Master's Degree in
Mechatronic Engineering

Optimizing Surgical Scheduling in Healthcare Services: Models and Algorithms for Minimizing Makespan in No-Wait Scenarios

Supervisor
Prof. Marco Ghirardi

Co-supervisor
Dott. Roberto Bargetto

Candidate
Federico Lauritano

April 2024

Abstract

This thesis addresses a practical challenge encountered in the management of operating rooms within healthcare institutions. When dealing with the issue of scheduling a list of patients with varying surgery durations, an incorrect decision can result in significant time inefficiencies across the overall schedule, leading to a waste of resources that this work aims at minimizing. The problem of organizing tasks is well studied in the context of scheduling theory and the general idea is to search for a solution that satisfies all the requirements presented and at the same time minimizes a certain quantity indicating the quality of a solution. The first step in solving a scheduling problem is identifying what are the Decision Variables (DVs) that best represent the problem within the context. The following step regards translating the limitations of the problem into mathematically rigorous Constraints that correctly fit the DVs previously chosen. Finally, the Objective Function (OF) is chosen as the key parameter to be optimized. In analogy with scheduling theory we can consider the rooms of the facility as machines that can perform one task at the time, and the patients as jobs composed of several ordered tasks. In the context of the presented problem we consider a facility in which only one Operating Theater (OT) is available, composed by a single Anesthesia/Observation Place (AOP) room and a single Operating Room (OR). The single patient is seen as a sequence of three operations to be performed: Anesthesia (A) to be performed only in AOP, Surgery (S) to be performed only in OR and Awakening (AW) to be performed in any of the two rooms. The additional information obtained by the context translates into constraint that impose several complications to the problem, like the No-Wait constraint between tasks of a patient and the No-overlap of different patients inside the same room.
The problem is presented in two variants: the AOP Scheduling Problem (AOPSP) and the Integrated AOP and OR Scheduling Problem (IAOPORSP). In the AOPSP variant the order of the patients is predetermined, thus leaving the decision to only the placement of the AW. In the IAOPORSP variant the ordering of patients must also be determined together with the AW placement. Both variants consider as OF the Makespan of the schedule, so the amount of time between the beginning of the first task and the ending of the last task in the OT.
Due to the vast number of potential solutions, exhaustively exploring all possibilities is impractical and time-consuming. Therefore, an heuristic algorithmic approach is essential to efficiently solve the problem. This thesis details the methodologies employed to tackle both problems, starting with several modeling techniques for the purpose of using a commercial solver to attempt the resolution, and then progressing to various algorithmic strategies designed to match the solver's outcomes while reducing computational time. Finally the results will be showcased by comparing the discrepancy between the outcomes of the most effective algorithm found and the optimal result obtained by the commercial solver.

Table of Contents

Acronyms

A Anesthesia

AOP Anesthesia-Observation Place

AOPSP Anesthesia-Observation Place Scheduling Problem

AW AWakening

BG Backward Greedy

BPP Bin Packaging Problem

DV Decision Variable

FG Forward Greedy

GFG Generalized Forward Greedy

IAOPORSP Integrated Anesthesia-Observation Place and Operating Room Scheduling Problem

ID Input Domain

ILP Integer-Linear Programming

KP Knapsack Problem

LB Lower Bound

LP Linear Programming

MILP Mixed-Integer-Linear Programming

OD Output Domain

OF Objective Function

OR Operating Room

OT Operating Theatre

QP Quadratic Programming

S Surgery

SNKSA Sequential Neighborhood Kernel Search Algorithm

WSAP Wedding Seating Arrangement Problem

Chapter 1

Introduction

1.1 Historical overview on Scheduling

The purpose of scheduling is to optimally allocate resources to minimize costs for a project. Resources can take the form of money, natural resources and, often, time, which is strongly correlated with the money expense of a project. The idea of correctly using resources to avoid wastage is so basic that to find a moment in history in which this concept has begun to be applied would be almost impossible. It can be argued that resource allocation played a pivotal role in the evolutionary trajectory that led to humans becoming the most advanced species on the planet. This assertion is grounded in the principle that access to a greater abundance of resources enhances survival prospects. Consequently, it follows logically that more efficient utilization of resources would further improve chances of survival. This concept has been partially analyzed by S. Ólafsson (1996) [1]; his paper explores the utilization of evolutionary game theory for the purpose of improving allocation of resources in a complex network.
According to Hyatt et al. (2006) we have proofs of basic scheduling operations for projects of several millenia ago, like the construction of Pyramids or organization in military combat [2]. One of the most used tool for graphical representation of scheduling problems, Joseph Priestley is considered the father of Bar Charts, with his *Chart of Biography* being a notorious example from 1765, shown in Figure 1.1. Moving closer to modern era, in 1916 we find the first scheduling problem explicit discussion from the American engineer Henry Gantt, a very important name when talking about scheduling. As explained by Potts et al., it is around the 1950 that a considerable amount of articles regarding what we now consider scheduling started to appear [4]. His paper provides a really interesting insight on what the main research topics have been over the decades following the 1950s, with an interesting classification of the main topics of interest that can be resumed in five main blocks:

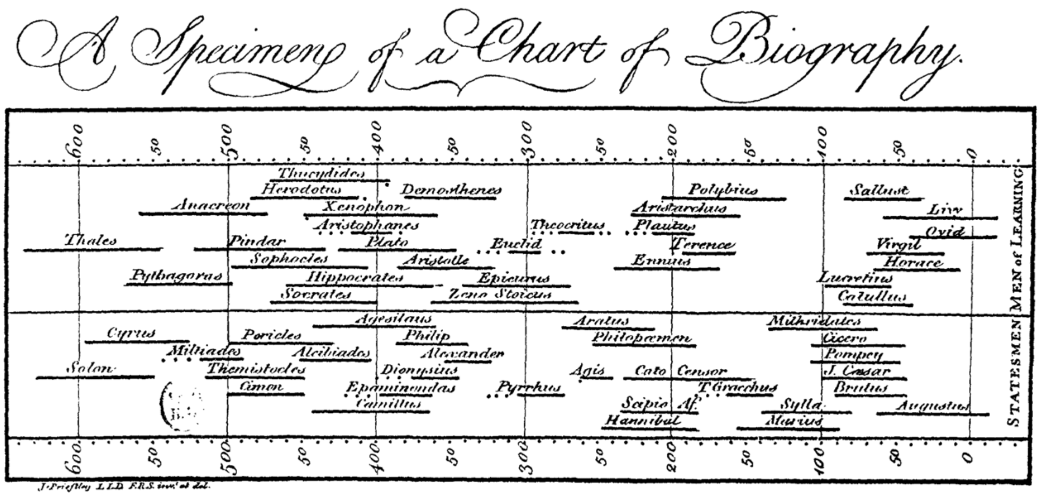

Figure 1.1: A Specimen of a Chart of Biography. Source: [3]

- **Combinatorial analysis**: The problem of scheduling is based on the fact that given a problem with certain requests to fulfill, the solution space is big enough to make the complete analysis of all the possible solutions impractical. Combinatorial analysis tries to understand how big is this feasible solutions space, a key element in understanding a problem.
- **Branch and bound**: One of the first techniques developed for solving problems of great dimensions. The main introduction of this methodology is the idea of reducing the solution space with considerations based on a *branching rule* specific for the problem.
- **Complexity and classification**: The foundations of the current scheduling concept make this period one of the most significant, marked by the definition of computational complexity theories and a classification of scheduling problem types.
- **Approximate solutions**: Many problems discovered over the years turned out to be *NP-Hard*, improperly meaning "hard to solve", and for these kind of problems finding an optimal solution is often difficult and expensive. The idea of getting close enough to an optimal solution started to spread in this period, with approximation algorithms and heuristics solutions as the one studied in this thesis work.
- **Enhanced scheduling models** : This is the current period of research for scheduling, it is now hard to determined a specific argument the research focuses on since the topics of interested range in a wide variety of problems and approaches.

1.2 Problem introduction and purpose of this thesis

In the context of healthcare facilities time is one of the resources that directly impact both the service provider and the service receivers. Minimizing the amount of time a patient spends inside a hospital directly translates to a lower monetary cost for the healthcare facility and, more importantly, to a lower risk of contracting additional diseases for the patient. This is a well known factor in the field of medicine, and it is in fact good practice to reduce as much as possible the length of the hospital stay for any patient. According to Delgado-Rodríguez et al. (1990) [5] the risk of nosocomial infections rises monotonously with the duration of the hospital admission, and this is clearly a bad thing. From the point of view of the care provider instead, operating rooms are among the most costly part of the facility to run, and thus it is again crucial to minimize wastage of time inside the facility. J. Raft et al. performed a really interesting study in which they attempt to compute the cost of running a surgical theater, and the results show the importance of reducing as much as possible the time wastage [6].
For the specific interest of this work, the only consideration that will be made regards the scheduling of the patients inside the OT from the point of view of ordering and placement in the correct rooms. No attention is given to material resources usage and human labor availability, and the assumption of not having unexpected outcomes in the hospital will be considered, as in failed surgeries or delays imposed by the moving of patients.
The case analysed will consider an healthcare facility with only two rooms available for the whole surgery planning, however there is good reason to believe that once the work will be done it will be easy to scale the problem to instances having greater room availability. The first room is the Operating Room, where the actual surgery occurs, whereas the other available room is the Anesthesia/Observation Place. The purpose of each room requires discussing the patients modeling first. A process that needs to undergo three phases to complete the surgery is considered: first an Anesthesia that can be performed only in the AOP, immediately after the Surgery that can only be performed in the OR, and lastly the Awakening that can occur in any of the two rooms. The objective of the study is to minimize the amount of time the OT has to be kept open, so make the last AW of the day end as soon as possible. The challenge of this problem arises on many aspects; the patients obviously cannot overlap inside a single room, and thus each room can host a single patient at the time. Moreover, no time can pass between any of the three phases for each patient and the following, so the beginning of the S must coincide with the ending of the A and the beginning of the AW must occur at the same time the S finishes. Wasn't for the AW that can be placed in any of the two rooms, the problem would have a much lower complexity, but this constraint makes this real life problem an incredibly peculiar one to model. While for a human being finding a decent schedule is quite

easy and intuitive, finding a very good schedule, and even worse the best schedule among the possible ones, turned out to be a very hard task.
The just described problem has been approached in two variants, an easier case in which the ordering of patients is already determined and only the placement of each AW must be fixed, and a much more complex variation in which also the ordering of patients must be determined. The pool of patients to schedule has been assumed to be in the range of 11 to 19 patients, which makes a reasonable amount of patients for a single workday in an OT. This consideration has been preceded by reasoning on the average duration of the processes in study, however given the extremely high variance for the single tasks of a surgery, dependent on factors like the patient characteristics and surgery to perform, it has been chosen to generalize the problem as much as possible to make the duration of the processes irrelevant for the study. The validation dataset, later defined, accounts for this consideration without going into details not relevant from a scheduling point of view.

1.2.1 Approaches for solving a scheduling problem

When tackling a scheduling problem there are several roads that can be taken to solve it optimally. In this brief section the main techniques will be illustrated without going too deep into the technical explanations, kept later for the methodologies chosen. With each methodology a deeper analysis from known papers will be provided when available.

Linear Programming:

It is the workhorse of solving scheduling problems. The idea is to model the problem as a set of linear relationship between the relevant measurable quantities of the problem, together with a linear objective in therms of what the outcome of the problem is. The relationship between variables limits the amount of possible states the system can take, and ensures that a solution respecting all the constraints is indeed a feasible solution for the problem. The objective is a metric that allows to understand how good is a certain solution for the problem.
Once the constraints and objective are defined, it is usually necessary to rely on a dedicated software capable of finding a good solution without the need of searching among all the possible ones, since usually this kind of problems have an amount of feasible solutions far greater than what can be checked individually. This thesis relies on the commercial software Gurobi (version 10.0.3), from Gurobi Optimization, LLC, for solving the linear models created of the problems. A slight variation of this method will be used in the MILP section of this thesis. MILP is LP with the

addition of integer variables, usually necessary to model choices that are either yes or no. Alexandra Sariel (1994) made an interesting comprehensive overview on LP [7].

Quadratic Programming:

This is a generalization of LP, it essentially allows the objective to have a more complex formulation (being quadratic and not only linear). For the specific problem studied in this thesis QP was a feasible tool indeed but not a necessary one, as the modeling of the problem can be made without any necessity of using quadratic relationships. For this reason this method has not been considered. K. G. Murty (2010) published a general overview on the topic, describing how the quadratic approach is a generalization of the linear one and as such has a more complex structure for solution searching [8].

Simulation-based optimization:

Simulation-based optimization utilizes computational models to evaluate potential solutions. It involves constructing a simulation model that represents the problem domain and using it to assess candidate solutions. This approach can handle stochastic elements and uncertainty, making it suitable for real-world problems. Given the deterministic nature of the problem under study, this technique was also not considered. Satyajith Amaran et al. (2014) analysis of this technique was performed relatively recently, considering that this methodology requires quite a big computational power capability for the system performing it [9].

Meta heuristic Search:

These algorithms iteratively explore the solution space, often guided by heuristic (i.e. approximate, not exact) rules, to efficiently find solutions. They are widely used in various fields for problems with large or irregular solution spaces where exact methods are impractical. Among the previously described methodologies this is the one that can probably better solve the problem, as it has already been used for solving very efficiently similar problems. Very relevant example of this is the study performed by Wang Meng-meng (2011) in which a Simulated Annealing algorithm is used for solving a scheduling problem in the topic of supply chains [10]. Since the topic has already been studied deeply and with very promising results, it was not attempted in this specific thesis to build a simulated annealing algorithm, but it has been taken as a good reference for constructing a dedicated algorithm. Anupriya Gogna and A. Tayal (2013) made an extensive description of the Meta heuristic techniques developed in the last two decades [11].

Ad hoc Algorithms:

When a problem has a very peculiar structure, or when it has very specific constraint, it is possible to rely on an algorithm created only for that specific problem. This is the case for this thesis work, where the atypical nature of the problem makes it advantageous to find a dedicated methodology. While a solution of this kind cannot be applied to any other problem, since even the smallest difference from the original request makes the algorithm useless, it is also true that for a problem like the one under study (i.e. a lot of small instances compared to few big ones), it is better to have a dedicated very fast solution compared to a more general but slow one. Considering that the logistic of changing approach in the medical field makes it very unlikely that the model for studying surgery procedures will change soon, it is reasonable to look for a dedicated algorithm for this problem without thinking about the flexibility of the algorithm.

1.2.2 Current status of research on the topic and my contribution

As explained in 1.1 the interest in scheduling problems has started not more than 50 years ago, and as so today there is still a lot to discover. Research on the topic can be divided into two main directions: a vertical one where the interest is more oriented toward finding a general approach at solving general problems, and a more horizontal direction in which the aim is finding specific solutions for specific problems, as in the case under study. The trade-off between the two is that general methods offer a broader applicability, allowing for a quicker adaptation across different problems in a shorter and cheaper amount of time. On the other hand a specific solution, while necessitating longer development times and deeper understanding of the domain of the problem, do not guarantee flexibility in moving toward a different situation.
The research landscape is marked by efforts to balance these aspects, aiming to leverage the depth of ad-hoc solutions and the breadth of general methods. This ongoing exploration reflects a broader goal within operational research: to develop methodologies that are both widely applicable and capable of integrating specific knowledge to enhance performance in particular contexts.
Once the horizontal direction of research is clear, it is of no surprise that while the topic of scheduling patients for surgery has been already investigated in the past (from both the medical world and engineering-mathematical world), no specific case identical to the one in consideration was found. F. Dexter et al. (2002) published an article about the scheduling of patients but without the constraint of having only two rooms available and using a different model for the surgery procedure [12].
A. Saremi et al. (2013) published a study on different approaches for optimizing the day surgery appointments in a Canadian hospital and the results obtained by

applying said methods, highlighting a significant improvement in efficiency and also in patients satisfaction after the procedure [13]. While this last article considers a multi-stage environment as this thesis does, it also takes into account several factor that are considered irrelevant from a scheduling point of view and that in this thesis have been ignored (but that are nonetheless relevant when considering the real life application of the problem). No research on the specific case considered in this thesis has been found.
The ultimate goal of this work is to use a General existing solution, provided by the MILP models of the problems processed by the commercial solver, to develop an ad hoc algorithm capable of using much less computational resources to obtain a result as close as possible to the optimal one in quality.

1.2.3 Solutions representation

To show the results of the optimization process a dedicated-structure Gantt chart will be used, as depicted in the example of figure 1.2.
The two rows of the chart represent the two rooms available in the OT, with time

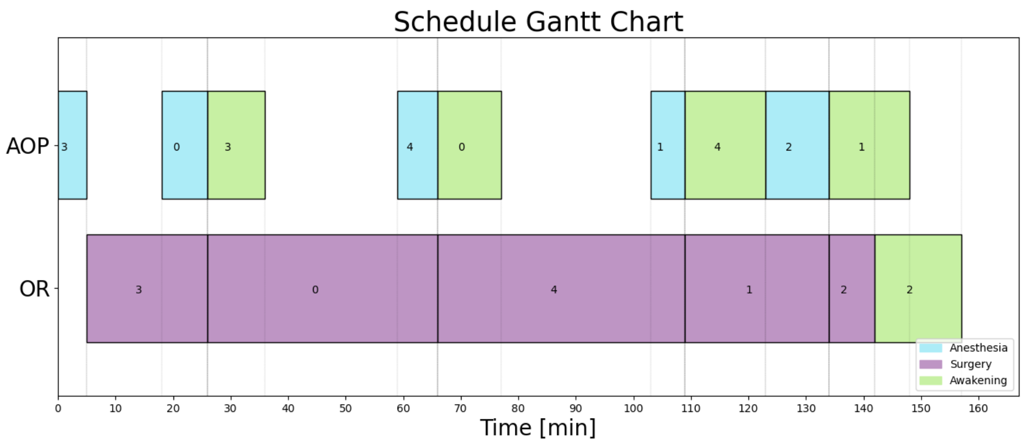

Figure 1.2: Example of schedule with 5 patients.

expressed in minutes along the horizontal axis and the tasks performed in a certain time frame grouped by color for distinguishing A,S and AW.
Notice how the Completion of any A is vertically aligned with the beginning of the associated S, and same applies to any S and associated AW, regardless of where AW occurs.

1.3 Other comments

The focus of this thesis is the exploration of algorithms, and as such, detailed attention to the precise implementation of the tests conducted will not be emphasized. The entirety of the research was carried out using Python (version 3.10.5) as programming language, with the algorithms executed on the author's personal computer. While all materials related to the study, including scripts, datasets, and implementations (even those that did not yield successful outcomes) are available for the sake of completeness (GitHub: LaFede99)it is pertinent to clarify that I, the author, do not possess a background in computer science. Consequently, I do not regard the implementation of the algorithms as exemplary. For this reason, discussions will not focus into the execution time of the code in an absolute sense; rather, comparisons will be drawn solely on the relative performance between two algorithms.

Chapter 2

Theoretical Fundamentals

2.1 Problem and algorithm complexity

Problems

The word "problem" doesn't need a mathematical definition. It is the general meaning of a real world situation that requires a solution. In therms of what finding a solution can signify, there is no specific categorization: finding a solution may imply positioning objects in certain locations, having a simple yes-no answer for a decision, or even deciding a value for something that has to be set. The important concept is that a problem is a real world situation in which there is a decision to make that, in some way, affects the state of the system, and the intention is to control the system in a way that is advantageous.
The initial step in solving a problem involves finding a methodical approach to tackle it; this first step represent the modeling phase of the problem. Modeling is a procedure that allows to translate the system under investigation into a set of rigorous mathematical objects. This parallelism has to be rigorous in the sense that the representation must translate all the relevant details of the situation into rules such that the problem can then be translated back to a tangible form, once the solution has been found, without any loss of relevant information in the process.

Definition 1 (Problem's Model)**.** *A model is a set of mathematically coherent variables (and associated constraints), expressions and concepts, correctly abstracting the problem without loss of relevant information in the translation phase (in both directions of translation).*

Working with a mathematical model enables the application of rigorous principles, already known from existing theory, to the object within the model, ensuring a clear and structured analysis. Moreover, by extracting the system from its real world context, a (correct) model eliminates extraneous details and distractions allowing

for a focused examination of the core dynamics at play. This decoupling from the real world scenario enhances the capability of solving complex problems by focusing on the essential aspects, as long as no key relationship inside the system is lost in the translation phase. The topic is well explained by Dündar et al. (2012) in the article "Mathematical Modelling at a Glance: A Theoretical Study" [14].
The components of the model that are worked on are the decision variables, that represent the input of the system (the part we can interact and play with), the system outcome (the information we are interested in that changes with the input variables), and the relationship among all the elements. A specific model, may also have additional input variables that are not to be tuned but that characterize the specific situation we are working on, distinguishing models of the same structure but with different characteristic.

Definition 2 (Instance). *An instance of a model is a set of input characteristic that modify the behavior of the system without changing the way the model can be approached. Are sometimes called the parameters of the model, they don't change during the optimization phase.*

Algorithms

Once the model has been correctly defined the successive step is finding an approach that allows the resolution of the problem to the desired extent. Generally speaking the problems under study are part of the Combinatorial Optimization domain, where the Input variables of the system must follow the constraint that only a finite amount of possible input combinations can exist.

Definition 3 (Input Domain). *The Input Domain (ID) is the set of all possible combinations of the Inputs parameters that a model can take. It can be finite or infinite, discrete or continuous.*

Definition 4 (Output Domain). *The Output Domain (OD) is the set of all possible output values produced by the system in response to given inputs. It can be finite or infinite depending on the ID, discrete or continuous.*

Once the finitness of the ID has been confirmed, a procedure for exploring it can be determined. The most basic, but unreasonable for big instances, procedure may be described as enumerating all the possible states of the system, obtained by considering the whole ID, and then proceeding to evaluate all the solutions obtained to determine the best one. This approach would indeed lead to the best solution, but in many cases the dimension of the ID makes it impossible to explore all the feasible cases.

Definition 5 (Optimality of a solution)**.** *Optimal solutions are the solutions that, once defined the OF of the model, have a better objective function value compared to all other possible solutions. Optimality may be local, when defined only within a subset of the ID, and unique, when no more than one solution leads to optimal OF. Optimality always exist within a finite ID, but the uniqueness is not guaranteed.*

Instead of exploring all the possible solutions, it is often better to explore the ID by using a systemic algorithm. According to Yanofsky et al. (2006), an algorithm is defined as a set of equivalent programs that perform the same function [15]. A less rigorous but more understandable definition is that an algorithm is a sequence of exact steps (important when specifying that a computer must be able to perform it). Usually an algorithm is used for performing a sequence of operations. If well structured, this sequence can also solve a problem.
In the context of Combinatorial Optimizaton we are interested usually in algorithms that iteratively move inside the ID without the need of exploring it all, or that allow to build a solution without the need of exploring the ID explicitly at all.

