathematical programming. *Annals of Information Systems*, 10, 2009.

H. Marchand, A. Martin, R. Weismantel, and L. Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123(1-3):397–446, nov 2002.

S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, USA, 1990.

N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

N. Mladenovic and P. B. Hansen. Variable neighborhood search. *Computers and Operations Research*, 24(11):1097–1100, 1997.

P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts - towards memetic algorithms, 1989.

Pablo Moscato and J. F. Fontanari. Stochastic versus deterministic update in simulated annealing. *Physics Letters A*, 146(4):204–208, 1990.

University of Michigan. Research space guidelines, 2012. URL `http://www.provost.umich.edu/space/other/ResearchSpaceGuidelines.pdf`.

I. Osman and G. Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63(5): 511–623, October 1996.

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

R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization: An overview. *Swarm Intelligence*, 1:33–57, 2007.

J. Puchinger and G. R. Raidl. Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In *First International Work-Conference pm the Interplay Between Natural and Artificial Computation, Part II*, volume 3562, pages 41–53, 2005.

J. Puchinger, G. R. Raidl, and U. Pferschy. The multidimensional knapsack problem: Structure and algorithms. *Informs Journal on Computing*, 2009.

G. R. Raidl. A unified view on hybrid metaheuristics. In *Proceedings of the Third international conference on Hybrid Metaheuristics*, HM'06, pages 1–12, Berlin, Heidelberg, 2006. Springer-Verlag.

G. R. Raidl and J. Puchinger. Combining (integer) linear programming techniques and meta-heuristics for combinatorial optimization. In *Hybrid Metaheuristics*, pages 31–62. Springer, 2008.

M. G. C. Resende and C. C. Ribeiro. Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 219–249. Kluwer Academic Publishers, 2003.

L. Ritzman, J. Bradford, and R. Jacobs. A multiple objective approach to space planning for academic facilities. *Managament Science*, 25(9):895–906, 1980.

P. Ross, D. Corne, and H.L. Fang. Improving evolutionary timetabling with delta evaluation and directed mutation. In *PPSN III: Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature*, pages 556–565, London, UK, 1994. Springer-Verlag.

G. Schrimpf, J. Schneider, H. Stamm-Wilbrandt, and G. Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159(2):139–171, 2000.

R. Sharpe. Optimum space allocation within buildings. *Building Science*, 8(3):201–205, 1973.

M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, first edition, 1996.

P. N. Strenski and S. Kirkpatrick. Analysis of finite length annealing schedules. *Algorithmica*, 6 (3):346–366, 1991.

E. G. Talbi. A taxonomy of hybrid metaheuristics. *Journal of Heuristics*, 8(5):541–564, 2002.

M. Tamiz, D. Jones, and C. Romero. Goal programming for decision making: An overview of the current state-of-the-art. *European Journal of Operational Research*, 111(3):569 – 581, 1998.

N. T. Trung and D. T. Anh. Comparing three improved variants of simulated annealing for optimizing dorm room assignments. In *Computing and Communication Technologies, 2009. RIVF '09. International Conference on*, pages 1 –5, july 2009.

N. T. Trung, T. N. Tuan, and D. T. Anh. Informed simulated annealing for optimizing dorm room assignments. In *Intelligent Information and Database Systems, 2009. ACIIDS 2009. First Asian Conference on*, pages 265 –270, april 2009.

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

Ö. Ülker and J. D. Landa-Silva. Designing difficult office space allocation problem instances with mathematical programming. In *SEA - Symposium of Experimental Algorithms*, pages 280–291, 2011.

Ö. Ülker and J. D. Landa-Silva. Evolutionary local search for solving the office space allocation problem. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 3573–3580, june 2012.

Ö. Ülker and J. D. Landa-Silva. Analysis of office space allocation problem using mathematical programming. In *To be decided*, pages 1–25, 2013a.

Ö. Ülker and J. D. Landa-Silva. Two neighbourhood iterated local search algorithm for space allocation problem. In *Workshop on Hybrid Meta-heuristics 2013*, pages 1–15, 2013b.

S. Voss, I. H. Osman, and C. Roucairol, editors. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

C. Voudouris and E. Tsang. Guided local search. *European Journal of Operational Research*, 113 (2):469–499, 1999.

D. L. Whitley, V. S. Gordon, and K. E. Mathias. Lamarckian evolution, the baldwin effect and function optimization. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature – PPSN III*, pages 6–15, Berlin, 1994. Springer.

W. E. Wilhelm. A technical review of column generation in integer programming". *Optimization and Engineering*, 2:159–200, 2001.

H. P. Williams. *Model Building in Mathematical Programming*. John Wiley and Sons, fourth edition, 1999.

I.H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers in Data Management System, San Francisco, CA, USA, second edition, 2005.

D. H. Wolpert and W. G. Macready. No free lunch theorems for search. Tech. Rep. No. SFI-TR-95-02-010, Santa Fe Institute, Santa Fe, NM, 1995.

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67 –82, apr 1997.

D. H. Wolpert and W. G. Macready. Coevolutionary free lunches. *IEEE Transactions on Evolutionary Computation*, 9(6):721–735, dec 2005.

L. A. Wolsey. *Integer programming*. Wiley Interscience, first edition, 1998.

M. H. Yang and W. C. Chen. A study on shelf space allocation and management. *International Journal of Production Economics*, 60-61:309–317, 1999.

S. H. Zahiri. Fuzzy multi-objective PSO: An approach for office space allocation. *Iranian Journal of Electrical and Computer Engineering*, 8(2):61–70, 2009.