Computational complexity

When dealing with a problem, it is important to understand the difference between the complexity of the problem and the complexity of the algorithm. The complexity of a problem is a property related to the dimension of the ID and to how hard it is to actually solve the problem. The complexity of an algorithm instead, describes the relation between the dimension of the instance that is fed to the problem, and the amount of computation required to perform the algorithm. Since there is not a unique way of solving a problem, it is clear that the same problem can be solved using different algorithms of different complexity. Lower complexity obviously translates to lower amount of time needed to perform computation, and so less "energy" consumption in executing it. (whether we refer to raw energy we use in applying by hand the algorithm or actual electrical energy consumed by the CPU of the computer performing the computation). Often an algorithm performs some decision during execution, leading to different behavior between different executions, and different length to perform them. To evaluate the complexity we always consider the worst case possible when running the algorithm.
Calling n the dimension of the instance under investigation, and calling A the algorithm we are interested in, we express "A is $\mathcal{O}(f(n))$" if the Algorithm A has the function f(n) as upper bound for the time complexity of execution, namely if the function f will always take more time than algorithm A to complete if both are fed the same input n. Formally:

Definition 6 ($\mathcal{O}(f(x))$)**.** *We say $g(x) = \mathcal{O}(f(x))$ if there exist some positive con-*

stants C and k s.t. $g(x) \leq C \cdot f(x)\ \forall x \geq k$

A special distinction in the classification occurs when we consider the nature of the function f(x). We say that the computational complexity of the algorithm A is Polynomial if, given the previous definitions, f is a polynomial function of x. This distinction is very important because the polynomial nature of the upper limit makes it easy to understand how fast does computation time rise, according to the maximum exponent of the polynomial. If we fix the speed at which the computing machine executing the algorithm performs operations, and compare polynomial algorithms with other kind of function, it is clear how the increase in computation is much more manageable in the polynomial case. This can be observed in the contents of table 2.1.

Type of function	**size n**					
	10	**20**	**30**	**40**	**50**	**60**
n	0.00001 seconds	0.00002 seconds	0.00003 seconds	0.00004 seconds	0.00005 seconds	0.00006 seconds
$\mathbf{n^2}$	0.0001 seconds	0.0004 seconds	0.0009 seconds	0.0016 seconds	0.0025 seconds	0.0036 seconds
$\mathbf{n^3}$	0.001 seconds	0.008 seconds	0.027 seconds	0.064 seconds	0.125 seconds	0.216 seconds
$\mathbf{n^5}$	0.1 seconds	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
$\mathbf{2^n}$	0.001 seconds	1 second	17.9 minutes	12.7 days	35.7 years	366 centuries
$\mathbf{3^n}$	0.059 seconds	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{12} centuries

Table 2.1: Amount of time required for a computer of 10^6 operations/second to perform f(n) operations, as f changes.

It is obvious that the aim when searching for an algorithm that has to perform some task should be to find a polynomial algorithm whose degree is as low as possible, with lowest possible complexity being $\mathcal{O}(1)$ which represents an algorithm with fixed execution time regardless of the dimension of the instance.

2.2 Scheduling problems classification

Scheduling theory is a branch of Applied Mathematics, which is a branch of Operations Research. It's concerned with mathematical formulations and solution methods of problems of optimal ordering and coordination in time of certain operations (Zinder et al., 2021)[16] . The purpose of this field is usually optimizing a specific time-related quantity, often minimizing it, to reduce as much as possible the cost of the operation under investigation. The applications of this range from production systems, logistics, and as this thesis shows even non industrial situations. It is important to distinguish Scheduling theory from Queuing theory; the two are similar but whilst the second handles cases in which the amount of process to study are not finite, Scheduling theory studies environment in which the amount of "jobs", as will later be defined, is limited. To make a more comprehensive explanation, an example can be made:

- Queuing: A cafè is a great example of such a system. The amount of processes to handle (clients, orders, table seating...) is not known in advance, and thus the considerations to be made are based on statistics and averages, the distribution of the events occurrence can be used to predict the best way to run the system. We consider the task to perform as a flow of task, not a discrete amount.

- Scheduling: The OT considered in this thesis is a great example of this environment. The Tasks to perform are discrete in quantity, and the duration of each task is well defined in advance. We already know that a specific day of the schedule will contain a specific set of jobs, so no considerations are needed on the statistical distribution of the jobs duration (and the frequency of occurrence is not something imposed, rather something to decide).

It can be noted how the two systems are similar, and often it is possible to move from one approach to the other by varying some details. If for example, the OT scheduling considered in the example were to include the arrival of patients from the emergency room of the hospital, using Queuing theory would become the most effective way to approach the problem.
In the following section a detailed definition of the main components of a scheduling problem will be provided: Jobs, Machines, Parameters associated, Environment and Rules.

2.2.1 Formal definitions

We start from the classification of the Jobs. A job is a specific object that needs to undergo a series of tasks to complete the procedure in study. The machine is what

can perform the tasks that the jobs need to undergo. Before introducing a formal classification of the environment, several parameters must be defined.
Let's call **I** the set of tasks that can be performed on a job. It is not necessarily required that a job has to undergo all the operations inside **I**, but at least one operation has to be performed otherwise the job would be unprocessed. Let's also call **J** the set of jobs that will be processed in the environment. Ultimately we define as **M** the set of all machines in the environment.
I, J, M are discrete and finite sets.
In table 2.2 the variables and parameters useful for the formulation are defined.

Variable	**Definition**
$\mathbf{P}(i,j,m)$	Processing Time, the amount of time needed by machine m to perform task i on job j.
$\mathbf{r}(j)$	Release Time, if present, provides the moment job j can start to be processed by the system.
$\mathbf{d}(j)$	Due date, if present, provides the moment job j should be completely processed.
$\mathbf{C}(i,j)$	Completion Time, the moment task i of job j terminates being processed.
$\mathbf{B}(i,j)$	Begin Time, the moment task i of job j starts being executed. Unless preemption is allowed, $\mathbf{C} = \mathbf{B} + \mathbf{P}$.
$\mathbf{F}(j)$	Flow time of job j, defined as $C_l - \mathbf{r}$, with $\mathbf{C_l}$ being $\mathbf{C}$ of last task of j.
$\mathbf{L}(j)$	Lateness, defined as $\mathbf{C_l} - \mathbf{d}$, with $\mathbf{C_l}$ being $\mathbf{C}$ of last task of j.
$\mathbf{T}(j)$	Tardiness, defined as $\max(\mathbf{L}, 0)$.
$\mathbf{E}(j)$	Earliness, defined as $\max(-\mathbf{L}, 0)$.
$\mathbf{U}(j)$	Unit penalty, defined as 0 if $\mathbf{C_l} \leq d$, 1 otherwise, with $\mathbf{C_l}$ being $\mathbf{C}$ of last task of j.

Table 2.2: Parameters and variables in scheduling problems

The variables presented are not used in all problems, usually only a subset of the above are. It is also important to notice that while mostly coherent, the nomenclature does not seem to be standardized under a standardization agency, and thus may vary a bit. For a much deeper analysis on the subject i suggest the reading "Scheduling: Theory, Algorithms, and Systems" from Michael L. Pinedo (2012) [17].

2.2.2 Three-field classification

The general scheduling problem may be categorized according to the notation:

$$\alpha|\beta|\gamma$$

where:

$\boldsymbol{\alpha}$: Describes the machine environment under study. It is represented by a letter and a number using the format $\alpha_1\alpha_2$.
α_1 can be one among the following:

- $\emptyset$: single machine environment, when only one machine is available for the process.
- P: Identical Parallel machines, they can all perform the same operations at the same speed.
- Q: Uniform parallel machines, they can all perform the same operations but they may have different speed of execution.
- R: unrelated parallel machines, the machines are not identical, thus they each have a different speed and the jobs may take different times on different machines (without considering the speed of operation)
- O: Open shop, each job is composed of a set of tasks, each of which is assigned to a specific machine but without the constraint of an order for the execution for the tasks.
- F: Flow shop, each job is composed of a set of ordered tasks, each assigned to a specific machine in the correct order, all identical for all the jobs.
- J: Job shop, each job is composed of a set of ordered tasks, each assigned to a specific machine in the correct order, and different jobs may have different tasks and order for the tasks.

α_2 represents the number of machines in the environment, so it is a positive integer.

$\boldsymbol{\beta}$: Describes the characteristics of the jobs, it is composed by $\{\beta_1, \beta_2, \beta_3\}$, where all three elements can be $\emptyset$ to imply no additional information is needed. If however the elements are not all $\emptyset$ we have that:

- If $\beta_1 = pmtn$ we have that preemption is a possibility, namely we can interrupt a job's processing to resume in a second moment.
- If $\beta_2 = prec$ a precedence relation between the jobs (not the tasks of the jobs necessarily) may be established.

- If $\beta_3 = r_j$ Different jobs may have different release times.

γ : represents the Optimality Criterion, and tells which is the function to be optimized, either maximized or minimized. This function is a combination of the parameters explained in 2.2, and the relationship between these parameters implies whether the problem can be treated as linear, quadratic, or else. One interesting property to note is connected to what will be discussed in the problems characterization. The following definitions and considerations are explained in detail by A. Sprecher et al. in the paper "Semi-active, active, and non-delay schedules for the resource-constrained project scheduling problem" (1995) [18].

Definition 7 (Regular performance measure)**.** *An Optimality criterion is said to be Regular if it is a non-decreasing function of the jobs completion times* $\boldsymbol{C}$*.*

Definition 8 (Semi-Active schedule)**.** *A schedule is called Semi-Active if no operation can be anticipated without the need of delaying other operation or violating some constraint.*

It can be proven that for all problem with regular performance metrics there exists at least one optimal semi-active solution.

Important variation

Briefly discussed by Stéphane Dauzère-Pérès et al. in "The flexible job shop scheduling problem: A review" (2023)[19], one important peculiarity for the purpose of this thesis is the Flexible Job Shop Problem. This variation makes it so that each operation of a job can be performed only on a subset of the available machines, which is the correct way of describing the fact that in the problem approached in this thesis the patient may need to go back to an already visited room, the AOP, when the AW is scheduled to occur there (thus making the graph of the process cyclic, which is atypical in scheduling). It is relevant to note that the therm "Flexible" is not specific to this variation, but is used more as a gateway when a scheduling problem slightly differs from a standard known one.

2.3 Linear programming

"Mathematical programming is one of the most important techniques available for quantitative decision-making. The general purpose of mathematical programming is finding an optimal solution for allocation of limited resources to perform competing activities" (C.Chandra et al., 2016) [20]. Linear programming is the unfold of mathematical programming when only Linear functions are used for the various components of the program. All must be expressed in linear form, either directly or through linearization processes starting from previously nonlinear forms. The following section is inspired by the book "Scheduling: Theory, Algorithms, and Systems" from and Michael L. Pinedo [17]

2.3.1 General model

The general structure of a linear programming model is composed by:

- **Decision variables**: The mathematical representation of the quantities in play for the problem. They can be continuous, integers, or boolean. Continuous variables lead to easier problems, whereas integer and boolean usually imply an increase in model complexity (and the problems associated with these kind of variables are specifically called Integer-Linear-Programming, ILP). Among the DV there may be variables that do not directly represent a quantity of the problem and that are not strictly decisional, but only of support for other variables, usually necessary in the process of linearization. The set containing all the possible configurations of the DVs is the domain of the problem; representing all the possible states the system can take. Usually the variables are bounded to be positive, since a negative variable can be obtained through difference of positive variables; this allows to reduce the Domain of the problem imposing a lower bound to the variables.
- **Objective**: The main purpose of the model, the quantity that has to be optimized. To be optimized means that it has to be either minimized or maximized. To have an LP model this quantity must be linearly proportional to the previously defined DVs. The OF can too be continuous or discrete, and the implications on any of the two possibilities reflects directly in the methodologies that can be applied, as will be discussed later.
- **Constraints**: The rules relating the DVs. These represent the conditions that characterize the problem, that translates into a reduction of the dimension of the Domain of the model. The constraints must also be expressed in linear form with respect to the DV. Constraints have also indirect effect on the OF. They are written either as equalities or inequalities, with the latter being less strict than the first. As long as a solution, i.e. a set of unique values for the DVs, applied to the constraints does not generate false statements (unverified equalities/inequalities), then the solution proposed is a feasible solution for the problem.

2.3.2 Linearization techniques

The following section is inspired by the description provided by Gass and Saul I. in "Linear Programming: Methods and Applications" (2010) [21].

Max-min constraint

The Max-min methodology is useful for linearizing the condition in which the objective function (or other variables) requires the maximization of the minimum element in a set of variables. Formally we write:

$$\max(\min(\mathbf{v_1},\, \mathbf{v_2}, \dots,\, \mathbf{v_n}))$$

with $\{\mathbf{v_1},\, \mathbf{v_2}, \dots,\, \mathbf{v_n}\}$ subject to the already structured set of constraints of the problem.
This expression is not linear in the variables, and thus cannot be directly used in a LP formulation. To avoid this problem we introduce a support variable Y, not necessarily of the same type of the DV (continuous, integer or boolean), and we rewrite the problem in the following way:

$$\max(\mathbf{Y})$$

subject to all the constraint of the problem, in addition to the following constraints:

$$\begin{gathered}\mathbf{Y} \leq \mathbf{v_1} \\ \mathbf{Y} \leq \mathbf{v_2} \\ \dots \\ \mathbf{Y} \leq \mathbf{v_n}\end{gathered}$$

This formulation is now linear and can be used in a LP problem.

Min-max constraint

The Min-max methodology is the counterpart of the Max-min, aiming at minimizing the maximum between a set of variables. Formally we write:

$$\min(\max(\mathbf{v_1},\, \mathbf{v_2}, \dots,\, \mathbf{v_n}))$$

with $\{\mathbf{v_1},\, \mathbf{v_2}, \dots,\, \mathbf{v_n}\}$ subject to the already structured set of constraints of the problem.
This expression is again not linear in the variables, and cannot be directly used in a LP formulation. Similarly to the previous case, we introduce a support variable Y, not necessarily of the same type of the DV (continuous, integer or boolean), and

we rewrite the problem in the following way:

$$\min(\mathbf{Y})$$

subject to all the constraint of the problem, in addition to the following constraints:

$$\begin{gathered}\mathbf{Y} \geq \mathbf{v_1}\\ \mathbf{Y} \geq \mathbf{v_2}\\ \dots\\ \mathbf{Y} \geq \mathbf{v_n}\end{gathered}$$

This formulation is now linear and can be used in a LP problem.

Min-abs constraint

Similar to the previous two methodologies, the Min-abs techniques allows to minimize the module of a variable (or of a set of variables, by applying it more than once to different variables). The non linear formulation of the model is:

$$\min |\mathbf{v_i}|$$

with $\mathbf{v_i}$ subject to the constraints of the problem.
By noticing that:

$$|\mathbf{e}| = \max(\mathbf{e}, -\mathbf{e})$$

the problem can be translated in another nonlinear form, which however is already known to have a linear representation, the $\min(\max(v))$ formulation. After introducing the support variable Y, the linearized problem thus becomes:

$$\min(\mathbf{Y})$$

subject to:

$$\begin{gathered}\mathbf{Y} \geq \mathbf{v_i}\\ \mathbf{Y} \geq -\mathbf{v_i}\end{gathered}$$

The problem is once again linear in the DV and the formulation can be used for a LP.

Big-M constraint

The Big-M constraint has the purpose of creating constraints in which a continuous variable and a boolean one interact (sometimes integers in general, but less common).

The idea is that the boolean variable should "enable" the continuous variable to be different from zero, and this is used to bound the continuous variable according to yes/no choices. Say there is a variable $\mathbf{Y} \in \{0, 1\}$ and $\mathbf{x} \in \mathbb{R}^+$, the nonlinear definition of Y would be;

$$Y = \begin{cases} 1 & \text{if } \mathbf{x} > 0 \\ 0 & \text{otherwise} \end{cases}$$

The above may be intended as "Y follows x", but it is important to remember that in LP the variables are not one following the other, it's just a combination of the variables that happens to be feasible. If any of the two were to be constrained to a specific value, then the other would be constrained as well.
Expressing the above formulation in linear form we may write:

$$\mathbf{x} \leq \mathbf{M} \cdot \mathbf{Y}$$

with M being a constant positive parameter big enough to comprehend all the cases for which the constraint may be called. It is important to notice that using just an absurdly big value for M usually leads to slow execution in commercial solvers, thus why it is important to properly chose it.
As can be seen in table 2.3, the above constraint ensures that when the binary variable is set to 0, the continuous variable can be the result of a feasible solution if and only if it is also 0.

	Y=0	**Y**=1
x=0	Ok	Ok
x> 0	X	Ok

Table 2.3: Big-M constraint feasibility conditions

Sometimes it is also necessary to ensure that when Y=1, $x \neq 0$, and this is obtained with an additional constraint to the Big-M:

$$\mathbf{m} \cdot \mathbf{y} \leq \mathbf{x}$$

with m being a sufficiently small value compared to x. This additional constraint is less common than the Big-M but was added for completeness.

Disjunctive constraint

The disjunctive constraint is a linearization technique whose purpose is to formalize the constraints that express either one or another scenario. Informally we characterize the Domain of the problem to be constrained either in one way, or in another, without fixing a priori the decision. To make a more comprehensible description, we may look at an example as follows, where $\mathbf{C_i}$ represents the completion time of a job, and $\mathbf{p_i}$ represent the processing time of the same job.

$$\mathbf{C_i - p_i \geq C_j \vee C_j - p_j \geq C_i}$$

This constraint translates to "Either job **i** starts after job **j** has finished, or job **j** starts after job **i** is."
This is a very important constraint in LP but at the moment the formulation is not linear. To change this we introduce a support variable $\mathbf{Y} \in \{0, 1\}$ that represents the two choices. Informally we want to have so that if $\mathbf{Y}$=0, one of the two situations must happen, vice versa if $\mathbf{Y}$=1 the other one must occur.
Other than the support variable we introduce a Big-M like constraint, with a sufficiently large $\mathbf{M}$, for the purpose of splitting the constraints into two dependent but different ones. The formulation can now follow:

$$\begin{gathered}\mathbf{C_i - p_i \geq C_j - M \cdot Y} \\ \mathbf{C_j - p_j \geq C_i - M \cdot (1 - Y)}\end{gathered}$$

With this technique it is ensured that, depending on the value of $\mathbf{Y}$, only one of the two constraints works on reducing the boundary for the Domain of the model. The constraint is now linear and can be used in a LP formulation.

The previous methodologies can be explored in more detail in the work of I. Grossman "Review of Nonlinear Mixed-Integer and Disjunctive Programming Techniques" (2002) [22].

2.4 Objective Function: LP vs MILP considerations

This section aims at giving a better understanding on why the kind of problems tacked by this thesis are not as easy as they may seem. In the context of LP it has already been expressed that the OF must be a linear combination of the DVs.
When the domain of the model, so the values that the DVs can take in a feasible condition, is a continuous space (so all the elements of the domain are individually continuous), it can be easily proved that the range of the OF is also a continuous set. This consideration holds for both scalar and vectorial OFs, but in the context of the problem under investigation scalar values are enough. The continuity property does not usually hold when among the DVs there are discrete valued variables, which is the definition of a MILP.
In particular, it is trivial to prove that a linear combination of only discrete variables

leads to a discrete OF, and this follows an important consideration; since the analysis is not performed on a continuous domain, Calculus theory cannot be applied to the problem directly. It is in fact true that Calculus regards the study of infinitesimally small change, another (improper) way of describing continuous functions. When this property is not present anymore all the methodologies that are usually applied to LP (Convex optimization, Gradient descent,...) are not applicable in a direct way.

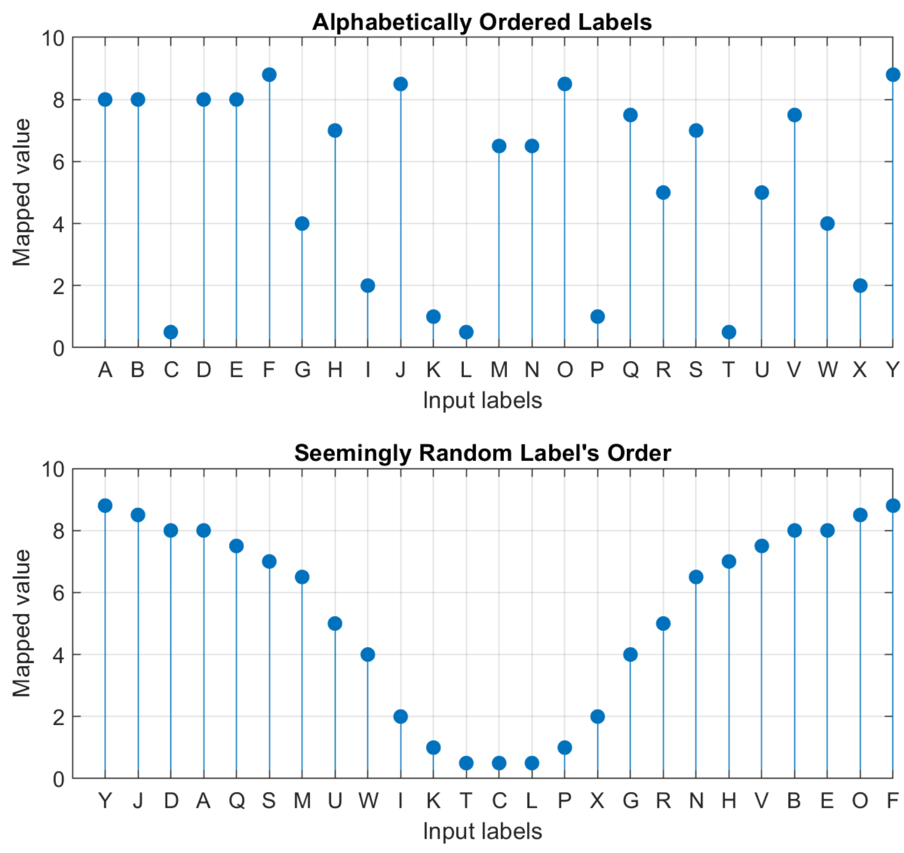

Figure 2.1: Example of different trends obtained from different representation of the same Map.

Furthermore, the nature of the problem under consideration provides a discrete OF with discrete DVs that do not follow an intrinsic ordering, as in the case of continuous variables. This implies that it is not possible to write or draw the OF in explicit form, since the ordering along the abscissa can be chosen in a non unique way. This consideration is a direct consequence of the domain of the problem having a combinatorial nature instead of the classic continuous one.
Image 2.1 shows the different shapes that a Map can take when the ordering of the abscissa is not properly determined, highlighting the advantage of using a certain ordering rule against another. The focus here is on the fact that when arranging the

abscissa in alphabetical order of the labels no trend is visually recognizable. On the other hand, visualizing the same data in a different arrangement, seemingly random, uncovers a specific local behavior of the map. The example was created ad-hoc for the purpose of visualizing and emphasize the concept previously described.
The topic will be further discussed in future sections; the challenge of finding a good representation for a specific solution to the problems will require the design of new methods aimed at linking similar/close solutions. These similar solutions are what later will be defined as Neighboring, an important concept in Algorithms for Optimization problems. P. Hanses et al. provided a comprehensive analysis on the definition of a neighborhood for combinatorial problems in his Paper "Variable neighborhood search: Principles and applications"[23].

Chapter 3

Problems definition

3.1 AOPSP

The Anesthesia/Observation Place Scheduling Problem regards only the decision of where to make the AW of each patient occur. The set of patient to process and the respective duration of each procedure is deterministic and known a priori, together with the ordering that the patients will follow for the procedure. Since the various processes for different patients may overlap in time (there are two rooms) the order will be defined as: Patient **i** comes before patient **j** if the beginning of A of patient **i** occurs before the beginning of A of patient **j**.
This definition is used since if similar rules were applied also on S and AW, the ordering would be trivially of one patient at the time in the system, which is feasible but the worst possible semi-active schedule
The A of each patient will naturally occur in the AOP, and the S will be performed in the OR immediately after the completion of the A. This part of the patient schedule has no decision to perform, A and S can be seen as a single process occurring in two rooms since the relative position is fixed by the problem constraints, but is crucial in determining the time available for the AW. To decide where the AW will occur it is necessary to understand how the block building will be performed, to understand where the AW can fit or not, to avoid idle time.
It is important to notice that the decision taken for each patient is independent of the placement of the others from feasibility prospective. Even though an optimal schedule will intuitively have lots of fitting in between idle time of patients, any combination of placements will result in feasibility. This is because, even if a bad placement (more on what makes a placement bad later) is requested, the patient can just slide in front moving the whole schedule ahead without breaking the ordering constraint and the positioning of any other patient. It is a direct consequence that all possible solutions of the problem are feasible. In picture 3.1 several placements are displayed for the same order of patients, to show the rearrangements the schedule undergoes to keep the feasibility of each possible solution.

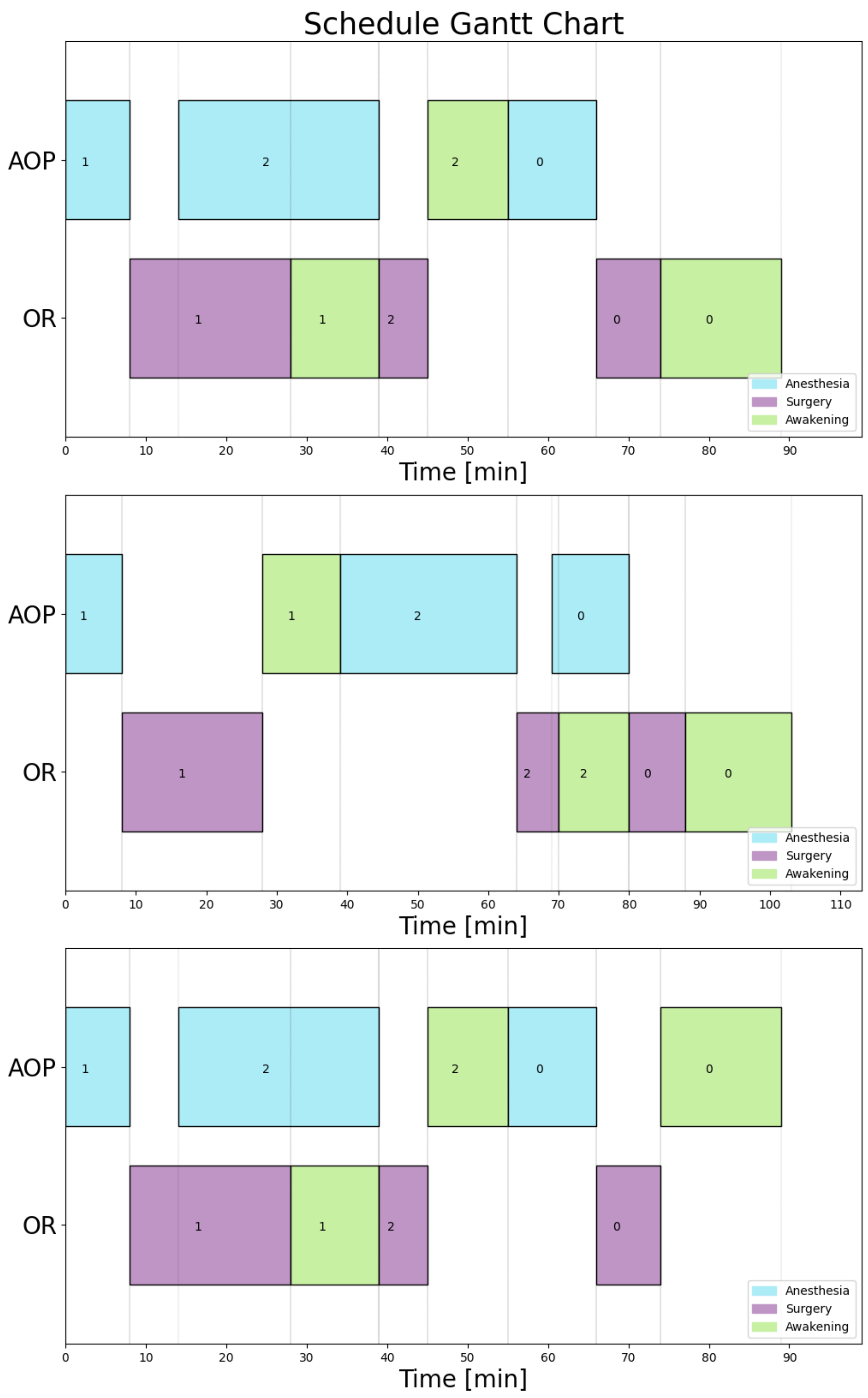

Figure 3.1: Several schedules for the same set of 3 patients processed in the same order. All feasible at cost of additional idle time.

3.1.1 Classification: machines and jobs characteristics

We will call **N** the number of patients to be scheduled.
The machines for this problem are the rooms the patients go in. Each room can only process one patient at the time, and perform a subset of the tasks that the patients must undergo. For the AOP the subset is $\{A, AW\}$, whereas for the OR it is $\{S, AW\}$. The ordering among different tasks is given in the form of a precedence constraint, with Anesthesia-Surgery-Awakening taking place in this order only. No preemption across different machines can occur, so once the AW of a patient starts in a room it cannot be interrupted to continue in another machine or in another time.
The jobs, here the patients, are bound by a precedence constraint dictated by the imposed ordering, have no release times but must follow a no-wait scenario, meaning that no time can pass between operations. Obviously no two tasks of the same patient can overlap in time. It is relevant to notice that technically the precedence is not a constraint, but rather part of the problem structure. By considering the Completion times as DVs such a requirement would become a constraint bounding the Domain space. However, as will be more clear in the following sections, the optimal choice for DVs will be the placement. In fact, when only *semi-active schedules* are considered, the Completion times are uniquely identified from the placements. As a consequence the imposed ordering just translates the more general problem, later defined, into a variation of a two identical parallel machines assignation problem. For each patient, one simply needs to decide in which room the AW occurs, and then the schedule can be constructed accordingly.
This considerations make the problem not canonical, since it cannot be described by the categorization explained in section 2.2.2. Indeed the presence of A and S before AW (linked by the no-wait) occupies the machines for time where no AW is occurring. In other words the placement of a patient stalls the machines for the amount of times necessary to process the accessory tasks A and S.
Considering all of the above, and accounting for what explained in section 2.2.2, the problem may be denoted to as a Flexible variation of the:

$$\mathbf{P2}|\{\emptyset, \mathbf{prec}, \mathbf{r}*\}|\max(\mathbf{C_{i,j}})$$

where $\mathbf{C_{i,j}}$ refers to the completion time of the task inside the system, making $\max(\mathbf{C_{i,j}})$ the make span of the schedule. The element"$\boldsymbol{r^*}$" refers to a presence of release times that dynamically change with the choices made by other patients.

3.1.2 Complexity and instances dimension

Since the only decision to take in this problem regards the placement of each patient's AW, and since the choices are independent, the amount of feasible solutions is equal to the combinations obtainable with a set of N binary variables. Calling **CPX** the dimension of the ID of the problem, we have that:

$$\mathbf{CPX_{AOPSP} = 2^N}$$

The complexity of the problem increases quickly due to the exponential law, yet remains readily manageable by the context. In fact, in a general facility, instances involving 15 patients already surpasses average utilization of a working day. Even a worst case in which there are 19 patients to schedule the total amount of cases to explore would be in the order of $\approx 5 \cdot 10^5$, which is easily manageable for modern day computing machines.

3.1.3 Similarity and differences with known problems

The AOPSP has a very deep connection with a famous problem in combinatorial optimization: the Knapsack problem (KP). Briefly, KP regards finding the optimal way of allocating resources inside a finite availability container, or in the variation of multiple sacks minimizing the number of containers used (Bin Packaging Problem, BPP). The criteria for this request is visualizing how much does a single resource fill the container (whether in the form of volume occupied w.r.t. the volume of the container, the cost of the resource with respect to the budget, and so on). The "filling" property is usually referred to as Weight of the resource and the containing capability as the Limit of the container. A deeper overview on the topic can be found on the paper "Where are the hard knapsack problems?" by David Pinsinger (2005) [24].
The similarity with the AOPSP comes from the fact that, locally, each patient must be placed between the previous and the following patients. In therms of the KP, we have that the length of the task under investigation (either A, S or AW depending on the situation) can be seen as the Weight. Consequently the slot of time available, whether it is an unconstrained gap (front of the schedule) or a gap constrained by two other tasks, is the Limit. This parallelism has been very useful in the process of finding an algorithm for solving the problem. Remembering that a patient can be awakened in any room without the risk of compromising the feasibility of the solution, there are several differences to highlight:

- The patient has a weight given by the length of the task, however each task is locally fixed in time with the other two of the patient. Finding a slot big enough to fit the investigated task may not be enough for solving the placement decision. (the solution would still be feasible if we just consider the placements as DV, but the resulting schedule would have patients shifted to the right with considerable amount of idle time)

- The tasks are meant to be fit in two different rooms without the possibility of distancing them in time, so the problem becomes more similar to the Multidimensional 0-1 Knapsack, a variation in which both the Weight and the Limit are multidimensional. The topic is investigated by Arnaud Fréville (2004) in his work "The multidimensional 0–1 knapsack problem: An overview" [25]. With this similarity the cardinality would equal the amount of rooms available, but the no-wait constraint makes the AOPSP different enough to require a completely different approach from that occurring in the mentioned study.
- Whilst it might be intuitive that maximizing the Weight within the Limit leads to better the solutions thanks to the removal of idle time from the schedule, the problem has a peculiar behavior when it comes to the issue; while in the KP positioning a resource in the knapsack does not influence the system other than removing availability, in the AOPSP the choice on a patient, while not conditioning the feasibility of other choices, leads to a perturbation of the time gaps present in a considerably large neighborhood of the patient. This makes it so that the optimal choice for positioning a patient cannot be taken locally, but only globally. The topic will be further discussed in following sections.
- The fact that the available time depends on the choices made on other patients makes it so that the KP similarity refers more to the localized choice of a patient. Each one is the resource that has to fit in a knapsack whose Limit is decided by the choices performed by the neighborhood patients. Each choice is taken creating new boundary conditions for the individual KPs generated.

3.2 IAOPORSP

The Integrated AOP and OR Scheduling Problem has a more complicated structure compared to the previous problem. Whilst the environment and the jobs are almost the same, not having a fixed order increases the amount of choices to perform significantly.
Relieving the imposed order constraint makes the IAOPORSP a relaxation of the AOPSP, but the problem completely changes in prospective. In optimization theory relaxations are used for simplifying the solution search of a problem, increasing the

problem domain by neglecting certain constraints. In this case, relaxing the ordering constraint, the dimension of the Domain explodes, and since all solutions of the AOPSP are feasible, no improvement is made on the ease of solving.
Later will be discussed that the IAOPORSP also has the property that all the correctly formulated solutions (using the ordering and the placements for DVs) are feasible.
Once again the duration of the procedures is deterministic and known a priori. Considering the order of execution among the DVs, the same rule used for the AOPSP for determining the precedence will be used: Patient **i** is scheduled before patient **j** if the beginning of the A of patient **i** occurs before the beginning of the A of patient **j**, with no constraint on S and AW. Once again no time can pass between the single tasks of a patient, and a room cannot process more than one patient simultaneously. The A of the patients must be performed in the AOP room, the S in the OR room, and the AW in any of the two room, with no possibility of interruptions/switching rooms mid-task.
It is again relevant to notice that only *semi-active schedules* will be considered for solutions, as an infinite amount of solutions exist if we consider schedules with idle time before a patient.

3.2.1 Classification: machines and jobs characteristics

All the considerations on the machine environment made for the AOPSP hold for the IAOPORSP as well (3.1.1). The jobs must not handle a precedence constraint, and no release time is considered globally, since any job can be placed in any position. If a focus on the local schedule of a specific patient is observed, one might argue that the processing of auxiliary tasks from other patients (S and AW) may be seen as a release time, but such an approach would be unprofitable for the purpose of the solution search.
Following the considerations made for the AOPSP, the problem might be classified as a Sequence-Dependent Setup variation of the:

$$\mathbf{P2}|\{\emptyset,\emptyset,\emptyset\}|\max(\mathbf{C_{i,j}})$$

The "Sequence-Dependent Setup" refers to the fact that each room will not be able to accept new patients until the time required to process A and S of other patients expires. This amount of time is not easily determined and is consequence of the choices made on a large neighborhood of the patient under consideration.

3.2.2 Complexity and instances dimension

The placement part of the problem may be tackled as in the AOPSP variant, and leads to 2^N possible combination of placements, where N is the number of patients to schedule. The ordering part is exactly the problem of finding how many arrangements can be created using N object without replacement. The observation comes naturally from imagining that the first patient may be any of the **N** positions, the second any of the remaining (**N**-1), and so on. The total number of possible combinations for the ordering of patients is then ***N*!**.
Since the ordering and placement are independent choices, the amount of possible schedules can be obtained by multiplying the two numbers together, obtaining the complexity of the problem:

$$\mathbf{CPX_{IAOPORSP}} = \mathbf{2^N} \cdot \mathrm{\mathbf{N}}!$$

The above function is always greater in value than the one for the AOPSP, since ***N*!** is always greater than one except for the trivial case of one single patient. Once **N** grows over 1, difference in combinations between the two variants of the problem is so big that plotting it would be useless for the purpose of comparing.
To give an idea of how much does this function rise rapidly, it is sufficient to notice that for the AOPSP a number of 19 patients generated a domain dimension of $\approx 5 \cdot 10^5$ (3.1.2); such number is reached, and surpassed, with only 7 patients by the IAOPORSP domain.
Since the context of the problem is the same discussed for the AOPSP, an average of 15 patients for instance will be considered also for this variation, with peaks of 19 patients for testing purposes. With 19 patients the number of possible combinations to explore is around $\approx 6.38 \cdot 10^{22}$, a number that far exceeds the processing capability of modern day processors.

Trivia: Under the absurd assumption that each combination would require only 50 Floating Point Operations to be tested, the fastest computer available as of 2023 (OLCF-5, Oak Ridge Leadership Computing Facility [26]) would require (considering constant peak speeds of 1.1 exaflops) approximately one month to test all the combinations for 19 patients. To do so, the energy consumption (considering the 22.7 MW declared) to perform such a process would equal the energy released by the nuclear bomb "Little Boy".

3.2.3 Similarity and differences with known problems

The problem has several similarities with many different known problems. Whilst none present the exact same structure of the one under investigation, many of them lead to a better understanding of the approach necessary to solve the issue. A brief description of the most relevant ones follows.

Wedding Seating Arrangement Problem (WSAP)

The problem under discussion regards the placements of **n** guests in **n** seats. Usually the seats are divided into groups (the tables), and the weight of the decision is exploited in the preference of each guest with respect to the preference for other guests at the table. The constraints given by the preference can be expressed either by binary absolute preference (guest a can or can't sit at the same table of guest b) or in the form of preference range (guest **i** would like to seat near guest **j** with a score out of 10, to describe the preference level). Both approaches can be used at the same time and the final objective will be the total amount of satisfaction of the guest.
The WSAP can be interpreted as a variation of the BPP, where the tables represent the sacks to fill. The difference relies in the fact that the Weights of the guests are not static but depend on the other guests placements. A good parallelism with the IAOPORPS can be made: The preference a guest expresses towards another one can be interpreted as how well do two patients fit in the schedule, which depends on the placements of the other patients in the schedule.
Solutions for the WSAP problem are not, as of today, expressed in closed form. Only heuristic solutions have been proposed up to now. Rhyd Lewis and F. Carroll explored in detail the problem in their paper "Creating seating plans: a practical application"[27]. They also present the results obtained by the algorithm implemented on a commercial website [28], whose functioning is unfortunately not provided but that shows very good results compared to other approaches presented.

Traveling-Salesman Problem, TSP

One of the most famous and discussed problem in Combinatorial Optimization, the TSP studies the possible ways of ordering the **n** cities of a map with the purpose of minimizing the distance traveled. The cost of going from a city to another is usually expressed as the distance between two cities, and so each city has a specific cost associated to each other one in the map. The just mentioned set of costs is usually expressed in a $\mathbf{n} \times \mathbf{n}$ matrix. The task of finding the best itinerary depends only on

the order chosen for the trip, the DVs to consider are the same that are used for the ordering part of the IAOPORSP. The Weight associated to a travel is represented in the IAOPORSP by how well do two patients fit together in the schedule, and here lies the first difference between the two problems. As earlier discussed the problem has a sequence-dependent nature for the patients positioning, which can be interpreted by the TSP as if the Weight of going from city **i** to city **j** would depend on what cities were visited before city **i**. Moreover the possibility of choosing where to place the AW of a patient adds another layer of complexity to the problem, not considered in the TSP, which may be interpreted as if the travel from one city to another could be made using different methods that could also influence the weight associated to the travel to the previous and following cities.
Currently the TSP is considered to be an NP-hard problem, so no algorithm has been found capable of solving the problem in polynomial time. Whilst not analyzed further, the fact that the IAOPORSP has several layers of complexity added on top of the TSP, together with considerations made during the study, leads to believing that the problem is also NP-hard, but further study would be needed to rigorously prove this statement.
The TSP has been studied by several papers over the years, the one by M. M. Flood "The Traveling-Salesman Problem"[29] offers a good overview on the topic.

Unrelated parallel machine scheduling with past-sequence-dependent setup time

The problem is presented by Chou-Jung Hsu et al. (2011) in "Unrelated parallel machine scheduling with past-sequence-dependent setup time and learning effects" [30]. The discussion is similar to that of the problem under study in this thesis, and offered a great overview on the variation related to the sequence dependent setup time and the consequent approaches. Unfortunately the setup considered in the paper is only proportional to the length of the already processed jobs, while the one considered for the IAOPORSP is not linearly dependent on such a factor, with a more unpredictable behavior.
Relevant to notice that the mentioned paper found a polynomial time algorithm for the solution.

Chapter 4

MILP modeling

4.1 Preface

Before explaining the models of the two problems, there are a couple of important considerations to make. The problems require the minimization of the Makespan, and for a human being it is trivial to consider that, regardless of the schedule chosen, the patients should start as soon as possible. It is intuitively clear that adjusting the entire schedule by a set duration, such as ten minutes, would directly increase the Objective Function (OF) by the same increment. However, this straightforward insight does not extend to the solver's operational logic. Given the combinatorial complexity of the problem at hand, Gurobi examines an extensive array of potential solutions. During this exploration, when it identifies a solution that currently offers the best OF, it does not evaluate whether removing idle time at the beginning could further enhance the objective (while maintaining the same solution conceptually). The topic under discussion is what has been previously defined as having solutions that are not *Semi-Active* (8). This oversight is naturally rectified if the solution search is allowed to reach completion. However, should the solver's runtime be excessively prolonged and consequently interrupted, inefficiencies would arise. This wasted times, though, can be effectively mitigated through post-processing of the Gurobi solution.
Another important remark is that the amount of decision variables of a model doesn't reflect the problem complexity. As shown before the complexity for the AOPSP problem is exponential with the number of patients to schedule, and having many more variables is not a sign of increased complexity of the problem, but rather of the algorithm, as previously discussed in chapter 2.1. This implies an increased amount of variables and constraints, that may reflect into a heavier or lighter computational load for the solver, reason for which several models for the IAOPORSP have been considered.

4.2 Time-based model for the AOPSP

4.2.1 DVs and parameters

The *Anestesia/Observation Place Scheduling Problem* imposes a priori the order in which the patients must be processed. It follows that the only decision to take is the placement of the AW of each instance. Since the order is already determined the Makespan of the schedule is easily obtainable as the completion of the last patient of the provided ordering. Calling **N** the number of patients to schedule, the following DVs have been chosen:

$$\mathbf{C}[i,j] \in \mathbb{Z}^{+} \tag{4.1}$$

$$\mathbf{OR}[j] \in \{0,1\} \tag{4.2}$$

with

$$\mathbf{i} \in \{1,2,3\}$$
$$\mathbf{j} \in \{1,2,\dots,\mathbf{N}\}$$

Variable **C** represents the completion time of task **i** for patient **j**. Variable **OR** assumes the value of 1 if the j-th patient AW occurs in the OR. It is important to notice that even if $\mathbf{C} \in (\mathbf{3} \times \mathbf{N})$, the complexity of the model does not follow the dimension of **C** as the values of the Completion times are intrinsically consequent from the values of **OR**, as previously stated in section 4.1.
For the parameters of the model, since a BIG-M constraint will be used and considering the dimension of the problem, a value of

$$\mathbf{M} = 3000 \tag{4.3}$$

has been set, considering that a value equal to the sum of all processing times of all tasks is sufficient to guarantee the correctness of the constraint formulation without wasting computational power with uselessly large values. The processing times of the single tasks of each patient follow the same structure of the completion times as in:

$$\mathbf{P}[i,j] \in \mathbb{Z}^{+} \tag{4.4}$$

4.2.2 Model formulation

Minimize:

$$\mathbf{C}[3,\mathrm{N}]$$

Subject to:

$$
\begin{aligned}
&\mathbf{C}[1,1] - \mathbf{P}[1,1] = 0 & & \quad (4.5)\\
&\mathbf{C}[2,j] - \mathbf{P}[2,j] = \mathbf{C}[1,j] & \forall j & \quad (4.6)\\
&\mathbf{C}[3,j] - \mathbf{P}[3,j] = \mathbf{C}[2,j] & \forall j & \quad (4.7)\\
&\mathbf{C}[1,j+1] - \mathbf{P}[1,j+1] \geq \mathbf{C}[1,j] & \forall j \in 1,\dots,N-1 & \quad (4.8)\\
&\mathbf{C}[3,j+1] - \mathbf{P}[3,j+1] \geq \mathbf{C}[3,j] - \mathbf{P}[3,j] & \forall j \in 1,\dots,N-1 & \quad (4.9)\\
&\mathbf{C}[2,j+1] - \mathbf{P}[2,j+1] \geq \mathbf{C}[2,j] + \mathbf{P}[3,j] \cdot \mathbf{OR}[j] & \forall j \in 1,\dots,N-1 & \quad (4.10)\\
&\mathbf{C}[1,j+2] - \mathbf{P}[1,j+2] \geq \mathbf{C}[3,j] - \mathbf{M} \cdot \mathbf{OR}[j] & \forall j \in 1,\dots,N-2 & \quad (4.11)\\
&\mathbf{C}[3,j] - \mathbf{P}[3,j] \geq \mathbf{C}[1,j+1] - \mathbf{M} \cdot \mathbf{OR}[j] & \forall j \in 1,\dots,N-1 & \quad (4.12)
\end{aligned}
$$

Constraint 4.5: Fixes the beginning of the first job at t=0, actively avoiding idle time before starting the schedule.

Constraint 4.6: Imposes the No-wait constraint between A and S of each patient.

Constraint 4.7: Imposes the No-wait constraint between S and AW of each patient.

Constraint 4.8: Ensures the ordering by imposing that A of a patient starts after the start of another one's A.

Constraint 4.9: Ensures the ordering by imposing that AW of a patient starts after the start of previous patient's AW unless the current awakens in the OR.

Constraint 4.10: Ensures no overlapping between the S of a patient and the S of the previous, or in case the previous awakens in the OR, ensures no overlapping between patient's S and previous AW.

Constraint 4.11: Ensures no overlapping between A of a patient and the Aw of the patient two places behind in the schedule, when the current awakens in the AOP.

Constraint 4.12: Ensures no overlapping between AW of a patient and the A of the following one, when the AW of current patient occurs in the AOP.

4.2.3 Model overview

The model described exhibits a notably low computational complexity, enabling Gurobi to solve instances of significantly larger scale than those targeted in this thesis work with near-instantaneous execution to optimality. This efficiency has obviated the need for further exploration of alternative models for the AOPSP.

4.3 Time-based model for the IAOPORSP

4.3.1 DVs and parameters

Since the *Integrated Anestesia/Observation Place and Operating Room Scheduling Problem* also requires the decision on the position of the patients, the complexity of the model will inevitably rise. This requires the need of additional variables with respect to the previous model. The chosen decision variables are:

$$\mathbf{C}[i,j] \in \mathbb{Z}^+ \tag{4.13}$$

$$\mathbf{OR}[j] \in \{0,1\} \tag{4.14}$$

$$\mathbf{prec}[i1,j1,i2,j2] \in \{0,1\} \tag{4.15}$$

$$\mathbf{Max\_completion} \in \mathbb{Z}^+ \tag{4.16}$$

with

$$\mathbf{i,\ i1,\ i2} \in \{1,2,3\}$$
$$\mathbf{j,\ j1,\ j2} \in \{1,2,\ldots,N\}$$

To model the possibility of changing the order among the patients the DV 4.15 is used. It is a binary variable that assumes the value of 1 in case that task **i1** of patient **j1** starts before task **i2** of patient **j2**. For the parameters of the model the same considerations of the previous one can be made (4.2.1), and we consider:

$$\mathbf{M} = 3000 \tag{4.17}$$

$$\mathbf{P}[i,j] \in \mathbb{Z}^+ \tag{4.18}$$

Since now it is not know a priori who will the last patient be after the optimization, a support variable to identify the last completion of the schedule is needed (4.16), and it will be characterized by the constraint of the model.

4.3.2 Model formulation

Minimize:

$$\mathbf{Max\_completion}$$

Subject to:

$$\mathbf{Max\_completion} \geq \mathbf{C}[3,j] \qquad \forall j \qquad (4.19)$$

$$\mathbf{C}[1,j] - \mathbf{P}[1,j] \geq 0 \qquad \forall j \qquad (4.20)$$

$$\mathbf{C}[2,j] - \mathbf{P}[2,j] - \mathbf{C}[1,j] = 0 \qquad \forall j \qquad (4.21)$$

$$\mathbf{C}[3,j] - \mathbf{P}[3,j] - \mathbf{C}[2,j] = 0 \qquad \forall j \qquad (4.22)$$

$$\mathbf{C}[2,j1] - \mathbf{P}[2,j1] - \mathbf{C}[2,j2] + \mathbf{M} \cdot \mathbf{prec}[2,j1,2,j2] \geq 0 \qquad \forall j1,j2|j1 \neq j2 \qquad (4.23)$$

$$\mathbf{C}[2,j1] - \mathbf{P}[2,j1] - \mathbf{C}[2,j2] + \mathbf{M} \cdot \mathbf{prec}[2,j1,3,j2] + \\ + \mathbf{M} \cdot (1 - \mathbf{OR}[j2]) \geq 0 \qquad \forall j1,j2|j1 \neq j2 \qquad (4.24)$$

$$\mathbf{C}[3,j1] - \mathbf{P}[3,j1] - \mathbf{C}[2,j2] + \mathbf{M} \cdot \mathbf{prec}[3,j1,2,j2] + \\ + \mathbf{M} \cdot (1 - \mathbf{OR}[j1]) \geq 0 \qquad \forall j1,j2|j1 \neq j2 \qquad (4.25)$$

$$\mathbf{C}[1,j1] - \mathbf{P}[1,j1] - \mathbf{C}[1,j2] + \mathbf{M} \cdot \mathbf{prec}[1,j1,1,j2] \geq 0 \qquad \forall j1,j2|j1 \neq j2 \qquad (4.26)$$

$$\mathbf{C}[1,j1] - \mathbf{P}[1,j1] - \mathbf{C}[3,j2] + \mathbf{M} \cdot \mathbf{prec}[1,j1,3,j2] + \\ + \mathbf{M} \cdot \mathbf{OR}[j2] \geq 0 \qquad \forall j1,j2|j1 \neq j2 \qquad (4.27)$$

$$\mathbf{C}[3,j1] - \mathbf{P}[3,j1] - \mathbf{C}[1,j2] + \mathbf{M} \cdot \mathbf{prec}[3,j1,1,j2] + \\ + \mathbf{M} \cdot \mathbf{OR}[j1] \geq 0 \qquad \forall j1,j2|j1 \neq j2 \qquad (4.28)$$

$$\mathbf{C}[3,j1] - \mathbf{P}[3,j1] - \mathbf{C}[3,j2] + \mathbf{M} \cdot \mathbf{prec}[3,j1,3,j2] + \\ + \mathbf{M} \cdot (\mathbf{OR}[j1] + \mathbf{OR}[j2]) \geq 0 \qquad \forall j1,j2|j1 \neq j2 \qquad (4.29)$$

$$\mathbf{prec}[1,j1,1,j2] + \mathbf{prec}[1,j2,1,j1] = 1 \qquad \forall j1,j2|j1 \neq j2 \qquad (4.30)$$

$$\mathbf{prec}[2,j1,2,j2] + \mathbf{prec}[2,j2,2,j1] = 1 \qquad \forall j1,j2|j1 \neq j2 \qquad (4.31)$$

$$\mathbf{prec}[3,j1,3,j2] + \mathbf{prec}[3,j2,3,j1] = 1 \qquad \forall j1,j2|j1 \neq j2 \qquad (4.32)$$

$$\mathbf{prec}[2,j1,3,j2] + \mathbf{prec}[3,j2,2,j1] = 1 \qquad \forall j1,j2|j1 \neq j2 \qquad (4.33)$$

$$\mathbf{prec}[1,j1,3,j2] + \mathbf{prec}[3,j2,1,j1] = 1 \qquad \forall j1,j2|j1 \neq j2 \qquad (4.34)$$

Constraint 4.19: Characterizes the support variable **Max_completion** to be greater than every completion time of the schedule, thus allowing the linearization of the OF being $\min(\max_j(\mathbf{C}[3,j]))$.

Constraint 4.20: Imposes that no patient can start before time t=0. The way this constraint is formulated is what creates the problem discussed in section 4.1 of potential idle time at the beginning of the schedule.

Constraint 4.21: Imposes the No-wait constraint between A and S of each patient.

Constraint 4.22: Imposes the No-wait constraint between S and AW of each patient.

Constraint 4.23: Ensures that no overlap between two S of two different patients can occur by using two BIG-M constraint applied to **prec**.

Constraints 4.24 and 4.25: Ensure that no overlap between S of a patient and AW of another can occur by using a BIG-M constraint applied to **OR** and **prec** DVs.

Constraint 4.26: Ensures that no overlap between two A of two different patients can occur by using two BIG-M constraint applied to **prec**.

Constraint 4.27 and 4.28: Ensure that no overlap between A of a patient and AW of another can occur by using a BIG-M constraint applied to **OR** and **prec** DVs.

Constraint 4.29: Ensures that no overlap between two AW of two different patients can occur in a very specific case in which both patient's AW occurs in the AOP.

Constraint 4.30, 4.31, 4.32, 4.33, 4.34: Characterize the **prec** variable by imposing that either a patient comes before another or viceversa.

4.3.3 Model overview

The model under consideration carries a significantly greater computational burden compared to that for the AOPSP. This increase in complexity is anticipated, aligning with the discussions in 3.2.2. Despite its accuracy, this specific formulation demands excessive computational resources to achieve full optimization for problem instances of a scale similar to the one addressed. Consequently, further research has been directed towards developing a more computationally efficient model, as will be shown in the following section.

4.4 Assignment-based model for the IAOPORSP

4.4.1 DVs and parameters

In an effort to mitigate the computational demands of the model, a novel approach divergent from the previous one has been employed. In the work of Unlu et al. (2010) is shown that selecting a different set of DVs can significantly reduce the computational load on the solver when addressing the same problem [31]. In this revised model, rather than encoding the precedence among patients through a binary four-dimensional variable that encompasses all possible precedences among tasks, a simpler two-dimensional variable is introduced. This Precedence Variable intrinsically models the ordering relation between patients by representing who is scheduled in what position of the final order. This methodological pivot not only simplifies the problem representation but also aims at enhancing the solver's efficiency by reducing the complexity of the decision variable space.

$$\mathbf{C}[i,j] \in \mathbb{Z}^+ \tag{4.35}$$

$$\mathbf{OR}[j] \in \{0,1\} \tag{4.36}$$

$$\mathbf{Z}[j1,j2] \in \{0,1\} \tag{4.37}$$

With

$$\mathbf{i} \in \{1,2,3\}$$
$$\mathbf{j,\ j1,\ j2} \in \{1,2,\ldots,N\}$$

While the mathematical definition is equal to the previous formulations the DV $\mathbf{C}[i,j]$ is conceptually different in what it represents. Instead of being associated to the completion time of patient j, this time the variable is associated with the completion time of the j-th patient of the schedule. This small change will be reflected in the constraints formulation of the model and will lead to a significant change in optimization weight. As previously stated variable $\mathbf{Z}[j1,j2]$ instead represents whether patient j1 is placed in position j2 of the schedule, again conceptually different from the previously used (4.15) DV. To make everything coherent variable $\mathbf{OR}[j]$ is again associated with the placement of the AW of the j-th patient. Other parameters of the model follow:

$$\mathbf{M} = 3000 \tag{4.38}$$

$$\mathbf{P}[i,j] \in \mathbb{Z}^+ \tag{4.39}$$

$$\mathbf{S}[i,j] = \sum_{k=1}^{N} \mathbf{P}[i,k] \cdot \mathbf{Z}[k,j] \tag{4.40}$$

4.4.2 Model formulation

Minimize:

$$\mathbf{C}[3,N]$$

Subject to:

$$\mathbf{C}[1,1] - \mathbf{S}[1,1] = 0 \tag{4.41}$$

$$\mathbf{OR}[N] = 1 \tag{4.42}$$

$$\sum_{j1=1}^{N} \mathbf{Z}[j,j1] = 1 \qquad \forall j \tag{4.43}$$

$$\sum_{j1=1}^{N} \mathbf{Z}[j1,j] = 1 \qquad \forall j \tag{4.44}$$

$$\mathbf{C}[2,j] - \mathbf{S}[2,j] = \mathbf{C}[1,j] \qquad \forall j \tag{4.45}$$

$$\mathbf{C}[3,j] - \mathbf{S}[3,j] = \mathbf{C}[2,j] \qquad \forall j \tag{4.46}$$

$$\mathbf{C}[1,j+1] - \mathbf{S}[1,j+1] \geq \mathbf{C}[1,j] \qquad \forall j \in 1,\dots,N-1 \tag{4.47}$$

$$\mathbf{C}[2,j+1] - \mathbf{S}[2,j+1] \geq \mathbf{C}[2,j] \qquad \forall j \in 1,\dots,N-1 \tag{4.48}$$

$$\begin{aligned}\mathbf{C}[2,j+1] - \mathbf{S}[2,j+1] \geq \mathbf{C}[2,j] + \mathbf{S}[3,j] -& \\ - \mathbf{M} \cdot (1 - \mathbf{OR}[j])& \qquad \forall j \in 1,\dots,N-1\end{aligned} \tag{4.49}$$

$$\mathbf{C}[3,j] - \mathbf{S}[3,j] \geq \mathbf{C}[1,j+1] - \mathbf{M} \cdot \mathbf{OR}[j] \qquad \forall j \in 1,\dots,N-1 \tag{4.50}$$

$$\mathbf{C}[1,j+2] - \mathbf{S}[1,j+2] \geq \mathbf{C}[3,j] - \mathbf{M} \cdot \mathbf{OR}[j] \qquad \forall j \in 1,\dots,N-2 \tag{4.51}$$

Notice how **S** is just a notation variable and not a DV nor a parameter.

Constraint 4.41: Imposes the beginning of the first patient at time t=0, thus avoiding any idle time before starting the schedule.

Constraint 4.42: Imposes the placement of the last patient's AW to be in the OR, since no improvement could be obtained by having it in the AOP.

Constraints 4.43, 4.44: Characterize the **Z** variable by imposing that no two patients can be scheduled in the same position simultaneously.

Constraint 4.45: Imposes the No-wait constraint between A and S of each patient.

Constraint 4.46: Imposes the No-wait constraint between S and AW of each patient.

Constraint 4.47: Ensures the ordering of patients by imposing that A of a patient starts after the start of previous patient's A.

Constraint 4.48: Ensures that no overlap between two S of two different patients can occur in the OR.

Constraint 4.49: Is a continuation of constraint 4.48, ensures that no overlap can occur between the S of a patient and the AW of the previous one in case the previous AW is placed in the OR. This constraint incorporates a BIG-M method, which cannot be straightforwardly applied as in 4.10. The differentiation stems from the fact that, in this context, the processing times are not static parameters but rather the outcomes of multiplying a parameter by a DV. Consequently, these processing times cannot be directly multiplied by **OR** without compromising the model's linearity.

Constraint 4.50: Ensures no overlap between AW of a patient and A of the previous, when AW of current patient occurs in the AOP.

Constraint 4.51: Ensures no overlap between AW of a patient and A of the patient scheduled two position ahead in the schedule, when current patients AW is placed in the AOP.

4.4.3 Model overview

The model in consideration can be seen as a generalization of the model used for the AOPSP explained in section 4.2, with the only difference in formulation being given by the fact that using **S** for the processing times requires some adjustments to avoid non-linearity during constraints formulation.
The model shows a marked enhancement over the time-based formulation (4.3), with computation time dramatically decreased from several hours for instances involving 15 patients to just a few minutes for scheduling 16 patients; a task impossible in reasonable times with the prior model. The increased efficiency of this new algorithm will be explored in subsequent sections.

Chapter 5

Methodology

5.1 Algorithms for AOPSP

Table 5.1 describes all the variables used in the algorithms for the AOPSP. The latter are categorized between DVs and fixed parameters. Additionally each algorithm will present support variables for the already illustrated linearization methods. The algorithms assume that the processing times for the task to perform are already structured according to the imposed order.

Variable	**Definition**
N	Number of patients to schedule.
I=$\{1,..N\}$	Set of patient's indexes. $i \in I$
J=$\{1,2,3\}$	Set of task's indexes. $j \in J$
A(i)	Duration of A for i-th patient of the schedule. Also referred to as **P(j=1,i)**
S(i)	Duration of S for i-th patient of the schedule. Also referred to as **P**(j=2,i)
AW(i)	Duration of AW for i-th patient of the schedule. Also referred to as **P**(j=3,i)
$\mathbf{OR_{AW}}(i)$	Positioning of i-th patient of the schedule. $\mathbf{OR_{AW}}$(i)=1 if patient **i** AW occurs in the OR, $\mathbf{OR_{AW}}$(i)=0 otherwise.
$\mathbf{B_A}(i)$	Beginning of A for i-th patient of the schedule.
$\mathbf{C_A}(i)$	Completion of A for i-th patient of the schedule.
$\mathbf{B_S}(i)$	Beginning of S for i-th patient of the schedule.
$\mathbf{C_S}(i)$	Completion of S for i-th patient of the schedule.
$\mathbf{B_{AW}}(i)$	Beginning of AW for i-th patient of the schedule.
$\mathbf{C_{AW}}(i)$	Completion of AW for i-th patient of the schedule.

Table 5.1: Parameters and variables for AOPSP algorithms

For each task it obviously holds that $\mathbf{C_j}(i) = \mathbf{B_j}(i) + \mathbf{P}(j, i)$.

5.1.1 Forward greedy (FG)

Overview

This algorithms searches the solution by building the schedule one patient at a time going forward from the first. The decisions are based solely on the placement of previously processed patients and the processing times of both the current and the next patient in line. No interest is given to the effect of the choice on the patients ahead in the schedule. Since the algorithm picks the best route among a set of currently available options, it is classifiable as a Greedy algorithm, hence the name. The algorithm doesn't need to explicitly explore partial solutions, but rather builds the schedule based on 3 statements, checked for every patient. The statements can be either True or False and have been built such that in each set of statements considered, one and only one of the three is verified. Given which one it is, a certain approach is used for determining the patient's placement. It must be noted that the exploration of the ID is implicit (and incomplete); the three statements have been built by collecting all the possible situations a new patient could find when entering the system and noticing that the positioning could be obtained by verifying three conditions. This consideration only accounts for local decisions, similarly to a fixed width decision tree that only explores a portion of the whole domain.
The algorithm runs in $\mathcal{O}(3 \cdot \mathbf{N})$, a linear execution time. It is relevant to notice that the number of statements is not equal to the possible placements of the patient, as this case shows.
The FG approach proved to be non optimal, but with a considerably small margin of error from the optimum. Moreover the study performed to build the FG lead to relevant discoveries for understanding the problem, hence why is here presented.

Support variables and Statements

In table 5.2 the support variables for the algorithm are declared.

Variable	Definition
$\mathbf{C_{OR}}$	Current time of availability of the OR, so completion of the last task scheduled to be in the OR at time of computation.
$\mathbf{C_{AOP}}$	Current time of availability of the AOP, so completion of the last task scheduled to be in the AOP at time of computation.

Table 5.2: Support variables for AOPSP Forward Greedy Algorithm

The three statements used for the identification of patient's **i** scheduling follow:

$$\mathbf{DF1}: \quad \mathbf{A}(i+1) \leq \mathbf{T} \tag{5.1}$$

$$\mathbf{DF2}: \quad \mathbf{T} < \mathbf{A}(i+1) \leq \mathbf{T} + \mathbf{AW}(i) \tag{5.2}$$

$$\mathbf{DF3}: \quad \mathbf{A}(i+1) > \mathbf{T} + \mathbf{AW}(i) \tag{5.3}$$

where:

- $\mathbf{T}$=$\mathbf{B_A}(i) + \mathbf{A}(i) + \mathbf{S}(i) - \mathbf{C_{AOP}}$ is used only for readability purposes.
- 5.1 checks whether the A of the next patient could be placed between the available time in the AOP and the beginning of the AW of current patient, in case the AW of current patient was scheduled in the AOP. In picture 5.1 can be seen that since A of patient "**2**" is short enough to fit between AW of patients "**0**" and "**1**", the schedule allows for "**1**" to Awaken in the AOP.
- 5.2 is in case the previous condition is not met, and checks whether the A of next patient could be placed in the AOP if $\mathbf{OR_{AW}}$(i)=1 were to be considered. In such case, the time between the end of **AW**(i) and the availability of the AOP would be big enough to fit **A**(i+1). In image 5.2 it's clear how the A of patient "**2**" is too long to fit between patient's "**0**" and "**1**" AW, so the AW of patient "**1**" if forced in the AOP and the A of patient "**2**" starts as soon as possible such that the completion coincides with the completion of patient "**1**" AW.

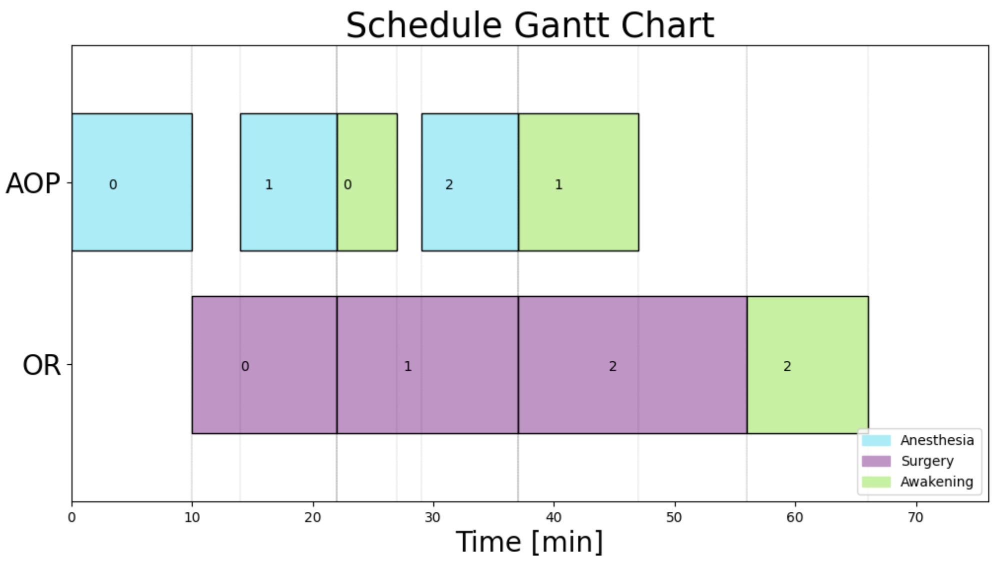

Figure 5.1: Example of placement for patient "**1**" when his statement DF1 is verified.

- Ultimately 5.3, if the other conditions are not met, is a safe-net for when **A**(i+1) is so long that it can't fit in the schedule without creating idle time in the OR. In picture 5.3 the A of patient "**2**" is so long that idle time is inevitably created in the OR, so long as this order has to be maintained.

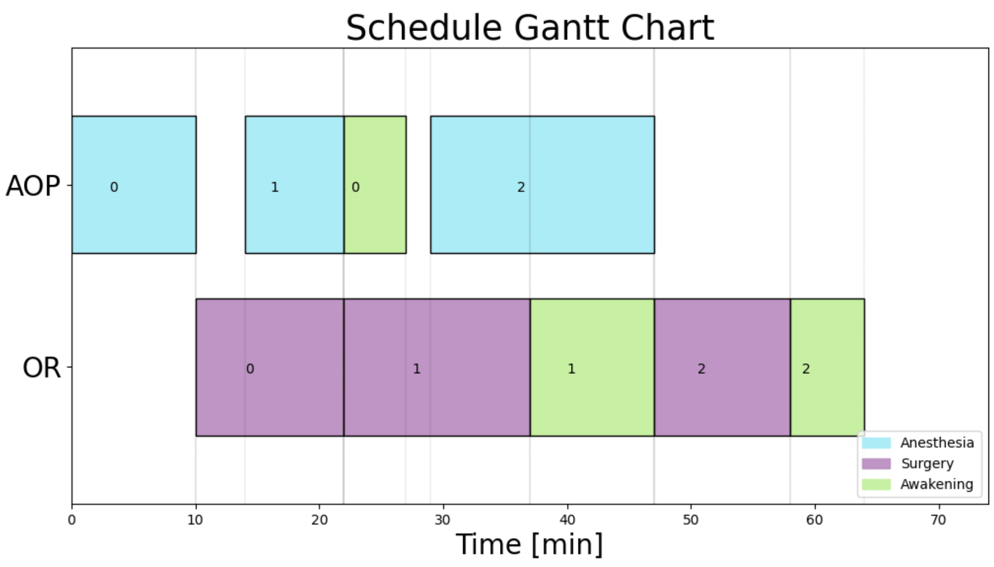

Figure 5.2: Example of placement for patient "1" when his statement DF2 is verified.

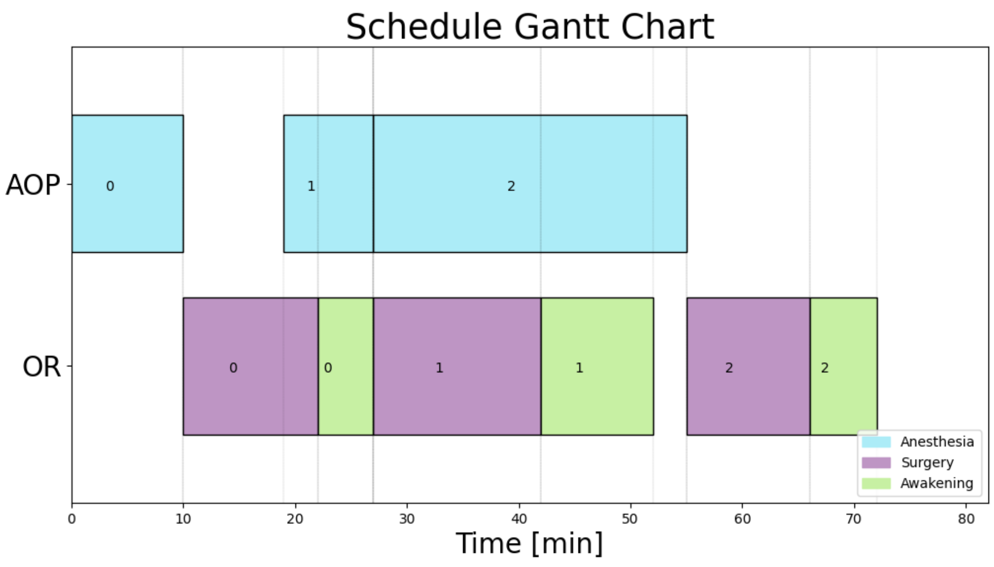

Figure 5.3: Example of placement for patient "1" when his statement DF3 is verified.

Algorithm

The algorithm is initialized as follows:

$$\mathbf{B_A}(1) = 0 \tag{5.4}$$

$$\mathbf{C_{OR}} = 0 \tag{5.5}$$

$$\mathbf{C_{AOP}} = \mathbf{A}(1) \tag{5.6}$$

$$\mathbf{OR_{AW}}(\mathbf{N}) = 1 \tag{5.7}$$

where:

- 5.4 Fixes the beginning of the schedule, that must start with the Anesthesia of the first patient at t=0.
- 5.5 Initializes the available time of the OR at t=0. Even though it could already be fixed at t=$\mathbf{A}(1)$+$\mathbf{S}(1)$, this step is necessary to make the algorithm iterative over the patients.
- 5.6 Initializes the available time of the AOP at t=$\mathbf{C}(1,1)$.
- 5.7 Forces the last patient's AW in the OR. This is done since, after the last patient, the OR will be inevitably empty whereas the AOP may still contain some process from the second to last patient.

For the main logic, which is repeated over each patient, there are two sections. The first fixes the completion times of S and AW. This part is equal for all patients, since even if the placement of AW is unknown yet, the time of completion is fixed by the no-wait. Secondly a cascade of if-else statements checks which approach to use for placement.
For the completion of S and AW it'll be:

$$\mathbf{B_S}(i) = \mathbf{B_A}(i) + \mathbf{A}(i) \tag{5.8}$$

$$\mathbf{B_{AW}}(i) = \mathbf{B_S}(i) + \mathbf{S}(i) \tag{5.9}$$

If the verified statement is found to be **DF1**, then the following instructions are executed:

$$\mathbf{OR_{AW}}(i) = 0 \tag{5.10}$$

$$\mathbf{B_A}(i+1) = \mathbf{B_{AW}}(i) - \mathbf{A}(i+1) \tag{5.11}$$

$$\mathbf{C_{OR}} = \mathbf{B_{AW}}(i) \tag{5.12}$$

$$\mathbf{C_{AOP}} = \mathbf{B_{AW}}(i) + \mathbf{AW}(i) \tag{5.13}$$

Else, if the true statement turns out to be **DF2**, the algorithm executes:

$$\mathbf{OR}_{\mathbf{AW}}(i) = 1 \tag{5.14}$$

$$\mathbf{B}_{\mathbf{A}}(i+1) = \mathbf{B}_{\mathbf{AW}}(i) + \mathbf{AW}(i) - \mathbf{A}(i+1) \tag{5.15}$$

$$\mathbf{C}_{\mathbf{OR}} = \mathbf{B}_{\mathbf{AW}}(i) + \mathbf{AW}(i) \tag{5.16}$$

$$\mathbf{C}_{\mathbf{AOP}} = \mathbf{B}_{\mathbf{A}}(i+1) + \mathbf{A}(i+1) \tag{5.17}$$

Lastly, if the patient conditions makes **DF3** the correct statement, the following instructions are implemented:

$$\mathbf{OR}_{\mathbf{AW}}(i) = 1 \tag{5.18}$$

$$\mathbf{B}_{\mathbf{A}}(i+1) = \mathbf{B}_{\mathbf{A}}(i) + \mathbf{A}(i) \tag{5.19}$$

$$\mathbf{C}_{\mathbf{OR}} = \mathbf{B}_{\mathbf{AW}}(i) + \mathbf{AW}(i) \tag{5.20}$$

$$\mathbf{C}_{\mathbf{AOP}} = \mathbf{B}_{\mathbf{A}}(i+1) + \mathbf{A}(i+1) \tag{5.21}$$

5.1.2 Backward greedy (BG)

Overview

This algorithm has been designed right after the FG. The reasoning for the following section requires the knowledge of the results from the first algorithm, but for the sake of synthesis it is sufficient to know that the first algorithm generates non-idealities because local decisions go in contrast with future placements, delaying the whole schedule. To attempt to solve this problem, a backward approach has been chosen for the successive algorithm. The idea is that creating a schedule from the last patient going backward would allow for the decision to be made with knowledge of the future requirements of the schedule. Considering the patient currently under investigation to be the one in position **i**, the algorithm takes into consideration:

- The current situation of the schedule, so information provided by the placement of the patients **j** > **i** and the time gap available in the AOP, if present.
- The duration of the procedures that patient **i** must undergo, with respect to the available times.
- The duration of the A of patient **i-1**, which has not yet been processed but that is accounted for to avoid the non-idealities that appeared with the FG.

The algorithm has once again a building approach rather than an exploring one for the solutions. Consequently, a linear complexity is obtained as for the FG. The increment in information utilized implies an increase in computation that brings the

total complexity to $\mathcal{O}(4 \cdot \mathbf{N})$, still a negligible computational load.
Since the BG picks among a set of possible choices the best one, even though the concept of "best" for this algorithm is slightly more global than in the FG, the correct classification is still that of a greedy approach, hence the name.
The methodology of the BG is similar to the first attempt, a set of statements has been chosen to classify all possible situations a patient could find when entering the system. Note that this time the statement are not built such that only one can be true, but a certain combination of statements will be used for each case.
The number of situations a patient can find entering the system rises to six from the three of the FG. Thanks to the proper choice and combination of Statements it is possible to reduce the building approaches to only four.
Since going backward implies that the AW, the only task whose placement might change, is not "in front" of the partial schedule, only one support variable will be needed for the algorithm, to account for the possible gap in the AOP where the AW of patient **i-1** might be placed.
This algorithm proved to be better than the FG but still non optimal. The knowledge required to choose correctly the placement of the patient under investigation turned out to include further patients in the schedule.

Support variables and initialization

In table 5.3 the support variables for the algorithm are declared.

Variable	Definition
GAP	Next moment after $\mathbf{C_A}(i+1)$ where AOP is not free anymore. Used to compute the possible time available to fit $\mathbf{AW}$(i).

Table 5.3: Support variable for AOPSP Greedy Backward Algorithm

The statements used for the identification of the i-th patient follow:

$$\mathbf{DB1}: \quad \mathbf{AW}(i) \leq \mathbf{GAP} - \mathbf{C_A}(i+1) \tag{5.22}$$

$$\mathbf{DB2}: \quad \mathbf{A}(i+1) \leq \mathbf{S}(i) \tag{5.23}$$

$$\mathbf{DB3}: \quad \mathbf{A}(i+1) \leq \mathbf{AW}(i) + \mathbf{S}(i) \tag{5.24}$$

$$\mathbf{DB4}: \quad [\mathbf{S}(i) + \mathbf{AW}(i) \leq \mathbf{AW}(i-1) + \mathbf{A}(i+1)] \ \&.. \quad ..[\mathbf{AW}(i) \leq \mathbf{AW}(i-1)] \ \& \ [i > 2] \tag{5.25}$$

where:

- 5.22 checks if it is possible to place $\mathbf{AW}$(i) in the time slot right after $\mathbf{A}$(i+1). Picture 5.4 clearly shows a situation in which $\mathbf{AW}(1)$ is too long to be placed between **A**(2) and **AW**(2). Notice that if another patient was present after "**2**", then the **GAP** would have been possibly constrained by **A**(3).

- 5.23 checks if **S**(i) is long enough to be placed in parallel to **A**(i+1) such that $\mathbf{B_A}(i+1) \geq \mathbf{B_S}(i)$ and $\mathbf{C_S}(i) = \mathbf{C_A}(i+1)$. Picture 5.5 depicts a situation in which the length of **S**(1) enables the placement of **AW**(1) in the AOP whilst the parallel positioning between **S**(1) and **A**(2) leaves enough space for **A**(1) to have no interference.
- 5.24 checks if **S**(i) together with **AW**(i) are long enough to be placed in parallel to **A**(i+1) such that $\mathbf{B_A}(i+1) \geq \mathbf{B_S}(i)$ and $\mathbf{C_{AW}}(i) = \mathbf{C_A}(i+1)$. The situation of picture 5.6 is analogous to that of point 5.23, but this time **S**(1) alone is not sufficient to be placed in parallel to **A**(2), so **AW**(1) is sequentially placed in the OR.
- 5.25 checks whether or not **AW**(i-1) could be placed after **A**(i) and before **A**(i+1). This statement can't be True for **i**=1 and **i**=2. In picture 5.7 the length of **S**(2) would allow for **AW**(1) to be placed in the **GAP** above it, but thanks to this condition it is possible to notice that keeping **AW**(1) in the OR allows for **AW**(0) to have enough space to be positioned in the AOP.

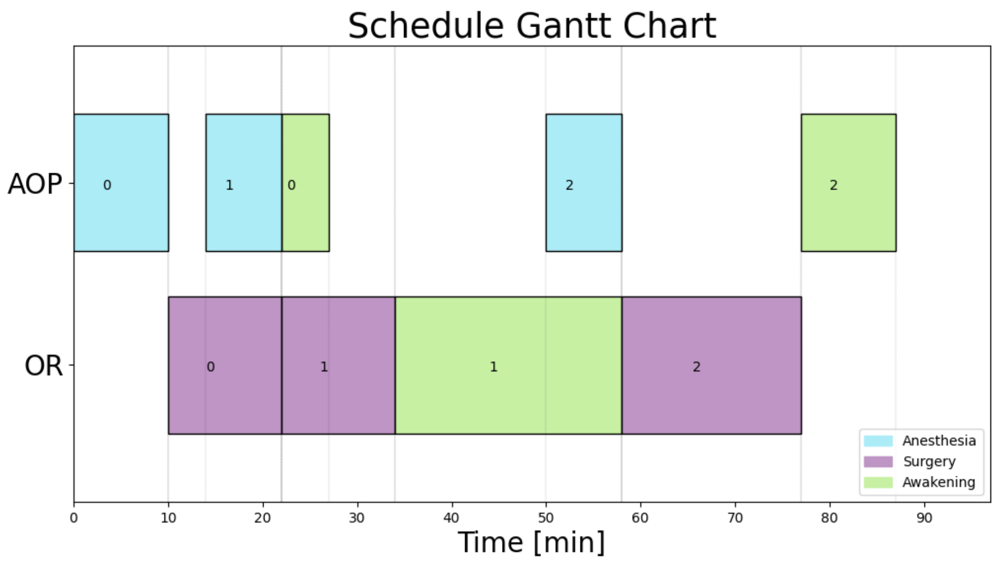

Figure 5.4: Example of schedule in which DB1 is not verified and consequent idle time.

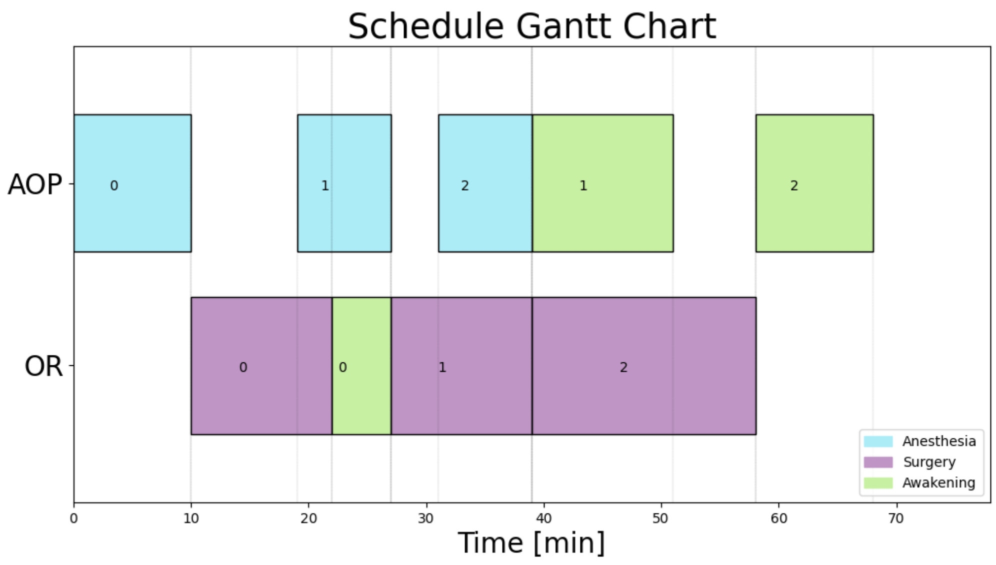

Figure 5.5: Example of schedule in which DB2 is verified and consequent possible placement.

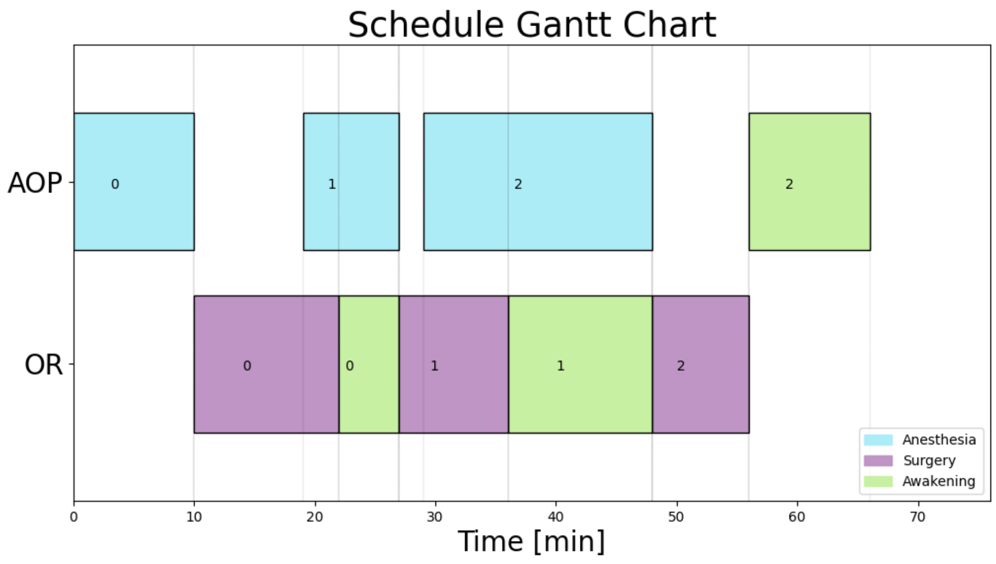

Figure 5.6: Example of schedule where DB3 is verified and consequent possible placement.

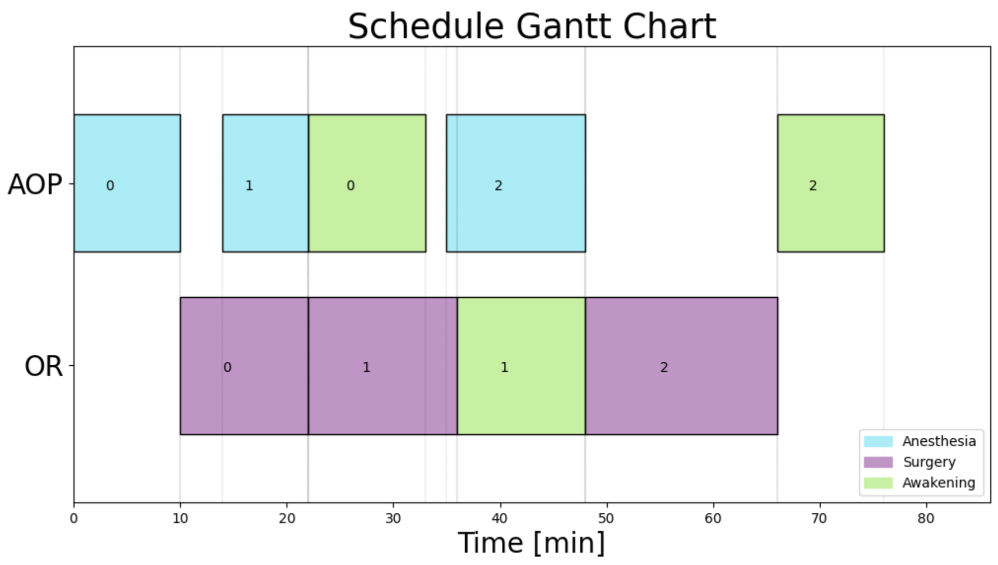

Figure 5.7: Example of schedule where DB4 is verified and consequent possible placement.

Algorithm

The algorithm is initialized by fixing variables of the last patient as follows:

$$\mathbf{C_{AW}(N)} = 0 \tag{5.26}$$

$$\mathbf{AW_{OR}(N)} = 1 \tag{5.27}$$

$$\mathbf{C_S(N)} = -\mathbf{AW(N)} \tag{5.28}$$

$$\mathbf{C_A(N)} = -(\mathbf{AW(N)} + \mathbf{S(N)}) \tag{5.29}$$

$$\mathbf{GAP} = 1000 \tag{5.30}$$

where:

- 5.26 fixes the end of the schedule at t=0. This is done solely for simplicity; at the end of the computation the whole schedule will be shifted to begin at t=0. The shifting magnitude will be equal to the make span.
- 5.27 fixes the placement of the last AW in the OR, for the same reason described in 5.7.
- 5.28 and 5.29 fix the completion for the last patient's S and A, according to the no-wait constraint.
- 5.30 initializes GAP to a random value big enough to not interfere with the first iteration's logic.

Moving to the main logic, each patient **i** will have the four statements evaluated. Successively, the following combinations, in the presented order, are considered:

IF: (**DB1**=**True** & **DB2**=**True** & **DB4**=**False**) then the following logic is implemented:

$$\mathbf{OR_{AW}}(i) = 0 \tag{5.31}$$

$$\mathbf{C_{AW}}(i) = \mathbf{C_A}(i+1) - \mathbf{AW}(i) \tag{5.32}$$

$$\mathbf{C_S}(i) = \mathbf{C_A}(i+1) \tag{5.33}$$

$$\mathbf{C_A}(i) = \mathbf{C_S}(i) - \mathbf{S}(i) \tag{5.34}$$

$$\mathbf{GAP} = \mathbf{C_A}(i+1) - \mathbf{A}(i+1) \tag{5.35}$$

Else, IF (**DB1**=**True** & **DB2**=**True** & **DB4**=**True**), the program executes:

$$\mathbf{OR_{AW}}(i) = 1 \tag{5.36}$$

$$\mathbf{C_A}(i) = \mathbf{C_A}(i+1) - \mathbf{A}(i+1) - \mathbf{AW}(i-1) \tag{5.37}$$

$$\mathbf{C_S}(i) = \mathbf{C_A}(i) + \mathbf{S}(i) \tag{5.38}$$

$$\mathbf{C_{AW}}(i) = \mathbf{C_S}(i) + \mathbf{AW}(i) \tag{5.39}$$

$$\mathbf{GAP} = \mathbf{C_A}(i+1) - \mathbf{A}(i+1) \tag{5.40}$$

Again, if the previous condition is not met, the program checks if the patient falls into the cases for which:
(**DB1**=**True** & **DB2**=**False** & **DB3**=**True**) OR (**DB1**=**False** & **DB3**=**True**) In case it does, it runs:

$$\mathbf{OR_{AW}}(i) = 1 \tag{5.41}$$

$$\mathbf{C_{AW}}(i) = \mathbf{C_A}(i+1) \tag{5.42}$$

$$\mathbf{C_S}(i) = \mathbf{C_{AW}}(i) - \mathbf{AW}(i) \tag{5.43}$$

$$\mathbf{C_A}(i) = \mathbf{C_S}(i) - \mathbf{S}(i) \tag{5.44}$$

$$\mathbf{GAP} = \mathbf{C_A}(i+1) - \mathbf{A}(i+1) \tag{5.45}$$

Ultimately, only if all other cases are not verified, the program checks whether: (**DB1**=**True** & **DB2**=**False** & **DB3**=**False**) OR (**DB1**=**False** & **DB3**=**False**) and then executes:

$$\mathbf{OR_{AW}}(i) = 1 \tag{5.46}$$

$$\mathbf{C_A}(i) = \mathbf{C_A}(i+1) - \mathbf{A}(i+1) \tag{5.47}$$

$$\mathbf{C_S}(i) = \mathbf{C_A}(i) + \mathbf{S}(i) \tag{5.48}$$

$$\mathbf{C_{AW}}(i) = \mathbf{C_S}(i) + \mathbf{AW}(i) \tag{5.49}$$

$$\mathbf{GAP} = \mathbf{C_A}(i+1) - \mathbf{A}(i+1) \tag{5.50}$$

Note that for this last case the check on the variables is redundant, as all the possible feasible combinations of the statements are explored over the course of the four checks.

As previously explained, once the algorithm has successfully created the schedule, the whole needs to be shifted to obtain $\mathbf{B_A}(1) = 0$. This is done by summing to all the completion times/begin times the current absolute value of $\mathbf{B_A}(1)$, which is also the make span of the schedule.

5.1.3 Modified Branch and Bound (B&B) - Best algorithm

Overview

This algorithm exploits the knowledge obtained from the two attempts illustrated before and many more that are not presented in this work. It has been found that the optimal placement requires knowledge of several patients surrounding the one under investigation. It has also been proved that a greedy choice, only considering few patients, could lead to a globally non optimum solution. No knowledge on how far was required to look to assert the ideal choice was present at the time of creation of this algorithm, so a conservative approach has been exploited. To tackle the issue a mix of exploratory and building approach has been selected. The idea is to exploit a B&B logic but avoiding:

- The explosion of the explored solutions to the whole ID, making the algorithm just a brute force attempt. The interest has been diverted toward an algorithm capable of only exploring a linear function of the Instance dimension.
- The pruning of locally bad partial solutions that might contain an underlying globally optimum solution.

The exploration has been decided to start from the first patient moving forward; when the last patient is reached a set of not anymore partial solutions is available.

Since no further exploration is now needed, the algorithm can proceed to pick the best solution and conclude the computation.

Possible cases and logic

Similarly to what illustrated in the previous two attempts, the whole set of possible states a new patient could find when entering the system is depicted in the following images and tables.

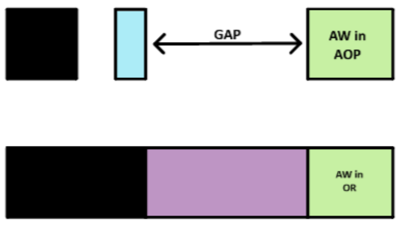

Figure 5.8: New positioning case C1

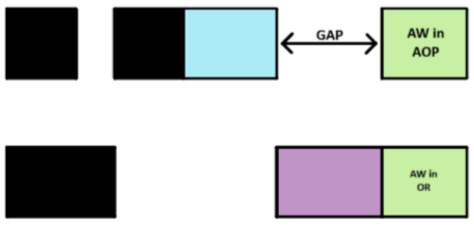

Figure 5.9: New positioning case C2

The black blocks represent part of the schedule already determined, providing the boundary conditions for the insertion of the new patient, depicted with the same colors used in the solution representation (1.2.3). Notice that only when the AW occurs in the AOP a GAP can be generated.

Case	**Explanation**
C1	In this situation the AOP is freed before the OR, so there is room for the A of a patient. This A is small enough to fit without idle time in the OR, and since no other constraints are present the patient can be scheduled to awaken in both OR or AOP. Image: 5.8
C2	This scenery has the AOP being freed after the OR, with a possible GAP where a short A could fit, but the one of patient under consideration is too long to fit and so everything is shifted to the right, causing a lot of idle time. The AW of the new patient is consequently unconstrained. Image: 5.9

Table 5.4: Explanation of possible new positioning cases C1 and C2

Almost all the possible situations can evolve in two ways, whether the AW of the new patient is placed in the AOP or in the OR, for a total of 11 cases.
Each time a new iteration occurs, a new patient is analyzed and a set of currently alive partial solutions exists. The program generates the partial solutions obtained

by placing, in both ways, the new patient on all the alive set. Each new solution will belong to one of the 11 cases presented.
Subsequently, if the above are grouped again into a set where $\mathbf{AW_{OR}} = 1$ and one where $\mathbf{AW_{OR}} = 0$, then it's possible to process the two with different rules.

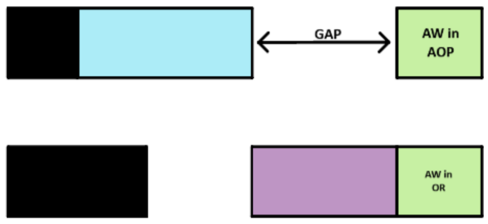

Figure 5.10: New positioning case C3

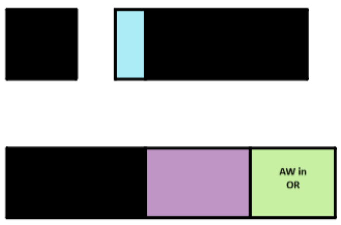

Figure 5.11: New positioning case C4

Case	Explanation
C3	In this situation the starting condition is the same of C1, but the length of the new A forces inevitably idle time in the OR. The AW is not constrained and can be placed everywhere. Image: 5.10
C4	Situation analogous to C3, here the **GAP** is big enough for A to fit and thus no idle time is created. Since the length of S is smaller than the black block in the AOP, the AW is forced to occur in the OR. If the length of S+AW is still shorter than the black box, the following patient will experience case C5. Image: 5.11

Table 5.5: Explanation of possible new positioning cases C3 and C4

In the FG and BG attempts the aim was finding the best solution to determine where to proceed. Now the focus has been shifted to determining criteria that could highlight partial solutions mathematically bound to perform worse than some other existing partial solutions. Once these weak leafs are found, they are pruned to avoid useless computation in the iterations ahead.

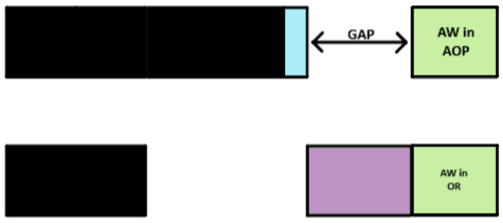

Figure 5.12: New positioning case C5

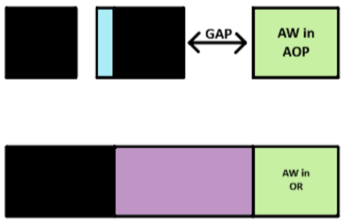

Figure 5.13: New positioning case C6

Case	Explanation
C5	In this scenario the AOP gets freed after the OR without the presence of a **GAP** where to place an A. The new patient is force in the position depicted with a high amount of idle time in the OR. The AW is unconstrained. Image: 5.12
C6	This case the AOP is freed after the OR but the presence of a **GAP** big enough to fit the A of the patient allows for a good fit between patients. Image: 5.13

Table 5.6: Explanation of possible new positioning cases C5 and C6

Dedicated pruning methods must be identified to obtain this, one for each of the two subsets. Once the pruning phase has been completed, the algorithm can continue with the generation of the increasingly big partial solutions. Note that even if no dedicated pruning methods were chosen, the fact that all possible placement could be categorized into at most 11 cases implies that even by only comparing solutions belonging to the same group could lead to the reduction of the problem to maximum $11 \cdot \mathbf{N}$ solutions per iteration.
For the purpose of representing the partial solutions, the same support variables used for the FG and BG will be used:

Variable	Definition
$\mathbf{C_{OR}}$	Current time of availability of the OR, so completion of the last task scheduled to be in the OR at time of computation.
$\mathbf{C_{AOP}}$	Current time of availability of the AOP, so completion of the last task scheduled to be in the AOP at time of computation.
GAP	Amount of time available in the AOP between other tasks where A of the new patient could be placed.

Table 5.7: Support variables for AOPSP B&B Algorithm

Pruning of $\mathbf{AW_{OR} = 1}$

When the placement occurs in the OR it can be noted that the **GAP** in the AOP disappears as there are no task after the A of the previous patient. As a consequence, the following considerations can be made:

- No task needs placement in a constrained **GAP**, so the variables of interest to classify the partial solutions are $\mathbf{C_{OR}}$ and $\mathbf{C_{AOP}}$ only. For the sake of making the algorithm iterate, **GAP** can be fixed to 0.
- It must be remembered that all partial solutions under investigation have the same patient as last positioned, so the duration of the procedures is the same among different solutions.
- Since the AW will always occur after the A, it is possible to describe the current OF of the partial solution as the completion of the AW.
- It is not sufficient to compare only the OF to prune one solution. Two solutions having the same OF may differ for the space available in the AOP to fit the A of the successive patient. The quantity to check should be $\mathbf{C_{OR}} - \mathbf{C_{AOP}}$, which represents how big of an A can be placed in the AOP in this situation.
- The same consideration holds when reversed, considering only $\mathbf{C_{OR}} - \mathbf{C_{AOP}}$ without the OF may lead to incorrect solutions.

The summary of all these considerations lead to the following pruning rule for the group of OR placed patients. Considering two solutions A and B to compare, using the apex x^A for the variables associated to solution A and x^B for those associated to B, we have:

$$IF \quad (\mathbf{OF^A} \leq \mathbf{OF^B} \quad \& \quad \mathbf{C^A_{AOP}} \leq \mathbf{C^B_{AOP}})$$

then solution B can be pruned. It obviously hold that if both inequalities are not verified, solution A can be pruned. If only one of the two is, then both solution are kept alive.
This condition can only remove useless solutions since, given that the OF of A is smaller than that of B, and given that more space is available in the AOP to fit a patient, it follows that solution A can hold all possible placement that B could receive and even more. Even if the two solutions happen to have the same values for the interest variables, it is still a good idea to prune one of the two, since the same solution would come from both ends.

Pruning of $\mathbf{AW_{OR} = 0}$

When the placement occurs in the AOP the following considerations can be made:

- The **GAP** this time does not disappear, and the width may be influenced only by the AW of the previously scheduled patient. Furthermore, the influence occurs only if said patient's AW was placed in the AOP.
- If the **GAP** is not influenced by the AW of the previous patient, then it's length is equal to the S of current patient.
- Similarly to the other case, the current OF is equal to the completion of the AW of the current patient, but this parameter is not sufficient to decide whether to prune a solution or not. Even if one solution has a smaller OF compared to another, the absence of a big enough **GAP** to fit an A in the future might compromise the quality of the solution.
- Again, the previous consideration holds in reverse; **GAP** alone is not sufficient to delete one solution.

Calling again A and B two solutions to compare, and using x^A for the variables associated to solution A and x^B for those associated to B, this time we have:

$$IF \quad (\mathbf{OF^A} \leq \mathbf{OF^B} \quad \& \quad \mathbf{GAP^A} \geq \mathbf{GAP^B})$$

then solution B can pruned. The same considerations hold as for the $AW_{OR} = 1$ case: if both inequalities are not verified, A can be pruned. If only one inequality is verified, none is deleted.
The reasoning also follows that of the other case: if one solution has a better objective function and has a bigger gap to place tasks, then it is capable of receiving all the placements of the other solution and thus the second becomes redundant.

Algorithm

The program starts by taking the first patient and generating the two partial solutions obtained with both the placements. This first stage has obviously no pruning part as only one solution for each class exists. Successively, the following steps repeat:

1. The whole set of currently alive solutions is sprouted: the amount of solutions will double since each solution obtains new developments with $\mathbf{AW_{OR}} = 1$ and $\mathbf{AW_{OR}} = 0$.
2. The set is separated into two subsets according to the positioning of last patient. Let's call **AW1** and **AW0** the two sets.

3. Using the rule specifically made for **AW1**, all the possible couples of partial solutions inside the set are compared, and if necessary the losing solutions are pruned.

4. The same step is performed for **AW0**.

5. The alive solutions in **AW1** and **AW0** are joined again in a new set to start the successive iteration, if possible.

Once the program arrives at last patient, the usual comparing may be neglected and the identification of the best solution among the currently alive ones can be performed. Note that this time no other factor other than the OF has to be considered, nor a distinction between the placement of the last patient.

Empirical results

The algorithm proved to be optimal in all instances tested, always finding the best possible solution. This was expected as the logic never deletes solutions that may lead to good results, but only those that are rigorously proved to be non optimal. The dimension of the explored part of the ID is always a linear function of the instance dimension. The exact computation of the complexity would require a deep analysis of the dynamics at play, however accounting for the considerations made during the overview is safe to consider an upper bound for the complexity of $\mathcal{O}(11 \cdot \mathbf{N})$. Empirical results point more toward a complexity of $\mathcal{O}(4 \cdot \mathbf{N})$, as will be shown in the results section of this discussion. The execution time, as predicted by the complexity, is negligible even for huge instances.
If the equal signs are removed from the pruning rules, a bigger set of final solutions appears after execution, many of which having the same OF. This implies that the optimal solution to such a problem exists but is not unique.

5.2 Algorithms for IAOPORSP

The methodology applied to the IAOPORSP follows the one used for the AOPSP in many aspects. The variables used for implementing the algorithms are almost identical to those presented in table 5.1, with some tuning performed to account for the change of order of the patients. A support variable will be used for holding the ordering of the patients. Instead of referring to a patient using the number that represent the order in line, this time the patients will be referred to with the number they initially get assigned in the instance presentation. Clearly the number associated to a patient is not relevant for the scope of solving the problem, as long

Variable	**Definition**
$\mathbf{N}$	Number of patients to schedule.
$\mathbf{I} = \{1, ..N\}$	Set of patient's indexes. $i \in I$. They just serve the purpose of labeling, no property is shared between number i and positioning of a patient.
$\mathbf{J}$=$\{1, 2, 3\}$	Set of task's indexes. $j \in J$
$\mathbf{A}(i)$	Duration of A for patient labeled i.
$\mathbf{S}(i)$	Duration of S for patient labeled i.
$\mathbf{AW}(i)$	Duration of AW for patient labeled i.
$\mathbf{OR_{AW}}(i)$	Positioning of patient labeled i. $\mathbf{OR_{AW}}$(i)=1 if patient **i** AW occurs in the OR, $\mathbf{OR_{AW}}$(i)=0 otherwise.
$\mathbf{B_A}(i)$	Beginning of A for patient labeled i.
$\mathbf{C_A}(i)$	Completion of A for patient labeled i.
$\mathbf{B_S}(i)$	Beginning of S for patient labeled i.
$\mathbf{C_S}(i)$	Completion of S for patient labeled i.
$\mathbf{B_{AW}}(i)$	Beginning of AW for patient labeled i.
$\mathbf{C_{AW}}(i)$	Completion of AW for patient labeled i.
$\mathbf{ORD}(i)$	Variable used to store the information on the ordering of patients. **ORD**(i) contains the positioning of the patient labeled i.

Table 5.8: Parameters and variables for IAOPORSP algorithms

as the processing times are correctly associated to each patient. For each task it obviously hold that $\mathbf{C_j}(i) = \mathbf{B_j}(i) + \mathbf{P}(j, i)$.
To better understand the variable **ORD** (see table 5.8), let's consider a brief example: say the schedule is composed of three patients, labeled respectively 1,2 and 3. **ORD** is a vector of dimension three. If patient 1 has to be processed as second, patient 2 has to go first, and patient 3 last, then the variable will have values:

$$\mathbf{ORD} = [2, 1, 3]$$

Whilst this choice of representation may seem counter intuitive, the implementation for the programs becomes much simpler and direct during the computation.

5.2.1 Generalized Forward Greedy (GFG)

Overview

Since a good knowledge of the problem has been obtained during the tackling of the AOSPS, for this new problem the approach has been changed to a more direct one. This first algorithm will be used as a starting point for the subsequent implementations. In detail, the result of this first algorithm will be fed into methods whose purpose is not to find a solution from scratch, but to improve an existing one. More detail will be provided in the dedicated sections.
Note that the only question discussed in this section will regard the ordering of patients and not the best way to construct a schedule once the order has been determined, since once the order is fixed the problem decays into the already solved AOPSP. Knowing in advance the combinatorial nature of the problem, the first objective has been the creation of a greedy algorithm capable of constructing in polynomial time a solution, even if not perfect. This is obtained through the following observations:

- The first patient of the schedule has inevitably idle time in the OR, since no patient before can fill the room whilst the first is undergoing Anesthesia. Consequently, a good choice for the first patient may be the one having the shortest duration for A.

- The following patients can be chosen by selecting a certain auxiliary variable, function of the patient under investigation and the current schedule arrangement, computing this variable for all possible patients still available, and selecting the one with the best objective. It is like solving a small scheduling problem at each iteration. This method is obviously classified as a greedy approach, which was already proven to be non optimal for the problem, but may serve as a solid base for future processing of the solution.

- While searching for a locally good patient to place is easy, determining the variables to minimize for each allocation is not a simple task, so once again the problem is tackled by classifying the current arrangements into possible scenarios. The placement of the AW of each patient will be decided after the computation of the auxiliary variable to attempt the reduction of the non idealities.

Since the partial solutions will be built one patient at the time with a small exploration of the possible placements for the patients, the GFG uses a mix of building and exploring approach.
Considering the algorithm's characteristics, it stands to reason that larger pools of available patients for scheduling would likely yield better results. With a broader

selection available, the algorithm has a greater array of options to choose from, potentially leading to more optimal outcomes.

Auxiliary variables for computation

To perform the decision for each new insertion in the schedule, for each available patient a set of four variables may be computed. Not all four will be used in the decision process, but the comparison is essential in building the desired logic.
Let's call **prev** the label of the last scheduled patient, **curr** the label of the patient to be scheduled now and **prpr** the label of the second to last scheduled patient, if it exists. Here follows the definition of the the auxiliary variables:

$$\mathbf{DG1} = ||(\mathbf{S}(prev) + \mathbf{AW}(prev) - \mathbf{A}(curr)|| \tag{5.51}$$

$$\mathbf{DG2} = [\mathbf{S}(prev) - \mathbf{A}(curr)]^{+} \tag{5.52}$$

$$\mathbf{DG3} = ||\mathbf{AW}(prev) + \mathbf{S}(prev) - \mathbf{AW}(prpr) - \mathbf{A}(curr)|| \tag{5.53}$$

$$\mathbf{DG4} = [\mathbf{S}(prev) - \mathbf{AW}(prpr) - \mathbf{A}(curr)]^{+} \tag{5.54}$$

where:

- 5.51: This variable accounts for when the AW of the previously scheduled patient occurs in the OR. In this case, the aim is to find the current patient such that the A could fill the whole time in the AOP in parallel to S and AW of the previous occurring in the OR. The smaller this quantity is, the better the fit will be. No constraint is necessary on the sign of the variable, as even a longer A can be placed (with idle time appearing in the OR). This variable can be well understood by looking at pictures 5.8 and 5.10. In the first one A is very short compared to the available time in the AOP, and so idle time is created. Similarly, in the second picture A is too long and so idle time is created in the OR.

- 5.52: This variable mirrors the purpose of **DG1** for when the AW of the previous patient occurs in the AOP. The A of current patient must now fit inside a gap, therefore the variable will be considered only when $\mathbf{A}(curr) \leq \mathbf{S}(prev)$, since the available time is constrained by **S**(prev). If the condition is not met, the variable is set to a value big enough to be ignored. Once again the value of this variable represents how much idle time is left in the AOP, the less the better. In pictures 5.11 or 5.13 it's easy to see the A fitting inside the **GAP**. In picture 5.9 is instead easy to visualize why only positive values of **DG2** are used, since when A is too long the patient must be scheduled on the right creating a lot of idle time in both rooms.

- 5.53: This variable considers the case in which the patient twice before in the schedule awakens in the AOP, while the previous AW occurs in the OR. In this scenario the next patient faces the end of the AW of **prpr** in the AOP, and so the time slot to fill is smaller than the one considered by **DG1**. Since the time available is not constrained, the quantity can have either sign. This situation is analogous to the one described for **DG1**, again depicted in picture 5.8 or 5.10. In this case, the black block in the AOP is representation of the AW of two patient before in the schedule.

- 5.54: The last variable accounts for when the previous patient awakens in the AOP and the one before awakens also in the AOP. As for **DG2** the time available in the AOP will be bounded but this time from **AW**(prpr) and **AW**(prev), so only patients providing positive values are considered during execution. The scenario is observable in picture 5.9 if the fit can't happen, or 5.11 and 5.13 when **DG4** is positive.

The logic of the program requires the identification of the case currently inside the system; depending on the situation only two of the four auxiliary variables are computed, for each patient available, to evaluate the possible effect of each patient on the schedule.
Each time a patient is determined to be the next in the queue, the AW placement of the previous patient is fixed, while the placement of the current one will be fixed when the successive one will be determined.
The selective section of the program can be resumed:

- If the last patient was selected because the smallest computed parameter at the previous step turned out to be either **DG1** or **DG3**, then the successive search is performed using **DG1** and **DG2** as evaluation metrics.

- If the last patient was selected because the smallest computed parameter instead turned out to be either **DG2** or **DG4**, then the following search is performed using **DG3** and **DG4** as evaluation metrics.

Algorithm

The program is initialized by finding the patient with minimum A among the set of all patients, and removing it from the pool of available ones.
To find the patient second in position, the program computes the value of **DG1** and **DG2**, a total of $2 \cdot (\mathbf{N} - 1)$ values, for all patients available. It then compares all the values to identify the smallest, and selects the patient associated to that value to be the next in line. The program keeps also track of whether the minimum found

value was **DG1** or **DG2**.

1. A new search is performed, according to the winning value for the previous patient, the two parameters identified by the selective section are computed for all available patients.
2. The program searches for the minimum value among the ones computed and places the new patient as last in position, then:
   - If the obtained value is **DG1**, then the old patient AW is scheduled in the OR, and the successive search will evaluate **DG1** and **DG2**.
   - If the obtained value is **DG2**, then the old patient AW is scheduled in the AOP, and the successive search will evaluate **DG3** and **DG4**.
   - If the obtained value is **DG3**, then the old patient AW is scheduled in the OR, and the successive search will evaluate **DG1** and **DG2**.
   - If the obtained value is **DG4**, then the old patient AW is scheduled in the AOP, and the successive search will evaluate **DG3** and **DG4**.

This is performed until no more patients are left to schedule.

The algorithm described turned out to be non optimal as expected, but with results not far off from the ideal solutions. The final complexity of the algorithm is linear in the instance dimension, with most operations being simple parameters computation with simple mathematical formulas.
Instead of attempting different algorithm for this problem the focus will now shift toward improving the results obtained by the one just discussed, as an exact algorithm for the problem is not believed to exist at this point.

5.2.2 Neighborhoods

The idea for improving the results currently obtained is to find a method for starting from an existing solution and moving to similar ones with better results. To do this, several definition are needed:

- Simple Neighborhood : To identify similar solutions is not an easy task. When dealing with continuous domain variables, an easy concept of similarity is obtained by varying the input of the system of a small quantity. This is what is usually performed, implicitly, when using a Gradient Based method for solving a problem. The issue under consideration however, as explained in section 2.4, has a discrete nature, and as such no concept of "slightly perturbing the input" makes sense. An alternative definition of neighborhood for combinatorial structures instead can be used: One solution is said to be neighborhood to another if it is possible to go from one to the other by applying an exact algorithm. The nature of the algorithm, so the type of perturbation introduced, defines the neighborhood under consideration.

- Complex Neighborhood : If instead of applying the same algorithm over and over to generate new solutions we consider a mix of different algorithms, then the result of the algorithms and the order of application become relevant matters in the generation of new solutions. The neighborhood generated will be much bigger and thus more various solutions can be found. It has to be noted that the difference between simple and complex neighborhoods is just for explanatory purposes; a series of algorithms is still an algorithm, and as so, simple and complex neighborhoods do not differ in concept.
- Jumping rule: When creating a new solution is also important to set a rule to use to decide if the new explored one is good enough to be considered a new starting point for the neighborhood search. This rule may be as simple as jumping when a better objective function is found, but more complex decisions might come into play.

From now on, the term Neighborhood will refer to the whole set of possible solutions obtainable with one iteration of a specific method to an existing solution. In the following sections several methods for generating Neighborhoods are defined together with the algorithms necessary to obtain them. To remind, currently the solution in expressed by the ordering of patients (a list of numbers/labels) and the positioning in the OR. Given that once the order is fixed, the optimal OR placement can be found by implementing the previously discussed Modified B&B 5.1.3, the only variable considered for the neighborhood generation is the ordering **ORD**.

Swapping

The first and simpler way of generating new solutions is swapping the position of two patients. Since the whole neighborhood obtained with this method will be explored, it's easier to work in respect to the positioning, namely swapping the i-th patient with the j-th patient rather than swapping the patient whose label is i with the one whose label is j.
The only inputs required to perform such a task on **ORD** are the positions of the two patients to swap. The program just proceeds to swap the patient by inverting the two numbers, as in the following example with N=7:

Swap(i=2, j=4):

$$\mathbf{ORD} = (4,6,3,1,5,2,7) \rightarrow (2,6,3,1,5,4,7)$$

Since **Swap(x,y)=Swap(y,x)** and **Swap(x,x)** is pointless, the total number of new solutions that can be generated by a single iteration of this method is $\mathbf{N}\cdot(\mathbf{N}-1)/2$. Obviously several iterations could explore up to $\mathbf{N}!$ solutions, the whole ID.

Sliding

Another possible method for generating Neighborhoods is to slide one patient until it reaches another desired position moving the other patients accordingly. The same consideration made in the previous method holds: since all possible combinations will be explored the easier choice for the inputs of this function are the initial position of the patient to move and the desired destination. The general request will be to slide the i-th patient of the schedule until it reaches the j-th position. This algorithm differs from the previous because while before only the two positions of interest were affected by the program, now all patients between the starting and finishing position will change in order.
Let's call **who** the old position of the patient to slide, and **where** the position where the patient must end up. An example is here provided on what the outcome should be, with N=7:

Slide(who=2, where=5):

$$\mathbf{ORD} = (4,6,3,1,5,2,7) \rightarrow (3,6,2,1,4,5,7)$$

The algorithm must distinguish between when the patient is slid forward or backward. The logic is reported below (in pseudo Python language).

```
def Slide(who,where):
    if (who > where):
        for i in {1..N}:
            if (where <= ORD[i] < who):
                ORD[i] += 1
            elif (ORD[i] == who):
                ORD[i] = where
    else:
        for i in {1..N}:
            if (who < ORD[i] <= where):
```

```
        ORD[i] -= 1
    elif (ORD[i] == who):
        ORD[i] = where
```

The result of one iteration of this method is a set of $N \cdot (N-1)$ new solutions to explore. Note that part of the neighboring solutions are the same obtained with the Swap method, namely when **who** and **where** differ of only 1 position. Once again applying this method more than once can generate any of the **N**! possible solutions.

Group Sliding

The last method discussed in this work is a generalization of the Sliding, where instead of sliding only one patient, a group of successive patients is slid together. This method has been considered because if the starting solution, before improvements, has very good fit between a group of patients (found by the GFG), then moving the whole group might lead to better results.
The inputs necessary to perform such a variation are several:

- **who**: The first patient of the group to be slid.
- **how_many**: How many patients, excluding the first, are part of the group to be slid.
- **where**: Position where the first patient of the group has to end up.

This time the inputs are constrained, since choosing a number of patients for the group too big could cause the group to exceed the number of patients. Example:

$$\mathbf{N} = 10,\ \textbf{who} = 8,\ \textbf{how\_many} = 4,\ \textbf{where} = 2$$

is an unfeasible request, as there aren't 4 patients after the eighth to bring along. Similarly, a request like

$$\mathbf{N} = 10,\ \textbf{who} = 2,\ \textbf{how\_many} = 4,\ \textbf{where} = 8$$

is again unfeasible since there is not enough space to put the last three patients of the group.
It clearly holds that for **how_many** = 0 the method resembles the simple Slide. Once again, the algorithm differs from the simple Swap since even the patients not part of the group might face a change of order if they are between the old and the new position of the group. Here an example details the desired outcome of the method, with N=7:

Group_slide(who=2, how_many=2, where=4):

$$\mathbf{ORD} = (4, 6, 3, 1, 5, 2, 7) \rightarrow (6, 3, 5, 1, 2, 4, 7)$$

The logic to implement the above follows (without the feasibility checks):

```
def Group_slide(who, how_many, where):
    if (who >= where):
        for i in {1..N}:
            if (where <= ORD[i] < who):
                ORD[i]+= how_many+1
            elif (who <= ORD[i] <= who + how_many):
                ORD[i] += -who +where
    else:
        for i in {1..N}:
            if (who <= ORD[i] <= who + how_many):
                ORD[i] += where - who
            elif (who + how_many < ORD[i] <= how_many + where):
                ORD[i] += -how_many - 1
```

If a fixed dimension of how_many is considered, the number of possible neighboring solutions is $(\mathbf{N} - \textbf{how\_many}) \cdot (\mathbf{N} - \textbf{how\_many} - 1)$. Summing up over all possible values of how_many, the total number of solutions obtainable by this algorithm is:

$$\sum_{n=1}^{\mathbf{N}} n(n-1) = \frac{\mathbf{N}^3 - \mathbf{N}}{3}$$

Even if this method is not particularly expensive, computation wise, to explore $\approx \mathbf{N^3}$ solution would lean more toward a brute exploration method, which is surely functional but out of the scope of this work. For this specific reason, and considering the knowledge obtained by testing the GFG, it has been decided to use this method (in later implementations) with a fixed value of **how_many=2**, for a total group dimension of 3. This value makes sense as it gives the opportunity to the scheduler to find groups of 3 people with a very good fit and move it around without adding idle time between the tasks in the AOP.

5.2.3 Sequential Neighborhood Kernel Search Algorithm (SNKSA)

The dimension of the ID to explore, thanks to the results obtained with the AOPSP, has been reduced from $\mathbf{2^N} \cdot \mathbf{N}!$ to just $\mathbf{N}!$. Whilst this is a big reduction, the dimensions are still too big to explore in completeness. Moreover, even if the problem of finding the ideal positioning no longer exist, when the order changes the patients might change placement, and this is an unpredictable behavior in the search phase of a solution (if not by manually testing and solving the AOPSP associated to the new order). This considerations all reflect in the concept, stressed many times over the course of this thesis, that the OF is not a polynomial function of the input. Additionally, the combinatorial nature of the input makes predicting the effect of changing order extremely difficult.

Overview

The last algorithm presented is, as anticipated, a combination of all the work done for the other problem and the current one.
The idea is to select a method, among the three presented before, and applying it iteratively to the initial solution until a local minimum of the OF is reached. When this happens, and so no more improvements are available, the algorithm switches to another of the three methods, actively changing the way of generating neighborhoods in an attempt to escape the local minimum. The logic can be repeated indefinitely, as long as one wants and as long as at least one of the three methods can find a new minimum. Each new search will add at most $\mathbf{N^2}$ computations in the worst case (the Group Sliding), making this algorithm quite light in it's most basic form.
Note that if a method A is utilized until no more improvements are found, the application of one iteration of another method B introduces a shift in the starting Order (if a better solution is found by B). Consequently, applying again A may lead to the potential rediscovery of improvements. The consequence of this is that the methods can be applied over and over as much as one wants, and there is no risk of incurring in infinite recursions since the jump is performed only when the OF improves. At one point either no improvements are found or no improvement exists.
The order of execution of distinct Neighborhood searches is relevant when discussing the results of the improvements, as it dictates what solution gets fed to the successive methods in line.
Since it is not possible to determine an optimal routine of methods to perform (each solution will get different improvements), and since the model is up to now relatively light, the idea is to improve the results obtained by testing several routines.
A Kernel is defined as the ordered list of methods applied to a solution to search for an improvement. If the length of the Kernel is bounded to be lower than a certain amount, it is possible to enumerate all the possible Kernels obtainable if the amount of methods is also bounded.
In particular with 3 methods and a Kernel's length of 3, knowing that applying twice

in a row a method is useless, the amount of possible configurations is 12. These 12 sequences can be seen as 12 distinct ways of generating new Complex Neighborhoods.
Bounding the length of the Kernel might seem a counter intuitive idea but it's essential in keeping the amount of explored solutions reasonably low. For example, a length of 4 would produce 24 possible Kernels, a length of 5 would produce 48. Since the final algorithm will test and compare, at each iterations, all possible Kernels, keeping this value low is essential.
The order of application of the methods is relevant for the final result, hence why limiting the search to only one method at the time might result in losing promising solutions that might require a change in initial solution to obtain better global results.

Algorithm

The initialization occurs by applying the GFG algorithm to the considered instance, generating a temporary solution **S**.
In the case studied the 12 Kernels that can be applied are always the same; if a different setup is used, then now the program should generate the set of Kernels to test. For each possible Kernel, let's call **First, Second** and **Third** the three methods to perform. Remember that **First≠Second** and **Second≠Third**.
The logic for searching trough a Kernel is:

1. Use method **First** to generate the immediate neighborhood of S.
2. If a solution S^{first} exists in the neighborhood s.t. OF(S^{first})<OF(S), then S=S^{first} and go back to step 1, otherwise proceed.
3. Use method **Second** to generate the immediate neighborhood of S.
4. If a solution S^{second} exists in the neighborhood s.t. OF(S^{second})<OF(S), then S=S^{second} and go back to step 3, otherwise proceed.
5. Use method **Third** to generate the immediate neighborhood of S.
6. If a solution S^{third} exists in the neighborhood s.t. OF(S^{third})<OF(S), then S=S^{third} and go back to step 5, otherwise proceed.

Each Kernel will result in a solution (not necessarily unique). After all 12 have been generated, the program selects the one with the best OF, and proceeds to repeat the previous step using this new solution as a starting point for more 12 distinct searches. This step is repeated until no more improvements are found. Each time

a new solution is selected among the 12 best ones, the Kernel that provides that solution can be saved. The final solution obtained by the algorithm can be seen as the one obtained by imposing on the initial order a Kernel constructed by stacking all the Kernels that provide the winning solutions over the various iterations.
Since the single methods are iterated without bounding the number of repetitions, and so is the amount of Kernels stacked one after the other, the algorithm is not polynomial. In reality, no more than 10 repetitions are ever performed for both steps, so it is mathematically incorrect but practically coherent to say that the algorithm is quadratic in the Instance dimension.

Chapter 6

Validation and results

6.1 Consideration from failed attempts

The attempts performed on the AOPSP lead to an important observation regarding the way optimal solutions are built. This observation also helped during the solving of the more general IAOPORSP.
Let's consider a simple instance in which only three patients need scheduling. The order of execution for the patients is enforced, so the case under study falls into the AOPSP. This example is voluntarily made short for the sake of explanation, but the concept holds for longer instances too. Calling the three patients **P0**, **P1** and **P2**, the Processing times of the three are reported in the table below.

	P0	**P1**	**P2**
A [min]	15	13	35
S [min]	23	43	56
AW [min]	10	24	18

Table 6.1: Processing times for tasks of three patients, example of non triviality of solution.

In picture 6.1 two feasible solutions of the instance are presented. Pay attention to the fact that the two Gantt are not the same length and so the schedule above has a worst OF.
The focus of this example is on the placement of the first patient. If the schedule is built according to the FG algorithm presented before (5.1.1), the awakening of the first patient is easily fixed in the AOP. This is because the length of the surgery of patient **P0** allows for the anesthesia of patient **P1** to be placed between A0 and AW0.
If the instance happened to be only two patients long, then this would actually lead to a better OF; it is in fact true that the OF considering only the first two patients has value 105, whereas the OF of the other example presented has value of 115.

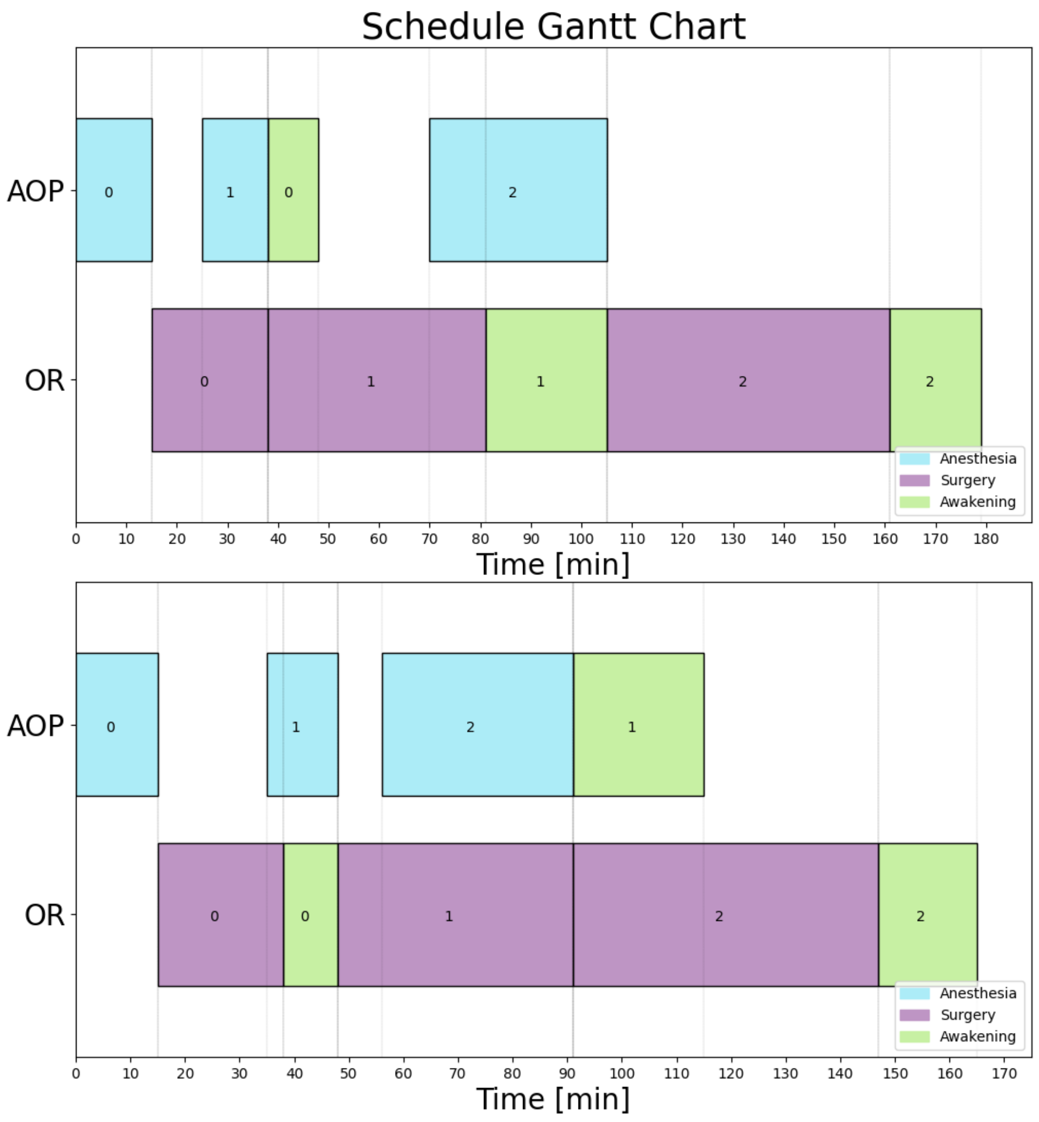

Figure 6.1: Two feasible schedules, example of non triviality of solution.

The schedule on top now continues, and the AW of patient **P1** is forced in the OR, since if it was scheduled in the AOP there would not be enough room to fit AW1 in the AOP. This would lead to the previously mentioned case of picture 5.9. Because of this the overall schedule's OF reaches a value of 179.
Let's now consider the second example. Here the AW of patient **P0** is forced in the OR even though it could have been placed in the AOP. As previously stated this decision, if applied only to two patients, leads to a worse schedule. On the other hand now, after the ending of A1 there is not AW0, meaning that more space is available to fit A2. As a consequence, AW1 can be placed in the AOP, leading to a huge reduction in OF. This schedule, constructed using counter intuitive reasoning, leads to a better OF of 166.

This example shows clearly that the decision on where to place the AW of patients cannot be made locally. To know the optimal placement one needs to know in advance the duration of the procedures of the patients ahead of the one under study. On the other hand, knowing only the duration of the patients ahead would not be sufficient, as those will need knowledge of the patients further in the schedule and so on. This implies that the whole schedule depends on all the patients, and not by a sequence of local choices.

6.2 Validation

Classification of data: real and generalized instances generation

The problem under study has been initially dealt with realistic assumptions on the duration of the tasks to perform. While no real data were collected, it has been considered that a realistic duration of the Surgery would be much greater than the duration of the two other tasks. This consideration creates an issue in the verification of the effectiveness of the algorithms.
It can be seen in picture 6.2, when considering a surgery much longer than A and AW, that the scheduling becomes trivial with all tasks except for the surgery itself scheduled in the AOP. Whilst it is true that a lot of idle time is present in the AOP,

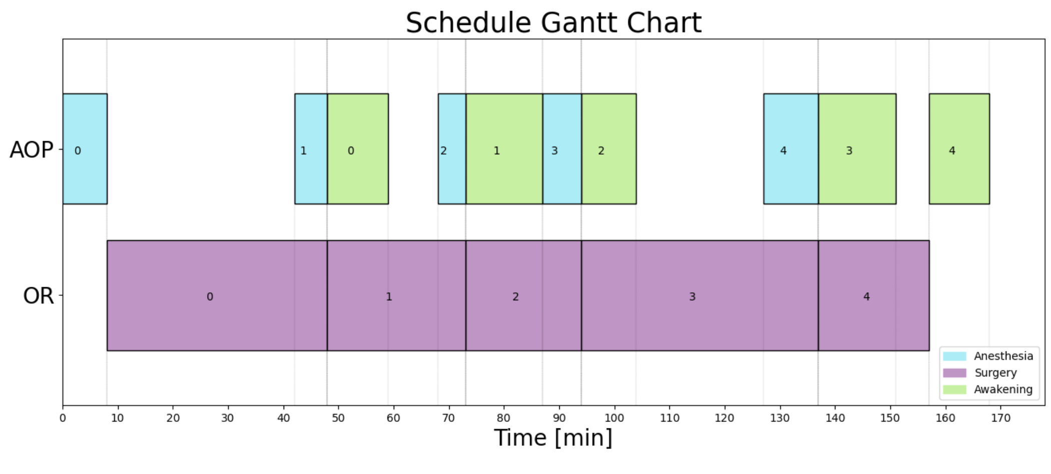

Figure 6.2: Example of schedule with only long surgeries.

it is also true that no improvement can be made; the presence of idle time is fixed by the duration of the surgeries.
For this reason more general datasets have been used, considering less realistic values for the tasks to better highlight the performances of the algorithms.
All the tasks duration have been generated using a random variable with uniform

distribution bounded by min-MAX boundaries. A total of 4 cases (**N1-N2-N3-N4**) have been considered to account for all possible tasks duration. In table 6.2 the 4 cases are illustrated.

Task limit	**Case**			
	N1	**N2**	**N3**	**N4**
$\mathbf{A_{min}}$	10	10	5	5
$\mathbf{A_{MAX}}$	100	100	40	40
$\mathbf{S_{min}}$	10	5	10	5
$\mathbf{S_{MAX}}$	100	40	100	40
$\mathbf{AW_{min}}$	10	5	5	10
$\mathbf{AW_{MAX}}$	100	40	40	100

Table 6.2: Limits for the tasks in the 4 different cases used for validation.

Where:

- Case **N1** will be used as a reference of general tasks. All tasks can have the same duration.
- Case **N2** is a modified version in which the anesthesia has a longer than average duration.
- Case **N3** is a modified version in which the surgery has a longer than average duration. This case is the most realistic with respect to the proposed problem.
- Case **N4** is a modified version in which the Awakening has a longer than average duration.

Instance dimensions and validation set

As previously stated, the instances that would be realistically used in an hospital facility have a dimension not bigger than 15 patients. For the purpose of testing, the number of patients considered in the various instances will be:

$$N = [11,\ 13,\ 15,\ 17,\ 19]$$

For a total of 5 different lengths.

Considering 4 cases, 5 different lengths for the instances, and 20 instances per combination, a total of 400 schedules have been used for testing purposes. 100 tests per each case-length have been used.

6.3 Results presentation

6.3.1 AOPSP

As previously stated the B&B algorithm presented (5.1.3) for the AOPSP resulted to be optimal, meaning that all instances fed are solved finding the best possible objective function, matching the results obtained by the commercial solver.
From the speed prospective the solver and the algorithm are almost on par. More than 100 patients are necessary to make a difference in the two appear, with the algorithm being slightly faster. Considering the almost instantaneous execution of both for such a big instance however, the difference is considered negligible.
It is interesting to look at the amount of alive and pruned solutions after each new patient is added to the system. Calling $\mathbf{AVG_k}$ the average number of pruned leafs on each level and $\mathbf{AVG_L}$ the average number of alive ones after pruning, we get:

$$\mathbf{AVG_k} = 2.29, \ \ \mathbf{AVG_L} = 2.3$$

The numbers make sense, as after the pruning the alive solutions are sprouted into twice the initial amount, and half of the new leafs are on average pruned. This all comes to the result that the solution generation is stable and does not explode into $\mathbf{2^N}$ solutions.
Whilst not mathematically rigorous, one might say that the algorithm is $\mathcal{O}(4 \cdot \mathbf{N})$, since in all instances tested no more than 4 alive solutions have ever been found.

6.3.2 IAOPORPSP

The IAOPORSP problem has not been solved optimally. Various algorithms are here presented.
Each instance has been run first on the commercial solver with an execution time limit of 15min (900s); on each run both the best solution and the best Lower Bound (LB) found by the solver have been kept. Obviously if the solver manages to find the optimum solution in less than 900s the OF and LB will coincide. Successfully, for each algorithm analyzed, the OF is saved to be later processed. In table 6.3 all the algorithms results considered are presented. For each instance the results are normalized with respect to the LB according to the Gap formula:

$$Gap_{\,alg} = \frac{OF_{alg} - LB}{LB}$$

Each of the 20 combinations tested (4 cases and 5 lengths) has a total of 20 tests. From the associated results the mean and variance for each combination have been extracted and are presented below. All the data presented will be given in normalized

Algorithm	**Explanation**
LB	Lowest bound found by the solver during the 900s execution. It's a theoretical guess found by the solver during exploration, which is not necessarily the same as the best OF found at the moment. It's a good reference to consider when one want's to know how good another solution is.
Gur	The OF found by the solver in the 900s. If exploration is completed before time limit is reached, Gur and LB should be equal.
Gre	OF obtained by applying only the GFG algorithm to the instance.
Swa	OF obtained by applying only the Swapping method to the GFG solution, until no more improvements are found.
Sli	OF obtained by applying only the Sliding method to the GFG solution, until no more improvements are found.
Gsl	OF obtained by applying only the Group Sliding method to the GFG solution considering groups of 2 patients, until no more improvements are found.
G2K2	Application of the Sequential Neighborhood Kernel Search Algorithm with a total group dimension (for the Group Sliding method) of 2 and a Kernel length of 2.
G4K2	Application of the Sequential Neighborhood Kernel Search Algorithm with a total group dimension (for the Group Sliding method) of 4 and a Kernel length of 2.
G3K3	Application of the Sequential Neighborhood Kernel Search Algorithm with a total group dimension (for the Group Sliding method) of 3 and a Kernel length of 3. Referred in this work as SNKSA.

Table 6.3: Explanation of algorithms used in validation.

form, without adding the subscript each time. Clearly it holds that the lower the Gap for an algorithm, the better it works.

Case	**Gur**	**Gre**	**Swa**	**Sli**	**Gsl**	**G2K2**	**G4K2**	**G3K3**
N1	0.0056	0.0968	0.0503	0.0528	0.0541	0.0328	0.0351	**0.0327**
N2	0.0027	0.0633	0.0117	0.0105	0.0126	0.0061	0.0061	**0.0059**
N3	0.0024	0.0812	0.0282	0.0278	0.0323	0.0141	0.0145	**0.0130**
N4	0.0039	0.1860	0.0804	0.0838	0.0875	**0.0552**	0.0562	0.0562

Table 6.4: Averaged Gap of each algorithm on each of the four cases tested. Data have been averaged over the number of patients.

N	**Gur**	**Gre**	**Swa**	**Sli**	**Gsl**	**G2K2**	**G4K2**	**G3K3**
11	0.0000	0.0995	0.0462	0.0442	0.0501	0.0275	0.0283	**0.0244**
13	0.0000	0.0958	0.0448	0.0445	0.0468	0.0265	0.0283	**0.0256**
15	0.0001	0.0971	0.0481	0.0538	0.0555	**0.0286**	0.0319	0.0317
17	0.0101	0.0991	0.05397	0.0596	0.0593	0.0389	0.0411	**0.0382**
19	0.0177	0.0926	0.0583	0.0616	0.0590	**0.0423**	0.0460	0.0436

Table 6.5: Results obtained by the algorithms, focus on the average Gap for the various instances lengths of case **N1**.

N	**Gur**	**Gre**	**Swa**	**Sli**	**Gsl**	**G2K2**	**G4K2**	**G3K3**
11	0.0000	0.0674	0.0109	0.0074	0.0112	0.0038	**0.0035**	0.0036
13	0.0000	0.0626	0.0095	0.0091	0.0107	**0.0040**	0.0048	0.0045
15	0.0001	0.0444	0.0065	0.0081	0.0088	0.0034	0.0027	**0.0023**
17	0.0034	0.0737	0.0129	0.0112	0.0124	0.0060	0.0063	**0.0057**
19	0.0099	0.0685	0.0188	0.0166	0.0200	**0.0132**	**0.0132**	0.0134

Table 6.6: Results obtained by the algorithms, focus on the average Gap for the various instances lengths of case **N2**.

	Gur	**Gre**	**Swa**	**Sli**	**Gsl**	**G2K2**	**G4K2**	**G3K3**
11	0.0000	0.0394	0.0208	0.0203	0.0234	0.0154	0.0154	0.0166
13	0.0000	0.0379	0.0152	0.0211	0.0191	0.0185	0.0111	0.0134
15	0.0002	0.0396	0.0183	0.0155	0.0147	0.0148	0.0130	0.0126
17	0.0060	0.0343	0.0140	0.0209	0.0132	0.0119	0.0115	0.0116
19	0.0053	0.0249	0.0104	0.0115	0.0114	0.0089	0.0091	0.0089

Table 6.7: Standard deviation of Gap, algorithms vs instances lengths, for case **N1**

	Gur	**Gre**	**Swa**	**Sli**	**Gsl**	**G2K2**	**G4K2**	**G3K3**
11	0.0000	0.0306	0.0144	0.0116	0.0130	0.0049	0.0057	0.0057
13	0.0000	0.0280	0.0143	0.0123	0.0135	0.0089	0.0092	0.0087
15	0.0000	0.0204	0.0098	0.0132	0.0104	0.0057	0.0043	0.0041
17	0.0074	0.0361	0.0153	0.0135	0.0155	0.0101	0.0101	0.0079
19	0.0163	0.0291	0.0233	0.0211	0.0251	0.0187	0.0186	0.0195

Table 6.8: Standard deviation of Gap, algorithms vs instances lengths, for case **N2**

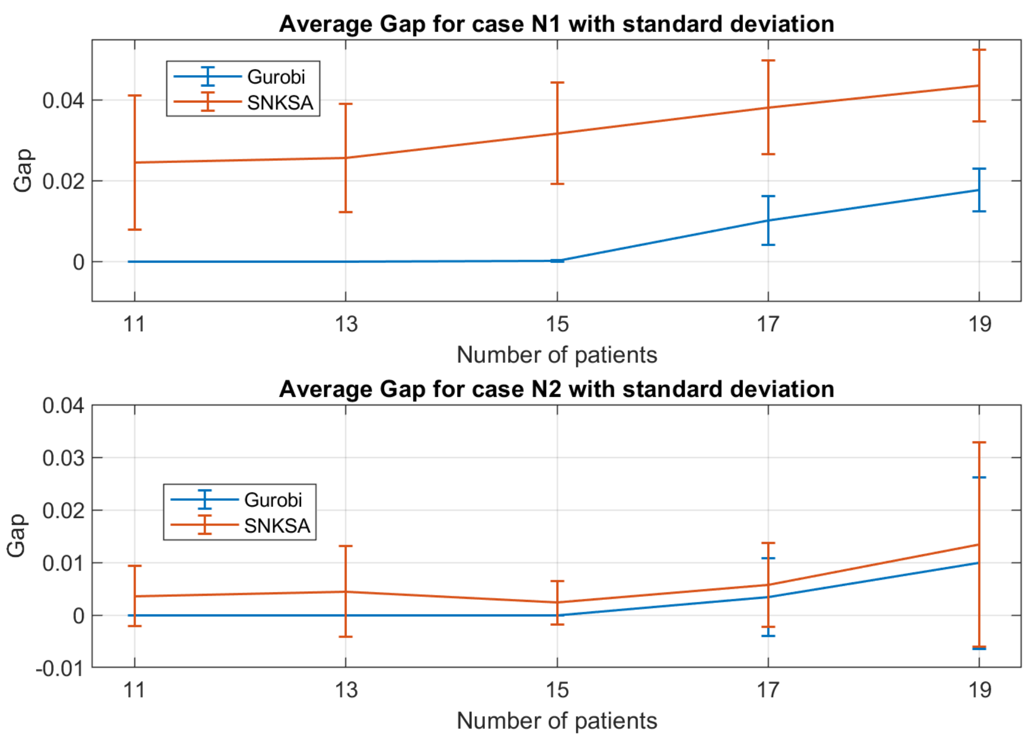

Figure 6.3: Error plots of Gap obtained by Gurobi and SNKSA-G3K3, with standard deviation, vs length of instances considered in cases **N1** and **N2**.

N	**Gur**	**Gre**	**Swa**	**Sli**	**Gsl**	**G2K2**	**G4K2**	**G3K3**
11	0.0000	0.0698	0.0265	0.0280	0.0346	0.0146	0.0139	**0.0115**
13	0.0000	0.0679	0.0198	0.0212	0.0277	0.0112	0.0115	**0.0104**
15	0.0002	0.0890	0.0270	0.0270	0.0317	0.0123	0.0127	**0.0113**
17	0.0027	0.0880	0.0282	0.0263	0.0318	0.0117	0.0122	**0.0114**
19	0.0090	0.0912	0.0397	0.0368	0.0356	0.0209	0.0221	**0.0203**

Table 6.9: Results obtained by the algorithms, focus on the average Gap for the various instances lengths of case **N3**.

N	**Gur**	**Gre**	**Swa**	**Sli**	**Gsl**	**G2K2**	**G4K2**	**G3K3**
11	0.0000	0.1893	0.0818	0.0802	0.0786	**0.0512**	**0.0512**	0.0540
13	0.0000	0.1667	0.0786	0.0850	0.0990	0.0537	0.0538	**0.0553**
15	0.0001	0.1875	0.0832	0.0949	0.0893	**0.0573**	0.0606	0.0596
17	0.0022	0.1871	0.0715	0.0742	0.0801	0.0489	0.0482	**0.0471**
19	0.0171	0.1993	0.0868	0.0848	0.0907	**0.0649**	0.0673	0.0651

Table 6.10: Results obtained by the algorithms, focus on the average Gap for the various instances lengths of case **N4**.

	Gur	**Gre**	**Swa**	**Sli**	**Gsl**	**G2K2**	**G4K2**	**G3K3**
11	0.0000	0.0401	0.0234	0.0221	0.0241	0.0191	0.0191	0.0190
13	0.0000	0.0358	0.0170	0.0183	0.0184	0.0124	0.0119	0.0114
15	0.0003	0.0313	0.0164	0.0150	0.0180	0.0084	0.0109	0.0094
17	0.0087	0.0336	0.0246	0.0178	0.0231	0.0130	0.0133	0.0153
19	0.0145	0.0415	0.0275	0.0248	0.0259	0.0192	0.0188	0.0188

Table 6.11: Standard deviation of Gap, algorithms vs instances lengths, for case **N3**

	Gur	**Gre**	**Swa**	**Sli**	**Gsl**	**G2K2**	**G4K2**	**G3K3**
11	0.0000	0.0793	0.0352	0.0278	0.0247	0.0247	0.0240	0.0232
13	0.0000	0.0612	0.0241	0.0230	0.0263	0.0172	0.0165	0.0182
15	0.0000	0.0554	0.0211	0.0234	0.0269	0.0130	0.0125	0.0120
17	0.0067	0.0321	0.0224	0.0249	0.0232	0.0161	0.0126	0.0154
19	0.0122	0.0357	0.0190	0.0161	0.0221	0.0128	0.0127	0.0126

Table 6.12: Standard deviation of Gap, algorithms vs instances lengths, for case **N4**

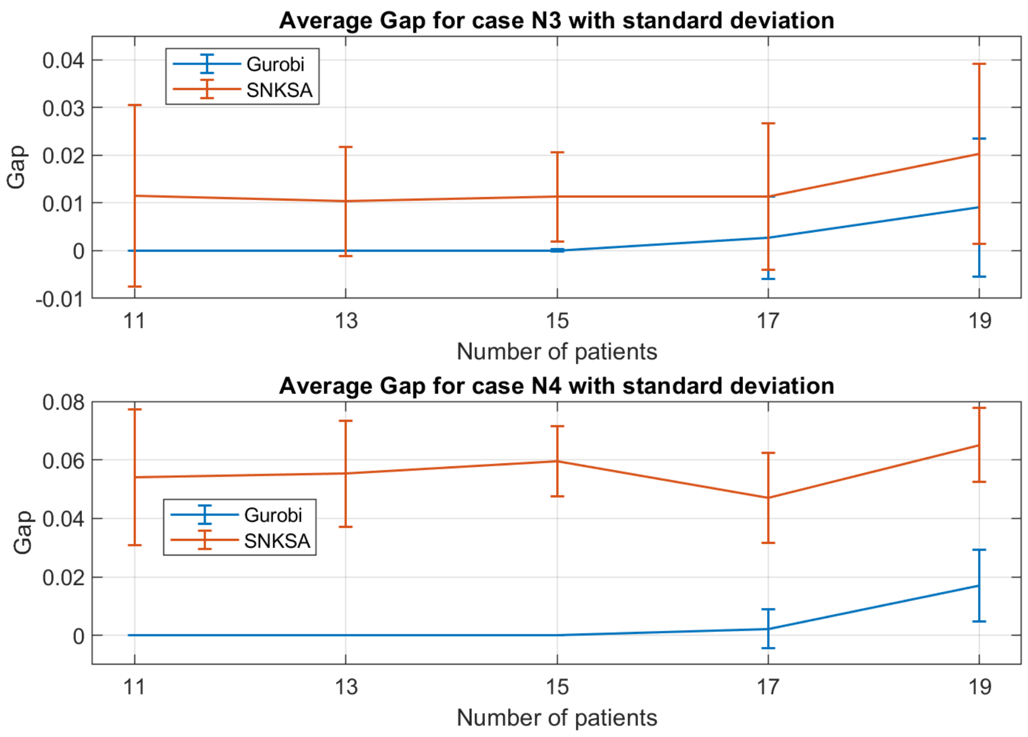

Figure 6.4: Error plots of Gap obtained by Gurobi and SNKSA-G3K3, with standard deviation, vs length of instances considered in cases **N3** and **N4**.

6.3.3 Considerations

The tables above show the results obtained in the validation section.
General results of table 6.4 show that among the algorithms selected, SNKSA has an overall better performance when averaging over the instance dimensions. This is not the case for case **N4**, when the anesthesia exceeds the duration of the other two task. It has to be noted though that the difference between the two best algorithms is marginal, leaving the overall best results to the SNKSA.
Tables below provide a better insight on the single cases considered, with case **N3** being the most similar to the initial problem presented. In this specific scenario the SNKSA obtains results always within 0.01 from the solution provided by Gurobi. Whilst this is true for the mean value of the experiment performed, it is also true that the variance of the results obtained is slightly higher than that obtained by the commercial solver.
By far the worst case analyzed is **N4**, where the OF provided by the algorithm is almost 0.05 points away from the solution provided by the solver.
Overall the algorithm turns out to work slightly better than expected. It is true that the performance of the commercial solver has not been reached, however considering the huge reduction in computation time provided, and the fact that it is possible to increase the ID exploration of the algorithm by removing the limiter on the Group Slide method, the results are considered satisfactory enough for the moment.

6.4 Conclusions and future work

This thesis provided an insight on two peculiar problems regarding the scheduling of patients for surgical procedures commonly occurring in healthcare facilities. The study aimed at finding a dedicated solution for both problems, and comparing the results obtained with what is currently a popular commercial solver on the market. While the solver capabilities allow for a very good solution of the problems, the general approach used requires enormous computational requirements. Purpose of this job was finding a dedicated algorithm for each problem capable of reducing the needs for the solving phase. The first of the two problems, the AOPSP, has been successfully comprehended and a linear algorithm has been found capable of optimally solve any instance even of dimension much greater that those of interest for a hospital facility.
For the IAOPORSP, generalization of the first, several attempts at finding an optimal algorithm have been presented. While none turns out to have the same effectiveness of the commercial solver, the quality of the solutions provided is considerably

good. For the specific case under study, where surgery time is considered longer than anesthesia and awakening, the best algorithm presented returns solutions only 2% off, on average, from the known best possible solution to the instance, which is an impressive result considering the challenge of the problem.
Several insight are presented with the purpose of getting a better comprehension of the problem, and several times during the course of this work the exploring capabilities of the algorithms have been limited to preserve the lightness of execution. Considering the above factors it is reasonable to believe that the work presented could further improve it's results if associated with actual data from real healthcare facilities and more importantly a better context for the problem. Once the requirements for a specific facility, in therms of how many patients does it need to process and what is the distribution of typical surgeries to perform, are defined it would be possible to tune the algorithms presented to obtain the best possible performance for the specific case.
This work is considered a good starting point for the topic discussed, which has not been previously examined in a public context as far as the candidate knows. Whilst very good results have been obtained, it is believed that further analysis could results in even greater improvements on the topic. A great interest must be given to the scheduling system inside hospitals, as a huge reduction in therms of costs and time for the facilities would result in a better allocation of resources that are, at the moment, being wasted due to inefficiencies.

List of Figures

List of Tables

Bibliography

[1] Sverrir Olafsson. Resource allocation as an evolving strategy. *Evolutionary Computation*, 4(1):33–55, 1996.

[2] Canberra Hyatt and Patrick Weaver. A brief history of scheduling. *Melbourne, Australia: Mosaic Project Services Pty Ltd*, 2006.

[3] Joseph Priestley. A specimen of a chart of biography. `https://en.wikipedia.org/wiki/A_Chart_of_Biography`, 1765. Accessed: 15/02/2024.

[4] Chris Potts and Vitaly Strusevich. Fifty years of scheduling: A survey of milestones. *Journal of the Operational Research Society*, 60:S41–S68, 05 2009.

[5] M. Delgado-Rodríguez, A. Bueno-Cavanillas, R. Lopez-Gigosos, J. Dios Luna-Castillo, J. Guillén-Solvas, O. Moreno-Abril, B. Rodríguez-Tuñas, A. Cueto-Espinar, R. Rodríguez-Contreras, and R. Gálvez-Vargas. Hospital stay length as an effect modifier of other risk factors for nosocomial infection. *European Journal of Epidemiology*, 6:34–39, 1990.

[6] J. Raft, F. Millet, and C. Meistelman. Example of cost calculations for an operating room and a post-anaesthesia care unit. *Anaesthesia, critical care pain medicine*, 34 4:211–5, 2015.

[7] Alexandra Sariel. An introduction to linear programming. 1994.

[8] K. G. Murty. Quadratic programming models. pages 445–476, 2010.

[9] Satyajith Amaran, N. Sahinidis, B. Sharda, and S. Bury. Simulation optimization: a review of algorithms and applications. *4OR*, 12:301 – 333, 2014.

[10] Wang Meng-meng. Simulated annealing algorithm used in supply chain scheduling problem. *Technology and Economy in Areas of Communications*, 2011.

[11] Anupriya Gogna and A. Tayal. Metaheuristics: review and application. *Journal of Experimental Theoretical Artificial Intelligence*, 25:503 – 526, 2013.

[12] F. Dexter and R. Traub. How to schedule elective surgical cases into specific operating rooms to maximize the efficiency of use of operating room time. *Anesthesia Analgesia*, 94:933–942, 2002.

[13] A. Saremi, P. Jula, T. Elmekkawy, and G. Gary Wang. Appointment scheduling of outpatient surgical services in a multistage operating room department. *International Journal of Production Economics*, 141:646–658, 2013.

[14] Sefa Dündar, Burcin Gokkurt, and Yasin Soylu. Mathematical modelling at a glance: A theoretical study. *Procedia - Social and Behavioral Sciences*, 46, 12 2012.

[15] N. Yanofsky. Towards a definition of an algorithm. *ArXiv*, abs/math/0602053, 2006.

[16] Ya.B. Zinder and V.V. Shkurba. Scheduling theory. `http://encyclopediaofmath.org/index.php?title=Scheduling_theory&oldid=51613`, 2021. This article was adapted from an original article by Ya.B. ZinderV.V. Shkurba (originator), which appeared in Encyclopedia of Mathematics - ISBN 1402006098. See original article. 2021 is last editing, publishing date not found.

[17] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Springer, 2012.

[18] A. Sprecher, R. Kolisch, and A. Drexl. Semi-active, active, and non-delay schedules for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 80:94–102, 1995.

[19] Stéphane Dauzère-Pérès, Junwen Ding, Liji Shen, and Karim Tamssaouet. The flexible job shop scheduling problem: A review. *European Journal of Operational Research*, 314(2):409–432, 2024.

[20] C. Chandra and J. Grabis. Mathematical programming approaches. pages 151–172, 2016.

[21] Saul I. Gass. *Linear Programming: Methods and Applications.* Dover Books on Computer Science Series. Dover Publications, 5 edition, 2010.

[22] I. Grossmann. Review of nonlinear mixed-integer and disjunctive programming techniques. *Optimization and Engineering*, 3:227–252, 2002.

[23] P. Hansen and N. Mladenović. Variable neighborhood search: Principles and applications. *Eur. J. Oper. Res.*, 130:449–467, 1998.

[24] David Pisinger. Where are the hard knapsack problems? *Comput. Oper. Res.*, 32:2271–2284, 2005.

[25] Arnaud Fréville. The multidimensional 0–1 knapsack problem: An overview. *European Journal of Operational Research*, 155(1):1–21, 2004.

[26] Wikipedia contributors. Frontier (supercomputer) – wikipedia, the free encyclopedia, 2024. [Online; accessed 6-March-2024].

[27] Rhyd Lewis and F. Carroll. Creating seating plans: a practical application. *Journal of the Operational Research Society*, 67:1353–1362, 2016.

[28] Wedding Seat Planner. `http://www.weddingseatplanner.com/`, Not available. Accessed: 2024.

[29] M. M. Flood. The traveling-salesman problem. *Operations Research*, 4:61–75, 1956.

[30] Chou-Jung Hsu, W. Kuo, and Dar-Li Yang. Unrelated parallel machine scheduling with past-sequence-dependent setup time and learning effects. *Applied Mathematical Modelling*, 35:1492–1496, 2011.

[31] Yasin Unlu and Scott Mason. Evaluation of mixed integer programming formulations for non-preemptive parallel machine scheduling problems. *Computers Industrial Engineering*, 58:785–800, 05 2010.